Accelerating Approximate Data Analysis with Parallel Processors

vorgelegt von
M. Sc.
Martin Kiefer
ORCID: 0000-0001-8662-4591

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

Doktor der Naturwissenschaften
- Dr. rer. nat. -

genehmigte Dissertation

Promotionsausschuss:

Vorsitzende:	Prof. Dr. Begüm Demir, Technische Universität Berlin
Gutachter:	Prof. Dr. Volker Markl, Technische Universität Berlin
Gutachter:	Prof. Dr. Zsolt István, Technische Universität Darmstadt
Gutachter:	Prof. Dr. Jens Teubner, Technische Universität Dortmund
Gutachterin:	Prof. Dr. Eleni Tzirita Zacharatou, IT University of Copenhagen

Tag der wissenschaftlichen Aussprache: 2. Februar 2023

Berlin 2023

Acknowledgements

I want to express my deepest gratitude to the numerous individuals who have made the completion of this Ph.D. thesis possible. Each person has contributed in their own unique way, and their support has been invaluable.

First and foremost, I would like to express my profound gratitude to Volker, the head of our group and my thesis advisor. His wisdom, mentorship, and unwavering belief in me have been instrumental in shaping this thesis. Without him, there wouldn't have been a Ph.D. to pursue. I would also like to thank the entire committee, consisting of Begüm, Volker, Eleni, Jens, and Zsolt, for their constructive feedback.

I am immensely grateful to my mentors, Max, Sebastian B., and Eleni, who have been indispensable sources of support and knowledge. Their guidance, friendship, and wisdom have helped me navigate the complexities of my research with confidence.

My heartfelt thanks go to my wife, Janine, and my son, Felix, who was born during the course of my Ph.D. journey. Their love, patience, and understanding have been a source of strength and inspiration, enabling me to persevere through the challenges of research and thesis writing.

My deepest gratitude goes to my parents, Olga and Jakob, for their unwavering support throughout my entire academic career. Their love and encouragement have been a constant presence, even as I pursued my career far from home.

I am thankful for my friends, particularly Sebastian K. and Bastian, who have been with me since the beginning of my academic journey. Special thanks to Sebastian K. are due — for our countless discussions, invaluable feedback, and the occasional Gin Tonic. I also extend my appreciation to Susi for her patience and understanding as she listened to my endless musings over the years.

My colleagues, especially Alexander, Andreas, Clemens, Dimitirios, Felix N., Jonas, Hary, Nils, Philipp, Ventura, and Viktor, have enriched my experience with discussions, support, and constructive feedback. Their camaraderie has made this journey an enjoyable and fulfilling one. I would like to thank Claudia, Melanie, Aman, and Lutz for their instrumental role in the success of our projects and the functioning of our team.

Finally, I want to express my profound gratitude to my late grandparents, David and Erna. Although we could not celebrate this accomplishment together, their memory will forever be cherished in my heart.

Abstract

Approximate data analysis trades off accuracy for faster or cheaper results. Instead of performing data analyses on the entire data, space-efficient *summaries* are constructed and evaluated. Applications from various domains, such as databases, computational biology, and machine learning, require accepting a loss in accuracy to gain the desired throughput and response times with limited resources. *Specialized parallel processors*, such as *graphics processing units (GPUs)* and *field-programmable gate arrays (FPGAs)*, promise substantial gains in the efficiency of approximate analyses. However, exploiting these architectures is challenging as it requires considering an architecture's properties and explicitly designing and implementing for parallelism.

In this thesis, we investigate novel approaches to maintaining and evaluating data summaries on parallel processors:

First, we propose *kernel density models* for GPU-accelerated *join selectivity estimation* in relational databases. Our estimators do not require common error-prone assumptions and typically provide higher accuracy than the state of the art. Furthermore, the estimators fit the massively parallel computations supported by modern GPUs. In contrast to central processing units (CPUs), our approach allows for larger model sizes and more accurate estimates within the same time budget.

Second, we propose *Scotch*, a holistic approach to implementing FPGA-accelerated *sketching* at the full rate of the interconnect. The framework generates hardware descriptions for a broad class of sketches based on a common abstraction and domain-specific language. Furthermore, Scotch automatically maximizes the sketch size to boost accuracy. As these tasks commonly require an FPGA expert, Scotch makes FPGA-accelerated sketching more accessible to software engineers.

Third, we propose an *optimistic data-parallel* architecture for FPGA-accelerated sketch summary maintenance. We share resources among inputs instead of pessimistically replicating all resources for every input. Thus, our optimistic architecture reduces the resources required to achieve given throughput for applications that tolerate stalls in processing due to resource congestion.

Overall, this thesis shows that specialized parallel processors can substantially increase the efficiency of approximate data analysis while the entry barrier for using them can be lowered substantially by systems and abstractions.

Zusammenfassung

Approximative Datenanalyse ermöglicht es die Genauigkeit von Ergebnissen gegen schnellere oder günstigere Berechnungen einzutauschen. Anstelle der Durchführung von Datenanalysen auf der Gesamtheit der Daten, werden speichereffiziente *Zusammenfassungen* erzeugt und ausgewertet. Anwendungen aus diversen Bereichen, wie Datenbanken, Bioinformatik und maschinellem Lernen, erfordern diesen Verlust von Genauigkeit, um Anforderungen bezüglich Durchsatz und Antwortzeiten mit begrenztem Ressourcenaufwand zu erreichen. Der aktuelle Trend zu *spezialisierten parallelen Rechenarchitekturen* wie *Grafikprozessoren (GPUs)* und *Field-Programmable Gate Arrays (FPGAs)* verspricht diesen Austausch zusätzlich zu verbessern. Jedoch ist die Verwendung dieser Architekturen nicht trivial, da sie es erfordert, die Eigenschaften der verwendeten Architektur einzubeziehen und Parallelität explizit beim Entwurf und der Implementierung zu berücksichtigen.

In dieser Arbeit werden Ansätze zur Erzeugung und Auswertung von Datenzusammenfassungen mit Hilfe paralleler Rechenarchitekturen vorgestellt:

Als Erstes stellen wir *Kerndichtemodelle* für GPU-beschleunigte Schätzung von *Joinselektivitäten* in relationalen Datenbanken vor. Unsere Schätzer benötigen hierzu keine der gängigen Annahmen und liefern bessere Ergebnisse als der Stand der Technik. Darüber hinaus passen die Schätzer gut zu den von modernen GPUs unterstützten massivparallelen Berechnungen, was im Vergleich zu Hauptprozessoren (CPUs) größere Modelle und dadurch genauere Schätzungen im selben Zeitbudget ermöglicht.

Als Zweites stellen wir *Scotch* vor, einen holistischen Ansatz für die Implementierung FPGA-beschleunigter Erzeugung von *Sketchzusammenfassungen* mit der vollen Rate des verwendeten Interconnects. Das Framework generiert Hardwarebeschreibungen für eine umfassende Klasse von Sketchingalgorithmen basierend auf einer gemeinsamen Abstraktion und einer entsprechenden domänenspezifischen Sprache. Darüber hinaus maximiert Scotch die Größe der Sketchzusammenfassung automatisch, um die Genauigkeit der Ergebnisse zu maximieren. Da diese Aufgaben üblicherweise einen FPGA-Experten erfordern, macht Scotch das FPGA-beschleunigte Erzeugen von Sketchzusammenfassungen zugänglicher für Softwareentwickler.

Als Drittes stellen wir eine *optimistische* Architektur für die FPGA-beschleunigte *datenparallele* Erzeugung von Sketchzusammenfassungen vor. Wir schlagen vor, Ressourcen zwischen parallelen Eingabewerten zu teilen, statt pessimistisch alle Ressourcen für jeden Eingabewert zu replizieren. Unsere optimistische Architektur verringert den Verbrauch von Ressourcen, um einen hohen Durchsatz über Datenparallelität zu erreichen, sofern die Anwendung Unterbrechungen durch Überlastung von Ressourcen toleriert.

Insgesamt zeigt diese Arbeit, dass spezialisierte parallele Rechenarchitekturen die Effizienz von approximierten Datenanalysen erheblich verbessern können und dass die Eintrittsbarriere für ihre Nutzung durch geeignete Systeme und Abstraktionen gesenkt werden kann.

Contents

1 Introduction

Many applications favor faster response times over result correctness, as exact results may be expensive or even infeasible to compute. For example, the query optimizer in relational databases requires the result sizes of query operators before executing the actual query to choose an efficient execution strategy [57, 135]. Thus, the optimizer must resort to estimates. *Approximate data analysis* allows such applications to trade off accuracy for faster or cheaper results [38, 58]. Instead of performing data analyses on the entire data, space-efficient summaries are constructed and evaluated. Choosing a favorable processor for constructing and evaluating summaries allows us to increase performance or support larger, more accurate summaries within the same budget.

In this thesis, we propose novel techniques that combine approximate data analysis with parallel processing hardware to (1) increase efficiency while (2) hiding the complexity resulting from using a parallel processor.

1.1 Motivation

Approximate data analysis is an essential strategy for evaluating high volumes of resting data and high-velocity data streams [38, 58]. While exact data analysis only allows for trading off between performance and resources spent (e.g., server nodes, energy consumption) on a one-dimensional axis, approximate analysis adds a dimension: accuracy. We visualize this trade-off in Figure 1.1. By creating and evaluating summaries in small space, approximate data analysis computes estimates for functions over the entire data efficiently [38, 58].

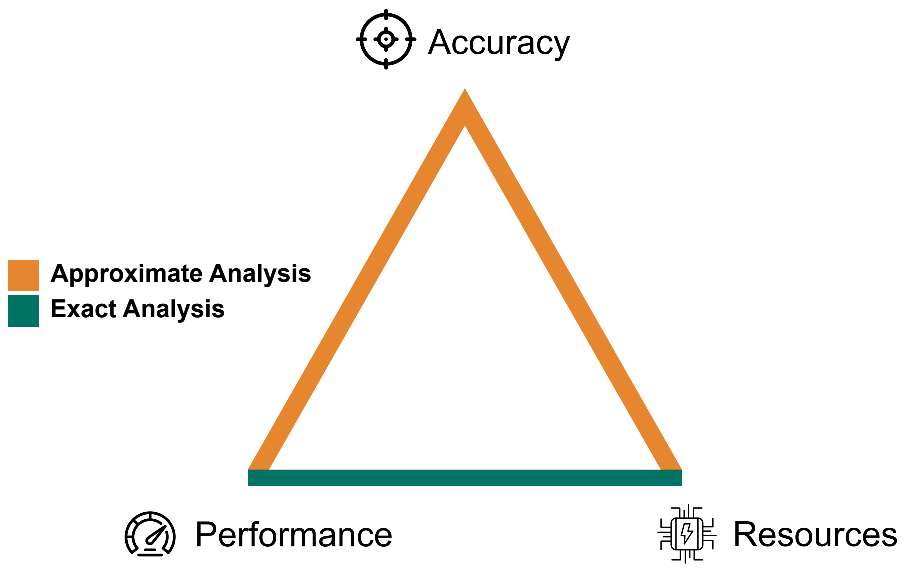


Figure 1.1: The trade-offs inherent to exact and approximate data analysis. Approximate analysis introduces the opportunity to trade off accuracy for higher performance and fewer resources spent.

While sacrificing accuracy to achieve higher performance or resource efficiency may seem counter-intuitive, applications surrounding us tolerate and sometimes even necessitate working with a loss in accuracy. For example, election polls estimate results from random samples that are only practical to compute exactly at an actual election. In computing, numerous applications of approximate data analysis exist: Web browsers employ probabilistic filters to validate URLs against small bitmaps to avoid shipping or exposing large blocklists [35]. Relational databases employ *selectivity estimation* to compute result sizes of individual operators based on statistics before the exact query executes [57, 135]. The estimates guide the query optimizer in finding an optimal execution plan. *Approximate query processing* answers relational aggregate queries with bounded error while offering interactive response times [1, 3, 26, 100]. Other notable examples include large-scale network analysis [49, 92, 154], machine learning [99, 133, 169], and computational biology [139].

At the same time, the trend toward parallel processors offers new opportunities for approximate data analysis. With clock frequencies limited by the power wall [13], processor vendors turned towards parallel architectures to put the additional transistors gained with the still-increasing transistor density to use. As a result, a heterogeneous landscape of parallel processing hardware has emerged: *Central processing units (CPUs)* employ multiple cores, simultaneous multi-threading, and vector instructions [8, 74].

Graphics processing units (GPUs) provide thousands of parallel hardware threads [8, 114]. *Field-programmable gate arrays (FPGAs)* allow for parallelism on the level of custom circuits [81, 151]. While parallel processors offer new opportunities to speed up data processing and analytics, employing them is not trivial: Algorithms, applications, and the inherent benefits and drawbacks of the architecture have to match. Furthermore, such processors expose parallelism to developers and require them to incorporate parallelism in their software explicitly.

Combining approximate data analysis and parallel processors promises highly efficient data analyses by further improving the trade-off between accuracy, performance, and resources. Depending on the algorithms employed, creating and operating on a summary can be a better fit for parallel processors than the exact analysis. Overall, we can create summaries faster or cheaper to support larger data volumes, speed up the evaluation to provide answers faster, or maintain larger summaries within the same budget to boost accuracy. Additionally, choosing a favorable processor improves energy efficiency [122]. Previous work has shown that approximation paired with fitting use cases justify the overhead of parallel design and programming [32, 70, 72, 82, 150, 154].

In this thesis, we introduce novel techniques for approximate data processing that, on the one hand, use parallel processors to improve efficiency while, on the other hand, hiding complexity introduced by the parallel processor behind systems and abstractions.

1.2 Use Cases

As applications, algorithms, and parallel processors have to be matched individually, this thesis covers two relevant use cases of approximate data analysis in which parallel processors have a substantial impact:

Use Case 1: GPU-Accelerated Selectivity Estimation Selectivity estimates are at the heart of query optimization in relational databases [57, 135]. As the optimizer has to choose from numerous equivalent execution plans before executing a query, it requires estimates on the result size of operators to assess the execution cost of candidate plans. These estimates are crucial to assess the execution cost of candidate plans, as inaccurate estimates can result in poor plans being chosen and consecutively lead to catastrophic query execution times [98, 101]. However, database systems commonly compute these selectivity estimates based on per-column summaries (e.g., most-common values, histograms) while assuming statistical independence of columns. While this enables quick estimates and avoids the cost of maintaining and evaluating multi-dimensional statistics, the independence assumption rarely holds for real-world data [97].

Heimel et al. have shown that GPU-accelerated *kernel density estimation (KDE)* is a promising technique to provide accurate selectivity estimates without the independence assumption [70, 71]. The authors have shown that KDE models are cheap to construct via sampling and can be optimized by learning from query feedback. The GPU serves as a statistical coprocessor to evaluate, update, and train larger model sizes than would have been possible on a CPU within the same time budget. The statistical coprocessor is completely transparent for users of the database system as it operates as part of the query optimizer. By improving plan quality via improved estimates, GPUs can not only accelerate in-memory databases but also indirectly accelerate I/O-bound systems that can not directly benefit from GPU-accelerated query execution.

While the approach has shown admirable results in terms of accuracy, the technique is limited to range queries. In particular, the approach does not trivially extend to joins which are among the most common operators in relational database systems, while being notoriously hard to estimate [101].

Use Case 2: FPGA-Accelerated Sketching Sketches are a popular class of summaries that can be constructed in limited memory and within a single pass over the input data [38, 58]. These attributes make the summary particularly popular in streaming use cases where data arrives at high rates and only preserving a history of data summaries is feasible. Sketches have been successfully applied in various domains, including machine learning [99], approximate query processing [37, 83, 157], large-scale network analyses [133, 140, 141, 153, 154, 169], and computational biology [139].

These applications inherently require summarizing large amounts of data that is potentially even in motion. We refer to this process as *sketching*. FPGAs have been identified as a promising architecture for sketching at the high throughput demanded by the applications [29, 92, 133, 139, 140, 141, 153, 154, 154, 169]. Embarrassingly parallel computations, a homogeneous memory access pattern, and the ability to merge summaries of the same size are a good match for custom circuits implemented on an FPGA. Furthermore, the circuit-level implementations enabled by FPGAs can provide hard guarantees on processing rates and consume at provide at least an order of magnitude better performance per Watt than CPUs [81].

While FPGA-accelerated sketching promises high throughput and energy efficiency, designing and implementing such accelerators is complex, incurs non-trivial trade-offs, and requires an expert in hardware design.

Overall, the two use cases show that parallel processors can increase the efficiency of approximate data analysis and benefit its applications. However, we have also highlighted issues with the state of the art from which we derive the three research challenges addressed in this thesis.

1.3 Research Challenges

In this thesis, we propose novel techniques advancing the state of the art in GPU-accelerated kernel density estimation and FPGA-accelerated sketching. We tackle the following research challenges:

Research Challenge 1: GPU-Accelerated Join Selectivity Estimation While Heimel et al. have shown made the case for GPUs as statistical coprocessors for selections with conjunctive range predicates [70], generalizing the approach to queries with arbitrary equijoins subject to selections would drastically extend the applicability of the technique. As joins are among the most common operators in databases and errors are known to propagate exponentially in the number of joins [110], the join selectivity estimation problem is crucial [101]. A naive approach that evaluates estimators for each value in the join domain is infeasible as the join domain is potentially huge. In Chapter 3, we show generalization techniques for KDE without iterating the join attribute domain or simplifying assumptions commonly violated in real-world data. The approach allows error-driven parameter optimization based on query feedback while benefiting from GPU acceleration.

Research Challenge 2: Generating FPGA Accelerators for Sketching at Line Rate While FPGAs have shown impressive throughput and energy efficiency for sketching and other data processing tasks, developing an FPGA accelerator is more involved than software programming: Without the services provided by software programming languages, operating systems, and modern CPUs (e.g., memory management), designing, implementing, and tuning FPGA-based sketching accelerators is a tedious and time-consuming process that requires the skillset of an FPGA expert. At the same time, the variety of algorithms and use cases warrants an abstraction that is more accessible to the developers of algorithms and software systems. In Chapter 4, we propose *Scotch*, a system that generates and optimizes FPGA accelerators for a broad class of sketching algorithms automatically. We provide a convenient abstraction to specify sketching algorithms in terms of user-defined functions, while a code generator does the heavy lifting of implementing sketching functionality. An iterative algorithm tunes the sketch towards the underlying FPGA automatically. Thus, Scotch simplifies integrating and maintaining FPGA-based sketching accelerators in software systems. The accelerators generated follow an architecture allowing for processing data at the full rate of the interconnect, often referred to as *line rate*.

Research Challenge 3: Optimistic Data Parallelism for FPGA-Accelerated Sketching As FPGAs are clocked at a few hundred Megahertz, sketching implementations must exploit data parallelism for high throughput. Maintaining one replica of the sketch per input value is easy to implement and necessary to guarantee processing at line rate. Thus, virtually all previous work on high-throughput sketching follows this strategy [29, 89, 92]. We consider replication *pessimistic* as it provisions for the worst case and, thus, leaves resources underutilized.

While sharing resources among inputs promises improved resource utilization, it also requires non-trivial mechanisms for conflict resolution and potentially stalling the architecture. In Chapter 5, we propose a viable *optimistic* sketching architecture that shares resources while introducing techniques to avoid stalls due to resource conflicts. Furthermore, we provide the theoretical framework to generate optimistic accelerators for a broad class of sketching algorithms from user-defined functions. We also employ an approximate query processing application to highlight that larger sketches are feasible in an optimistic architecture, which translates to significantly improved accuracy.

Each research challenge has a corresponding publication given in the next section.

1.4 Contributions and Impact

During our research, we have made the following contributions.

Conference Papers. We have published the main contributions of this thesis in the top-tier journal *Proceedings of the VLDB Endowment (PVLDB)*:

- Martin Kiefer, Max Heimel, Sebastian Breß, and Volker Markl: *Estimating Join Selectivities using Bandwidth-Optimized Kernel Density Models*, in PVLDB, Volume 10, Issue 13, 2085-2096, 2017.
- Martin Kiefer, Ilias Poulakis, Sebastian Breß, and Volker Markl: *Scotch: Generating FPGA-Accelerators for Sketching at Line Rate*, in PVLDB, Volume 14, Issue 3, 281-293, 2021.

- Martin Kiefer, Ilias Poulakis, Eleni Tzirita Zacharatou, Volker Markl: *Optimistic Data Parallelism for FPGA-Accelerated Sketching*, in PVLDB, Volume 16, Issue 5, 1113-1125, 2023.

Open Source Contributions. We have released all projects that are part of this thesis under Mozilla Public License 2.0:

- `https://github.com/martinkiefer/join-kde`. This repository contains our OpenCL code generators for GPU-accelerated join size estimation using KDE [87]. Furthermore, it contains all datasets, baselines, and scripts to reproduce our evaluation.

- `https://github.com/martinkiefer/scotch`. This repository contains our implementation of the Scotch [89] system for FPGA-accelerated sketching, including RTL generator, automated tuning, and I/O templates. Furthermore, it contains all baselines and scripts to reproduce our evaluation.

- `https://github.com/martinkiefer/optimistic-sketching`. This repository contains the RTL generator with extensions required for optimistic FPGA-Accelerated sketching. Furthermore, the repository integrates the generated sketching unit with Xilinx Vitis to support a wide range of FPGA devices in a GPU-like accelerator. Furthermore, it contains all datasets, simulators, baselines, and scripts to reproduce our evaluation.

Additional Contributions. The work described in this thesis has inspired an additional publication on approximate data analysis in distributed stream processing engines that is not discussed in this thesis:

- Rudi Poepsel-Lemaitre, Martin Kiefer, Joscha von Hein and Jorge-Arnulfo Quiané-Ruiz, Volker Markl: *In the Land of Data Streams Where Synopses Are Missing, One Framework to Bring Them All*, in PVLDB, Volume 14, Issue 10, 1818-1831, 2021.

1.5 Thesis Outline

The remainder of this thesis is structured as follows:

Chapter 2 establishes the background on approximate data analysis and parallel processors, focusing on summaries and architectures targeted in our work.

Chapter 3 generalizes GPU-Accelerated kernel density models for selectivity estimation to select-project-join queries.

Chapter 4 introduces Scotch, a system to generate FPGA accelerators for sketching at line rate without requiring an expert in hardware design and FPGAs.

Chapter 5 introduces an optimistic architecture for FPGA-accelerated data parallel sketching that trades guaranteed processing rates for improved resource utilization.

Chapter 6 concludes our findings and suggests future research.

2 Background

As the rate at which humanity generates data and the volume of stored data increase exponentially [142], techniques promising gains in speed and efficiency become more crucial. In the following sections, we provide background on the orthogonal approaches combined in this thesis:

Section 2.1 introduces the algorithmic approach by providing an overview of approximate data analysis based on data summaries. The section details the two main summaries discussed in this thesis — kernel density estimation and sketches.

Section 2.2 introduces the hardware approach by providing an overview of specialized parallel processors. The section discusses GPUs and FPGAs in detail, as they are the target processors of the approaches introduced in this thesis.

2.1 Approximate Data Analysis with Data Summaries

Approximate data analysis reduces a given input data set to smaller summaries and approximates functions over the input data based on them [38]. These estimates are usually subject to errors due to the lossy nature of summarization. *Sampling* is a popular and intuitive example of a summarization technique: Drawing and evaluating a random subset allows estimating quantities that would be infeasible to obtain on the entire population. However, estimating it via a random subset is sufficient. This approach is well-known from market research and election forecasts.

In this thesis, we cover two applications of approximate data analysis in particular: *Approximate query processing* computes approximate answers to relational aggregate queries aiming for significantly lower response times [1, 3, 100]. It usually implies a bounded estimation error. *Selectivity estimation* is a subproblem of query optimization in relational databases [57, 120, 135]. The query optimizer requires estimates on the result sizes of subsequent relational operators to assess the quality of logically equivalent execution plans. In this sense, it can be seen as a subproblem of approximate query processing. However, due to the time-critical nature of query optimization, methods usually focus on providing fast and accurate answers in practice rather than theoretical accuracy guarantees [21, 63, 70, 83, 135, 157].

An overview and common classification of summaries is given in [38]:

Samples are a representative subset of an overall population. The most common technique is a *simple random sample* in which elements are drawn independently with uniform probability. However, biased sampling techniques exist that sacrifice uniformity and independence to boost accuracy for some applications [48, 56, 157].

Histograms contain summary statistics for groups, so-called buckets, of the original population. Equi-width histograms are a popular representative: Instead of tracking item frequencies individually, they store the number of items falling into equally wide buckets spanning the data domain. Similarly, equi-depth histograms split the domain into differently-sized buckets, ideally capturing the same number of observations. Histograms are a well-known technique from data visualization [134]. More complex histograms exist that vary in terms bucketization rules [21, 63, 84] — especially in the multivariate case [21, 63].

Sketches are a recent technique that projects data onto a smaller domain by applying random transformations on the input data via families of hash functions and random seeds. Aggregations on top of such random values allow for estimating various quantities ranging from the number of distinct values in a stream [92], value frequencies [39], and complex relational queries [43, 83, 157]. As constructing sketches has a sublinear time and storage complexity in the number of summarized elements, the technique is popular in data stream processing [58].

Wavelets are a technique commonly used in signal and image processing. For example, the JPEG-200 image compression standard uses wavelets for compression [138]. Wavelet-based data summaries decompose the joint frequency distribution of the input data set into a weighted sum of so-called wavelet functions. Contributions identified as negligible according to their weight are dropped to reduce size. Wavelet summaries support estimating the result of relational aggregate queries [25, 106].

While this classification is not exhaustive because it misses techniques such as *quantile summaries* [58, 109], *kernel density estimators* [119, 126, 134], or *probabilistic graphical networks* [59, 155, 156], it suffices to set the stage for the summaries used in the remainder of this thesis.

Each data summary has individual properties that determine its applicability to a given problem. In particular, the following aspects are crucial:

Summarization Process: Creating a particular summary comes with an individual space and time complexity. In particular, summarization algorithms classify into *multi-pass* and *single-pass* algorithms based on the number of passes over the input data required. Single-pass summaries are particularly useful in streaming scenarios and for storage mediums with a high random access latency, such as hard disk drives or tape.

Supported Quantities and Computations: Data summaries can provide estimates for different quantities ranging from simple statistics, such as the frequency of values, to aggregates over complex relational queries. The algorithms to compute these quantities differ in their space and time complexity.

Accuracy: The accuracy of estimates derived from data summaries may vary depending on data characteristics such as order or distribution. Commonly, data summaries provide hard or probabilistic bounds on the error for estimated quantities.

All these aspects are affected by the size of the summary, e.g., the number of buckets in a histogram or the size of sample. The summary size controls the trade-off between accuracy and the effort to construct or evaluate the summary. As data summaries differ in their proprieties and how they are affected by their size, selecting an appropriate summary for a given application is a non-trivial task.

In the following, we will focus on the two classes of data summaries this thesis builds upon. First, in Section 2.1.1, we will discuss kernel density estimation, which can be considered a hybrid between histograms and sampling. Chapter 3 shows how this technique enables feedback-optimized join selectivity estimation on GPUs. Then, in Section 2.1.2, we discuss sketches. With the AGMS sketch, a sketch summary serves as a baseline in Chapter 3. Chapters 4 and 5 discuss techniques to construct sketch summaries efficiently on FPGAs with approximate query processing as an example application.

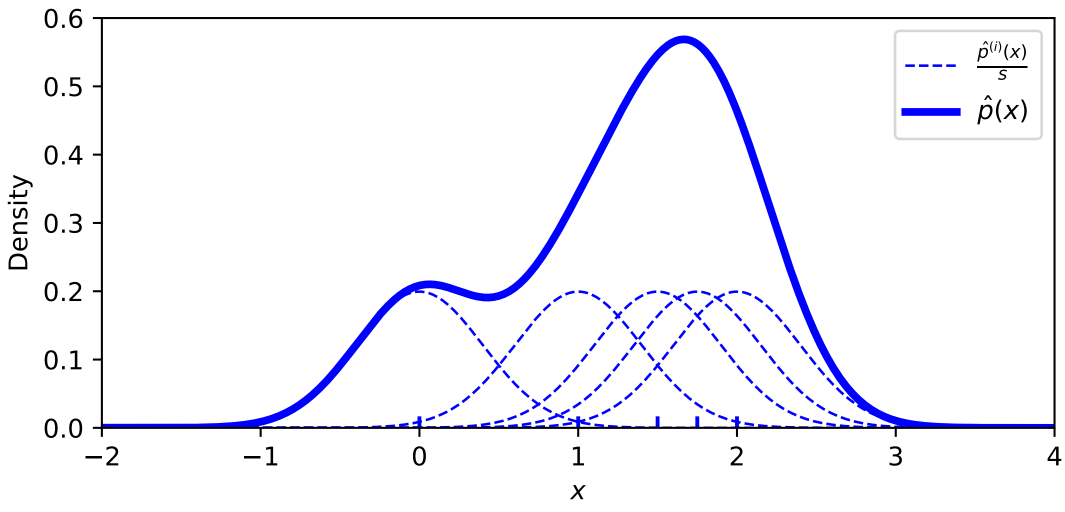


Figure 2.1: Kernel density estimator (line) for a sample (markers on the x-axis. The estimate is the sum of the individual contributions of kernels centered on each sample point (dashed).

2.1.1 Kernel Density Estimation

Kernel density estimation (KDE) is a data-driven, non-parametric method to estimate probability density functions [119, 126, 134]. We illustrate the technique for the univariate case in Figure 2.1: Based on a simple random sample drawn from a potentially unknown distribution, the estimate is the average of independent probability density functions, so-called *kernels*, centered at each sample point. The *bandwidth* parameter denotes the spread of the individual kernels and *smoothes* the distribution in the sample: A small bandwidth concentrates probability mass in the vicinity of sample points, while an increasing bandwidth spreads probability mass on larger regions around sample points.

Formally, based on a simple random sample $\mathcal{S} = \left\{\vec{t}^{(1)}, \ldots, \vec{t}^{(s)}\right\}$ of size s from a d-dimensional distribution, multivariate KDE defines an estimator $\hat{p}\left(\vec{x}\right) : \mathbb{R}^d \rightarrow \mathbb{R}$ that assigns a probability density to each point $\vec{x} \in \mathbb{R}^d$:

$$\begin{aligned}\hat{p}\left(\vec{x}\right) &= \frac{1}{s}\sum_{i=1}^{s} \hat{p}^{(i)}(\vec{x}) \\ &= \frac{1}{s \cdot |H|}\sum_{i=1}^{s} K\left(H^{-1}\left[\vec{t}^{(i)} - \vec{x}\right]\right) \end{aligned} \tag{2.1}$$

The density $\hat{p}^{(i)}$ denotes the individual contribution of each sample point to the overall density. It consists of the kernel function K, which can be any symmetric multivariate probability density, and the bandwidth H, which is a positive semi-definite symmetric matrix. A common simplification restricts multivariate kernels to the product of univariate kernels for each dimension.

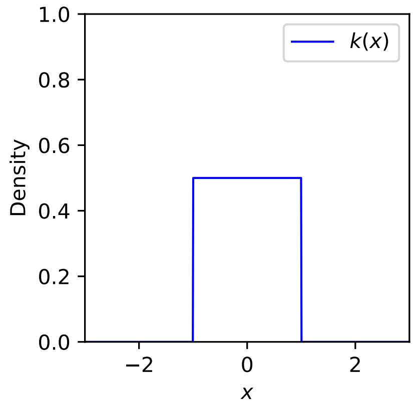
$$\begin{aligned}\hat{p}\left(\vec{x}\right) &= \frac{1}{s}\sum_{i=1}^{s}\hat{p}^{(i)}(\vec{x}) \\ &= \frac{1}{s\cdot\prod_{j=1}^{d}h_j}\sum_{i=1}^{s}\prod_{j=1}^{d}k\left(\frac{\left[\vec{t}^{(i)}-\vec{x}\right]}{h_j}\right)\end{aligned} \tag{2.2}$$

In this case, one bandwidth value h_j per dimension is sufficient as product kernels imply independence between dimensions. While product kernels impair estimation quality, they drastically simplify computations and allow for closed-form derivatives and integrals not available otherwise [70, 71, 134]. Note that independence in kernels does not imply independence in the overall estimator.

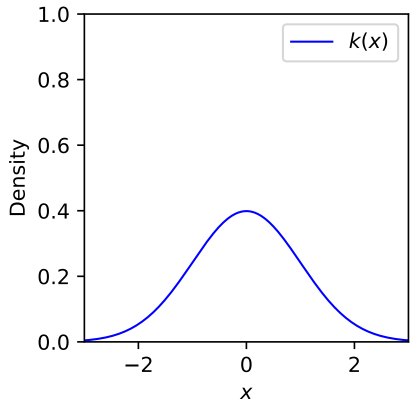
The Epanechnikov, Gaussian, Uniform, and Biweight are popular kernels shown in Figure 2.2. While the Epanechnikov kernel, a truncated second-order polynomial, is optimal in terms of the mean squared integrated error (MISE) [47], the loss of efficiency when choosing one of the other shown kernels is small [134]. Thus, kernel functions are commonly chosen based on desirable properties, e.g., being continuously differentiable or cheap to evaluate.

However, choosing the bandwidth is considered the most critical factor in minimizing the estimation error with KDE [134]. Figure 2.3 illustrates this for the univariate case: While the KDE model with $h = 0.037$ fits the target standard normal distribution well, a poor choice of the bandwidth leads to a suboptimal estimate. An overly large bandwidth ($h = 2$) results in an overly smooth distribution that is less reflective of the distribution in the sample. An overly small bandwidth (h=0.05) results in a spiky density estimate that overfits the distribution in the sample. The bandwidth selection problem is challenging. *Scott's rule* [134] is a popular rule-of-thumb solution to the bandwidth selection problem and assumes a normal distribution:

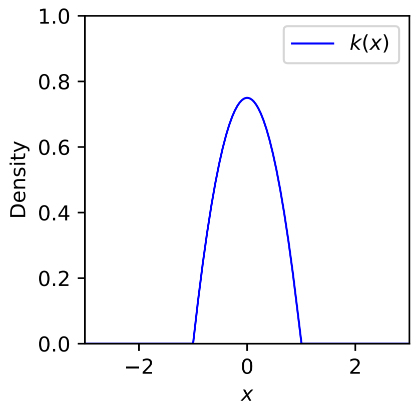
$$\hat{h}_j^{scott} = s^{-\frac{1}{d+4}}\cdot\hat{\sigma}_j \tag{2.3}$$



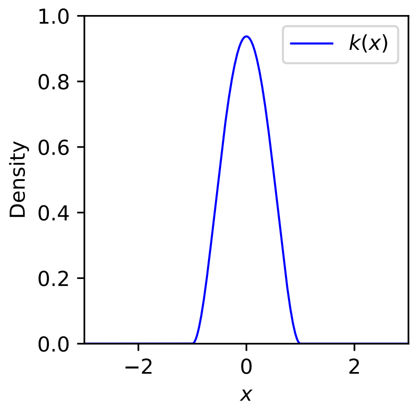
(a) Rectangular, $k(x) = \frac{1}{2} \cdot \mathbb{1}_{|x|\leq 1}$



(b) Gaussian, $k(x) = \frac{1}{\sqrt{2\pi}} e^{-\frac{1}{2}x^2} = \mathcal{N}_{0,h^2}(x)$



(c) Epanechnikov, $k(x) = \frac{3}{4} \cdot (1-x^2) \cdot \mathbb{1}_{|x|\leq 1}$



(d) Biweight, $k(x) = \frac{15}{16} \cdot (1-x^2)^2 \cdot \mathbb{1}_{|x|\leq 1}$

Figure 2.2: Popular kernel choices. $\mathbb{1}$ denotes the indicator function.

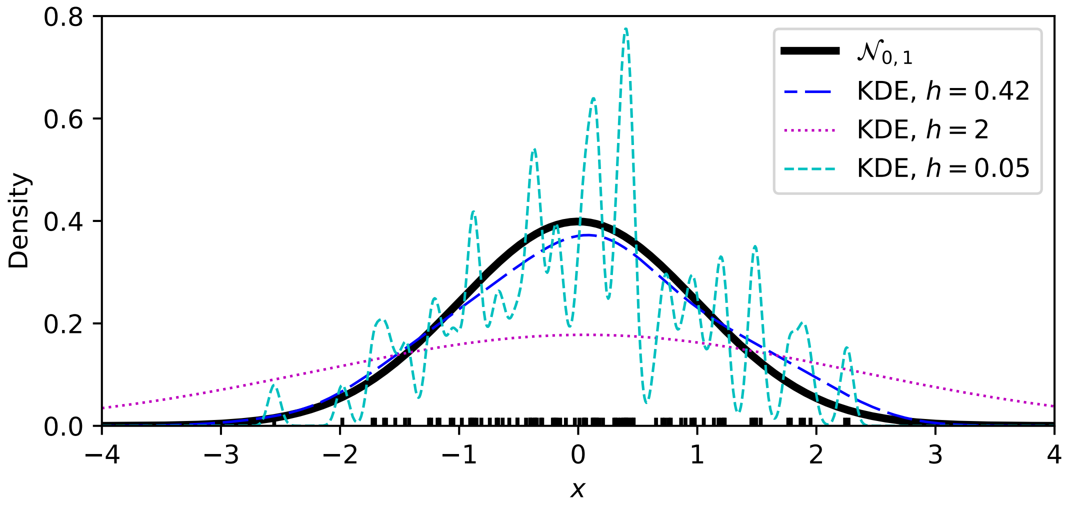


Figure 2.3: Kernel density estimates for a sample (markers on the x-axis) drawn from a standard normal distribution. We show KDE estimators with a well-fitting ($h = 0.042$, Scotts rule), underfitting ($h = 2$), and overfitting bandwidth ($h = 0.05$).

The variable $\hat{\sigma}$ denotes the sample standard deviation. Scott's rule contains three intuitions for the bandwidth parameter: A large sample is more representative of the generating distribution and, thus, requires less smoothing and, consequently, a smaller bandwidth. Increasing dimensionality and variance in the data increases the need for smoothing. As Scott's rule is easy to compute, it can also introduce large errors for non-normal distributions [134]. While more sophisticated methods provide significantly better estimates, they are also more computationally expensive. Popular bandwidth selection mechanisms employ cross-validation [15, 44, 131] or plug-in methods [137, 149, 160].

KDE as a Data Summary

KDE is not limited to approximating unknown probability densities. Using a sample from a dataset turns the technique into a data summary approximating the underlying distribution [63, 70, 71, 73]. By integrating the density estimate on a region, we can estimate the share of data points in the region. A hyper-rectangular region corresponds to a range query and translates to integrating each dimension of a product kernel individually. For a relation $R = (A_1, \ldots, A_d)$, a range query $Q = \sigma_c(R)$ has a selection predicate $c = l_1 \leq A_1 \leq u_1 \wedge \ldots \wedge l_d \leq A_d \leq u_d$. Given a sample S of size s drawn from R, we compute the estimate as follows:

$$\begin{aligned}\frac{|Q|}{|R|} &\approx \hat{p}(c)\\ &= \frac{1}{s}\sum_{i=1}^{s}\hat{p}^{(i)}(c)\\ &= \frac{1}{s}\sum_{i=1}^{s}\prod_{j=1}^{d}\quad\underbrace{\int_{l_j}^{u_j}\frac{1}{h_j}k\left(\frac{\left[\vec{t}^{(i)}-\vec{x}\right]}{h_j}\right)dx_j}_{\text{Gaussian: }\frac{1}{2}\left(erf\left(\frac{\vec{t}^{(i)}-u}{h\sqrt{2}}\right)-erf\left(\frac{\vec{t}^{(i)}-l}{h\sqrt{2}}\right)\right)}\end{aligned} \tag{2.4}$$

Thus, given that the integral of the univariate kernel k computes efficiently, a KDE model can provide estimates for range queries efficiently. For the Gaussian kernel used in our work, the integral requires the error function that erf ships with virtually all standard math libraries.

While evaluating the query on the underlying sample also yields an estimate of the share of qualifying values, the additional smoothing applied by the kernel density estimator helps to model data points not included in the sample. This smoothing is similar to buckets in histograms summarizing a region of data. Optimizing this bandwidth allows us to find an optimal balance between the data present in the sample and assuming values in the vicinity of the sample points.

If used as a data summary, additional techniques exist beyond optimizing the MISE solely based on the sample. In contrast to assuming an unknown density, the data the sample originates from may be available for querying. Especially in the context of selectivity estimation in relational databases, correct selectivities are available as a side-product of query execution. Heimel et al. have shown that closed-form derivatives exist for range queries and a Gaussian product kernel, enabling optimizing for the estimation error directly [70]. When optimized over a set of representative queries or query feedback collected in the execution engine of a database $Q_1, \ldots, Q_n$, the bandwidth selection problem translates to the following optimization problem [70, 71]:

$$\arg\min_{\vec{h}} \sum_{i=1}^{n} \mathcal{L}\left(\frac{|Q_i|}{|R|}, est(Q_i, \vec{h})\right)$$

The function *est* denotes the estimate provided by a KDE model for a given query with the given bandwidth; $\mathcal{L}$ denotes a loss function. This approach outperformed Scott's rule and cross-validation for the squared estimation error as $\mathcal{L}$ [70]. Furthermore, techniques for error-based sample maintenance exist [70, 88].

Constructing a kernel density estimator requires drawing a simple random sample of the input data and selecting the bandwidth. Drawing random samples is a well-understood problem [116]: At worst, reservoir sampling yields a random sample in a single pass over the input data [158]. The algorithm maintains a random sample incrementally by replacing sample elements with new observations in a randomized process. More efficient sampling schemes are available in relational databases as the number of tuples or blocks is known, and random access based on randomly drawn identifiers is possible [116]. Ideally, each tuple in the sample requires only one access to the base relation. Furthermore, algorithms for updating random samples in the presence of insertions, deletions, and updates to the original dataset exist [116, 117].

Computing the bandwidth using Scott's rule is cheap as it requires a single scan over the sample. If error-driven bandwidth optimization is applied based on gradient-based optimization [70], the cost depends on the optimization algorithm: Online optimization with stochastic gradient descent [14] requires only one evaluation of the estimator per training query to compute the gradient, each being in $O(s \cdot d)$. Batch optimization algorithms, e.g., MMA [145] or L-BFGS [113], require many evaluations and iterations over the set of training queries but provide higher accuracy [70].

When used as a data summary without further assumptions on the input distribution, kernel density estimation lacks error bounds. However, the underlying independent sample is a data summary with probabilistic error bounds from the Chebyshev, Hoeffding, and Chernoff bounds [38]. While these bounds can not be extended directly to KDE due to the additional smoothing applied, a bandwidth close to zero is equivalent to sample evaluation. Thus, assuming that the bandwidth was successfully optimized with an according loss function, KDE provides at least as good results as the uniform sample it is based on.

In Chapter 3, we propose new techniques to extend kernel density estimation to the more general class of relational queries that contain conjunctive equality and range predicates together with equijoins.

2.1.2 Sketches

The term *sketch* commonly denotes summaries for streaming data that exploit randomization in the construction process by employing hash functions with desirable statistical properties [38]. They have been successfully applied to various data-intensive tasks such as selectivity estimation [157], heavy hitter and change detection [154], data integration [170], and machine learning [99].

Unfortunately, the term appears in various other connotations and lacks a generally agreed definition. However, the term *linear sketch* has an agreed definition [38] that we will discuss here instead. While we establish and rely on a more general definition in Chapters 4 and 5, linear sketches help showcase the desirable properties of sketching algorithms we exploit in this work. Furthermore, we introduce AGMS, Count-Min, and Fast-AGMS as representatives of linear sketches that appear as baselines and target algorithms in the following chapters.

A linear sketch [38] is a linear transformation of the input data. Formally, given a vector $\vec{v} \in \mathbb{R}^k$ that contains the frequency or each of the n elements in the input domain, a linear function $M : \mathbb{R}^k \to \mathbb{R}^m$ with $k >> n$ describes the sketching procedure. This is equivalent to matrix multiplication with a matrix $M \in \mathbb{R}^{m \times k}$. Accordingly, a linear sketch $s \in \mathbb{R}^n$ is constructed as:

$$\vec{s} = M\vec{v} \tag{2.5}$$

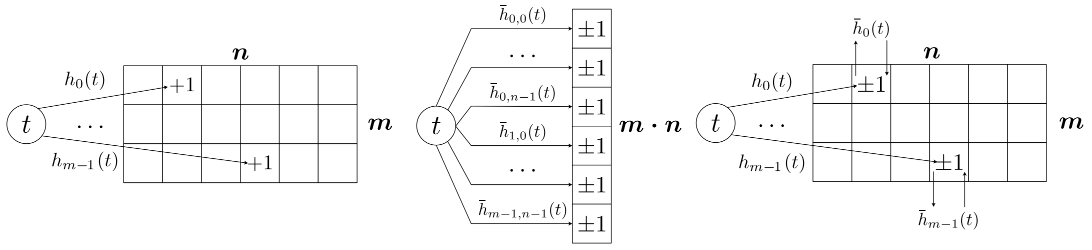
The sketching algorithm determines the construction of M. Defining sketching as a linear transformation helps us to highlight essential properties that also apply to the more general class:

Mergeability [2]: Two sketch summaries $\vec{s_1}$ and $\vec{s_2}$ of the same size constructed over distinct frequency vectors v_1 and v_2, can be combined into a single ketch summary of the same size by addition as $M(\vec{v_1} + \vec{v_2}) = M\vec{v_1} + M\vec{v_2}$.

Single-Pass Summary Construction: Given that each appearance of an item in the input data stream e has a corresponding frequency vector v_e, we can construct a sketch incrementally by applying M and merging, i.e., $\vec{s} := \vec{s} + M\vec{v_e}$. Thus, a single pass over the input data suffices to construct a sketch summary.

Out-of-order processing: Following the previous argument, the order of updates is irrelevant, as addition is commutative and associative.

Constant memory consumption: Given that the size of sketch k and the vector entries are fixed, memory consumption is constant in the number of input elements. [1]



(a) Count-Min (b) AGMS (c) Fast-AGMS

Figure 2.4: Three popular sketches referenced throughout this thesis.

Constant update time: As the only structure modified over time is the sketch itself and, assuming entries have a fixed size, update times are independent of the number of previously observed elements. [1]

Combining single-pass summary construction and constant memory consumption makes sketches particularly useful for stream processing. In addition, mergeability and out-of-order processing allow for parallelization and creating a logical summary of the union of multiple separate data streams.

However, the definition of a linear sketch in Equation 2.5 is not practical for implementation, given that v contains the full frequency distribution of the input data and makes a data summary obsolete. Fortunately, to the best of our knowledge, the matrix M is always sparse in practice and contains only one non-zero entry per column that is determined at random. This property allows us to make updates to entries of the sketch for each appearance of a new value individually by incrementing an entry selected by a random hash function $h : \{0, k-1\} \rightarrow \{0, m-1\}$ with desirable statistical guarantees.

We will introduce the Count-Min, AGMS, and Fast-AGMS sketches with their construction, estimated quantities, and accuracy guarantees in the following. Sketches are zero-initialized in all cases. Figure 2.4 visualizes their construction.

Count-Min (CM): The Count-Min sketch [39] allows tracking upper bounds on value frequencies for a stream of input items. Given a sketch s and hash-function h, the update for an item t is $s[h(t)] := s[h(t)] + 1$. Intuitively, we are tracking the item frequencies of hash values instead of the full domain of the data stream. The space required to track frequencies reduces from k to n counters while accepting collisions in entries. Thus, the vector entry $s[h(t)]$ may not contain the actual frequency of t in the input data but

[1] Space and time complexity usually remain sublinear in the number of observed items when the sketch size is viewed as a function of the accuracy.

potentially an upper bound due to hash collisions. To mitigate the effect of collisions, m instances of the sketch are maintained with different hash functions h_i resulting in a sketch matrix $S \in \mathbb{R}^{m \times n}$ with the update processing being with $s[i, h(t)] := s[i, h_i(t)] + 1$ for every $i \in \{0 \ldots m-1\}$. As $s[i, h(t)]$ is an upper bound for every i, we can take the tightest upper bound over all sketch instances. Thus, the estimate $\hat{f}(t)$ of the frequency $f(f)$ is given as $f(t) \leq \hat{f(t)} = min_{i \in \{0 \ldots m-1\}} s[i, h(t)]$. In addition to this hard guarantee, probabilistic guarantees exist: Given hash functions h_i from a family of pairwise independent hash functions, Euler's numbers e, and the total number of observations N, choosing $m = ln(\frac{1}{\delta})$ and $n = \frac{e}{\varepsilon}$ yields $\hat{f(t)} \leq f(t) + \varepsilon N$ with probability at least $1 - \delta$ [39].

The CM sketch also supports computing estimates on the self-join size or the join size by multiplying matrix entries either with itself or with a second CM sketch with the same size and hash functions [39]. With auxiliary structures, CM sketches provide estimates for ranges of items without estimating the frequency of every individual value [39].

AGMS: The AGMS sketch [4, 5] allows for estimating self-join and join sizes. The number of entries in the sketch vector is $n = 1$. Given independent random variable $X_t \in \{+1, -1\}$ with equal probability of taking either option, an update to the sketch s for an input t applies $s := s + X_t$. Given input datasets A and B with corresponding sketches $s(A)$ and $s(B)$, the product $s(A) \cdot s(B)$ yields an unbiased estimator $\hat{J}$ of the join size J:

$$\begin{aligned}
E[\hat{J}] &= E[s(A) \cdot s(B)] \\
&= E\left[\sum_{i \in A} X_i \cdot \sum_{j \in B} X_j\right] \\
&= E\left[\sum_{(i,j) \in A \times B} X_i \cdot X_j\right] \\
&= \sum_{\substack{(i,j) \in A \times B, \\ i=j}} \underbrace{E\left[X_i \cdot X_j\right]}_{=1} + \sum_{\substack{(i,j) \in A \times B, \\ i \neq j}} \underbrace{E\left[X_i \cdot X_j\right]}_{\text{by independence } =0} \\
&= \left|\left\{(i,j) \mid (i,j) \in A \times B,\ i = j\right\}\right| \\
&= |A \bowtie B| = J
\end{aligned} \tag{2.6}$$

While pairwise independent random variables are sufficient to prove the estimator is unbiased here, 4-wise random variables minimize estimator variance [127]. Materializing random variables as a sequence of k random bits is an option. However, as k is potentially large, it is more convenient to use a member of a family of hash functions

$\bar{h} : \{0 \ldots k-1\} \rightarrow \{-1, +1\}$. In this case, we have $s := s + \bar{h}(t)$ for every input value t.

As a single AGMS sketch has high variance, it is essential to evaluate multiple instances of the sketch. The original estimator assumes a matrix of $m \cdot n$ counters, each being an independent instance of the sketch constructed over the entire data with a different hash function $h_{i,j}$ [4, 5]. After multiplying counters, the combined estimate $\hat{J}$ is computed by first taking the average in each of the n rows and then taking the median of these averages. Picking $m = 4\ ln\left(\frac{1}{\delta}\right)$ and $n = \frac{16}{\epsilon^2}$ yields an estimate that guarantees $|J - \hat{J}| \leq \epsilon \sqrt{|A \bowtie A||B \bowtie B|}$ with probability at least $1 - \delta$ [58].

By carefully constructing sketches as the product of multiple random variables, the sketch extends to joins over multiple relations. Moreover, as filters are equivalent to joining with the set of values qualifying the selection, the AGMS sketch estimates the result size of complex queries [43, 157]. However, estimator variance increases exponentially with each additional join [157]. As each update affects all $m \cdot n$ sketches, AGMS is expensive to construct.

Fast-AGMS (FAGMS): The Fast-AGMS sketch [37] reduces the number of updates required compared to plain AGMS. Like the CM sketch, the sketch is a matrix $s \in \mathbb{R}^{m \times n}$, and updates are scattered across each row by i by a hash function h_i. However, entries are updated with a hash function $\bar{h}_i$ according to the AGMS update procedure. Thus, compared to AGMS, the number of updates is reduced by a factor of n. The estimated join size from two sketches $sk(A)$ and $sk(B)$ requires performing an element-wise multiplication, adding the result for each row, and computing the median. The sketch essentially has the same space-accuracy tradeoff as plain AGMS [37].

While this sketch is faster to evaluate and requires fewer hash functions for random variable generation, it does not generalize to complex queries, as scattering with a hash function h requires a common attribute. Izenov et al. proposed new techniques to close this gap in selectivity estimation applications [83].

The independent hash functions h and $\bar{h}$ are created by initializing generator schemes for hash function families with random seeds. Throughout this thesis, we use the $H3$ scheme to implement h as it (1) generates the required pairwise-independent hash functions and (2) is cheap to evaluate and implement in hardware and software [123]. For the $+1/-1$ hash function $\hat{h}$, we use EH3 [50], which is a hashing scheme that efficiently generates 3-wise independent random variables while in practice providing at least as good results as 4-wise independent scheme [128]. We provide the definitions for these schemes as examples in Section 4.3.

2.2 Parallel Processors

Dennard scaling [42] predicted the speedups delivered by every new generation of CPUs for three decades. Essentially, Dennard observed that, with every new generation of transistors, (1) area and power consumption for transistors halves while (2) the operating frequency increases by 40%. These improvements allowed CPU vendors to fit twice as many transistors on a CPU within a fixed area and power budget, while increased operating frequencies ensured faster processing of instructions. Thus, every new generation of CPUs promised substantial speed-ups without requiring fundamental changes to existing programs.

However, Dennard scaling ended as the ever-shrinking size of transistors did not allow clock frequencies to scale exponentially due to physical limits and the resulting increased power consumption and heat dissipation [12] — CPU manufacturers hit the *power wall* around 2005 [13]. CPU vendors embraced parallelism as a new driver for performance and invested transistors in multiple cores, simultaneous multithreading, and vector processing [74]. However, exploiting this parallelism requires adapting algorithms and programs to concurrency and understanding the underlying processing architecture. C++ expert Herb Sutter summarized the new reality in a famous article in Dr. Dobb's Journal titled 'The free lunch is over.' [144].

The end of Dennard's scaling did not only cause CPU architectures to embrace parallelism but also opened the door for specialized parallel processors with different performance characteristics and programming models. Staying with Sutter's metaphor, the 'free lunch' was over for software developers — and hardware manufacturers began opening distinguished restaurants for the evergrowing performance hunger of paying customers. In particular, the following specialized architectures established themselves as alternatives to the well-known multi-core CPUs:

Graphics Processing Units (GPUs) are the most common specialized parallel processor as they are included in virtually all desktop computers, laptops, and smartphones. While formerly restricted to accelerating computer graphics tasks and communicating with an output display, GPU vendors established the architecture as a processor for general-purpose computations. Like a CPU, GPUs process instructions, but provide an order of magnitude more parallel processors at the expense of cache sizes and single-thread performance [8, 114].

Application-Specific Integrated Circuits (ASICs) are a more general approach to parallel processing that employs custom integrated circuits for a given task [81, 151].

ASICs allow experts to solve specific problems more efficiently by allocating transistors solely based on the task's requirements. Parallelism on a circuit level allows (1) selecting the degree of parallelism freely, (2) arbitrarily mixing data, pipeline, and task parallelism, and (3) synchronization with little to no overhead. However, creating ASICs requires long development cycles, substantial financial effort, and production in large quantities to be economical [96]. Examples of ASICs include Oracles SPARC-M8 CPU that included special-purpose data analytics accelerator units [40], Google's Tensor Processing Units for machine learning workloads [33], and ASICs for cryptocurrency mining [10].

Field-Programmable Gate Arrays (FPGAs) bridge the gap between ASICs and software [81, 151]. FPGAs provide low-level logic, memory, and other resources on a chip that are initialized and interconnected via software. FPGAs offer the benefits of custom circuitry for a problem but avoid the high initial cost of ASIC production [81, 96]. Furthermore, FPGAs are reprogrammable, while ASICs are static once manufactured. While FPGAs were primarily used for ASIC prototyping and enabling firmware updates in the past [81], FPGA vendors nowadays provide devices that specifically target the market of accelerated computing [77, 165].

The following sections elaborate on the architectures of the specialized parallel processors essential to the next chapters. We provide an overview of their architectures, introduce their programming models, and characterize fitting workloads. In Section 2.2.1, we elaborate on GPUs. In Chapter 3, GPUs accelerate KDE-based join selectivity estimation for relational databases. Furthermore, Chapters 4 and 5 compare against a GPU as a baseline. Section 2.2.2 elaborates on FPGAs. Chapter 4 shows how to implement and tune sketch summaries without expert involvement. Chapter 5 discusses an optimistic architecture for FPGA-accelerated sketching that improves resource utilization.

2.2.1 Graphics Processing Unit (GPU)

Desktop computers traditionally used graphics processing units for graphics computations and video output. In the mid-2000s, they were identified as a target for homogeneous operations on floating-point vectors and matrices [114]. Finally, the architecture opened to general-purpose computations [114]. In the following, we will provide an overview of the architecture, highlight the programming model and commonly used primitives, and characterize workloads suiting the architecture.

GPU vendors and frameworks use different terminology for equivalent concepts. As all experiments in this thesis are conducted on Nvidia GPUs, we use the terminology

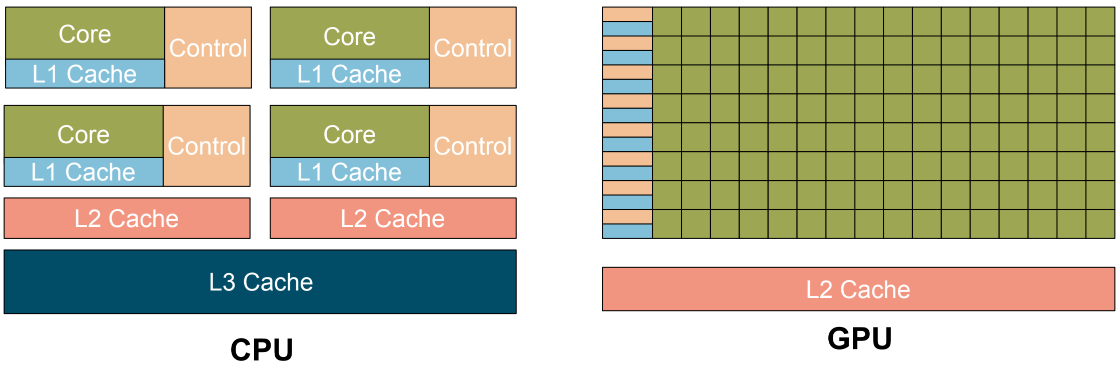


Figure 2.5: High-level architecture of CPUs and GPUs visualizing the provisioning of transistors to compute resources (cores), control flow handling (control), and cache levels. Figure adapted from [114].

from Nvidia's *Compute Unit Device Architecture (CUDA)* [114] platform and programming model throughout this thesis.

Architecture Overview

GPUs follow an architecture that favors the throughput of parallel computations over single-thread performance. Figure 2.5 visualizes the difference in the architectures of multicore CPUs and GPUs. GPUs invest a substantial amount of chip space into *cores* [2] that execute instructions in parallel [114]. However, significantly fewer resources are invested in caches and logic to handle and anticipate control flow. With their dedicated control logic, CPU cores can execute different programs and commonly include optimizations for single-thread performance, such as out-of-order processing of instructions or branch prediction [74]. In contrast, multiple GPU cores share an instance of control logic, form a *streaming multiprocessor*, and are designed to execute the same program (Single Instruction, Multiple Threads / SIMT). Thus, the architecture favors data parallelism, which applies the same operations to different data items.

GPUs serve as coprocessors in host systems operated by CPUs. *Dedicated* GPUs are physically separate chips located on expansion cards with periphery (cooling, I/O, external memory) [8]. Depending on whether the traditional purpose of graphics output is still served or not, we refer to them as graphics cards or accelerators, respectively. Such accelerators connect to the host system via an interconnect; usually, Peripheral Component Interconnect Express (PCIe) [8, 114]. We show an example setup for a

[2]We are using the term *core* in the CUDA connotation. It corresponds to the term floating-point-units and arithmetic-logic-units in CPU terminology.

high-end system with a GPU accelerator in Figure 2.6 and visualize the theoretical peak bandwidth for the interconnect and memory. We use this system to highlight critical properties of the architecture.

Most importantly, we see that the memory bandwidth of the High Bandwidth Memory (HBM) used with the graphics card is 3.8 times higher than the DDR4 [3] memory available to the host. A memory bandwidth this high is required to avoid starvation of the vast parallel compute resources due to data transfer from device memory. At the same time, this memory is also more expensive, explaining the two orders of magnitude smaller memory on the host accelerator. Programs only achieve the maximum device memory bandwidth if memory accesses from multiple threads to contiguous memory locations combine into a single large transaction. This technique is called *memory coalescing* and emphasizes the architecture's specialization on data parallelism. Besides the device memory, each streaming multiprocessor provides tens to hundreds of Kilobytes of dedicated on-chip memory that provides even higher bandwidth. This memory is traditionally explicitly available to the program but may also partially or fully serve as a cache in recent GPU architectures [114].

Another important observation is the rather small bandwidth of the PCIe interconnect compared to the CPU (6.5x) and the GPU memory bandwidth (50x). This scarcity leads to the *data transfer bottleneck* in which resources are underutilized as they wait for data to arrive via the interconnect. The data transfer bottleneck restricts the use cases of GPUs as it is futile to accelerate tasks on a GPU that can be processed faster than interconnect bandwidth on a CPU. While faster interconnects such as NVlink [64] have been specified and are already available, systems prone to the data transfer bottleneck remain state-of-the-art until this day [102].

For completeness, we mention *integrated* GPUs located on the same chip as a CPU and operating on the same main memory [8]. Main memory bandwidth replaces PCIe as a potential bottleneck. Integrated GPUs provide fewer compute resources due to the smaller memory bandwidth and scarce chip space shared with the CPU.

While CPUs use large caches, out-of-order execution, and speculative execution to hide the latency of instructions and DRAM access of individual threads, GPUs approach this primarily by switching between threads [159]: Each streaming multiprocessor simultaneously executes *warps* consisting of 32 threads in lock-step on its streaming multiprocessors. As these streaming multiprocessors provide sufficient resources, e.g., registers, and shared memory, to maintain the state of multiple warps, execution switches to a ready warp if the current warp waits for computations or memory access. While this

[3]Double Data Rate 4 Synchronous Dynamic Random-Access Memory

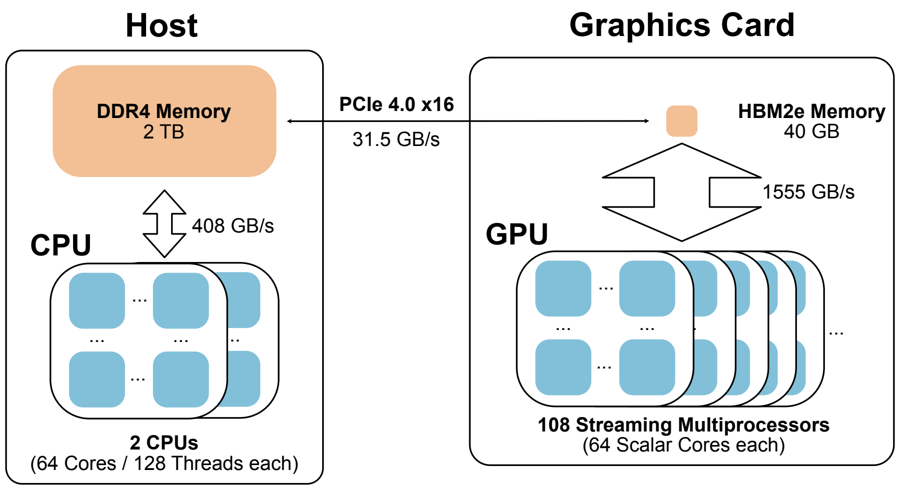


Figure 2.6: Data transfer bandwidths for server system with an AMD EPYC 7742 CPU and an Nvidia A100 GPU accelerator.

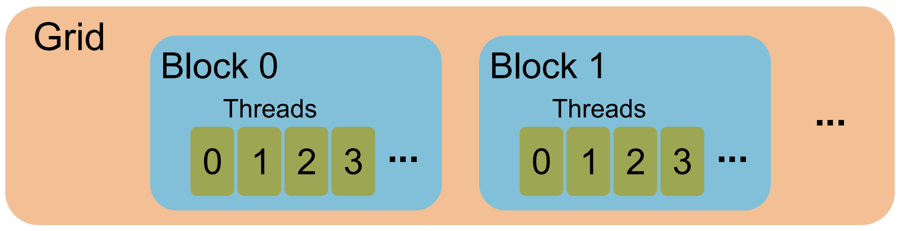


Figure 2.7: CUDA programming model. A kernel is executed on a grid consisting of multiple blocks that, in turn, consist of multiple threads.

does not decrease the latency of individual threads, it increases the overall bandwidth of performed computations. It also highlights the parallel nature of GPUs: Algorithms must exploit the massive parallelism provided by the architecture to exploit it fully.

Execution Model

GPUs are programmed using frameworks such as CUDA [114], OpenCL [61], or OneAPI [79]. These frameworks provide interfaces to transfer data and launch GPU programs, so-called *kernels*, from the host computer. Furthermore, GPU frameworks provide programming languages and compilers for kernels.

The size of a problem is given as a grid consisting of several blocks that, in turn, consist of several threads. While all threads execute the same kernel, a thread can access its position on the grid and, thus, have an individual state. Identifiers for blocks and threads have up to three dimensions. Figure 2.7 shows a one-dimensional grid.

```
// Kernel executed on the GPU
__global__ void vector_add(double *a, double *b, double *c)
{
    // Compute global id
    int id = blockIdx.x * blockDim.x + threadIdx.x;
    c[id] = a[id] + b[id];
}
... // Boilerplate code omitted

// Host function executing the kernel on a GPU
vecAdd<<<2048, 64>>>(a, b, c);
```

Listing 2.1: Vector addition in CUDA

Vector addition is a straight-forward example to showcase the CUDA programming model: The host and device code that computes $\vec{c} = \vec{a} + \vec{b}$ is given in Listing 2.1. First, it contains the kernel prefixed with the __global keyword. It computes a global thread id based on the block id, the dimensionality of the block, and the block-local thread id. After that, it adds the id-th entry and assigns the result to the output array. Second, we have the host call that launches the kernel on the GPU. It creates 2048 blocks consisting of 64 threads each and provides input pointers to the kernel. Thus, the kernel adds vectors with $2048 \cdot 64 = 131072$ entries.

As CUDA code is rather low-level, higher-level primitives describe data-parallel operations on input data array [65, 67]. Figure 2.8 shows common primitives:

Map evaluates a function f on each input value in the buffer and writes it to an output buffer. Formally, it sets $out[i] := map(f(in[i]))$.

Scatter writes data from an input buffer to designated output buffer positions given in an additional buffer. Formally, it sets $out[offset[i]] := in[i]$. Conditional scatter operations only write the i-th input if the i-th entry in an additional buffer is non-zero.

Gather loads data from offsets given in an additional buffer. Formally, it computes $out[i] := in[offset[i]]$.

Reduce: The primitive aggregates all entries in the buffer based on an associative and commutative function $\oplus$. Formally, it sets $out[i] := \oplus_i in[i]$.

Prefix Sum: The primitive subsequently evaluates an associative and commutative function $\oplus$ on elements in an input buffer and writes the results to an output buffer. Formally it sets $out[i] := in[i] \oplus out[i-1]$, given $out[i-1]$ was set previously and $out[0]$ is set to a neutral element with respect to $\oplus$.

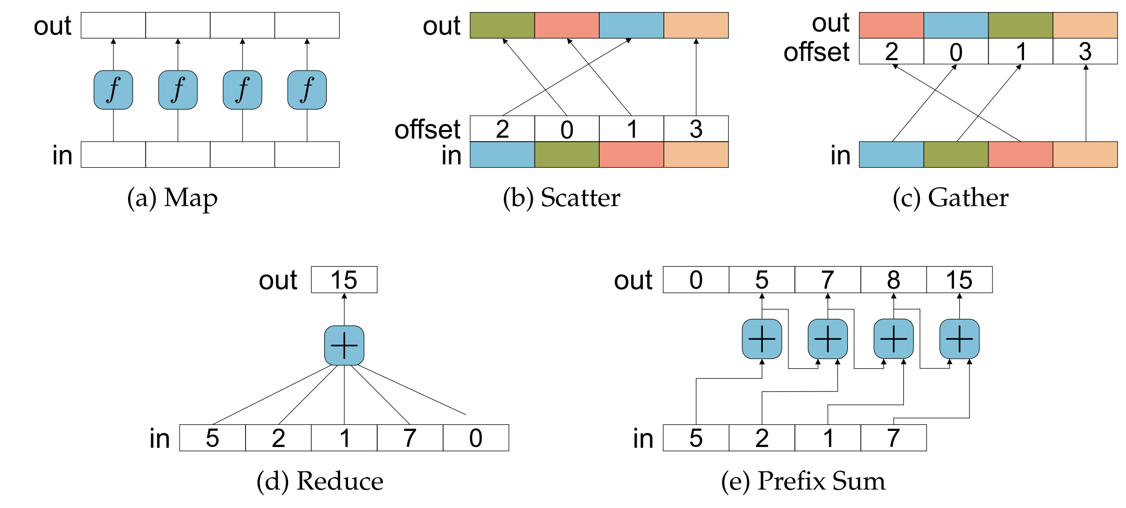


Figure 2.8: Data parallel primitives. Figure adapted from [69].

Libraries such as BoostCompute [147] [4] or Thrust [9] [5] provide optimized implementations for these primitives and, thus, reduce development overhead.

Figure 2.9 shows a filter operation implemented with data-parallel primitive: (1) The map primitive evaluates the filter predicate and encodes true as one and false as zero. (2) A prefix sum using addition is applied to the truth vector to compute the position of elements satisfying the predicate in an output buffer. (3) A conditional scatter reads entries from the input buffer and writes them to the position computed according to the prefix sum and conditioned on truth vector from the second step. Note that this multi-step procedure is necessary as threads must not only evaluate the predicate but also determine a position to write their results to.

Overall, we see that translating algorithms to the GPU programming model and primitives is not straightforward: CUDA requires a developer to model parallelism explicitly. While libraries provide convenient primitives, mapping a sequential algorithm to such primitives is not always possible or the most efficient approach.

Characteristics

To conclude our introduction of GPUs, we devise characteristics of tasks that fit a GPU's specialized processing capabilities:

High Parallelism: GPUs require an abundance of parallel threads to achieve peak performance. This abundance is crucial as scheduling ready warps during stalls not

[4] `http://boostorg.github.io/compute/`
[5] https://thrust.github.io/

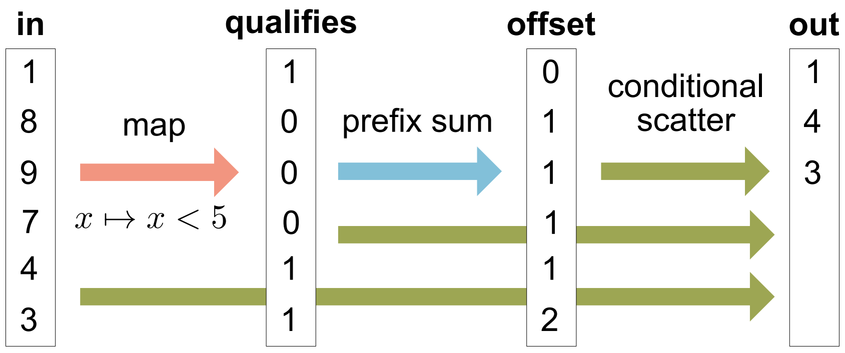


Figure 2.9: A filter implemented with data parallel primitives.

only improves the utilization of compute resources but is also the main mechanism for latency hiding.

Limited Branching: As threads are executed in lock-step and warps execute branches sequentially, threads diverging into many alternative branches reduces the degree of parallelism and, thus, reduces computational throughput.

Data Parallelism: As the GPU memory architecture needs coalesced memory accesses to achieve peak device memory bandwidth and branch divergence induces a performance penalty, data-parallel algorithms are favorable.

Limited synchronization: While thread synchronization on a block or grid level is possible, it prevents threads from executing further and limits the number of threads available for latency hiding. Thus, embarrassingly parallel tasks are ideal.

Floating point-intensive workloads: Originating from computer graphics workloads and encouraged by the trend towards deep learning, GPUs are well-known for their enormous floating point operation throughput. Recent devices provide floating point operation throughput in the order of teraFLOPs (10^{12} floating-point operations per second) [115]. Thus, compute-intensive floating-point-heavy workloads are a good match for GPUs, e.g., numeric simulations or machine learning.

Primitive operations: If the task maps to data-parallel GPU primitives, optimized library implementations from libraries are available that fully exploit the architecture.

Device Locality: In the presence of slow interconnects and the data transfer bottleneck, excessive transfers between host and device memory diminish or even negate the benefits of GPU acceleration. Hence, it is beneficial if data can reside on device memory, while transfers to the host are rare and small.

These characteristics help identify workloads suitable for GPU acceleration. However, they do not pose a list of necessary criteria for GPU acceleration: For example, relational query processing is usually not floating-point heavy but can be accelerated using GPUs in the absence of data transfer bottlenecks [16, 68, 71, 72, 102]. Neither is data locality required for GPU acceleration if compute throughput is the bottleneck.

2.2.2 Field-Programmable Gate Arrays

Field-programmable gate arrays are an architecture that, on the one hand, allows for creating custom circuits for a given task but, on the other hand, also allows changing these circuits based on software. They provide uncommitted logic resources on a chip, so-called *gate arrays*. A *bitstream* configures the device by initializing and connecting resources after manufacturing, making the architecture *field-programmable*. In the following, we will explore the FPGA architecture, which fundamentally differs from instruction processing architectures such as CPUs and GPUs. We will provide an overview of the architecture, discuss the programming model and development workflow, and characterize workloads suitable for the architecture.

FPGA vendors use different terminology for equivalent concepts. As we evaluate devices from multiple vendors, we use the vendor-independent terminology proposed by Teubner and Woods throughout this thesis [151].

Overview

FPGAs achieve reprogrammability by providing the three fundamental ingredients of electronic circuits in a reprogrammable fashion [151]: (1) Combinational logic, (2) memory elements, and (3) connections. Furthermore, there are (4) auxiliary components.

Combinational Logic: Lookup tables (LUTs) are the key concept to implementing reprogrammable logic. Table 2.1 shows LUTs that fully define the behavior of an XOR gate and a half-adder by exhaustively providing the output of the logical function for every input value. Figure 2.10 shows the FPGA realization of a lookup table: Four bits of programmable static RAM (SRAM) contain the exhaustive list of output bits for the 2-input logical function. The two input bits are processed in a tree of multiplexers that selects the SRAM cell corresponding to the input asserted. Recent FPGAs even support up to 6-input LUTs [6, 78]. The implementation of logic based on these look-up tables shows a notable difference compared to ASICs: While ASICs implement combinational logic directly from cascades of 2-input gates [6], the field-programmability feature of

[6]commonly NAND gates, as required by the CMOS fabrication process

Table 2.1: Look-up tables for an XOR gate and a half-adder

XOR		
in_1	in_2	out
0	0	0
0	1	1
1	0	1
1	1	0

Half-Adder			
in_1	in_2	sum	$carry$
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

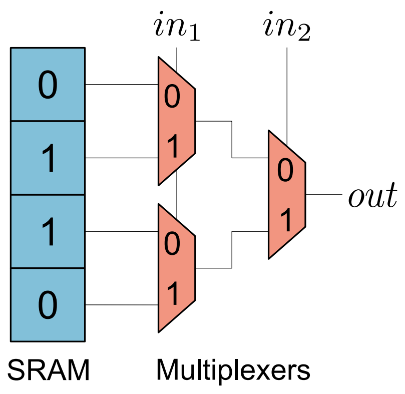


Figure 2.10: A two-input look-up table for an XOR operation

FPGAs comes at the expense of higher resource consumption for SRAM and multiplexing logic. Thus, FPGAs require a larger chip area and longer signal paths for the same logic.

Memory: Memory on the FPGA is essential to construct sequential logic from combinatorial logic. Sequential logic does not merely depend on the input signals but may also depend on a state stored in a memory element. On FPGAs, sequential logic is typically *synchronous*, meaning that memory elements hold their output for a clock cycle and change it at a clock edge. D-flip-flops provide this functionality by buffering LUT outputs for a clock cycle. Figure 2.11 shows *an elementary logic unit (ELU)* consisting of two LUTs with their accompanying D-flip-flops. An SRAM-controlled multiplexer determines whether the output is returned unbuffered or buffered by the D-flip-flop.

Synchronous sequential logic built from LUTs and flip-flops has a significant advantage: Timing analysis algorithms can validate that signals arrive at flip-flops within an interval that guarantees correct operation. Furthermore, D-flip-flops allow for pipeline parallelism in large combinatorial functions, which is the primary technique to avoid long signal paths and, thus, enable higher operating frequencies.

Connections: LUTs and their accompanying D-flip-flops need to be connected to allow for more complex logic that exceeds a single LUT. The *carry logic* shown in Figure 2.11 connects the two pairs of LUTs and Flipflops in the ELU. This carry logic provides an efficient mechanism to implement certain functions with dependencies between LUTs, e.g., adders and counters. *Logical islands* contain multiple ELUs and connect them for longer chains of carry logic.

Furthermore, a *switching matrix* provides a general mechanism to establish connections between ELUs across logical islands. Figure 2.12 shows a two-dimensional FPGA layout containing logical islands and the switching matrix: Each wire crossing in the switching matrix is a switch that establishes a connection if the corresponding SRAM cell is set. Recent FPGA architectures have transitioned from a two-dimensional to a three-dimensional FPGA layout that stacks multiple two-dimensional *super logic regions (SLRs)* [6]. However, communication across SLRs is limited.

Auxiliary Components: Common functionality is implemented directly on the FPGA fabric to save reprogrammable resources for user logic: While ELUs are sufficient to implement small random access memory, *Block RAM* elements provide high-density memory in the order of kilobytes. I/O blocks provide efficient implementations of the physical layer of communication protocols. Some FPGA devices even provide IEEE-compliant floating point units or an entire CPU.

FPGAs are very versatile, especially compared to GPUs: While GPUs exclusively operate as coprocessors, the availability of FPGA boards with various supported I/O connectors, FPGAs, and target domains invites applications besides on-chip or off-chip coprocessing [81]: FPGAs are also capable of operating autonomously in the data path.

Programming Model and Workflow

Given that recent FPGAs provide millions of LUTs [6], it is infeasible to implement a large logic design by manually initializing SRAM cells. Like ASICs, FPGA designs are commonly specified using hardware description languages, such as VHDL or Veriolog [151]. They provide the *Register-Transfer-Level (RTL)* abstraction that models sequential logic as a flow of signals between registers. [7] Hardware definition languages and RTL make describing complex designs more manageable and portable because they abstract from the specifics of the target architecture.

[7]Registers are synchronous state elements and map to D-flip-flops on an FPGA.

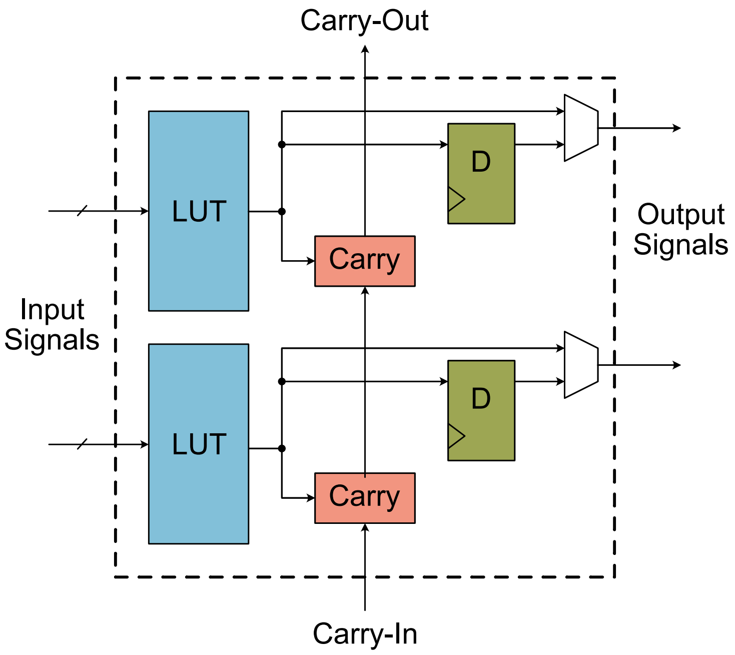


Figure 2.11: An elementary logic unit consisting of two LUTs, D-flip-flops, and carry Logic. Figure adapted from [151].

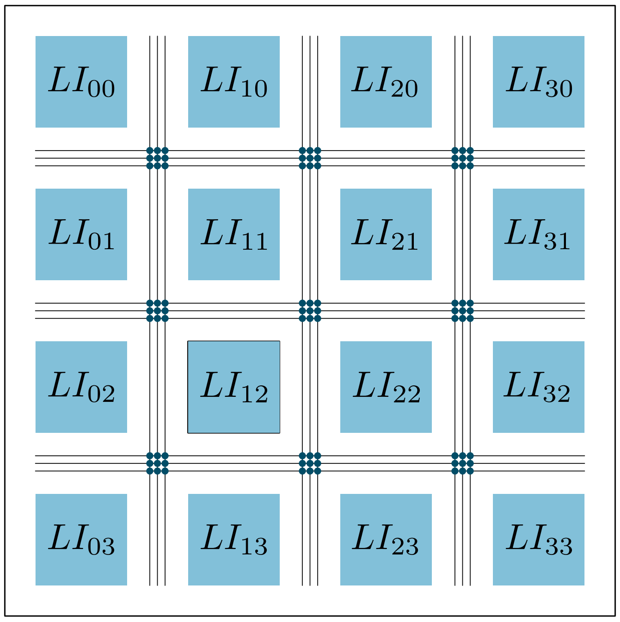


Figure 2.12: A two-dimensional grid of logical islands (LIs) with switching matrix. Black dots visualize optional connections controlled by SRAM cells. The switching matrix routes LI inputs and outputs. Figure adapted from [151].

```
 1 -- Standard includes for logic and arithmetic types
 2 library ieee;
 3 use ieee.std_logic_1164.all;
 4 use ieee.numeric_std.all;
 5
 6 -- The entity declaration: What goes in, what goes out
 7 entity adder_full_unsigned is
 8 port (
 9   clk : in std_logic;
10   add1 : in std_logic_vector(31 downto 0);
11   add2 : in std_logic_vector(31 downto 0);
12   sum : out std_logic_vector(31 downto 0)
13 );
14 end adder_full_unsigned;
15
16 -- Synchronous implementation of the process
17 architecture rtl_sync of adder_full_unsigned is
18 begin
19     process(clk) is
20     begin
21         if rising_edge(clk) then
22             sum <= std_logic_vector(unsigned(add1)+unsigned(add2));
23         end if;
24     end process;
25 end rtl_sync;
```

Listing 2.2: Synchronous 32-bit unsigned addition in VHDL

Listing 2.2 shows a synchronous unsigned addition implemented in VHDL. VHDL centers around entities and their architecture. Similar to interfaces in object-oriented programming, an entity declaration defines what input and output signals the entity receives. In our case (Line 7-14), we have two input signals for the operands and one output signal for the result. Furthermore, there is a clock signal for synchronization. The following architecture definition (Line 17-24) establishes the relationship between input and output signals. Line 21 contains the arithmetic: We assign unsigned arithmetic semantics to the operands, perform the addition, and cast back to a plain logic vector. The synchronous architecture obeys the clock signal by applying the assignment in a process construct and is conditioned on the rising edge of the clock. The output is buffered and only changes at the rising edge of each clock cycle. [8]

Between RTL and a functioning FPGA implementation lies an elaborate process performed by vendor-provided toolchains. In the following, we will introduce the workflow in Xilinx Vivado consisting of two steps [167]:

[8]We recommend the publicly available book 'Free Range VHDL' by Bryan Mealy and Fabrizio Taperro as a primer on VHDL [22].

Step 1: Synthesis maps the technology-independent hardware description to a logical representation consisting of primitives provided by the target FPGA. This translation includes not only a mapping to LUTs, but also higher-level primitives such as adders, multiplexers, or RAM.

Step 2: Implementation *places* the resources in physical locations on the FPGA and *routes* them to establish connections. *Timing analysis* verifies that the implementation satisfies all constraints to guarantee correct operations. In particular, timing analysis verifies that signals do not change in a safety interval around the clock edge. A *setup violation* occurs if the paths between registers are too long for the requested clock frequency. Signals potentially arriving too early at a flip-flop constitute a *hold violation*. Finding a placement and routing that satisfies all constraints is computationally intensive and dominates the compile time. If the implementation step finds such a valid solution, it yields a corresponding bitstream to initialize the SRAM cells of the FPGA accordingly.

Both steps incorporate constraints provided by the developer. In particular, the developer has to assign global input and output signals to physical pins of the FPGA and set clock frequencies according to the requirements of the application and the used I/O protocols.

The compilation process for FPGAs is notoriously slow compared to software - mainly due to the implementation step [151]. While small designs may only take minutes to compile, large designs on large FPGAs close to the capacity of the FPGA can take hours to days. Furthermore, hardware development for a given problem is more tedious because functionality commonly provided by the processor or operating system has to be implemented manually or with low-level library modules (e.g., memory management, I/O, or caching).

Besides RTL, FPGA vendors provide frameworks for *high-level synthesis* that translate imperative code in C/C++-based languages to RTL for supported devices, e.g., Xilinx Vitis [86] and OneAPI [76]. While HLS provides shorter time-to-product [29, 36] as it provides familiar programming languages and standard functionality (e.g., I/O, software interfaces), it is no silver bullet to translating efficient CPU and GPU code to FPGA implementations. It still requires thinking about the generated RTL, pipelining, and the target FPGA device: A developer must annotate code with FPGA-specific pragmas that guide the translation process and severely impact the RTL generated and its performance [36].

Characteristics

FPGAs offer many advantages of ASICs as they implement custom circuits with reprogrammable resources. Development cycles on FPGAs are shorter, and creating a product based on an FPGA does not require the high order quantities that make ASICs infeasible for small quantities [81, 96]. Furthermore, reprogrammability also allows for updates, while ASICs are static after production. However, these benefits come at the expense of efficiency: Reprogrammable logic requires more transistors than simply implementing the same functionality using transistors directly. This effect leads to three drawbacks compared to ASICs: (1) More chip space is required for the same logic. (2) Power consumption is higher as more transistors have to be powered for the same logic. (3) The signal path between registers is longer; thus, maximum supported clock frequencies are lower. While high-performance CPUs and GPUs support clock frequencies beyond 5 GHz, user-defined logic on recent FPGAs clocks at a few hundred MHz. Overall, the key reasons to select an FPGA to approach a particular problem instead of ASICs are reprogrammability and avoiding the high production quantities and long development cycles required for economic ASIC production [81].

As custom hardware on FPGAs can be any circuit and the market for FPGA-based accelerator devices is very diverse in terms of FPGAs and provided means of I/O and off-chip memory, it is harder to narrow down applications that fit the architecture. However, we highlight several properties in contrast to CPUs and GPUs that will be relevant for this thesis:

Custom Parallelism: Custom hardware can freely implement mixtures of parallelism strategies. While GPUs favor data parallelism with their SIMT processing model and CPUs favor task parallelism with their optimization on single-thread performance, FPGAs can be used to implement both arbitrarily. While CPUs and GPUs employ pipeline parallelism by processing instructions in pipelines, FPGAs can build arbitrarily deep pipelines. A program that requires numerous instructions to complete can become a single processing pipeline on an FPGA. Such deep pipelines allow an FPGA to consume one set of input values per clock cycle. Circuits operating in parallel naturally synchronize on clocks, while synchronizing threads in software incurs overhead.

As CPUs and GPUs operate at an order of magnitude higher clock frequencies and invest massive amounts of transistors in fast instruction processing and on-chip memory, outperforming them using an FPGA in terms of computational throughput is challenging. However, they can be 'outparallelized' for many applications by aggressively investing resources into task, data, and pipeline parallelism to make up for lower clock frequencies.

Hard Processing Guarantees: As FPGAs implement custom hardware, circuits can provide hard guarantees on the processing rate of input data. For example, an FPGA can implement hardware guaranteed to process data at a given rate. Such guarantees are harder to give on systems with CPUs and GPUs as the operating system, processor internals, and concurrent processes may affect the execution of a running program.

Energy Efficiency: Custom hardware allows implementing solely the hardware needed to solve a problem. Thus, FPGAs often have an edge in terms of more than performance per Watt [81, 122] as CPUs and GPUs are provisioned for general-purpose applications and have to power unused logic.

Flexibility: FPGAs are available with various interconnects and target application domains, ranging from embedded systems [125] over large-scale networks [111] to high-performance computing [166]. Compared to GPUs, they can operate in a standalone setup without an attached host computer or on the data path. Thus, using devices with multiple or faster interconnects can avoid the data transfer bottleneck common to GPUs.

Non-Interactive Compile Times: Compiling RTL to an FPGA bitstream can take hours to days and may even require several attempts if no valid placement and routing is found. The long compile times prohibit use cases that require recompiling FPGA designs interactively in a short time. For example, compiling a query execution plan to machine code for each incoming query is a popular strategy for CPUs and GPUs in online analytical processing [17, 112] that does not fit FPGAs. The entire circuit can change its behavior based on runtime-provided inputs (e.g., using control signals, memory initialization, or implementing instruction processing). However, this incurs a trade-off between flexibility and resource consumption and requires careful design of applications.

Overall, we summarize that FPGAs are a powerful architecture that can provide high performance and energy efficiency by implementing custom logic and exploiting circuit-level parallelism. However, this comes at a high cost in terms of development and compile times. Furthermore, designing and implementing hardware requires a different skill set compared to software development, raising the entry barrier for the technology.

3 GPU-Accelerated KDE for Join Selectivity Estimation

Join size estimation is one of the most crucial problems in relational query processing [101]. Estimates are commonly computed based on simple statistics and assumptions commonly violated in practice. In this chapter, we propose estimators for join selectivities based on GPU-accelerated kernel density models that do not require common assumptions. While the technique has been successfully applied to selections over base tables, this work constitutes an important generalization. It strengthens the case for GPUs as statistical coprocessors as more accurate estimates can be computed in the same time budget. Thus, we indirectly accelerate query processing by improving the statistics available to the optimizer. At the same time, our approach remains viable in I/O-bound databases that can not benefit from GPU-accelerated query execution directly.

This chapter is mainly based on our publication 'Estimating Join Selectivities using Bandwidth-Optimized Kernel Density Models' [87].

3.1 Introduction

In order to correctly predict the cost of candidate plans, the query optimizer of a relational database engine requires accurate information about the result sizes of intermediate plan operations [135]. The accuracy of these cardinality estimates has a direct impact on the quality of the generated query plans. Incorrect estimates are known to cause unexpectedly bad query performance [31, 80, 97, 103, 124]. In fact, due to the multiplicative nature of joins, errors in these estimates typically propagate exponentially through larger query plans [80]. This means that even small improvements can dramatically improve the information quality that is available to the query optimizer [80, 97].

A particularly challenging problem is to accurately and consistently predict the result size of joins [146]. The typical approach for this is to combine the information from base table estimators under the assumptions of uniformity, independence, and containment [146]. However, while easy to compute, this approach can cause severe estimation errors if any of the underlying assumptions is violated [31]. Multiple authors suggested specialized methods to tackle the *join estimation* problem, including sampling [56, 66, 98, 157], graphical models [59, 155], and sketches [95, 129]. However, none of them managed to manifest themselves as a generally viable solution, leaving the problem as one of the major unsolved challenges in research on query optimization [101].

In prior work, Heimel et al. introduced *bandwidth-optimized kernel density models (KDE)* as a way to compute multidimensional selectivity estimates. KDE is a data-driven, non-parametric method to estimate probability densities from a data sample [134]. They demonstrated that combining KDE with a query-driven tuning mechanism to pick the so-called *bandwidth* parameter optimally yields an estimator that typically outperforms the accuracy of state-of-the-art multidimensional histograms. Furthermore, the estimator is easy to maintain and to parallelize on graphics processing units [70]. In this work, we significantly expand upon their work and demonstrate how to estimate the result size of queries spanning multiple equijoins and base table predicates from bandwidth-optimized KDE models. In particular, we explain how to compute estimates from both joint models and combined base table models. We present pruning methods to reduce the computational overhead and demonstrate a mechanism to tune the bandwidth parameter automatically. Based on an extensive experimental evaluation, we found that our family of estimators matches and usually outperforms the accuracy of existing state-of-the-art join estimators like correlated sampling [157] or the AGMS sketch [129].

In the following two sections, we provide background on the join estimation problem and bandwidth-optimized KDE models. In Section 3.4, we lay the theoretical foundation of our work, explaining how to estimate join selectivities from a KDE model. Section 3.5 introduces pruning techniques to reduce the computational overhead, and Section 3.6 discusses strategies to fine-tune the bandwidth parameter of these models. Finally, Section 3.7 presents our experimental evaluation, and Section 3.8 concludes the chapter by summarizing our findings.

3.2 The Join Estimation Problem

Given a set of n relations $R_1, R_2, \ldots, R_n$, and a query $Q = \sigma_{c_1}(R_1) \bowtie_{\theta_1} \left(\ldots \bowtie_{\theta_{n-1}} \sigma_{c_n}(R_n)\right)$, where σ_{c_i} denotes a selection with (local) predicate c_i and $\bowtie_{\theta_i}$ a join with join predicate θ_i, our goal is to predict the fraction of tuples from the Cartesian Product $R_1 \times \ldots \times R_n$ that fall into the query's result. This *join estimation problem* is one of the classic problems from query optimization research [101], and improving the quality of join estimates has a direct and measurable impact on the plan quality produced by cost-based optimizers [31, 80, 94, 97, 103, 124, 143]. We consider the important subproblem where all joins are equijoins.

3.2.1 Classic Join Estimation

The classic way to estimate an equijoin between two tables R_1 and R_2 requires us to know the number of distinct keys $n_{R_1.A_1}$ and $n_{R_2.A_1}$ in the corresponding join columns. Assuming *uniformity*, each distinct key in R_i will appear $|R_i|/n_{R_i.A_1}$ times. Further assuming that the key domain from the table with fewer distinct keys is a subset of the other table's domain – the *containment* assumption –, and assuming that the local predicates c_1 and c_2 are *independent* of the join attribute, we arrive at the classic join estimation formula that is used by most query optimizers [135, 146]:

$$\left|\sigma_{c_1}(R_1) \bowtie \sigma_{c_2}(R_2)\right| \approx \frac{\left|\sigma_{c_1}(R_1)\right| \cdot \left|\sigma_{c_2}(R_2)\right|}{\max\left(n_{R_1.A_1}, n_{R_2.A_1}\right)} \tag{3.1}$$

While straightforward to derive and easy to compute, the underlying assumptions make Equation (3.1) susceptible to several sources of estimation errors that can cause substantially under- or overestimations of the join result size [80, 97, 146]. These deficiencies have inspired several researchers to investigate more sophisticated methods for estimating join result sizes. These methods can be broadly categorized into two classes: While *base table models* dynamically combine the information from individual estimators, *joint models* directly model the value distribution for preselected joins. Joint models usually produce more accurate estimates but are less flexible and harder to maintain than base table models.

3.2.2 Sampling-based Join Estimation

Sampling is a powerful tool to estimate selectivities for both individual and joined query results. Creating and maintaining a random sample from database tables is a well-understood topic [158], and we can directly produce estimates from base table samples by evaluating the actual query on them [66, 116]. However, joining base table samples has one major drawback: While it is an unbiased estimator to the selectivity, its variance is very high for small sample sizes due to missing join partners in the sample. While this problem can be mitigated using non-uniform methods like end-biased [48] or correlated sampling [157], the created samples are targeted to predefined joins. Another possibility is to build a joint model by sampling directly from the result of a particular join. Sampling from a join result, and maintaining said sample under updates, deletions and insertions, are well-understood problems and can be done efficiently in the presence of join indexes [27, 116]. Such a join sample generally produces better estimates and requires smaller sample sizes compared to evaluating the query based on base table samples [66, 98]. Leis. et al. showed that estimates computed from join samples significantly improve plan quality [98]. However, join samples are limited to queries that use the same join from which the sample was drawn.

3.2.3 The AGMS Sketch

The AGMS sketch is a probabilistic data structure to estimate the join size between two data streams [4, 5]. It extends to multiple joins and selections by representing them as a join with all values matching the selection predicate [43, 157]. We discussed its basic construction, estimation procedure, and properties in Section 2.1.2.

3.3 Bandwidth-Optimized KDE

We established the theoretical framework for KDE and its interpretation as a data summary in Section 2.1.1. In prior work [70], Heimel et al. demonstrated how to derive a selectivity estimator for multidimensional range queries based on KDE. They also demonstrated how to select the estimator's bandwidth parameter by numerically minimizing the estimation error. For this, they plugged the gradient of KDE's estimation error with respect to its bandwidth into an off-the-shelf numerical solver. They continuously fed this solver with training data obtained from user queries collected on-the-fly. This query-driven tuning mechanism allowed them to outperform the accuracy of state-of-the-art multidimensional histograms such as GenHist [62] or STHoles [21], while still offering the flexibility and maintainability of a sample-based method. Furthermore, Heimel et al. explained how the estimator is efficiently evaluated and maintained on GPUs [70, 88] and, thus, identified an interesting use case for GPUs in relational databases besides actual query execution [16, 72]: As relational query execution on GPUs commonly suffers from data transfer bottlenecks, supplying more accurate statistics can improve plan quality and, thus, indirectly speed-up query execution. In this scenario, the GPU acts as a *statistical coprocessor* that maintains and evaluates more accurate KDE models than could be evaluated on a CPU within the same time budget.

3.4 KDE-based Join Estimation

We will now discuss how to compute the result size of equijoin queries from KDE models. In particular, we introduce two basic strategies: A joint model that works by building a KDE estimator from a sample directly drawn from the join result, and a base table model that works by dynamically combining multiple base table KDE models. By combining base table models, we effectively join their estimated distributions and avoid the problem of empty join results for naive sample evaluation.

3.4.1 Estimating from a Join Sample

The straightforward method to estimate joins based on KDE is to build the model from a sample that we drew directly from the join result. Sampling data directly from a join result is well-understood and can be efficiently implemented [116]. Since KDE is inherently sample-based, this method allows us to build a KDE-based join estimator without having to change a single line of code of the base table model. The advantages and disadvantages of this approach are identical to naïve sample evaluation: While we expect the joint model to produce very accurate estimates [66], it is less flexible than any method that combines base table KDE models. In fact, while the latter model can provide selectivity estimates for any arbitrary equijoin, the former requires us to construct and maintain joint models for all potential joins in the query workload.

3.4.2 Combining Base Table Models

Let us now derive the estimation formula for computing join estimates from individual base table models. To simplify this derivation, let us first consider the case of predicting the result size of a two-way equijoin query with local predicates: $Q = \sigma_{c_1}(R_1) \bowtie_{R_1.A_1=R_2.A_1} \sigma_{c_2}(R_2)$. We will later generalize this to multiple joins. For each individual join key v, we can express the number of result tuples produced for that key as:

$$\begin{gathered}\left|\sigma(R_1)_{R_1.A_1=v\wedge c_1}\right| \cdot \left|\sigma(R_2)_{R_2.A_1=v\wedge c_2}\right| = \\ |R_1| \cdot p_1(R_1.A_1 = v \wedge c_1) \cdot |R_2| \cdot p_2(R_2.A_1 = v \wedge c_2)\end{gathered} \tag{3.2}$$

In this equation, the function $p_i(c)$ denotes the exact base table selectivity for a predicate c on table R_i. We can now define the join selectivity $J(Q) = {}^{|Q|}/_{|R_1|\cdot|R_2|}$ by summing Equation (3.2) over all keys $v \in A$, where A is the join domain, which is the set of distinct join keys. Note that A can be reduced to a subset of the distinct keys in the join columns due to local table predicates:

$$J(Q) = \sum_{v \in A} p_1(A_1 = v \wedge c_1) \cdot p_2(A_1 = v \wedge c_2) \tag{3.3}$$

We replace p_1 and p_2 by our base table estimators $\hat{p}_1$ and $\hat{p}_2$, and arrive at the join selectivity estimator $\hat{J}(Q)$:

$$\hat{J}(Q) = \sum_{v \in A} \hat{p}_1(A_1 = v \wedge c_1) \cdot \hat{p}_2(A_1 = v \wedge c_2) \tag{3.4}$$

Substituting the definition of a KDE estimator from Equation (2.4), we find that both estimators are evaluated and their results are multiplied. By distributivity, we can compute and multiply the individual contributions for all combinations of sample points in their respective samples S_1 and S_2, and sum over the products.

$$\begin{aligned}\hat{J}(Q) &= \frac{1}{s_1 \cdot s_2} \sum_{v \in A} \left(\left(\sum_{i=1}^{s_1} \hat{p}_1^{(i)}(A_1 = v \wedge c_1) \right) \left(\sum_{j=1}^{s_2} \hat{p}_2^{(j)}(A_1 = v \wedge c_2) \right) \right) \\ &= \frac{1}{s_1 \cdot s_2} \sum_{v \in A} \sum_{\substack{i=1 \\ j=1}}^{s_1, s_2} \hat{p}_1^{(i)}(A_1 = v \wedge c_1) \cdot \hat{p}_2^{(j)}(A_1 = v \wedge c_2)\end{aligned} \tag{3.5}$$

Assuming that the kernel functions used by our KDE models are *product kernels* [134], the following identity holds: $\hat{p}^{(i)}(A_1 = v \wedge c) = \hat{p}^{(i)}(A_1 = v) \cdot \hat{p}^{(i)}(c)$. Substituting this into Equation (3.5) allows us to isolate the join-specific parts of the computation:

$$\begin{aligned}\hat{J}(Q) &= \frac{1}{s_1 \cdot s_2} \sum_{v \in A} \sum_{\substack{i=1\\j=1}}^{s_1, s_2} \hat{p}_1^{(i)}(c_1) \cdot \hat{p}_1^{(i)}(A_1 = v) \cdot \hat{p}_2^{(j)}(c_2) \cdot \hat{p}_2^{(j)}(A_1 = v) \\ &= \frac{1}{s_1 \cdot s_2} \sum_{\substack{i=1\\j=1}}^{s_1, s_2} \hat{p}_1^{(i)}(c_1) \cdot \hat{p}_2^{(j)}(c_2) \cdot \underbrace{\left(\sum_{v \in A} \hat{p}_1^{(i)}(A_1 = v) \cdot \hat{p}_2^{(j)}(A_1 = v) \right)}_{\hat{J}_{i,j}}\end{aligned} \tag{3.6}$$

We refer to $\hat{J}_{i,j}$ as the *cross contribution*. Naively computing the cross contribution in a selectivity estimation scenario is infeasible, as the join key domain A is potentially huge and generally unknown at query optimization time. Instead, we have to exploit the properties of kernels to avoid explicitly summing over the entire join domain. Equation (3.7) — which is derived in Appendix A.1 —, provides a closed-form approximation to the cross contribution for a Gaussian kernel on integer attributes. The Gaussian kernel is a common choice for KDE-based estimators and is used in our experimental evaluation.

$$\hat{J}_{i,j} \approx \mathcal{N}_{t_1^{(i)}, (\delta_1^2 + \delta_2^2)}\left(t_2^{(j)}\right) \tag{3.7}$$

In this equation, $\mathcal{N}_{\mu,\sigma^2}$ denotes the probability density function for a standard normal distribution with mean μ and variance σ^2, $t_1^{(i)}$ denotes the i-th sample point from $R_1.A_1$, and δ_1 denotes the join estimator bandwidth for R_1.

3.4.3 Extending to Multiple Joins

In order to generalize our approach to multiple joins, we need to introduce the notion of equivalence classes. For two tables R_i and R_j that join on the attributes $R_i.A_l$ and $R_j.A_m$, we consider the pair of attributes equivalent $R_i.A_l \sim R_j.A_m$. Note that, by definition, this equivalence also holds transitively. We denote the equivalence class for a given attribute $R_j.A_m$ by $\Psi\left(R_j.A_m\right)$. Each equivalence class contains a set of attributes that have to be equal for all tuples in the join result. Based on this definition, we can now discuss how the cross contribution can be generalized to equivalence classes containing more than two relations and how we can compute the join selectivity for an arbitrary number of equivalence classes.

First, we consider the case of joins consisting of a single equivalence class $\Psi(R_1.A_1)$ containing n relations. Since all join attributes are in the same equivalence class, we

```
# 1) Apply sample pruning:
$S_1 = S_1 \setminus \left\{t_1^{(i)} \in S_1 \middle| p_1^{(i)}(c_1) < \theta\right\}$
$S_2 = S_2 \setminus \left\{t_2^{(i)} \in S_2 \middle| p_2^{(i)}(c_2) < \theta\right\}$

# 2) Sort $S_2$ by the join key (if necessary)
$S_2$ = sort($S_2$,$S_2.A_1$)

# 3) Apply Cross Pruning and estimate selectivity:
for $i$ in $\{1, ..., s_1\}$:
    j = binarySearch($t_1^{(i)}$ − maxdiff($\delta_1, \delta_2$), $S_2$)
    while $\left|t_1^{(i)} - t_2^{(j)}\right| \leq$ maxdiff($\delta_1, \delta_2$) $\wedge j \leq s_2$:
        $\hat{J}$ = compute$\hat{J}$($t_1^{(i)}, t_2^{(j)}, \delta_1, \delta_2$)
        $e$ += $\hat{p}_1^{(i)}(c_1) \cdot \hat{J} \cdot \hat{p}_2^{(j)}(c_2)$
    return $e/s_1 \cdot s_2$
```

Listing 3.1: Combining base table KDE models

can still sum over the shared join domain A. Assuming that each joined table R_i has a kernel density estimator model $\hat{p}_i$, we define the generalized cross contribution $\hat{J}_{o_1,\ldots,o_n}$ that needs to be computed for the cross product between all samples:

$$\hat{J}_{o_1,\ldots,o_n} = \sum_{v \in A} \prod_{i=1}^{k} \hat{p}_i^{(o_i)}(A_1 = v) \tag{3.8}$$

Appendix A.1 provides a closed-form approximation for the generalized cross contribution of a Gaussian kernel. Now, since the kernels for each dimension are independent, the final formula to compute the join selectivity for n equivalence classes $\Psi_1, \ldots, \Psi_k$ over a total of n relations can be computed by multiplying their respective generalized cross contributions J_i (omitting the sample offsets for readability) with the contributions for the local predicates c_j:

$$\hat{J}(Q) = \frac{1}{\prod_{i=1}^{n} s_i} \sum_{i_1=1,\ldots,i_n=1}^{s_1,\ldots,s_n} \prod_{j=1}^{n} \hat{p}_j^{(i_j)}(c_j) \prod_{j=1}^{k} \hat{J}_j \tag{3.9}$$

3.5 Efficiently Joining KDE Models

In the previous section, we derived the theoretical foundation for computing join selectivities from base table models. We now discuss how we can efficiently evaluate them in practice. Our estimator receives the relational query $Q = \sigma_{c_1}(R_1) \bowtie_{R_1.A_1=R_2.A_1} \sigma_{c_2}(R_2)$,

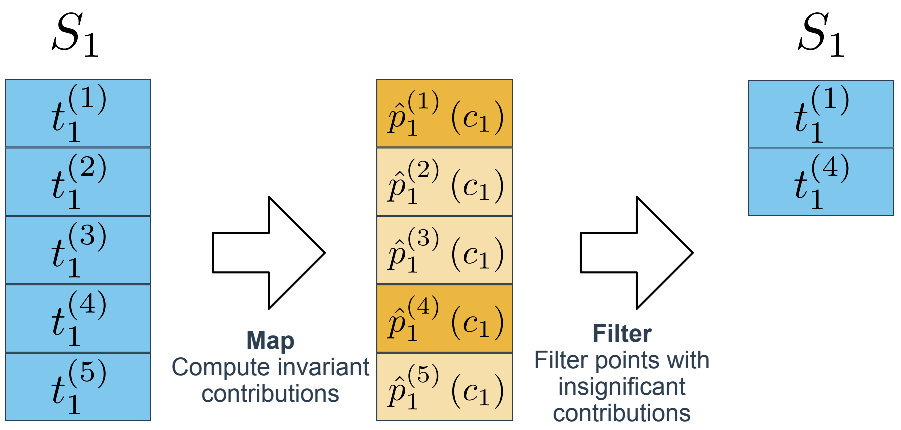


Figure 3.1: Sample pruning computes the invariant contributions and removes sample points with a negligible contribution.

as well as two KDE estimators with their respective base table samples S_1, S_2 and bandwidth vectors $\vec{\delta_1}, \vec{\delta_2}$. Based on Equation (3.6), we know that computing the join selectivity requires us to compute the cross contributions $\hat{J}_{i,j}$ for all $s_1 \cdot s_2$ pairs from the cross product of S_1 and S_2, incurring quadratic complexity. Accordingly, a naïve implementation would severely limit the scalability and applicability of our approach. To reduce the number of required computations, we now introduce two pruning techniques: *Sample Pruning* and *Cross Pruning*. Algorithm 3.1 illustrates these methods and the general selectivity estimation procedure.

Sample Pruning: If the local contribution for a particular sample tuple is sufficiently small, the contribution of every derived tuple from the cross product will be negligible. Thus, we can omit this sample point in all following computations, which we call *Sample Pruning* (Lines 2 – 3). Similar to pushing down selections in query execution plans, we can reduce the number of input tuples that have to be considered in the more expensive computation of the cross contributions. We chose the threshold as the inverse of the cross product size $\theta = \frac{1}{r_1 \cdot r_2}$, as this limits the overall error to the join cardinality estimate to at most one tuple.

As shown in Figure 3.1, sample pruning directly maps to the GPU primitives introduced in Section 2.2.1: We compute the invariant contribution using a map primitive and select sample points over the local contribution threshold using a filter operation.

Cross Pruning: Next, we compute the cross contributions (Line 10), multiply them with their corresponding local contributions and sum them up to compute the join selectivity (Line 11). In this part of the algorithm, we apply *Cross Pruning* to reduce the computational load: As the Gaussian kernel applies smoothing by distance, it's intuitive that the cross contribution for two sample points becomes negligible when the distance between the two points is very large. Again choosing the maximum tolerable error to be $\frac{1}{r_1 \cdot r_2}$, the maximum tolerable distance between two sample values is:

$$\begin{aligned} & \frac{1}{r_1 \cdot r_2} > \mathcal{N}_{t_1^{(I)}, \delta_1^2 + \delta_2^2}\left(t_2^{(j)}\right) \\ \Longleftrightarrow\ & \frac{1}{r_1 \cdot r_2} > \frac{1}{\sqrt{2\pi\left(\delta_1^2 + \delta_2^2\right)}} \exp\left(-\frac{1}{2}\frac{\left(t_1^{(I)} - t_2^{(j)}\right)^2}{\left(\delta_1^2 + \delta_2^2\right)}\right) \\ \Longleftrightarrow\ & \left|t_1^{(I)} - t_2^{(j)}\right| > \sqrt{-2 \cdot \ln\left(\frac{\sqrt{2\pi\left(\delta_1^2 + \delta_2^2\right)}}{r_1 \cdot r_2}\right)\left(\delta_1^2 + \delta_2^2\right)} \end{aligned} \tag{3.10}$$

To exploit this property, we first sort S_2 on the join attribute (Line 5). Next, instead of iterating over the cross product, the algorithm considers only tuples that are sufficiently close to each other by iterating over all tuples from S_1 (Line 7) and finding the first qualifying tuple from S_2 via binary search (Line 8). The function *maxdiff* computes the maximum distance according to Equation (3.10). Finally, we iterate over all qualifying tuples from S_2 (Line 9 – 11), compute the cross contribution $\hat{J}$ (Line 10), and iteratively compute the join selectivity (Line 11).

Cross pruning essentially requires us to perform a band join between the two samples. This is visualized in Figure 3.2. On a GPU, we implement this by scanning S_1 in a coalesced fashion and performing the look-ups in the sorted relation in parallel. Each thread accumulates the cross contribution for its part of the join result, aggregating it using the reduce primitive as a final step. The mixture of random memory access and cross contribution computations is a good fit for latency hiding in GPUs.

Sample pruning and cross pruning can significantly reduce the number of computations required to compute an estimate, in particular when the join and selections are very selective. Sorting, if necessary, can be done in $O(s_2 \log s_2)$, computing the local contributions and pruning can be done in a single pass over each sample [70]. We have to compute s_1 binary searches, each requiring at most $\log(s_2)$ accesses to S_2. The actual number of elements traversed in the inner while-loop is data-dependent. In the degenerate case of a join that is close to a cross product, we still need to traverse S_2 for every tuple in S_1, and the complexity of the overall algorithm remains $O(s_1 \cdot s_2)$. However, in the optimal case, we only have to check a handful of tuples from S_1, which

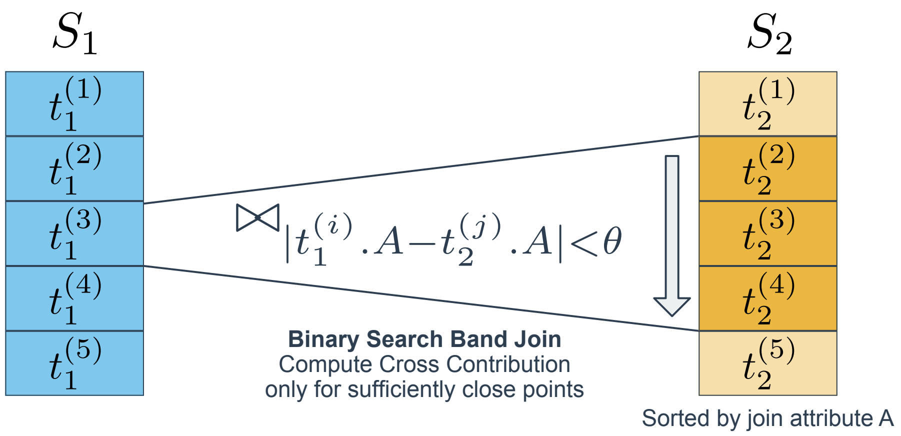


Figure 3.2: Cross pruning applies a band join instead of a full cross product and, thus, skips all combinations from the cross product resulting in a negligible cross contribution.

yields $O(s_1 \, log \, s_2)$. We argue that the degenerate case rarely appears in real-world data and provide an experimental evaluation on real-world data in Section 3.7.4.

3.5.1 Extending to Multiple Joins

Generalizing this algorithm to multiple joins requires only a few modifications. In particular, we must apply sample pruning to all base table samples. We pick a left-deep join order and sort the samples for the right-hand side of all join operators based on their join attribute. This way, we ensure that we must sort at most $j-1$ samples for a total of j joins. As we only allow bandwidth values such that the function values of the cross contribution never exceeds one, we can handle the joins by subsequent binary searches and apply cross pruning for each of them.

3.6 Bandwidth Optimization

Prior work [70] demonstrated that query-driven bandwidth optimization is crucial to the estimation quality of KDE-based selectivity estimators. During query execution, we observe the true selectivity of operators, which allows us to optimize the bandwidth based on the estimation error numerically. Since we cannot use sample or cross pruning to speed up the gradient computations for base table KDEs — a negligible contribution to the estimate does not imply a negligible contribution to the gradient —, we instead rely on a derivative-free, bound-constrained optimization algorithm. In particular, we use *constrained optimization by linear approximation* (COBYLA) [121] from nlopt [85]. We use the same algorithm for KDE over join samples.

We optimize the bandwidth based on the multiplicative error, which, given the true selectivity c and an estimate $\hat{c}$, is defined as:

$$m(c, \hat{c}) = \frac{max(c, \hat{c})}{min(c, \hat{c})} \tag{3.11}$$

If the estimate is larger than the actual selectivity, the multiplicative error is equivalent to the relative error; Otherwise, it is the inverse of the relative error. Thus, the smallest multiplicative error is 1.0, and over- and underestimations are equally penalized. The multiplicative error is the error metric of choice for cardinality estimation, as it minimizes error propagation in query plans and correlates directly with plan quality [110]. By optimizing for this error function, we ensure that the optimization translates to improved query plans.

Given a set of representative queries $Q_1, \ldots, Q_n$, we then optimize the bandwidth vectors for all tables $\vec{\delta}_1, \ldots, \vec{\delta}_m$ for the geometric mean over the estimation error:

$$\underset{\vec{\delta}_1, \ldots, \vec{\delta}_m}{\arg\min} \left(\prod_{i=0}^{n} m\left(J(Q_i), \hat{J}(Q_i)\right) \right)^{\frac{1}{n}} \tag{3.12}$$

Note that the estimate $\hat{J}$ depends on the bandwidth vectors $\vec{\delta}_1, \ldots, \vec{\delta}_m$. We can only optimize the bandwidth for attributes that are actually covered in the queries $Q_1, \ldots, Q_n$. We suggest collecting query feedback for all base table filters and subsequent join operators in a query plan. This only requires keeping track of intermediate results, which can be done with very little overhead. The optimization process can then be executed periodically or triggered by a database command. Note that bandwidth optimization does not block the KDE models, and thus, optimization and estimation can be interleaved.

3.7 Evaluation

In this section, we present the experimental evaluation of our KDE-based join estimators in terms of estimation quality and execution time. All experiments can be reproduced using the code and datasets from our public repository[1].

[1] https://github.com/martinkiefer/join-kde

3.7.1 Experimental Setup

The following datasets, workloads, and estimators were used in our experiments:

Estimators

We compared the following estimators:

Postgres: Our first baseline estimator uses the EXPLAIN feature of Postgres 9.6. Postgres uses the classic join estimation formula, relying on the independence assumption and 1D statistics (frequent values, histograms, and distinct values per attribute).

Table Sample (TS): Our second baseline estimator, which implements naïve sample evaluation based on uniform samples from the base tables.

Join Sample (JS): The final baseline estimator, which implements naïve sample evaluation based on a single uniform sample from the join result.

Correlated Sample (CS): A sampling-based estimator operating on biased samples constructed by using a common hash function on join attributes [157]. By correlating the samples on the join attribute, the problem of empty join results due to independence is avoided. Compared to other biased sampling algorithms, it does not require prior knowledge of the data distribution.

AGMS: We implemented the AGMS sketch with extensions to filter conditions as proposed in [157]. Random variables were generated by the EH3 hashing scheme, which was shown to be favorable in terms of hash size and generation efficiency [127]. We implemented range predicates using range-summation to avoid constructing sketches by adding each value in the range individually [127].

JS+KDE: Our KDE estimator based on a join sample, as described in Section 3.4.1.

TS+KDE: Our base KDE estimator based on table samples, as described in Section 3.5.

Note that we specifically evaluated against estimators that are comparable in terms of model construction and supported estimation operations (join subject to conjunctive base table predicates). In particular, all compared estimators can be constructed in a single pass over the data and can be maintained under updates.

All KDE models use bandwidth vectors optimized for the geometric mean of the multiplicative error on a set of 100 training queries.

Datasets

We conducted our experiments based on the following datasets that cover both synthetic and real-world examples:

SN (Shifted Normal): Synthetic dataset consisting of 100k tuples drawn from normal distributions $\mathcal{N}_{\mu_1,\Sigma}$, $\mathcal{N}_{\mu_2,\Sigma}$, and $\mathcal{N}_{\mu_3,\Sigma}$. Values were rounded to the closest integer to simulate a discrete dataset. The covariance matrix Σ was chosen as $\left(\begin{smallmatrix} 1200 & 1100 \\ 1100 & 1200 \end{smallmatrix}\right)$. The means were chosen as $\mu_1 = \left(\begin{smallmatrix} 500 & 700 \end{smallmatrix}\right)^T$, $\mu_2 = \left(\begin{smallmatrix} 600 & 700 \end{smallmatrix}\right)^T$ and $\mu_3 = \left(\begin{smallmatrix} 700 & 700 \end{smallmatrix}\right)^T$. Thus, all attributes are dense and highly correlated.

IMDb: Real-world dataset based on data from the Internet Movie Database[2], which we obtained using the python package IMDbPY[3] . Our queries use the tables `title` (3.5m tuples), `movie_keyword` (6m), `cast_info` (50m), `company_name` (300k), and `movie_companies` (4m).

DMV: Real-world dataset based on data from a Department of Motor Vehicles [75, 104]. The dataset contains information on cars, their owners, and accidents in six relations containing 23 columns. The relations contain between 269 and 430k tuples.

We used dictionary encoding to transform all string attributes into integers.

Query Workload

Every workload in our evaluation is defined by a prepared query statement and a workload strategy (Uniform, Distinct). The prepared statement contains up to three joins as well as selections with conjunctive range and equality predicates. The left-hand side of the selection predicates is a base table attribute, while the right-hand side is a parameter. We used the following algorithm to generate actual queries from the prepared statement based on the chosen workload strategy:

1. We compute the full join in the prepared statement while ignoring the selections, and project on the non-join-attributes in the prepared statement.
2. We select a tuple t from the join result based on the workload strategy: (Uniform) We select a tuple from the join result with uniform probability. (Distinct) We eliminate duplicates from the join result and draw a tuple with uniform probability.

[2]`http://www.imdb.com/interfaces`
[3]`http://imdbpy.sourceforge.net`

3. For attributes subject to equality predicates, the values on the previously drawn tuple t become the selection parameters. For an attribute A with range predicates, we need to provide an upper and a lower bound. We retrieve the minimum value min_A and maximum value max_A after applying the equality predicates. Defining the value of attribute A on the tuple t as $t.A$, we select the upper bound as $u_A = [T.A + (max_A - T.A) \cdot rand()]$ and the lower bound as $l_A = [T.A - (T.A - min_A) \cdot rand()]$. The function *rand* returns a random real value in $[0, 1)$.

The uniform strategy follows the distribution in the join result and therefore favors queries with higher selectivities. As the distinct strategy disregards the distribution of tuples in the join result, it favors queries with lower selectivities. Intuitively, a learning estimator should be more effective on the uniform workload as the queries focus on strongly represented regions. The distinct workload is harder to learn, as the estimator has to fit the entire dataset.

3.7.2 Estimation Quality

In the first set of experiments, we compared the accuracy of all estimators to get a feeling of how our KDE-based join estimators stack up against the state of the art. For these experiments, the size of base table samples was fixed to one percent, join samples and number of AGMS sketches were chosen to match the memory required by the base table samples. While the sample sizes for correlated samples vary inherently, we chose the sampling threshold to meet the target size in expectation. The actual experiment then consisted of optimizing the bandwidth of our KDE-based models on 100 training queries, followed by measuring the multiplicative estimation error for all estimators on another 100 queries from the selected workload. We ran this experiment for different query patterns over the dataset and both workloads, repeating it 20 times for each combination.

SN Dataset

Figure 3.3 illustrates the results of this experiment for the SN dataset. SN Q1 joins the tables generated with μ_1 and μ_2 on their first attribute. The remaining two attributes are subject to range selections. Methods based on uniform base table or join samples clearly outperform the other estimators on this workload by more than 60% in terms of the median estimation error. KDE on base tables provides a small improvement of up to 15% over plain base table sample evaluation, while join samples with and without KDE both provide close to perfect estimates. Correlated samples provide the worst estimates

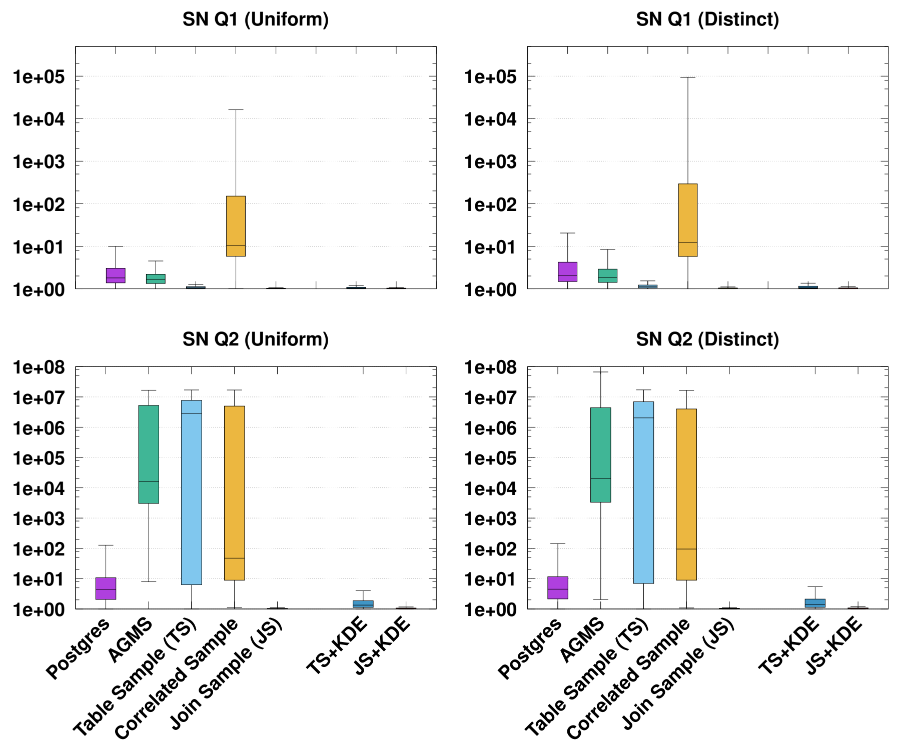


Figure 3.3: Estimation quality, SN dataset. The y-axis shows the multiplicative estimation error.

for this query and are off by one order of magnitude. As the join is not a PK-FK join, and the join attributes are skewed, the problem tackled by correlated samples does not arise.

SN Q2 adds the table generated with μ_3 to the join and introduces an additional range predicate to the query. As for SN Q1, join sample-based estimators produce close to perfect estimates. This is not surprising, as the complexity introduced by the additional join is handled in the sampling process. TS+KDE is superior to all other base table estimators and provides a better median estimation error by a factor of 4 than Postgres. In contrast, table samples, AGMS, and correlated samples are heavily affected by the additional joins and yield estimates that are off by up to seven orders of magnitude.

DMV Dataset

Figure 3.4 illustrates the results of this experiment for the DMV dataset. We evaluate three query patterns over the four main tables of the datasets. DMV Q1 consists of a single join and four base table selections (two range predicates, two equality predicates). DMV Q2 and DMV Q3 successively add a join. Furthermore, they add two range selections and one equality selection, respectively.

We observe that TS+KDE is superior to the other base table estimators and outperforms them by at least one order of magnitude in terms of the median estimation error in all experiments. Only Join Sample and JS+KDE perform better by up to a factor of four. While these estimators are very close regarding the median estimation error, JS+KDE improves the estimation errors above the median for IMDb Q2 and Q3.

For DMV Q1, correlated sampling outperforms Table Sample by more than an order of magnitude. However, for DMV Q2 and Q3, Postgres, Table Sample and Correlated Sample perform very similarly.

The AGMS sketch scales poorly with the introduced selections and performs worse than the other estimators in this experiment. Its median estimation error compared to Postgres is worse by two orders of magnitude — and the upper whisker and box boundary even extend beyond the plot boundaries. Given that the estimator variance for the AGMS sketch is proportional to the size of the cross product and selections are handled by joining with additional virtual tables [157], this behavior is expected.

IMDB Dataset

Figure 3.5 illustrates the results of this experiment for the IMDb dataset. IMDb Q1 joins two tables subject to one range and two equality predicates. IMDb Q2 and Q3 add a join and an equality selection predicate, respectively.

Join sample and JS+KDE provide close to perfect estimates for almost all experiments. IMDb Q3 with the distinct workload is the only exception to this: While both estimators have comparable median errors, JS+KDE shows a much better error distribution above the median as the upper box boundary and whisker improved by one and two orders of magnitude, respectively.

TS+KDE provides the best estimates among the base table estimators. We see drastic improvements of an order of magnitude over correlated samples and the AGMS sketch. Compared to Table Sample, we observe drastic improvements of more than an order of magnitude for IMDBb Q2 (Distinct), Q3 (Uniform), and Q3 (Distinct) — the estimates are comparable for all other experiments. Postgres estimation errors predominantly lie between one and ten, which is very competitive — especially for Q2 and Q3. However, TS+KDE still provides an improvement between factors of two and four for IMDb Q1 and Q2. For IMDb Q3, the provided estimates are comparable.

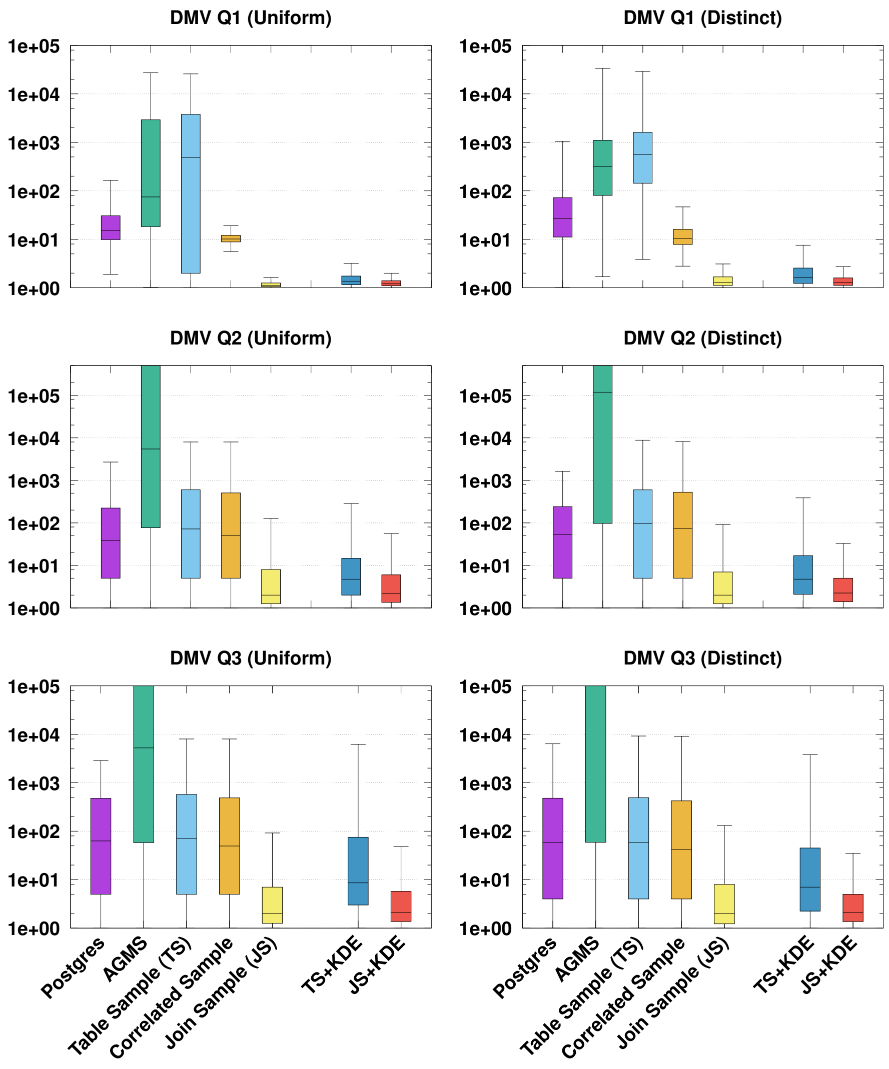


Figure 3.4: Estimation quality, DMV dataset. The y-axis shows the multiplicative estimation error.

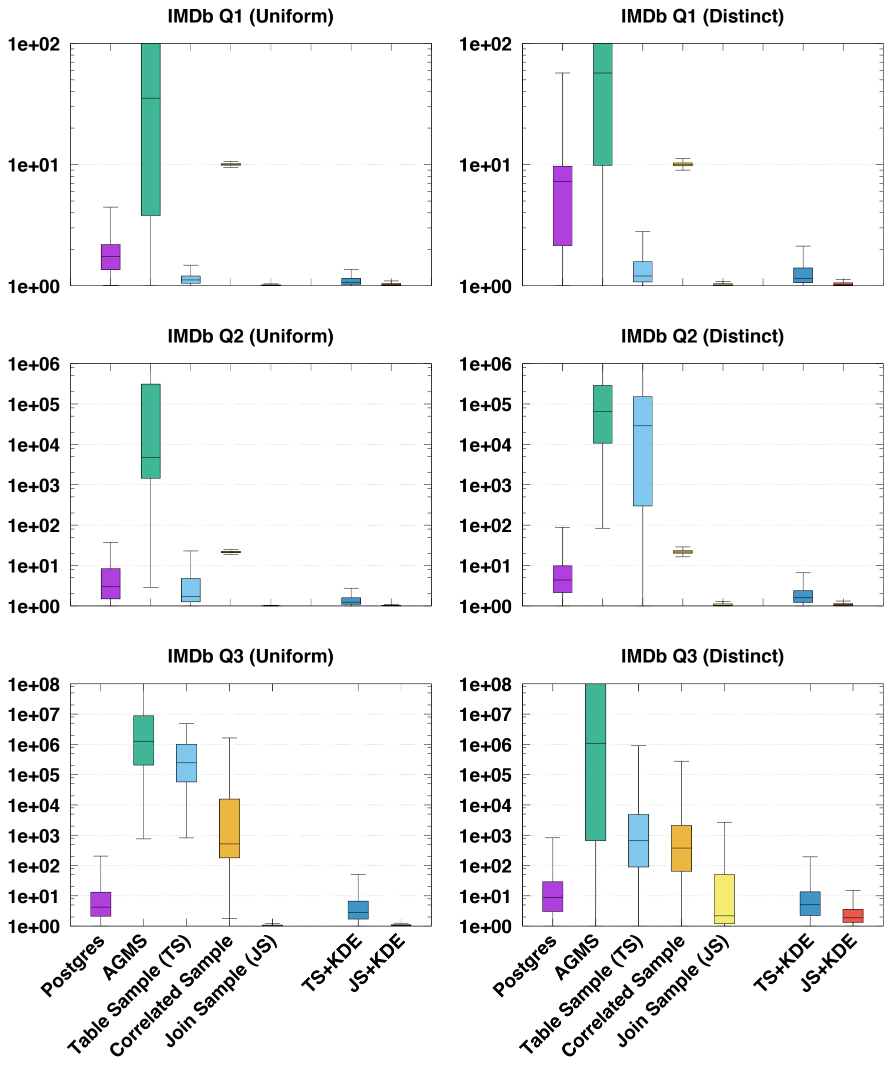


Figure 3.5: Estimation quality, IMDb dataset. The y-axis shows the multiplicative estimation error.

Discussion

Based on the results of our experiments, we can make the following general observations:

1. KDE-based join estimators generally perform better than the AGMS sketch and traditional estimators that rely on the independence assumption and 1D statistics.
2. KDE-based join estimators never perform significantly worse than their naïve sample evaluation counterparts or correlated sampling, but usually improve the estimates significantly.

3.7.3 Quality Impact of Model Size

In our second experiment, we investigate how the relationship between the different estimators changes with the model size. For this, we ran our previous experiments while increasing the sample ratio — and, accordingly, the other model sizes — by factors of two from 0.001 to 0.128. We report the geometric mean error for DMV Q1 (Uniform), as a representative for estimates on a single join, and DMV Q3 (Uniform), as a representative for multiple joins.

Figure 3.6 illustrates the results of this experiment for DMV Q1 with the uniform workload. There are a few noteworthy observations: First, KDE-based estimators are a significant improvement over naïve sample evaluation for small sampling fractions. This is clearly visible for TS+KDE: A sampling fraction of 0.001 is insufficient for naïve evaluation of base table samples, as the join between the two samples is likely to be empty causing estimation errors of three orders of magnitude and more. Adding a KDE model improves the estimation error by more than two orders of magnitude, which outperforms Postgres by a factor of two. While correlated samples tackle the same problem and bring substantial improvements over naïve base table samples for smaller sample sizes, KDE models are still clearly superior. Furthermore, we see JS+KDE bringing a 50% improvement over join sample evaluation for a sample size of 0.001.

Second, KDE-based estimators never perform significantly worse than Table Sample. While correlated samples bring improvements of an order of magnitude and more for small sample sizes, they converge slower to exact estimates. This causes an intersection point, after which Table Sample yields a smaller estimation error. This is consistent with our observations in Section 3.7.2, which showed that correlated samples are not always preferable to uniform table samples. As sample evaluation is in the parameter space of KDE-based estimators, their estimation error converges with their sample evaluation pendants for larger model sizes.

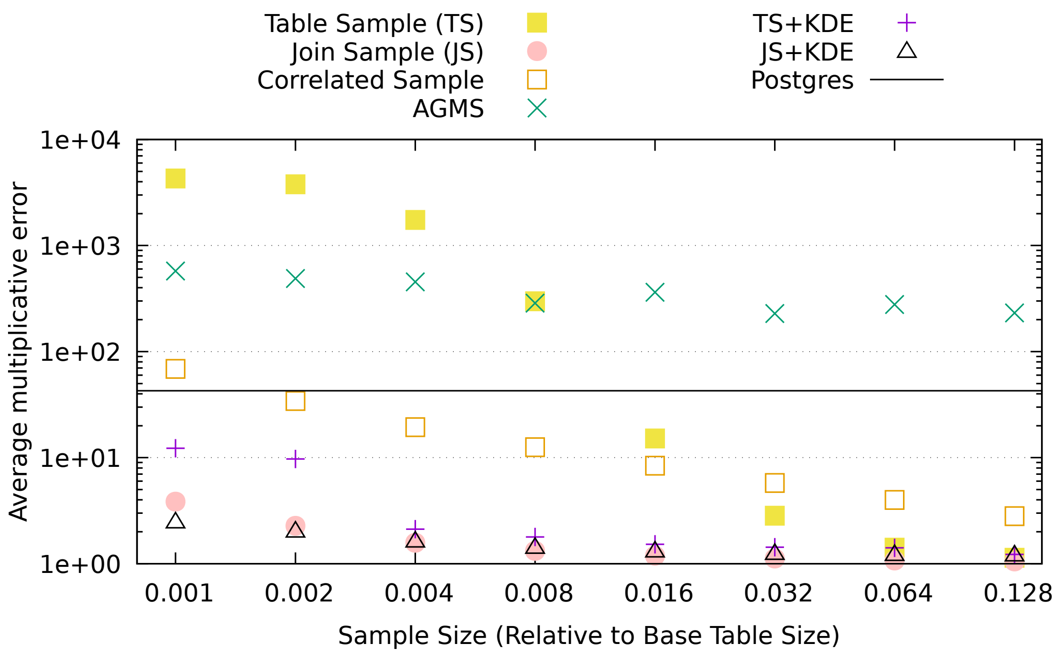


Figure 3.6: Estimation quality on DMV Q1 (Uniform) with growing sample sizes.

Figure 3.7 illustrates the results of this scaling experiment for DMV Q3 with the uniform workload, which adds two more joins to the query. The key observation is that larger sample sizes are required for the base table estimators to outperform Postgres by 10% or more. TS+KDE provides better estimates at sample size 0.008; correlated samples need twice as many points. A clear improvement for table samples is only visible at a sample size of 0.128. JS+KDE provides better estimation errors than join sample evaluation for sample sizes 0.001 to 0.004 by a factor between 1.5 and 2.

These experiments confirm that bandwidth-optimized KDE models can significantly improve the estimates computed from samples. Furthermore, the measured estimates were never significantly worse than the estimates provided by Postgres but are usually much better depending on the sample size and the workload.

3.7.4 Performance Evaluation

In our final series of experiments, we evaluated the runtime scalability of our estimators for increasing sample sizes.

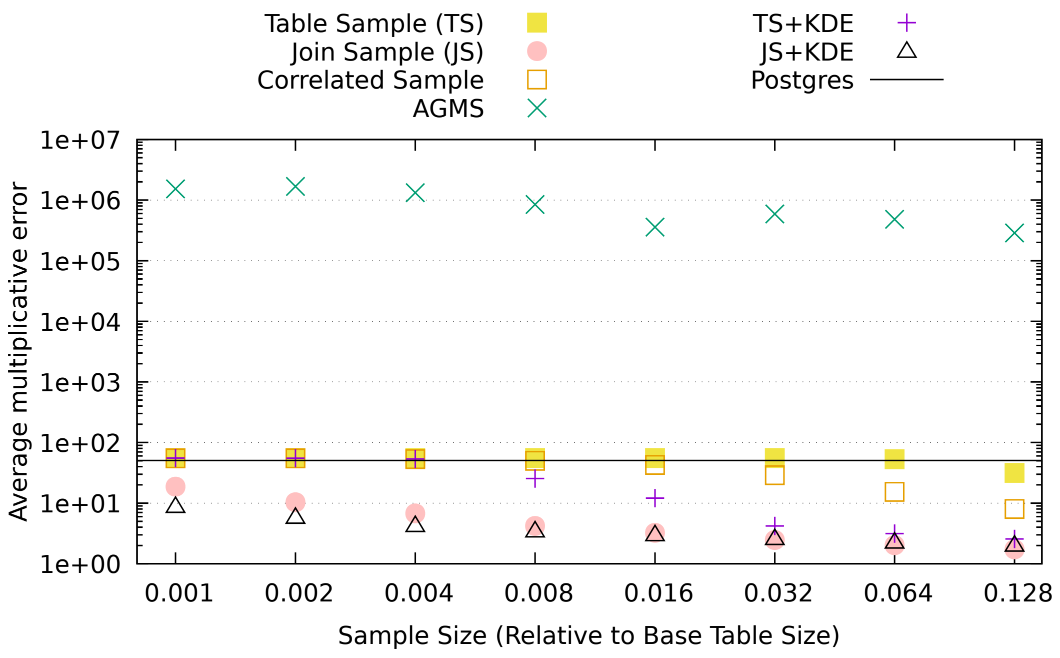


Figure 3.7: Estimation quality on DMV Q3 (Uniform) with growing sample sizes.

Setup

As demonstrated in the prior publication by Heimel et al. [70], KDE models are very well suited to be accelerated by graphics cards [70]. Accordingly, we implemented all estimators using custom OpenCL kernels and primitives as provided by the Boost.Compute framework[4]. Naïve sample evaluation on the join sample was implemented as a straightforward table scan, evaluation on the table samples was implemented by applying the local filter predicates on the sample, followed by performing a binary search join. Accordingly, both naïve estimators perform basically the same operations as our KDE estimators but with less computational overhead and the most aggressive pruning.

The experiment was conducted on a custom server with an Intel Xeon Gold 5115 CPU in two sockets, an NVIDIA V100 GPU accelerator, and 188~GB of DDR4 memory. The server was running Ubuntu Linux 22.04.1 with kernel 5.15.0. The accelerator was controlled by NVIDIAs 515.65.01 driver. If not stated otherwise, estimators were evaluated on the GPU.

We performed our experiments on the IMDb dataset, as it is the largest of our evaluated datasets. We repeated the previous experiment for two queries over the IMDb

[4]www.boost.org/libs/compute

dataset while reporting the average runtime in milliseconds instead of the estimation error. Furthermore, we compare JS+KDE and TS+KDE on a CPU and GPU to highlight the impact of GPU acceleration.

IMDB Q1 (Uniform)

Figure 3.8 shows the results for query IMDb Q1 using the uniform workload. The first observation is that the runtime for all estimators does not visibly increase until a sample size of 0.016. This is caused by the overheads introduced by the OpenCL framework, BoostCompute, and memory transfer dominating the runtime, which is roughly 0.7ms for table sample estimators, 0.1ms for join sample estimators and 0.1ms for AGMS.

Once the actual computation dominates execution times, we see an increase in the runtime for both AGMS (0.016) and JS+KDE (0.128). As join sample evaluation only requires computationally cheap comparisons and increment operations for every tuple, the framework overhead dominates throughout the entire experiment.

The runtime for Table Sample, TS+KDE, and Correlated Sample is close throughout the experiment. The TS+KDE is consistently slower than the other estimators. It is at most 60% worse than table or correlated sample, but it is less than 20% slower for most sample sizes. The performance not increasing linearly with the sample size shows the effectiveness of our pruning techniques: While computing the cross contribution naively would result in a quadratic increase, the growth behavior with pruning does not differ from Table Sample evaluation.

IMDB Q3 (Uniform)

To show the performance of our estimator for multiple joins, we repeated the experiment for IMDb Q3, which adds two joins and additional base table predicates. The results are shown in Figure 3.9. Again, we observe that for smaller sample sizes, the framework overhead dominates the execution time. However, it is above 1 ms for base table models, which is due to the additional number of involved samples. For AGMS and JS+KDE, the execution time increases linearly, which can be seen for sample size 0.032 and larger. For Join Sample, the overhead dominates for all sample sizes and the execution time barely increases.

In this experiment, TS+KDE shows a significant overhead over Table Sample and Correlated Sample: Correlated Sample and Table sample increase by a factor of 4.2 for Correlated Sample and by a factor of 1.8 from sample size 0.001 to 0.128. Thus, the increase in the runtime is not even linear. TS+KDE clearly diverges from the other

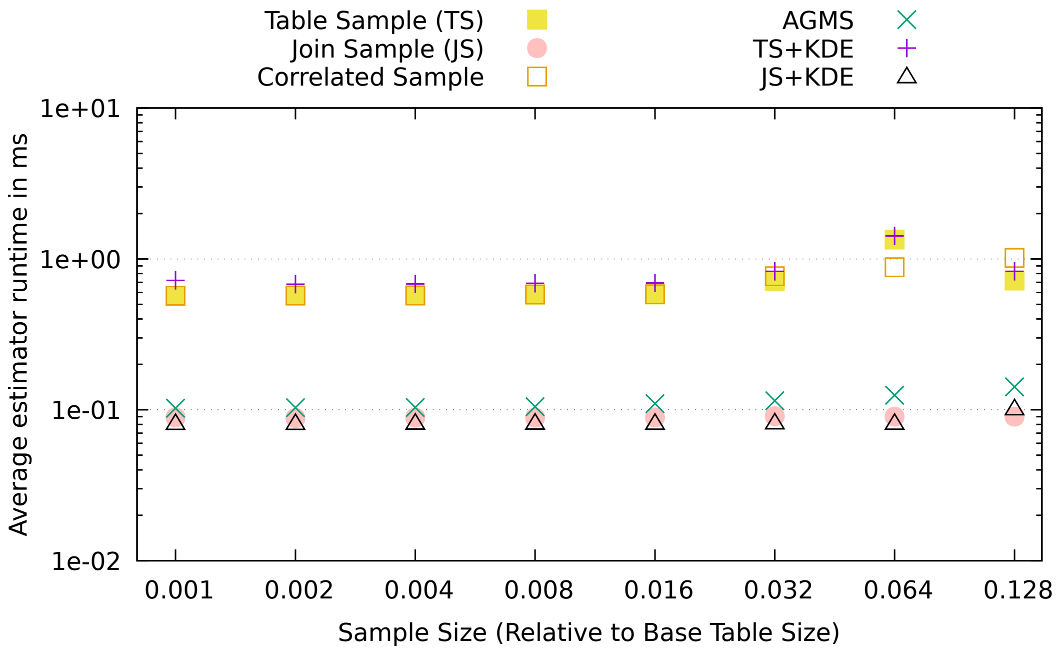


Figure 3.8: Estimation time on IMDb Q1 (Uniform) with growing sample sizes.

estimators showing a roughly quadratic growth. Between the smallest and the largest sample, the execution time increases by a factor of 24. Execution time is up to a factor of 20 larger when comparing TS+KDE to its sample evaluation counterpart.

While we see that the KDE overhead for large base table samples can be significant, our pruning techniques are still very effective for this query, as the runtime complexity without our pruning techniques would be quartic in the sample size.

Impact of GPU-Acceleration

Finally, we investigate the impact of GPU acceleration on the estimator runtime. Figure 3.10 compares the performance of our GPU and CPU implementation for IMDb Q1 (Uniform). We see that the estimator runtime for all implementations remains almost constant for sample sizes of up to 0.016. This is due to overhead introduced by OpenCL, BoostCompute, and kernel launches. Yet, GPU implementations consistently provide faster execution times by a factor of eight for TS+KDE and a factor of six for JS+KDE. For larger summary sizes and JS+KDE, estimator runtime increases linearly as the CPU as kernel evaluations dominate the execution time. On the GPU, computational overhead remains dominant, which widens the performance gap even further as sample sizes increase. For TS+KDE, we see execution time increase by up to a factor of 2 on both

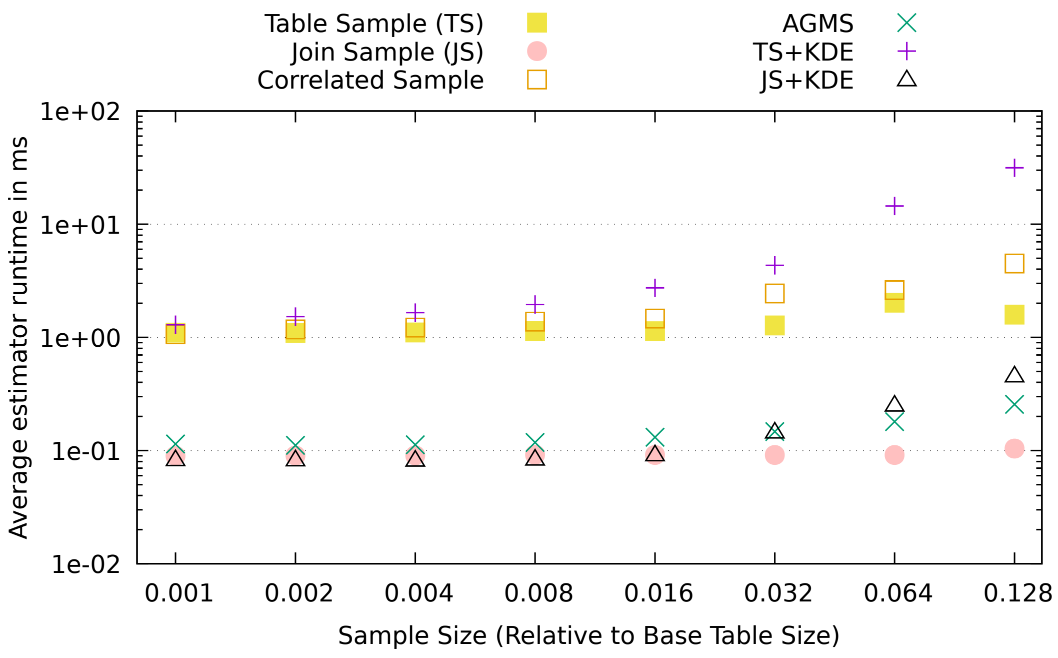


Figure 3.9: Estimation time on IMDb Q3 (Uniform) with growing sample sizes.

devices, while the performance gap is between a factor of six and ten.

Figure 3.11 shows the results for IMDb Q3 (Uniform). For JS+KDE, we see a similar behavior as before. The larger absolute sample sizes lead to both CPU and GPUs showing a linear performance increase for sample sizes starting at a relative sample size of 0.016, while the performance gap is between a factor of six and 27. For TS+KDE, we see an improvement by a factor of eight for small sample sizes, while the gap shrinks with increasing sample size. By sample size 0.128, the improvement has reduced to a factor of 4.6. This is explained by the more complex cross pruning algorithm: As GPU threads run in lock-step, skew in the number of tuples and search steps during cross pruning impairs efficiency. This does not manifest as strongly for CPU threads that operate independently.

Overall, we see that GPUs allow for larger model sizes within the same time budget. For small sample sizes, the smaller overhead of our GPU implementation allows us to compute estimates on less than 0.1 ms for JS+KDE and less than 1.1 ms for TS+KDE on average. Our CPU implementation can not meet these execution times. Based on the execution time improvement for the largest evaluated samples in IMDb Q3 Uniform, we can expect a factor of 27x larger samples asymptotically for JS+KDE. For TS+KDE, execution times clearly depend on the data distribution and query. Based on the results

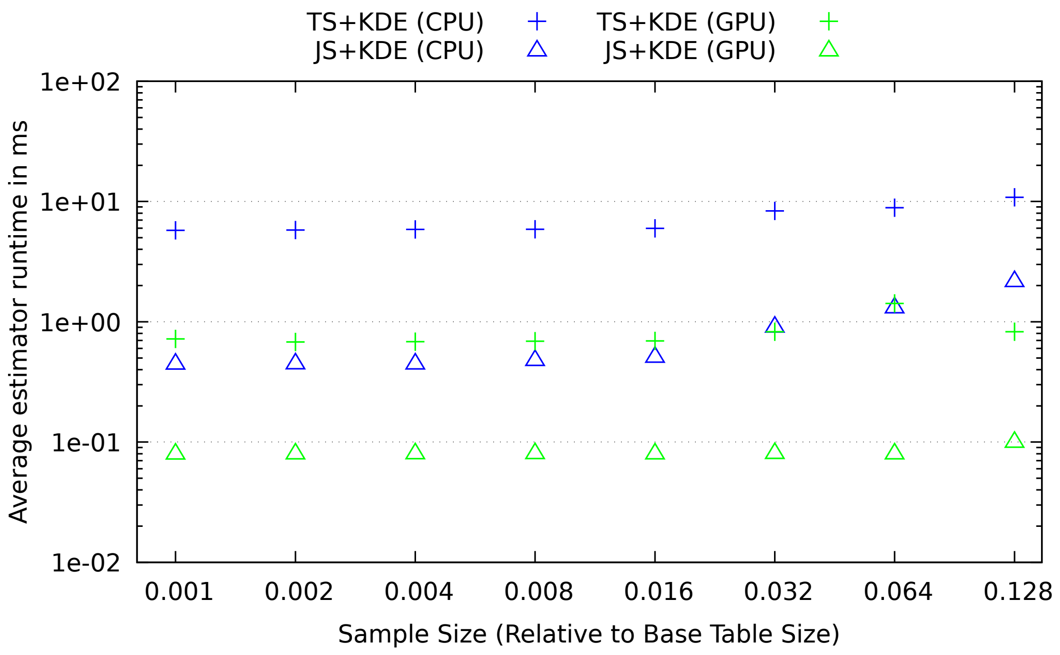


Figure 3.10: Estimation time for a CPU and GPU implementation on IMDb Q1 (Uniform) with growing sample sizes.

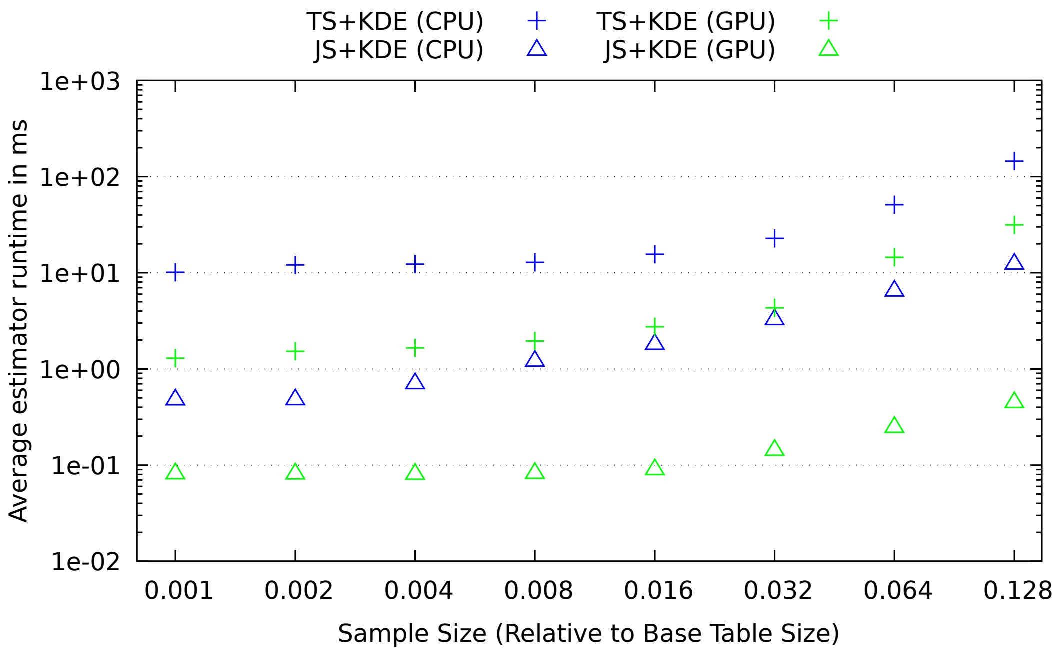


Figure 3.11: Estimation time for a CPU and GPU implementation on IMDb Q3 (Uniform) with growing sample sizes.

for IMDb Q3 (Uniform), we can expect at least a factor of four larger sample sizes. As we have shown in our previous experiment, such increases in the model size can result in substantial improvements in the multiplicative error.

3.8 Conclusion

In this chapter, we introduced a novel way to estimate join selectivities based on bandwidth-optimized Kernel Density Estimators. Existing models suffer from at least one of the following drawbacks: They (1) provide inaccurate estimates, (2) are expensive to construct, (3) are restricted to a single type of query, or (4) are expensive or impossible to maintain under changing data.

Our approach uses KDE models, which are constructed from base table or join samples, and provides an estimate to its underlying distribution. They apply smoothing to the sample distribution by placing probability density functions on all sample points, averaging them to compute the final estimate. The degree of smoothing is controlled by a hyper-parameter, the so-called bandwidth. Selecting this bandwidth parameter is essential for the estimation quality and can be done by performing numerical optimization over query feedback. KDE combines the flexibility and maintainability of a sample-based method, with the quality of state-of-the-art selectivity estimators.

We evaluated the quality of our approach using queries on both synthetic and real-world datasets. We found that KDEs provide significantly better join estimates than traditional methods in case one of the underlying assumptions is violated. Compared to naïve sample evaluation, our models can provide significantly improved results for relatively small sample sizes, while still converging to the same accuracy for larger samples. In practice, we suggest maintaining base table KDE models for all tables in a database as they usually provide better estimates for supported operators and data types (numeric attributes, dictionary encoded attributes). Joint KDE models can be added manually when additional accuracy is required for particular joins. Finally, we have confirmed that our generalization of KDE models to join selectivities strongly benefits from GPU-Acceleration as at least four times larger models can be evaluated in the same time budget.

Overall, this work strengthens the case for statistical coprocessing on GPUs to support query optimization in relational database systems.

4 Scotch: A Holistic Approach to FPGA-Accelerated Sketching

FPGAs have shown admirable performance for sketch construction in terms of throughput and energy consumption in prior work [30, 32, 130, 139, 153, 154]. However, developing and optimizing the RTL for sketch implementation is a time-intensive and cumbersome process conducted by an expert. In this chapter, we introduce the *Scotch* system to generate sketching RTL without expert involvement and to adapt the sketch size to the underlying FPGA and I/O requirements independent of the vendor and use case. The work shows that code generation and automatic optimization substantially lower the entry barrier to implementing FPGA-accelerated sketching. The generated accelerators deliver competitive performance compared to handwritten implementations and mostly outperform parallel software implementations in terms of energy efficiency and throughput.

This chapter is mainly based on our publication 'Scotch: Generating FPGA-Accelerators for Sketching at Line Rate' [89].

4.1 Introduction

Since analyses over sketch summaries shift the computational pressure from the analysis to the summary construction, maintaining the summaries at high throughput is critical. Implementations based on multi-core CPUs or GPUs have high energy consumption and often fail to deliver the bandwidth required to satisfy modern interconnects for network (100G Ethernet, Infiniband) and storage (PCIe 3.0+, SATA Express). Field-Programmable Gate Arrays (FPGAs) allow developers to construct custom hardware based on reconfigurable logic elements. Custom hardware allows for high degrees of parallelism, which enables data processing at line rate. These capabilities are a perfect match for the parallel computations over state commonly found in sketching algorithms.

However, implementing sketching algorithms on FPGAs is tedious. An FPGA expert is required to find an implementation that satisfies bandwidth constraints and resource limits while maximizing the sketch size for optimal accuracy. The expert has to make performance-critical design decisions, including memory architecture and pipelining of operations. Furthermore, maximizing the summary size requires time-consuming manual tuning. Previous research in the area focused on implementation strategies for individual sketches and use cases manually tailored to the FPGA [32, 130, 154].

In this work, we take a holistic perspective on FPGA-accelerated sketching by creating FPGA accelerators for an entire class of sketching algorithms without the need for explicit hardware description or manual tuning. We propose the *Scotch* framework that makes four main contributions:

1. Programming models to describe a variety of different sketching algorithms and ScotchDSL, a domain-specific language to implement user-defined functions for these models

2. A code generator producing a highly efficient hardware description based on ScotchDSL functions

3. An auto-tuning algorithm that maximizes the sketch size within the resources of the FPGA and target throughput

4. An extensive evaluation of our approach on various FPGAs that covers comparisons to CPU and GPU baselines in terms of throughput and energy-efficiency

In the following Section 4.2 we provide an overview of the Scotch system. We then introduce ScotchDSL and its programming model in Section 4.3. Section 4.4 explains the code generator and the generated hardware architecture. It is followed by extensions for data parallelism in Section 4.5 and a discussion of the limitations of the approach in Section 4.6. Finally, we introduce our auto-tuning algorithm in Section 4.7. Section 4.8 presents our experimental evaluation. Section 4.9 covers related work, and Section 4.10 concludes the chapter by summarizing our findings.

4.2 System Architecture

In this section, we provide an overview of Scotch and the accelerators generated. In Section 4.2.1, we first motivate and introduce the design requirements of Scotch. The following Section 4.2.2 shows the architecture of the accelerators generated by Scotch. Finally, Section 4.2.3 gives a high-level overview of the Scotch system.

4.2.1 Design Requirements

In the following, we describe the problems and highlight the design requirements that are the foundation of Scotch.

DR1: Lightweight Sketch Specification. While sketching is typically very concise in its mathematical definition, implementing it on an FPGA adds additional complexity. An FPGA expert is required since the developer needs hardware design knowledge to make architectural decisions. In particular, the expert has to pipeline computations and decide on a memory architecture. Scotch provides an intuitive programming model and domain-specific language to describe sketching. These descriptions are concise and close to their mathematical definition. Code generation replaces the tedious process of programming RTL for these functions. Efficient auxiliary components, such as memory, are generated automatically according to the requested sketch summary size without user involvement.

DR2: Automated Tuning. As a sketch summary's accuracy increases with its size, providing a large summary size is crucial for a sketching accelerator. However, finding a large sketch size within the FPGA's resources while meeting the operating frequency required by the interconnect is tedious. The developer has to vary the size and compile the accelerator by trial and error. This process requires an FPGA expert's intuition and is inconvenient given compile times in the order of hours. Scotch maximizes the sketch size for a given FPGA and interconnect without manual tuning. An auto-tuning algorithm systematically varies the summary size while being economical in the number of performed compilations.

DR3: Device and I/O Agnosticism. Various FPGAs and boards exist with different supported interconnects and target domains ranging from IoT applications to large-scale network processing. Implementations created for a particular setup are usually not easily portable to another. Scotch separates the implemented algorithm from the FPGA vendor, model, and board by encapsulating these details in an I/O module. RTL generation and tuning make no assumptions on the device or interconnect.

DR4: Constant Processing Rates. Compared to general-purpose processor architectures such as GPUs or CPUs, constructing custom hardware on FPGAs has a significant benefit: They can provide high throughput at a constant rate, which enables data processing at the full rate of the interconnect. However, this requires a careful design of all components implementing the sketching functionality. Scotch generates hardware that processes data at a constant rate. All components are scalable with the sketch size and fully pipelined, meaning that all components are divided into pipelined substages and process one set of input values per clock cycle. They do not require stalls or flushing pipelines due to data dependencies. For high-throughput interconnects, Scotch provides mechanisms to exploit data parallelism.

4.2.2 Accelerator Design

The accelerators generated consist of two components connected via a common interface:

Sketching Unit: Processes input values by storing and manipulating the sketch summary state. Furthermore, it exposes sketch summary state when requested via control signals. Scotch generates and optimizes the sketching unit.

I/O Controller: Implements off-chip communication via an interconnect, such as Ethernet or PCIe. It interprets signals arriving on the interconnect as input values or as requests to expose the sketch summary and drives the interface signals accordingly. The I/O controller has to be provided by an expert as, similar to drivers in operating systems, its implementation depends heavily on the used FPGA, board, and interconnect.

Separating sketching and I/O enables flexibility in terms of the interconnect (DR3).

4.2.3 Scotch

Scotch generates optimized hardware accelerators for a broad class of sketching algorithms. Users specify the sketching process in a convenient domain-specific language, while code generation and automated tuning replace the complicated and time-consuming RTL development and manual tuning of the summary size. Figure 4.1 illustrates the high-level system architecture.

A user implements sketching algorithms by providing user-defined first-order functions. They are arguments to higher-order functions supported by Scotch. User-defined functions are given in *ScotchDSL*, a domain-specific language that allows for

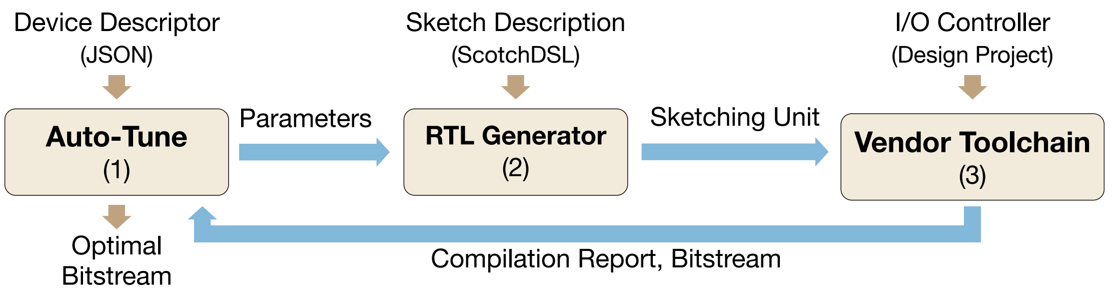


Figure 4.1: Scotch system architecture

an intuitive description of the sketching computations close to their mathematical formulation. ScotchDSL and the underlying programming model satisfy DR1 and are described in Section 4.3.

The *RTL generator* produces hardware description code for the entire sketching unit based on ScotchDSL functions. The generated RTL is fully pipelined, enabling high constant processing rates (DR4). The RTL generator and the generated hardware architecture are discussed in Section 4.4. A top-level project, which contains the I/O controller and constraints, instantiates and connects to the generated sketching unit yielding a complete design for the accelerator. Finally, the vendor-specific toolchain compiles the design, resulting in a bitstream to configure the FPGA and reports on resource utilization and timing.

Exploiting data parallelism is a common technique to achieve high throughput in parallel processor architectures. The RTL generator supports data parallelism by trading FPGA resources for higher maximum throughput (DR4). We discuss the underlying approaches in Section 4.5.

Scotch's *auto-tuning* algorithm maximizes the target sketch size within the resource limitations of the FPGA and clock rate constraints set by the I/O controller for interconnect used. Thus, it ensures high accuracy while creating a fully functional accelerator. The algorithm repeatedly parameterizes the RTL generator, compiles the project, and analyzes the resource consumption and timing reports in a feedback loop. The auto-tuning algorithm satisfies DR3 and is given in Section 4.7.

4.3 Sketch Specification

In this section, we introduce the sketch specification approach used in Scotch to satisfy DR1. In Section 4.3.1, we propose the *Select-Update model* that allows for a convenient description of sketching in terms of user-defined functions. It serves as the underlying

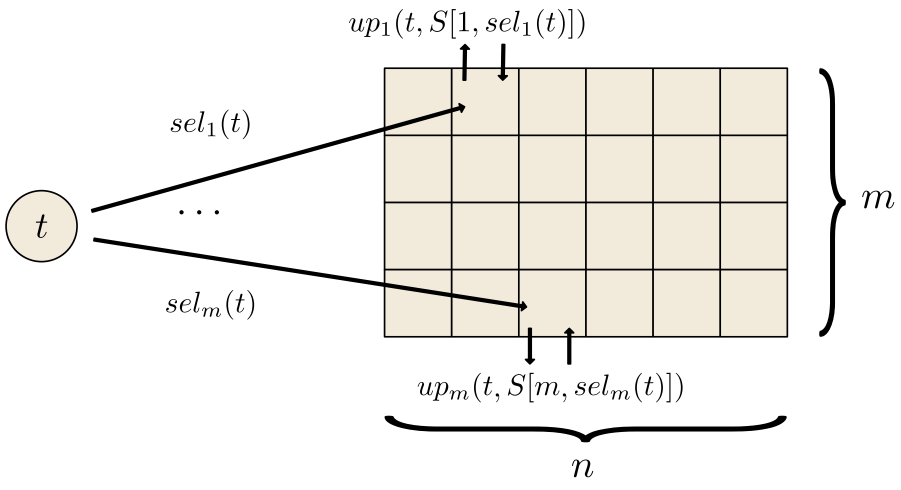


Figure 4.2: Select-Update model

programming model for Scotch. In Section 4.3.2, we introduce ScotchDSL, a domain-specific language allowing developers to specify these user-defined functions close to their mathematical formulation while being translatable to RTL.

4.3.1 Select-Update Model

The maintenance process of many popular sketching algorithms can be generalized as updating one entry per row of a matrix. Based on this observation, we propose the Select-Update model to describe sketching by specifying a *select function* that selects the entry in a row and an *update function* that determines the new value of the selected entry. Table 4.1 provides a non-exhaustive list of sketching algorithms fitting this model. The Select-Update model serves as a programming model for Scotch.

Formally, the Select-Update model defines an update to the sketch matrix as follows: A sketch matrix $S \in \mathbb{S}^{m \times n}$ is adapted for each observation $t \in \mathbb{T}$. For each row $i \in \{1 \dots m\}$, a selector function $sel_i : \mathbb{T} \to \{1, \dots, n\}$ determines an entry that is updated based on an update function $up_i : \mathbb{T} \times \mathbb{S} \to \mathbb{S}$. Formally, an update is defined as:

$$S\left[i, sel_i(t)\right] := up_i\left(t, S\left[i, sel_i(t)\right]\right), \quad i \in \{1 \dots m\} \tag{4.1}$$

The domains of the state $\mathbb{S}$ and value $\mathbb{T}$ are fixed-size bit sequences; their interpretation is left to the select and update function.

Select and update functions are either the same for all i or drawn from a family of functions based on a randomly drawn seed $\theta \in \Theta$, where the seed domain Θ is a fixed-size bit sequence as well. This allows for convenient definitions in a single function that takes the seed as an optionally used third argument. Thus, we specify $sel_i(t)$ and $up_i(t, s)$ by specifying $sel(t, \theta_i)$ and $up(t, s, \theta_i)$, respectively.

The Select-Update model includes sketching algorithms operating on a single column, such as AGMS [4] or MinHash [18]. In these cases, the specification of a select function is unnecessary, as there is only a single entry per row. This property allows for simplifications and optimizations, which we highlight throughout the chapter. We refer to these sketches as *column sketches*, while referring to the general case as *matrix sketches*. Sketches that operate on a single row are referred to as *row sketches*.

We provide the Select-Update model definitions of the sketches introduced in Section 2.1.2. AGMS is a column sketch; CM and FAGMS are matrix sketches.

Example 1 (CM): The CM sketch [39] we introduced in Section 2.1.2 requires a member of a family of two-wise independent hash functions computes the offset in each row. A common choice is *H3* [123], which computes an h-bit hash value for a k-bit key by using a random seed θ consisting of $h \cdot k$ bits.

$$sel(t, \theta) - 1 = \bigoplus_{j \in \{0 \ldots k-1\}} (t\,[j] \wedge \theta\,[j \cdot h + 1 \ldots (j+1) \cdot h]) \tag{4.2}$$

The subtraction in the first term accounts for one-based indexing. The operator $\oplus$ denotes a sequential bitwise XOR operation; the operator $\wedge$ denotes a bitwise logical AND between the j-th bit of t and j-th sequence of h bit in θ. The following update function denotes an increment to the selected state:

$$u(t, s, \theta) = s + 1 \tag{4.3}$$

Example 2 (AGMS): The AGMS sketch [4, 157] requires a member of a family of independent hash functions maping a k-bit key to $\{+1/-1\}$. The *EH3* family was found to be a good choice [50, 128]. The EH3 hash functions require a random seed θ consisting of $k + 1$ bits.

The function *eh*3 applies bitwise operations on the input value and the seed:

$$eh3(t, \theta) = h(t) \oplus \theta[k+1] \oplus \bigoplus_{r \in \{1 \ldots k\}} (\theta\,[r] \wedge t\,[r]) \tag{4.4}$$

$$h(t) = \bigoplus_{r \in \{1 .. \frac{k}{2}\}} t[2r-1] \vee t[2r] \tag{4.5}$$

Table 4.1: Sketching algorithms generalized by the Select-Update model and implementable in ScotchDSL

FM [53]	MinHash [18]	FAGMS [37]	CM [39]
AGMS [4]	HyperLogLog [52]	Bloom Filter [11]	Fast-Count [152]

The update function is then defined as:

$$u(t, s, \theta) = s + \begin{cases} 1 & \text{if } eh3(t, \theta) = 1 \\ -1 & \text{else} \end{cases} \tag{4.6}$$

Example 3 (FAGMS): Replacing the update function of the CM sketch with the AGMS update function yields FAGMS.

4.3.2 ScotchDSL

Scotch generates the sketching unit RTL based on user-defined select and update functions. A programmer provides these functions in ScotchDSL, a domain-specific language that describes the flow of computations from the input variables to the function result in terms of operations on bit vectors of arbitrary size. It is sufficiently expressive to implement the sketching behavior while ensuring that the computations translate to efficient hardware.

An algorithm implementation in ScotchDSL consists of an implementation of the user-defined functions and a descriptor file. The descriptor file contains the number of bits required for the state, input value, seeds, and auxiliary variables. ScotchDSL function implementations consist of consecutive variable assignments that set the value of a variable to the result of the expression on its right-hand side. Functions end with an assignment to the output variable.

ScotchDSL supports the following operations in the expressions:

1. Selecting an individual bit or a range of bits from a bit vector
2. Bitwise logical operations and comparisons
3. Signed and unsigned arithmetic operations and comparisons
4. Auxiliary functions that take variables as an argument and return a bit vector.

Besides simple assignments, we support conditional assignments and for-loops. For-loops have a fixed iteration range and replace repetitive assignments.

Operating on bit vectors allows the specification of algorithms closer to their mathematical definition (DR1). For example, the EH3 hash function given in Equation 4.4 requires computations on a 33-bit seed, which complicates an implementation in programming languages such as C/C++, as integer types are provided at a fixed granularity of bytes. ScotchDSL functions usually consist of a few lines and allow for quick customizations such as changing the state size or adding a frequency to an update.

As hardware definition languages operate on bit vectors in the same way, all expressions map to equivalent expressions in the target language VHDL. By providing a restricted set of operations, we ensure that the function code maps to pipelined RTL that is synthesizable on an FPGA (Section 4.4.2). Constructs that prevent a fully-pipelined design or are not synthesizable to FPGA hardware are not supported (e.g., data-dependent loops).

The ScotchDSL syntax borrows from VHDL. In the following, we provide ScotchDSL implementations for the previously introduced algorithms and highlight the language constructs. The descriptor files are regular JSON files and are omitted for the sake of space. Note that indexes in ScotchDSL are zero-based.

Example 1 (CM, Select): Listing 4.1 shows the implementation of the CM/H3 select function given in Equation 4.2. The code computes a 32-bit hash value (*offset*) from a 32-bit observation (*t*) based on a $32 \cdot 32 = 1024$-bit seed (*seed*).

Line 1 shows the regular assignment of a vector expression to a variable. It computes the first iteration of the sequential XOR in Equation 4.2. The expression *seed(31 downto 0)* selects the first 32 bits of the seed, and the & operator represents a bitwise AND. The built-in auxiliary function *expand(t(0), 32)* returns a bit vector consisting of 32 bits all set to the zeroth Bit of *t*. The output of the expression is stored in the auxiliary variable *x*. Lines 2-4 show a for-loop. It iterates from 1 to 30 using the variable *i*. The loop body computes the sequential XOR up to the *i*-th bit and seed by applying the XOR operator ˆ

```
1 x <= expand(t(0),32) & seed(31 downto 0) ;
2 for i in 1 to 30 {
3     x <= (expand(t(i),32) & seed((i+1)*32−1 downto i*32)) ^ x;
4 }
5 offset <= (expand(t(31),32) & seed(32*32−1 downto 31*32)) ^ x ;
```

Listing 4.1: Select function for CM with H3

```
1 p <= parity(seed(31 downto 0) & t);
2 h <= parity((t(30 downto 0) | t(31 downto 1))
3                     & '1010101010101010101010101010101');
4 outstate <= p ^ h ^ seed(32) = '1' ?
5                     signed(state)+1 : signed(state)−1;
```

Listing 4.2: Update function for AGMS with EH3

to the last value of x. Line 5 computes the last iteration of the sequential XOR and stores the result in the output variable *offset*.

Example 2 (AGMS, Update): Listing 4.2 shows the implementation of the update function for AGMS with a 32-bit input value and state. Line 1 computes the result of the sequential XOR operation in Equation 4.4. It computes the bitwise AND operation between the first 32 bits of the input variable *seed* and the input value t. Finally, it computes the full sequential XOR by using the *parity* auxiliary function provided by Scotch. Similarly, Line 2-3 computes the non-linear function h given in Equation 4.5. It computes a bitwise OR between the first and last 31 bits of the input value v. The parity function computes the sequential XOR; the bit vector literal in Line 3 ensures that only disjoint pairs of bits contribute to the parity. Finally, Lines 4-5 compute a $+1/-1$ update as shown in Equation 4.6 by using a conditional assignment of the form *var <= condition ? expr : expr*. In the condition, we compute the full result of the function $eh3$ given in Equation 4.4 and check whether the result is equal to the bit vector literal *'1'*. If the condition holds, the current state is incremented; otherwise, it is decremented. The built-in function *signed* assigns signed integer semantics to a bit vector for arithmetic.

4.4 RTL Generator

In this section, we introduce our approach to RTL generation for the sketching unit. In Section 4.4.1, we provide an overview of the RTL generator and the sketching unit architecture. Section 4.4.2 describes the translation of ScotchDSL functions to function units that perform all algorithm-specific computations. A pipelined RAM holds the summary state. We explain its architecture in Section 4.4.3. Compute units contain the

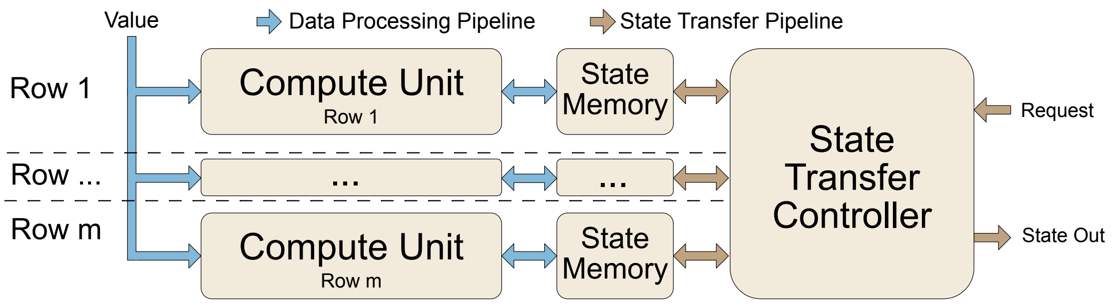


Figure 4.3: Sketching unit architecture

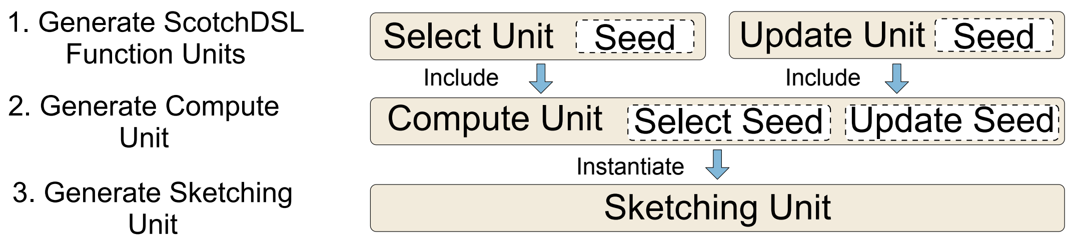


Figure 4.4: RTL generation process

ScotchDSL function units and perform all auxiliary operations. We explain the compute unit architecture and its components in Section 4.4.4. The state transfer controller retrieves and exposes the sketch state. We describe its architecture in Section 4.4.5.

4.4.1 Overview

The RTL generator creates a VHDL hardware description for the sketching unit based on ScotchDSL functions. The number of rows and columns for the sketch are input parameters and varied by Scotch's auto-tune algorithm. According to the desired state matrix shape, the RTL generator instantiates and parameterizes all components in the sketching unit. We provide an overview of the sketching unit architecture and then outline the RTL generation process.

Figure 4.3 shows the top-level architecture of the sketching unit. The sketching unit performs two tasks: First, it adjusts the summary state according to input values and, second, exposes it to the I/O controller. Each row of the sketch is represented by a dedicated compute unit and state memory, which operate independently and in parallel. The compute unit processes one input value per clock cycle and initiates read and write operations on the state memory. The state transfer controller exposes the sketch state when triggered by an outside request. It connects to the state memory of every row and dispatches read requests.

The RTL generator creates the sketching unit's hardware description in a three-step process shown in Figure 4.4. First, the generator translates ScotchDSL to function units. They implement the computation of the select and update functions while leaving the random seed as an open parameter. Second, it generates a compute unit by adding all auxiliary components. Third, the RTL generator creates the sketching unit. It instantiates a compute unit and state memory, sets the seeds, and adds the state transfer controller.

The sketching unit is fully pipelined to achieve high operating frequencies. The generator adjusts the interfaces and internals automatically according to the size of the state and input value of ScotchDSL functions. In the following, we detail the architecture and generation of the individual components. We assume sketching units consume one input value per clock cycle and discuss data parallelism in Section 4.5

4.4.2 ScotchDSL Function Units

ScotchDSL function units implement the sketch-specific computations in hardware: Select function units compute a row offset from the input value. Update function units compute the new state from the input value and previous state. The RTL generator implements them based on the provided ScotchDSL functions, for which we solve two problems: First, we have to translate the imperative ScotchDSL user-defined functions to pipelined RTL. Second, we have to ensure that the generated RTL does not introduce data hazards inside the update function unit.

The general translation mechanism consists of three steps:

Step 1: Abstract Syntax Tree (AST). The RTL generator parses the input function file and creates an AST.

Step 2: Dependency Graph. The RTL generator transforms the tree into a dependency graph that contains a node for every function input variable and assignment. It unrolls loops in the process. When the statement node B directly depends on the result of a node A, a directed edge from a node A to node B is inserted. The resulting dependency graph represents the flow of computations from input variables to the output variable.

Step 3: Function Unit. The RTL generator translates the assignment graph to RTL for the function unit. Each assignment node results in a synchronous component that computes the result and buffers the output. Edges between assignment nodes create connections in the top-level function unit. If necessary, the RTL generator adds buffers to ensure intermediates for the same input value arrive at the same clock cycle.

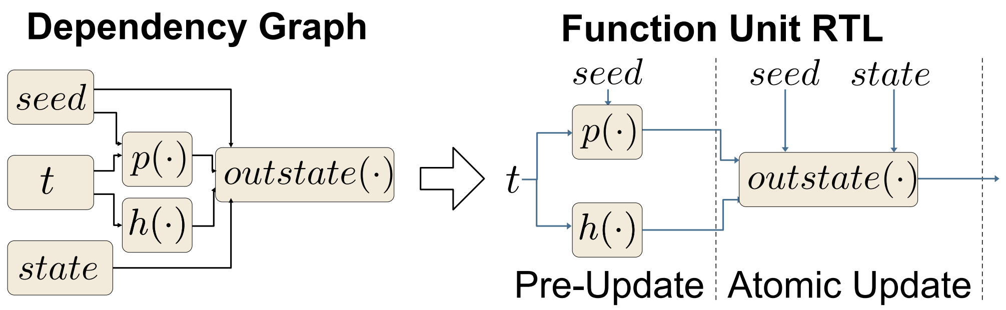


Figure 4.5: Dependency graph and function unit RTL generated for the AGMS update function (Listing 4.2)

The resulting function unit computes the ScotchDSL function in a pipelined fashion based on the input value and seed. As our architecture instantiates one dedicated compute unit per row, seeds are constants. Figure 4.5 shows the dependency graph and the function unit for the AGMS update function given in Listing 4.2.

While the general translation mechanism is sufficient for select functions, update functions also depend on the previous state. As the update function computes new state values in every clock cycle, data hazards occur if a later update overwrites a state used for a computation in the update function pipeline. Resolving these data hazards requires stalling or flushing the pipeline, which conflicts with DR4. Instead, the RTL generator prevents data hazards by ensuring only the output value computation accesses the previous state. If the function code violates this condition, the RTL generator inlines the computations for assignments required by the output value computation until the condition holds. We refer to the single computation of the new state as the *atomic update* while calling the rest of the update function pipeline the *pre-update*.

4.4.3 State Memory

The state memory holds each row's state entries in a BRAM-based pipelined random access memory architecture. A modern FPGA contains hundreds to thousands of BRAM blocks, each providing dense random memory in the order of kbits with configurable width and depth. BRAM blocks are dual-ported and process exactly one read and write operation per clock cycle to independent offsets when operating in *simple dual-port mode*. Maintaining the sketch size for large rows necessitates combining multiple BRAM blocks to provide sufficient memory depth.

While synthesis tools can automatically construct deeper RAM by combining k BRAM blocks, they naively multiplex and demultiplex reads and writes. This is feasible

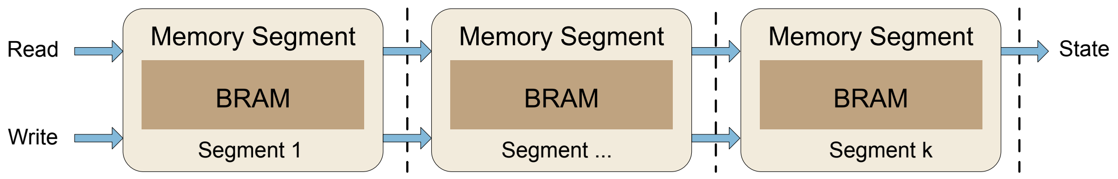


Figure 4.6: Pipelined memory architecture

for small values of k. However, prior work confirmed that this approach scales poorly compared to a pipelined memory architecture [154].

Figure 4.6 shows the pipelined memory architecture. It consists of k segments, where each segment contains a BRAM block and is exclusively responsible for a part of the address range. Read and write requests to an address are served by their corresponding memory segment. All other segments forward the request. Memory segments buffer requests before forwarding them to the next segment, creating a pipeline that consumes one read and write request per clock cycle with a latency of k clock cycles and a throughput of one read and write request per clock cycle. The width of the memory is the state size. The segment depth is a parameter to the code generator. Scotch sets it according to the depth supported by the target device's BRAM elements for the given state width.

The RTL generator simplifies the memory architecture for column sketches to a single register since random memory access is not required.

4.4.4 Compute Unit

Compute units process the input values for one row of the sketch by evaluating the select and update function and updating the per-row state according to the ScotchDSL function units. .

Overview

The compute units are pipelined and consist of several substages. Primarily, they consist of stages for the select and update function evaluation and stages accessing state memory. Furthermore, the RTL generator adds auxiliary stages to truncate the select function's output value and to prevent data hazards. Figure 4.7 shows the compute unit architecture with all substages. Note that the select, pre-update, data forwarding unit (DFU), and memory stages consist of several substages.

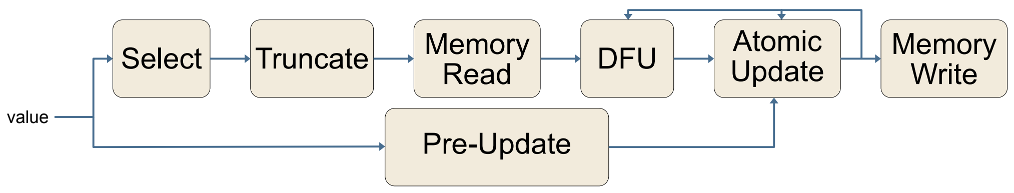


Figure 4.7: Compute unit architecture

Select: The select function unit computes the output of the select function given in ScotchDSL. As Scotch varies the number of columns in the auto-tune algorithm, we assume the select function provides sufficiently large offsets and truncate them to the required range in the truncate stage.

Truncate: We truncate the offset provided by the select function to the range $[0, n-1]$, n being the number of columns in the sketch.

Memory Read: The state memory retrieves the state for the previously computed offset.

DFU: As the compute unit consists of several substages with multiple clock cycles of latency, a state value read from memory for a particular offset can be overwritten by updates further down the pipeline. To prevent data hazards from causing lost updates, a data forwarding unit (DFU) tracks recent updates and ensures that only the most recent state values enter the atomic update stage. Section 4.4.4 introduces our novel fully-pipelined data forwarding unit architecture.

Pre-Update: A concurrent stage computes the inputs to the atomic update stage, as these computations do not depend on the state.

Atomic Update: The atomic update is computed based on the intermediates from the pre-update stage and the most recent state value arriving from the DFU. As the DFU can not see the very last update, the atomic update stage tracks its last computed state and uses it in case of two consecutive updates to the same offset.

Memory Write: The state memory stores the previously computed new state.

The RTL generator omits the select, truncate, and DFU stages as an optimization for column sketches since no random memory access is required.

Data Forwarding Unit (DFU)

The DFU resolves data hazards caused by the pipelined memory architecture. Data hazards occur when a state read from memory is altered by an update further down the pipeline. The DFU delays the computation of the atomic update to replace outdated states with the most recent value.

Figure 4.8 shows the architecture of our DFU. We first explain the key idea of our DFU and then introduce the optimized architecture used by the RTL generator.

Key Idea: Our DFU tracks the state-offset pairs leaving the atomic update stage in a shift register of size l. It compares state-offset pairs from the memory read stage to the shift register values in l stages. Each stage $i \in \{1 \ldots l-1\}$ compares the incoming state to the i-th and $(i+1)$-th least recent entry in the shift register. If one or both entries from the shift register coincide with the input offset, the most recent state in the shift register entries is passed to the next stage instead of the input state (3-way Compare-Forward). The last stage l only performs a single comparison with the most recent entry in the shift register (2-way Compare-Forward). This approach ensures that, when leaving the DFU, a state-offset pair has observed all updates caused by its second to $2l$-th successors.

Optimization: As 3-way Compare-Forwards are complex and reduce the scalability of the DFU, Scotch avoids them by splitting the DFU stages into three parallel pipelines. The i-th stage of the upper pipeline compares the input value to the i-th value in the shift register, while the lower stage compares the input value to the $(i+1)$-th value. This separation allows us to use simple 2-way Compare-Forward logic. However, the DFU has to track which pipeline carries the most recent state. If there is a match in the lower or upper pipeline exclusively, this pipeline carries the most recent state. If there is a match in both, the $(i+1)$-th value takes priority as it is more recent. In case there was no match, the previous priority remains valid. We perform these computations in a separate priority resolution pipeline with access to the results of the previous stage's comparisons. The final stage forwards the most recent state from either the l-th position in the shift register, the upper pipeline, or the lower pipeline. This requires a 2-way Forward based on the priority and an additional 2-way Compare-Forward.

The size l of the DFU that prevents all data hazards depends on the number of segments in the state memory k. There are a total of $k+l$ successors in the pipeline that may cause data hazards. The DFU and atomic update track the next $2l$ updates. Thus, picking $l = k$ results in a minimal DFU that prevents all data hazards.

4.4.5 State Transfer Controller

The state transfer controller exposes the state of the sketch summary to the I/O controller. It connects to the state memory of all rows and sequentially reads all $m \cdot n$ state values.

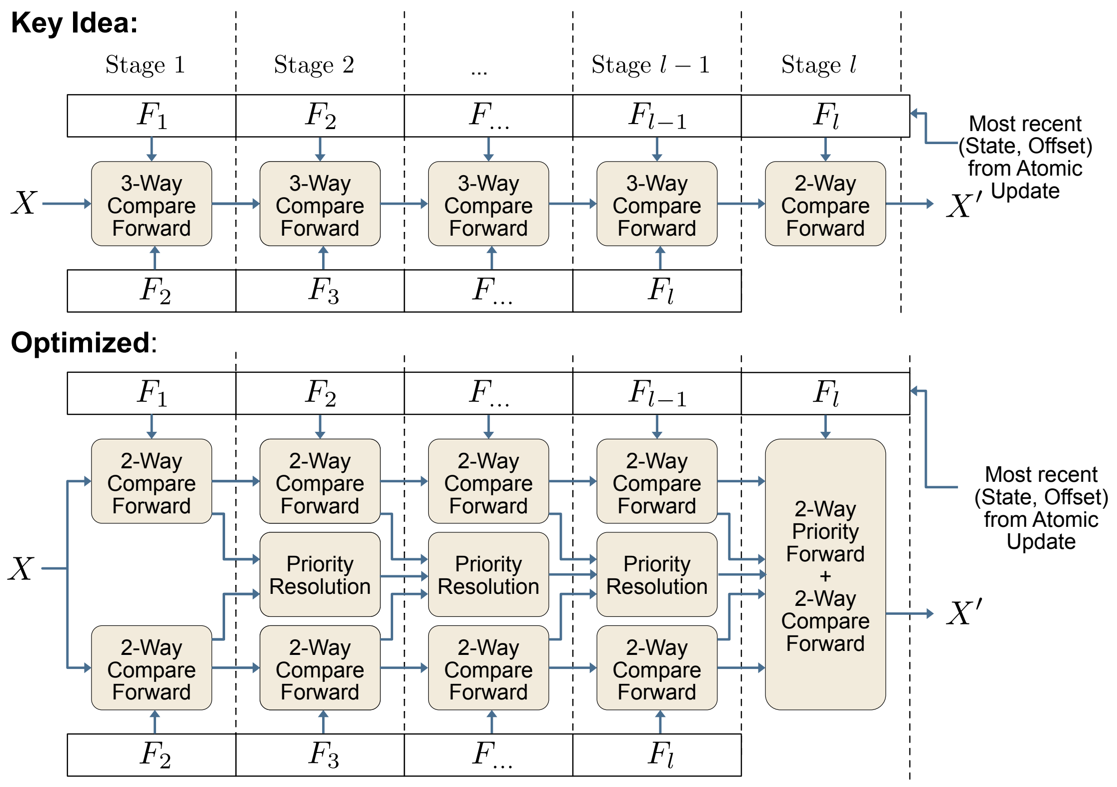


Figure 4.8: Data forwarding unit architecture

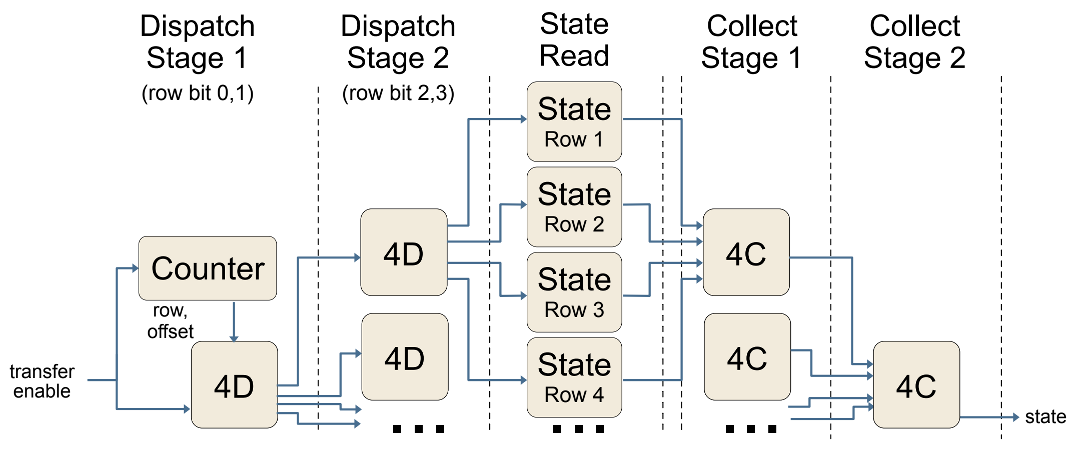


Figure 4.9: State transfer controller architecture

As Scotch aims to maximize the sketch size, the state transfer controller must scale with the number of rows. Particularly column sketches require state transfer controllers with a high number of connections.

Figure 4.9 shows the architecture of our transfer controller. A dispatch unit issues read and write requests to the per-row state memory, while a collect unit routes the result of the request to the output signals. Dispatch and collect units both follow a tree-shaped structure where each tree level is a pipeline stage. The pipelined tree structure prevents drops in the operating frequencies due to high fan-out in the dispatch unit and high fan-in in the collect unit by distributing the logic over several stages.

The dispatch unit maintains a counter for rows and offsets. The counters adjust after every request. When the I/O controller requests the next state value, several stages of 4-way dispatch logic (4D) route the selected offset to the selected row's state memory. As soon as the state memory serves the read request, 4-way collect (4C) logic routes the state value to the output signals.

As the per-row state memory has only one read port, value processing and state transfer are mutually exclusive. Thus, state transfer must wait until all updates are written to state memory.

4.5 Data Parallelism

The sketching unit described in the previous sections consumes one input element per clock cycle. However, this is insufficient to satisfy high-bandwidth interconnects such as 100G Ethernet. I/O modules interfacing such interconnects have to forward multiple input elements per clock cycle as the operating frequency of state-of-the-art FPGAs is limited to hundreds of Megahertz. Thus, data-parallel sketching units are required. This section describes the two mechanisms Scotch uses to provide data parallelism and satisfy DR4 for high-bandwidth interconnects: Section 4.5.1 introduces a general mechanism that exploits data parallelism by replicating components of the sketching unit. Section 4.5.2 introduces the *merged updates* for column sketches, which explicitly incorporates data parallelism to provide an improved resource utilization.

4.5.1 Replication

The most intuitive approach to data-parallel sketching with d simultaneous input values is maintaining d replicas of the sketch in parallel. The d separate sketches can be evaluated separately or merged into a single summary by an application. While the approach is simple, it also comes with approximately d times higher ELU and BRAM consumption.

If a data-parallel sketching unit with a replication factor of d is requested, Scotch generates d replicas of each compute unit and state memory. The components belonging to the same replica connect to the same input value pipeline. The state transfer controller is shared among replicas to save resources. The I/O controller may also instantiate the sketching unit several times to maintain entirely independent replicas and allow symmetric throughput for data processing and state transfer.

4.5.2 Merged Updates for Column Sketches

In the case of row sketches, we can implement data parallelism more efficiently as there is no random access required for each of the d input values. Instead of interpreting each of the d input values by a dedicated replica, we interpret them as one single input value and apply the update for d inputs simultaneously in the update function. As all sketches supported by Scotch are mergeable, this can be achieved by merging the updates of the d input values in a pipelined binary reduction and finally merging with the current state of the sketch. For example, in AGMS, we first accumulate all $+1/-1$ updates in a binary tree of adders, before adding the aggregate update to the current state of the sketch. As the sketch state is not replicated, the number of state registers and the state transfer controller remain independent of d. If the updates to merge are smaller than the overall state (e.g., the +1/-1 updates in AGMS), merging requires less logic than d independent updates in a replicated setup.

While this approach is more efficient in terms of resource consumption, it also requires an update function tailored towards the specific value of d and explicitly implementing merging in ScotchDSL. In Section 5.3, we establish a generalized framework that allows for creating equivalent merging and update logic for arbitrary *d* by specifying sketch updates in terms of user-defined map and reduce functions.

4.6 Discussion

The RTL generator relies on the structure implied by the Select-Update and Map-Apply model to generate fully-pipelined RTL (DR4). However, the underlying models and ScotchDSL also impose limitations on the supported algorithms. In the following, we will discuss these limitations.

Models: The supported models fix the memory access pattern for all supported algorithms, and the per-row memory is the only mechanism to store state. Each input value results in exactly one read and write operation to the state memory. This memory access pattern allows us to use a pipelined memory architecture that handles one read and write operation per clock cycle and resolves data hazards. Lifting this restriction would require us to stall the pipeline during conflicting memory operations and, thus, violate DR4. In particular, the supported models exclude streaming algorithms that require keeping a sorted list of values or complex data structures, such as Space-Saving [108] or Exponential CM [118]. Furthermore, algorithms that require an evaluation of the sketch to perform an update are not supported (e.g., CM-CU [60]).

ScotchDSL: ScotchDSL prevents for-loops with runtime dependent conditions. Supporting this would require us to build hardware that stalls while the loop iterates and, thus, violate DR4. Furthermore, ScotchDSL does not support floating point interpretations of bit vectors. Hardware support for floating-point operations is highly dependent on the device and requires vendor-specific modules that are hard to parameterize automatically. However, this would allow for floating-point state in already supported sketches and enable new sketches, such as Quantile sketches [55] or DDSketch [105]. Thus, we consider this an interesting direction for future work.

There is a well-known trade-off between flexibility, throughput, and resource consumption in hardware development [151]. In Scotch, the supported models and ScotchDSL provide enough freedom to implement popular sketching algorithms while limiting the flexibility to preserve DR4.

4.7 Automated Tuning

Scotch uses an auto-tuning algorithm to maximize the sketch size within the provided clock frequency constraints and resource limitations (DR2). Hence, it maximizes the accuracy of the sketch. In a nutshell, the algorithm performs an initial compilation, projects the maximum possible sketch size within the FPGA's resources, and performs a binary search to find the maximum sketch size still generating functional hardware.

Before tuning, the user fixes one of the matrix dimensions while the other is subject to optimization. Row and column sketches inherently fix one dimension. Matrix sketches usually have error guarantees in the form of an interval around the true value. The number of rows m determines the probability of the estimate falling into the interval; the number of columns n determines the interval's size [37, 39]. While the algorithm can optimize either dimension, a user will likely set the probability by fixing m based on the application and let Scotch minimize the interval by maximizing n.

We base our algorithm on the following conservative assumptions:

A1: Linear Resource Consumption. We assume ELU and BRAM consumption increases asymptotically linearly with the number of rows or columns. We use A1 to compute an upper bound for the optimization parameter. We base the assumption on the following observation: Increasing the number of rows results in a linear increase in the number of compute units and state memory components. The number of 4-dispatch, 4-collect units, and row counter bits in the State Transfer Controller grow logarithmically. Increasing the number of columns leads to a linear increase in memory segments and DFU stages. The number of bits required for offset registers and the number of LUTs for logic operating on them (e.g., comparisons) grow logarithmically.

A2: Global Optimum. We assume that there is a parameter b such that all parameters $x \leq b$ provide a functioning accelerator while all $x > b$ will lead to the optimization either failing due to timing or lack of resources. A2 justifies using a binary search. We base the assumption on the following intuition: As established in Assumption 1, resource consumption increases monotonically with the optimized parameter. Thus, FPGA resources will eventually exceed. Before this is the case, placement and routing by the toolchain, while satisfying timing, gets increasingly challenging and eventually impossible. In particular, the maximum operating frequency for the sketching unit decreases monotonically, which is the prevailing cause of timing failures.

Our auto-tuning algorithm consists of two steps:

Step 1: Initialization. The algorithm calls the RTL generator for an initial parameter r and compiles the accelerator using the vendor toolchain. Scotch estimates a parameter that exceeds 100% resource utilization and serves as a potential upper bound u. If the compilation for r has been successful, we define the initial search interval as $[r, u)$. If the compilation was unsuccessful due to timing or resources, the interval is $[0, min(d, u))$.

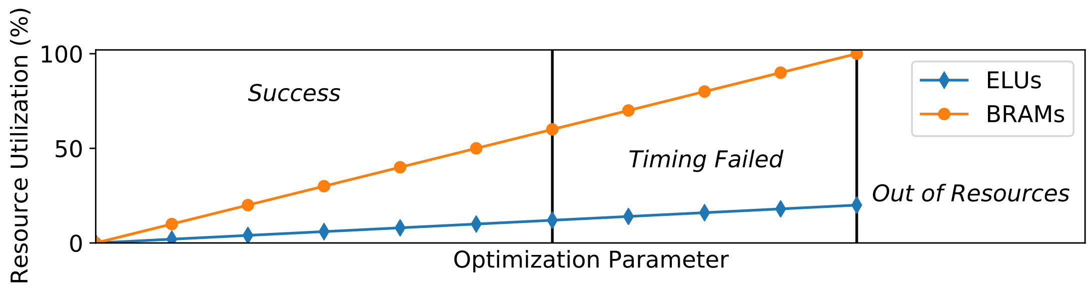


Figure 4.10: Auto-tune optimization space. BRAM and ELU consumption increase linearly with the optimization parameter. With an increasing optimization parameter, compilations are first successful, then fail due to timing, and finally fail due to a lack of resources.

Step 2: Binary Search. The algorithm performs a binary search in the interval. It repeatedly compiles the accelerator and checks whether the compilation was successful. As the algorithm converges, the lower bound either contains the maximum parameter with a successful compilation or is zero if no such parameter exists.

In Scotch, we extend the basic auto-tune algorithm by following adaptations: (1) A user may choose a relative difference between the upper and lower bound to speed up convergence. (2) The vendor toolchains use randomized algorithms for placement and routing. We account for their variance by trying five initial seeds for Intel Quartus Prime before considering a parameter as failed due to timing. For Xilinx Vivado, we vary implementation strategies for the same effect. (3) When BRAM is depleted, vendor toolchains try to implement the remaining memory segments less efficiently using ELUs. While only feasible if the fixed parameter is small, we double the estimated upper bound for row and matrix sketches to account for this edge case.

4.8 Evaluation

In this section, we evaluate the RTL generator and autotune algorithm. The Scotch system, algorithm implementations, and baselines are available in our public repository. [1]

4.8.1 Experimental Setup

We implement two column sketches ($n = 1$), two matrix sketches, and two row sketches ($m = 1$) as shown in Table 4.2. We order the three sketch types by the number of updates

[1] `https://github.com/martinkiefer/scotch`

Table 4.2: Sketching algorithms implemented

Column	AGMS [4]	MinHash (MH) [18]
Matrix	FAGMS [37]	Count-Min (CM) [39]
Row	Fast-Count (FC) [152]	HyperLogLog (HLL) [52]

applied to the state matrix for each input value.

We use the H3 family [123] for hashing values into arbitrary integer range in CM, FAGMS, MH, and HLL. For FC, we use an adaptation of the Polynomials-over-Primes scheme with a Mersenne Prime [123] to obtain a 4-wise independent hash function. For AGMS and FAGMS, we use the EH3 family for +1/-1 hashing [128]. We implement all algorithms for 32-bit input values. The state values for HLL are 6 bits wide; all other algorithms use 32-bit states.

Intel FPGA (A10, S10): We generate accelerators for two different Intel FPGA models: A midrange model (A10, Arria 10 GX 1150) and a high-end model (S10, Stratix 10 GX 2800). We use Intel Quartus Prime 19.3 as the vendor toolchain and Intel's Early Power Estimator to compute the power consumption.

Xilinx FPGA (XUS+, XUS): We generate accelerators for a recent Xilinx UltraScale+ FPGA (XUS+, XCVU7P) and an UltraScale FPGA (XUS, XCVU440). We use Vivado v2020.01 as the vendor toolchain.

CPU (Xeon): We run the algorithms on a server with two Intel Xeon Silver 4214 CPUs, each providing 24 hyper-threads. We use GCC 7.5 with OpenMP and vectorization compiler intrinsics (AVX512). We measure power consumption using powerstat 0.02.22 [91].

GPU (GeForce): We run the algorithms on a graphics card with an Nvidia GeForce RTX 2080 GPU using CUDA 10.2. We measure the power consumption using the nvidia-smi tool provided by CUDA.

FPGA accelerators use a minimal I/O template to focus on the performance of the sketching units in isolation. We use merged updates for column sketches if not stated otherwise. Our CPU and GPU baselines are hand-optimized, data-parallel, and fully utilize the architecture's parallelism. Measurements were taken on a machine running Ubuntu 18.04 for 20 iterations. We used a 2 GB uniform data set residing in main memory for the CPU and device memory for the GPU.

4.8.2 RTL Generator

We investigate the scaling behavior of the generated accelerators in terms of resource consumption and maximum operating frequency. For the sake of readability, we restrict to matrix sketches with $m = 8$. We vary the matrix size used by the RTL generator and compile with five different compilation seeds. We report the results for A10 as it has the lowest compile times (4-20x compared to S10) due to fewer resources and, thus, allowed for a more fine-grained analysis of the parameter space. Experiments for S10 and XUS+ have shown similar results and did not provide additional insights.

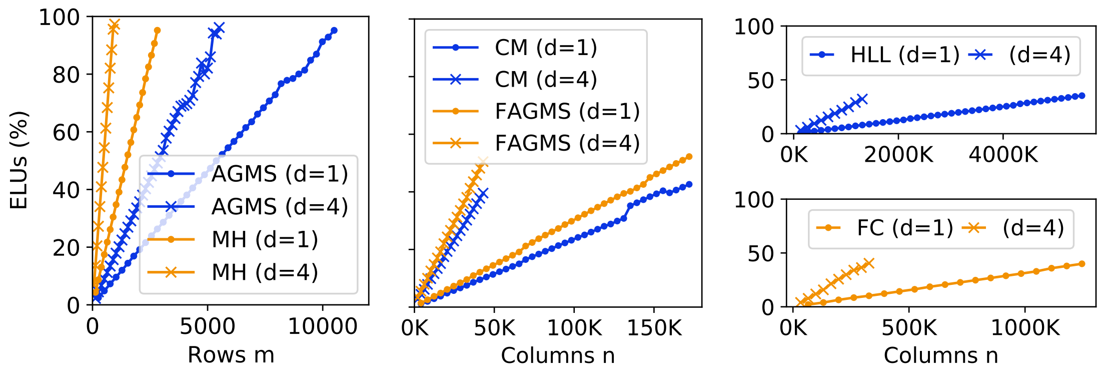
Resource Consumption

We investigate the resource consumption of generated sketching units for varying matrix sizes and data parallelism degrees $d = 1$ and $d = 4$. In particular, this allows us to validate that resource consumption increases linearly with the number of rows and columns (A1). Throughout all experiments, the number of BRAM blocks consumed is precisely the number of blocks assigned for the state memory. The ELU consumption for all algorithms is given in Figure 4.11. As ELU consumption varied under 0.01% for different seeds, we only report the maximum value. Overall, we observe that ELU consumption is approximately linear for all algorithms and both degrees of parallelism. Increasing the degree of data parallelism leads to a roughly proportional increase in resource consumption. For the column sketches shown in Figure 4.11a, we see that MH has up to 6 times higher ELU consumption, which is due to a more involved update function. For matrix sketches in Figure 4.11b, FAGMS variants show up to 30% higher ELU consumption than CM sketches due to more complex update logic. For the row sketches in Figure 4.11c, we observe HLL supporting much larger summary sizes, which is due to its smaller state. Neither matrix nor row sketches exceed 55% ELU utilization, as BRAM blocks are the dominating resource.

Summary: The experiment confirms that resource utilization is linear (A1). We observe that data parallelism comes at the cost of higher resource consumption.

Maximum Operating Frequency

Next, we investigate the impact of the state matrix size on the maximum operating frequency. As a low maximum operating frequency is the prevailing cause for failed timing, this experiment allows us to validate A2 of the auto-tuning algorithm. The maximum operating frequency varied by up to 110 MHz for different compilation seeds; therefore, we report the maximum over five runs.



(a) Column (b) Matrix (c) Row

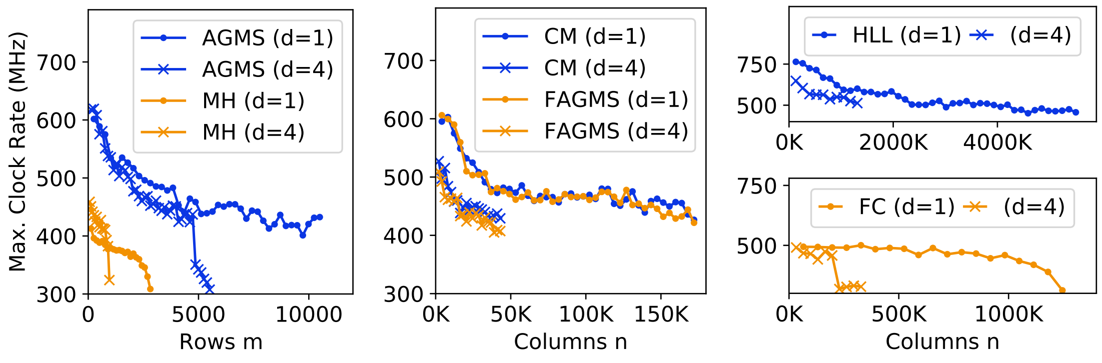
Figure 4.11: ELU consumption for various sketching units

Figure 4.12 shows the results. We observe that the maximum operating frequency decreases with a growing state matrix for all variants. However, it does not decrease strictly monotonically due to remaining variance. Almost all algorithms show a lower frequency for $d = 4$ consistent with the increased resource consumption. In Figure 4.12a, we see MH being the only exception to this, which we consider a toolchain artifact. We see that AGMS operates at an up to 200 MHz higher clock frequency than MH due to its simpler merged update function. For the matrix sketches shown in Figure 4.12b, we see operating frequencies between 400 and 600 MHz. Both algorithms decrease in a similar L-shaped pattern. For row sketches shown in Figure 4.12c, we observe HLL starting at a higher frequency of up to 763 MHz due to the significantly smaller state. FC starts at 500 MHz, decreases flatly, and shows drops of more than 120 MHZ for the largest sizes, indicating that its arithmetic-based hash function has become harder to place and route.

Summary: As the experiments confirm a trend towards a decreasing maximum operating frequency, A2 of the auto-tune algorithm is justified. While the algorithm may miss the global optimum due to remaining variance, it is an efficient alternative to a prohibitively expensive exhaustive search.

Impact of the Merged Updates

Finally, we conduct experiments investigating the benefits of merged updates for column sketches incorporating data parallelism. Figure 4.13 compares merged updates to replication for column sketches and a data parallelism degree of four. In terms of ELU consumption, we see that AGMS benefits most from merged updates, allowing for implementations that consume only half the resources and, thus, for a more than twice



(a) Column (b) Matrix (c) Row

Figure 4.12: Max. clock frequency for various sketching units

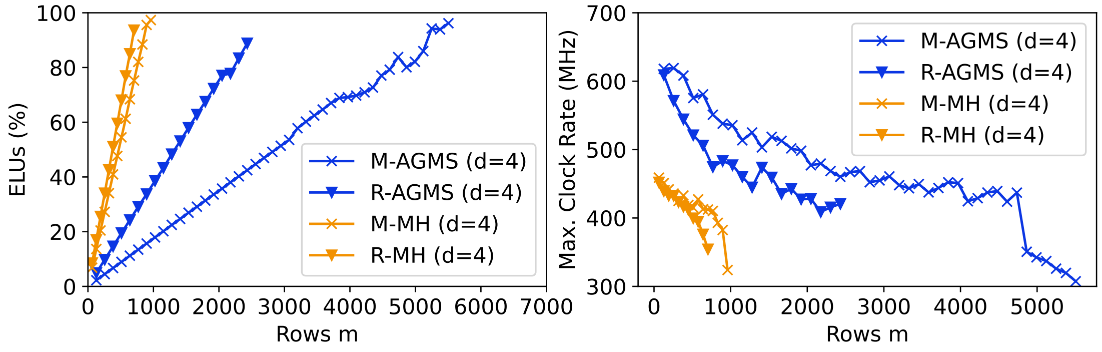


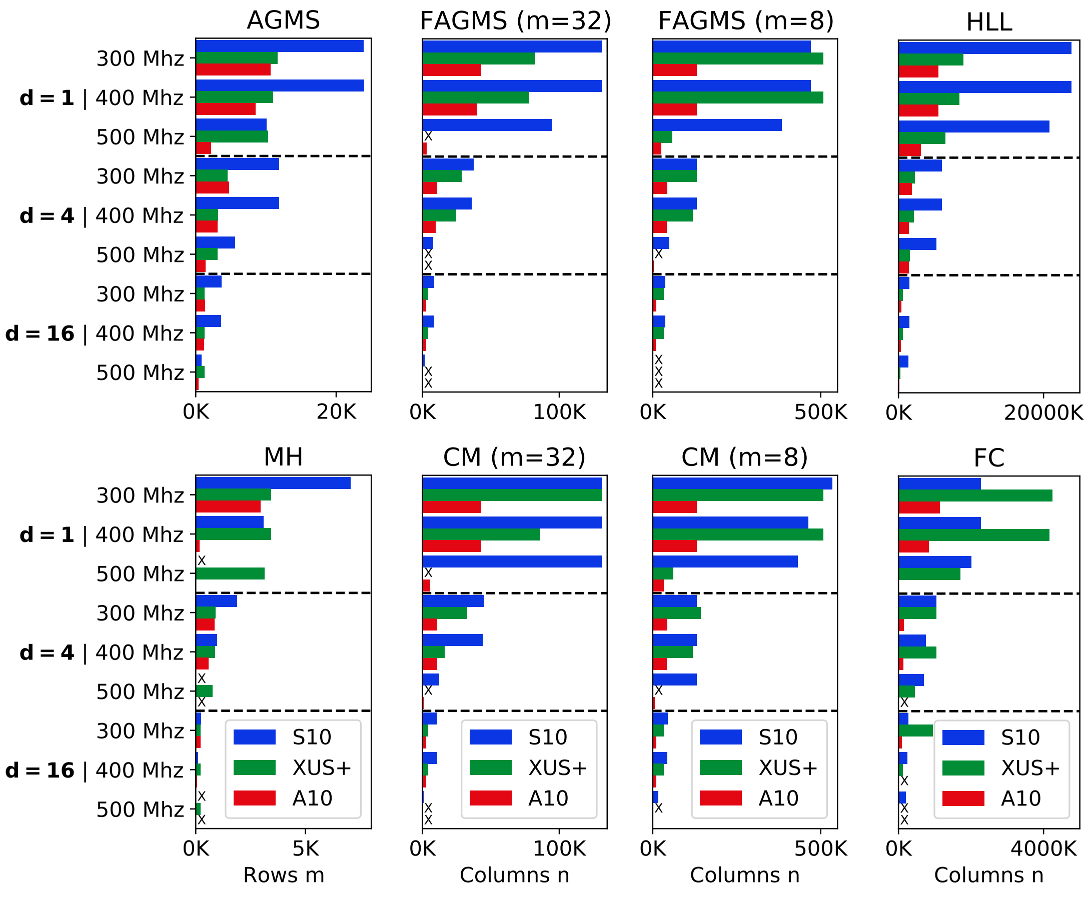
Figure 4.13: Comparison of merged updates (M) and replication (R) for data parallel accelerators

as high maximum number of columns. MH shows a smaller improvement of 25% due to the more expensive update logic that reduces the positive effect of shared resources in merged updates. Merged updates also show a positive impact in terms of the maximum operating frequency. The technique results in an up to 80 MHz improved frequency for AGMS and an up to 60 MHz improved frequency for MH.

Summary: Merged updates offer improved ELU efficiency and operating frequencies for data-parallel accelerators.

4.8.3 Automated Tuning

In the second set of experiments, we evaluate the performance of accelerators found by the auto-tuning algorithm in terms of summary size, throughput, and energy consumption. We generated accelerators for operating frequencies of 300, 400, and 500 MHz. Tuning



(a) Column (b) Matrix (c) Row

Figure 4.14: Summary sizes for FPGA accelerators on varying target devices with varying operating frequencies

starts with an initial parameter of 16 and stops at a relative difference of one percent. Tuning took between two hours and a week for A10, between four hours and six days for XUS+, and between one day and two weeks for S10.

Summary Size

First, we present the summary sizes found for the generated accelerators. This experiment allows us to showcase the effects of different target clock rates, degrees of parallelism, and device types for the generated sketches on the summary size.

Figure 4.14a shows the results for column sketches. We see that all devices are capable of generating accelerators at all operating frequencies for AGMS. For MH, with its more involved select and update function, summary sizes are between three and 49 times smaller. 500 MHz accelerators are either not possible for S10 and A10 or have less than five counters, while summary sizes for XUS+ are less affected by the increased frequency.

Figure 4.14b shows the results for matrix sketches. We observe that both algorithms perform similarly for the same number of rows as FAGMS and CM only differ in their update function. Decreasing the number of rows from $m = 32$ to $m = 8$ results in a roughly proportional increase of the summary size. As BRAM blocks are used, timing behavior for XUS+ changes as indicated by 500 MHz accelerators barely being possible. However, for 300 and 400 MHz, XUS+ even provides throughput competitive to S10. Figure 4.14c shows the results for row sketches. For HLL, we see S10 providing the largest summary sizes by up to a factor of 12. However, for FC, we see XUS+ providing up to 6.5 times larger summary sizes for $d = 1$ and $d = 4$ at 300 and 400 MHz. This observation suggests that XUS+ is better at implementing large rows with a 32-bit state than the other devices.

Overall, we see that increasing the degree of data parallelism always results in a decreasing maximized parameter. The decrease is not always proportional due to the effects of the target operating frequency. Increasing the operating frequency usually leads to a smaller parameter, especially when comparing 400 and 500 MHz.

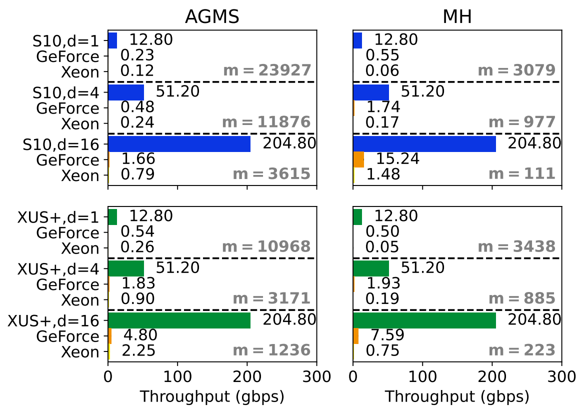
Summary: We see that the maximum summary size varies strongly depending on the algorithm, FPGA, target operating frequency, and parallelism degree. This shows the auto-tuning algorithm tailoring the summary size to the setup.

Throughput

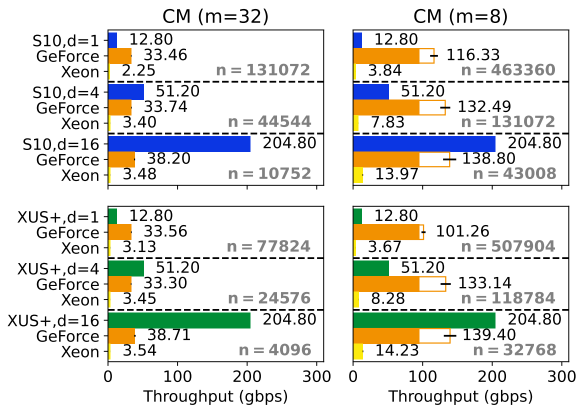
Next, we investigate our accelerators' throughput compared to a state-of-the-art CPU (Xeon) and GPU (GeForce). For the sake of brevity, we report on the larger S10 and XUS+ devices operating at 400 MHz. Our accelerators' throughput is $f \cdot d \cdot 32$ due to a static processing rate f.

Figure 4.15a shows the results for the column sketches. We see that our FPGA accelerators outperform the Xeon and GeForce baseline in all cases. Even the more competitive GeForce baseline shows an improvement ranging between a factor of 17 and 122 for AGMS and a factor of 13 to 60 for MH. FPGAs fully leverage their potential for column sketches: We generate hardware that updates tens of thousands of counters every clock cycle. CPU and GPU implementations need several cycles for an update and can not adjust all counters simultaneously due to limited parallel compute resources.

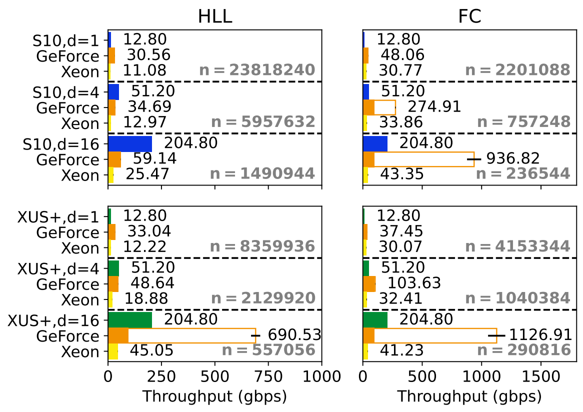
Figure 4.15b shows the results for CM with 8 and 32 rows as a representative for



(a) Column



(b) Matrix



(c) Row

Figure 4.15: Throughput for FPGA accelerators compared to Xeon and GeForce baselines

Table 4.3: Power consumption in Watt for 100 Gbit/s accelerators (S10, 390.625 MHz, $d = 8$) compared to baselines

Sketch	S10	Xeon	GeForce	Size
AGMS	33.05	152.79	184.32	n=1, m=6288
CM	69.85	176.62	213.99	n=84992, m=8
HLL	50.48	145.91	124.06	n=2981888, m=1

matrix sketches. We omit FAGMS as the results barely differ. Compared to our Xeon implementations, we see that all FPGA accelerators outperform it. For the GeForce baseline, we see a more competitive throughput: For $m = 32$, S10 and XUS+ need a data parallelism degree $d = 4$ to provide a clear advantage. For $m = 8$, the throughput provided by GeForce increases by a factor of four as the total amount of computations per input value has decreased by the same factor. However, the device can not achieve this throughput (entire bar) in practice, as PCIe poses a data transfer bottleneck and limits the processing throughput (filled bar). With a data parallelism degree of $d = 16$, the FPGA accelerators outperform GeForce by a factor of up to 2.6 considering PCIe and 10 to 40% when input data resides in device memory. FPGAs support various interconnects, allowing them to overcome transfer bottlenecks [151].

Figure 4.15c shows the results for row sketches. As before, FPGA accelerators with $d = 16$ are sufficient to outperform GeForce. When disregarding the PCIe bottleneck, we see two cases: For $n < 700k$, GeForce provides throughput up to an order of magnitude above the transfer bandwidth as only one update per data item is performed. For higher n, throughput decreases drastically due to cache misses.

Summary: The experiments show that automatically tuned FPGA accelerators can outperform software implementations on parallel architectures in many cases. For matrix and row sketches, data parallelism is essential to outperform a GPU.

Power Consumption

Next, we evaluate the power consumption of our sketching accelerators. We generate accelerators on Stratix 10 for AGMS, CM with $m = 8$, and HLL as representatives for column, matrix, and row sketches for 100 Gbit/s throughput (390.625 MHz, $d = 8$). We compare the power consumption to our CPU and GPU baselines at peak throughput. Measurements are obtained using software and do not include periphery.

Table 4.3 shows the results. We find FPGA accelerators consume between 2.5 and 5.6 times less energy than Xeon and GeForce.

Table 4.4: Scotch compared to [154] for CM (XUS, $m = 5, d = 1$)

Implementation	Operating Freq.	Throughput	Columns n
Hand-Written [154]	497 MHz	15.9 gbps	2^{16}
Scotch	503 MHz	16 gbps	2^{17}

Summary: The experiment shows that FPGA accelerators consume significantly less power than the CPU and GPU baselines.

4.8.4 Comparison to Hand-Written RTL

Finally, we compare Scotch to a state-of-the-art hand-written sketching implementation. Tong and Prasanna implemented CM sketching on XUS [154]. Their sketching approach is comparable as they guarantee static data rates and utilize a pipelined memory architecture with a DFU and $d = 1$. We compare their reported operating frequency, throughput, and summary size to a Scotch accelerator generated with the same FPGA and operating frequency. Table 4.4 shows Scotch found an implementation with twice as many columns.

Summary: Scotch competes with a manual implementation.

4.9 Related Work

Accelerating the construction of sketch summaries and their applications on FPGAs has been proposed in previous work. Our approach is related to the work of Tong and Prasanna, who used FPGAs for online heavy hitter and change detection in high throughput networks [154]. They implement CM sketching with guaranteed data rates using pipelined memory and a data forwarding unit. As shown in Section 4.8.4, our accelerators can operate at the same frequency and throughput for a data parallelism degree of one. However, we can provide even higher throughput by exploiting data parallelism. Other streaming algorithms implemented on FPGAs are Space Saving [150], Exponential CM [32], CM-CU [130], MinHash [139], and Bloom Filters [30].

High-level synthesis tools such as OpenCL [28, 41, 164] or VivadoHLS [51] allow for the generation of accelerators from C-like programming languages. While these frameworks are general-purpose and have been successfully applied in database acceleration [161, 162], Scotch exploits knowledge about the memory access pattern and semantics defined by the select-update model to provide fully pipelined sketching RTL; the tuning process is automatic, given a matrix size and user-defined functions. However, high-level synthesis tools can reduce the development effort and expertise required to develop I/O templates for Scotch.

Previous research has suggested automated tuning of toolchain and user parameters towards an objective function [20, 93]. It treats RTL and optimization parameters as a black box and, thus, requires general models and sufficient training. Scotch's automated tuning algorithm uses intuitive assumptions based on domain knowledge to optimize for the summary size using a practical approach that is logarithmic in the search space.

4.10 Conclusion

In this chapter, we introduced Scotch, a novel system for generating optimized sketching accelerators on FPGAs. It provides a full system stack covering all aspects, from sketch specification over code generation to automated tuning.

We evaluated Scotch for six sketching algorithms and three different FPGAs. We showed that Scotch tailors the summary size to the desired throughput and FPGA. We highlighted the inherent trade-off between throughput and summary size controlled via the operating frequency and degree of parallelism. We found that the accelerators can satisfy interconnects with a bandwidth of 100 Gbit/s and more. Scotch accelerators outperform CPU baselines by a factor of up to 300 in terms of throughput and by a factor of up to 4.6 in terms of energy consumption. Compared to a GPU baseline, FPGA accelerator throughput ranges from competitive to an improvement of a factor of 120 while consuming up to 5.6 times less energy. Furthermore, we found that Scotch accelerators compete with a manual FPGA implementation.

Overall, this work shows that Scotch produces highly efficient FPGA sketching accelerators without manual RTL programming and tuning. Thus, Scotch substantially lowers the entry bar for FPGA accelerated sketching by replacing an FPGA expert with code generation and automated tuning.

5 Optimistic Data Parallelism for FPGA-Accelerated Sketching

The architecture for FPGA-accelerated sketching introduced in the previous chapter implements data parallelism via replication. This chapter proposes an *optimistic* architecture that trades the hard processing guarantees provided by full replication for improved resource utilization by sharing resources among the processing logic for each input. We show that such an architecture is feasible and introduce techniques to avoid stalls due to resource congestion. Furthermore, we show that the larger summary sizes enabled by optimistic architectures translate to substantial gains in accuracy for an approximate query processing application.

This chapter is mainly based on our publication 'Optimistic Data Parallelism for FPGA-Accelerated Sketching' [90].

5.1 Introduction

The previous chapter on Scotch highlights that achieving high throughput for FPGA-based sketching accelerators particularly requires exploiting data parallelism. Scotch [89] and other recent approaches to FPGA-accelerated sketching [29, 92] implement data parallelism *pessimistically* by replicating the sketching hardware for each of the inputs and, thus, entirely avoiding concurrent accesses to state memory. However, while replication guarantees the desired throughput, it also increases resource consumption proportionally to the number of inputs. In particular, we identified state memory as the bottleneck of pessimistic architectures in Section 4.8.2. Thus, pessimistic data parallelism severely restricts sketch sizes and FPGA resources available to other functionalities, such as additional sketches.

This chapter proposes a novel *optimistic* architecture for FPGA-accelerated sketching with an improved trade-off between throughput and BRAM consumption. It makes the scarce state memory concurrently available to all input values instead of replicating it. As this architecture stalls to resolve resource conflicts in the presence of data skew, it does not guarantee constant processing rates (DR4, Section 4.2.1). However, we show how to exploit temporal locality to merge multiple updates into a single transaction and, thus, avoid stalling by reducing congestion.

In summary, we make the following contributions:

1. We propose a novel optimistic sketching architecture. Our architecture partitions the sketch state into independent *banks* available to all inputs, thus, implementing parallelism while reducing the consumption of the scarce state memory.

2. We propose merging techniques that mitigate resource congestion due to data skew by exploiting temporal locality.

3. We discuss and evaluate the impact of architecture parameters on resource utilization. Furthermore, we discuss the limitations of our optimistic architecture.

4. We implement FPGA-accelerated sketching on a Xilinx U250 FPGA data center accelerator for an approximate query processing application. We show that the optimistic architecture outperforms state-of-the-art CPU and GPU implementations' throughput while larger summary sizes boost accuracy.

To the best of our knowledge, this is the first work that performs FPGA-accelerated sketching for a general class of sketches in an optimistic architecture.

The following Section 5.2 describes our novel optimistic banked sketching architecture, while Section 5.3 presents our techniques to mitigate resource congestion. Sections 5.4 and 5.5 discuss the impact of different architecture parameters on resource utilization and the limitations of our architecture, respectively. Section 5.6 describes an application for optimistic FPGA-accelerated sketching, and Section 5.7 presents our experimental evaluation. Finally, we discuss related work in Section 5.7 before concluding in Section 5.9.

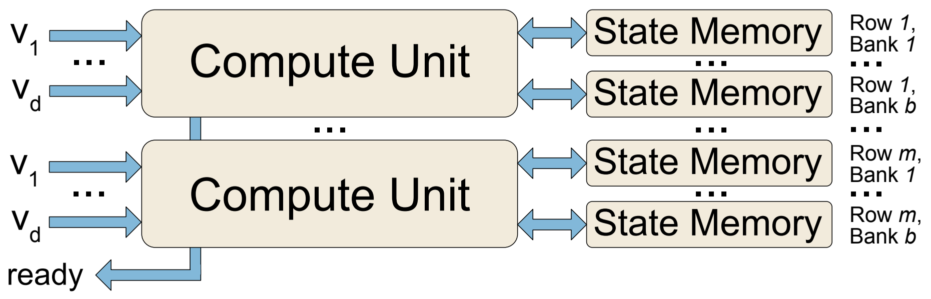


Figure 5.1: Optimistic data parallel sketching unit

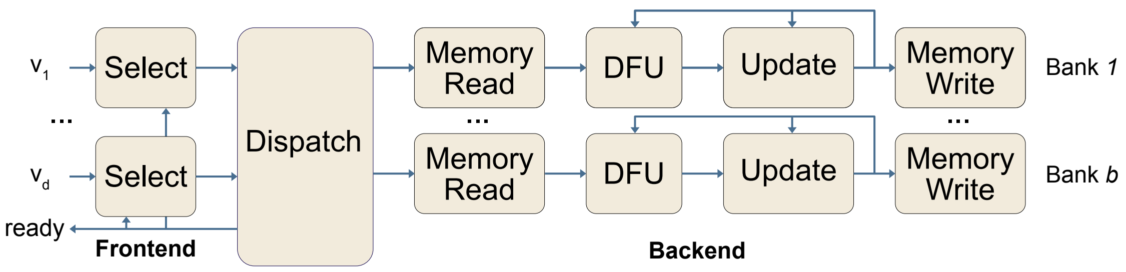


Figure 5.2: Optimistic compute unit architecture

5.2 Banked Sketching Architecture

In this section, we introduce our novel *optimistic* banked sketching architecture. Unlike a pessimistic architecture, the banked architecture makes the same state memory available to the processing logic of all d inputs simultaneously. Thereby, it reduces resource consumption for state memory by a factor of d, freeing resources for large sketch sizes, additional sketches, or other logic. However, it can also introduce stalls in case of congestion and thus requires additional logic to implement conflict resolution.

The banked sketching architecture maintains the sketch state in b pipelined *banks*, each serving a range of offsets. Figure 5.1 provides an overview of an optimistic sketching unit with multiple banks. Instead of processing each input value in a dedicated replica of the sketching unit, the same sketching unit processes all d input values. The per-row compute units calculate the d corresponding offsets and update the state in the offset's corresponding bank. As each bank processes one input value per clock cycle, conflicting accesses to the same bank require the compute unit to serialize operations and eventually stall the processing of further values.

5.2.1 Compute Unit Architecture

Figure 5.2 shows the optimistic compute unit architecture. It contains a select unit for each of the d input values that evaluates the select function. We refer to this part of the architecture as the *frontend*. A *dispatcher* connects the inputs to the corresponding banks, stalls the pipelines for conflict resolution, and signals stalls to the overall sketching unit. Each of the b banks consists of an independent set of stages: The memory read stage retrieves the state for a given offset, the update function computes the next state, and the memory write stage issues a write request to persist newly computed state. As reading and writing from memory incurs multiple cycles of latency, a data-forwarding unit (DFU) tracks updates. It ensures that no updates are lost before a state value enters the update stage. We refer to all stages behind the dispatcher as the *backend*.

Except for the dispatcher, all stages are also present in a pessimistic architecture. Note, however, that only the select and update stages are fully replicated for each input value and bank, respectively. The overall number of substages in the read, write, and DFU stage depends on the number of segments in the state memory and, thus, only depends on the size of the sketch matrix — but not on the number of inputs d or banks b.

The dispatcher arbitrates the d incoming offset-value pairs and follows the architecture in Figure 5.3. First, a b-dispatch unit dispatches the pair to the corresponding bank based on the offset. A FIFO queue $q_{i,j}$ buffers the offset-value pair assigned to bank i for input j. Since there is at most one new value per queue in each clock cycle and input, we can use the efficient FIFO queue building blocks shipped with vendor toolchains. A d-collect unit for each bank arbitrates conflicts by popping one element from a queue in each clock cycle in a round-robin fashion and forwards the element to its corresponding bank in the backend. If at least one queue in a dispatcher is full, the entire sketching unit stalls to avoid a potential loss of input data. Accordingly, the dispatcher signals a stall to the compute unit frontend. The compute unit signals a stall for its row to the overall sketching unit, which in turn signals a stall and stops accepting new input values.

5.2.2 Stall Rate

Given a fixed degree of data parallelism d, the banked architecture has two parameters: The number of banks b and the queue size qs. We will discuss the impact of parameters in terms of the stall rate, which is the fraction of clock cycles lost due to stalls. Choosing $b < d$ eventually causes the architecture to stall when values arrive at each clock cycle. As the architecture can only process b values during each clock cycle in the backend, the stall rate will be at least $1 - \frac{b}{d}$. Based on the above observation, we propose two versions

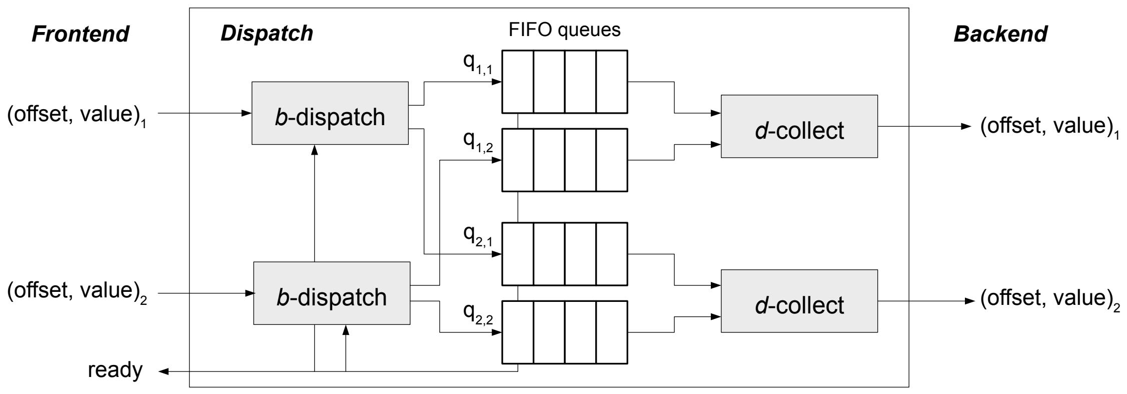


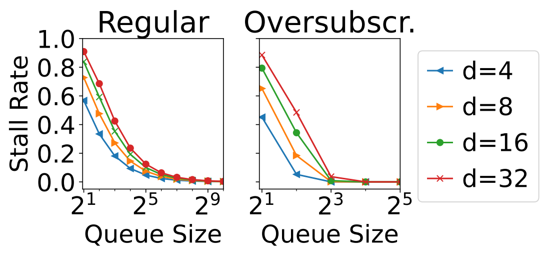
Figure 5.3: Dispatcher architecture

of the banked architecture that are physically capable of processing as many values as the pessimistic architecture: First, the *regular* architecture with $b = d$ has the minimum number of banks for d inputs. Second, the *oversubscribed* architecture with $b = d \cdot 2$ can process twice as many elements in the backend as the architecture can ingest. While this strategy doubles the number of connections and queues in the dispatcher, it potentially takes pressure off individual queues. We assume, without loss of generality, that d is a power of two as I/O interfaces commonly provide data at this granularity. Consequently, the number of banks b is a power of two, which drastically simplifies assigning offsets to banks in hardware.

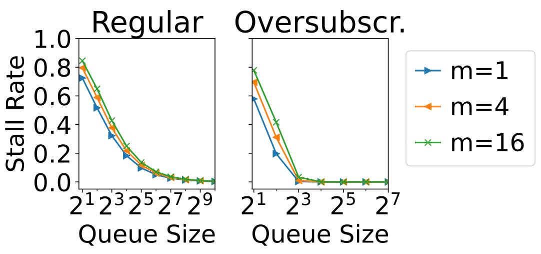
Given these two architectures, the stall rate depends on the number of inputs, banks, queue size, and offset distribution. In the rest of this section, we study their properties in more detail.

Uniform Bank Access

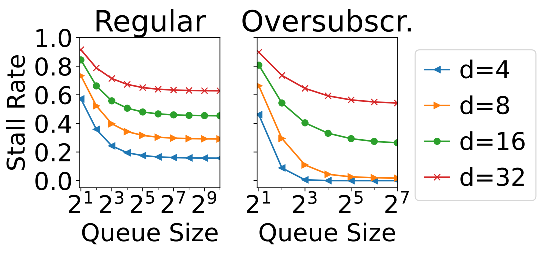
When accesses to the banks are uniform, queues prevent stalls due to random collisions. Figure 5.4a shows the impact of the queue size on the stall rate for a sketch with a single row and varying d on uniform offsets. We see stall rates approaching zero with increasing queue sizes for both architectures as the additional buffering suffices to prevent stalls due to offsets colliding on banks randomly. Most importantly, the regular architecture requires larger queues than the oversubscribed architecture for low stall rates. To reach stall rates below 1% for all values of d, the regular architecture requires a FIFO queue with $qs = 512$ entries, while $qs = 16$ suffices to prevent stalls completely in an oversubscribed architecture. As oversubscribed architectures can process twice as many input values simultaneously in the backend than arrive from the frontend, pressure on



(a) Uniform, $m = 1$ rows, varying inputs d



(b) Uniform, $d = 16$ inputs, varying rows m



(c) Zipf $\rho = 1.1$, $m = 1$ rows, varying inputs d

Figure 5.4: Simulated stall rates in a banked architecture

individual queues reduces, leading to smaller queues sufficing for low stall rates. While oversubscribed architectures contain twice as many queues, the accumulated queue size for the overall architecture remains 16 times smaller.

Impact of Multiple Rows

For $m > 1$ rows, the per-row sketching units process the input values independently, but stalls required by an individual row allow all other rows to reduce the number of buffered elements as well. Figure 5.4b shows the stall rates for a banked architecture with $d = 16$ and $m \in \{1, 4, 16\}$ rows for a uniform offset distribution. Although the probability of at least one row signaling a stall grows exponentially in a fully independent system, the dependent stalls in our architecture only lead to a moderate increase in the stall rate.

Skewed Bank Access

If there is skew in the distribution of accessed banks from the backend, e.g., due to heavy hitters and skewed data distributions, the stall rate can go up as high as $1 - \frac{1}{d}$ when all offsets target only one bank. Both architectures cannot sustain the full processing rate if a bank receives more than $\frac{1}{d}$ requests from the frontend, that is, more than one request per clock cycle on average. This implies that the regular architecture can only avoid stalls when the bank access pattern is uniform. The oversubscribed architecture tolerates moderate skew as the abundance of banks does not require all banks to operate at full capacity to avoid stalls.

Figure 5.4c shows the stall rates for a moderately skewed Zipfian distribution with support $\rho = 1.1$. For the regular architecture, we see that stall rates vary between 16 and 90% and find that queuing alone is not sufficient to mitigate the impact of skew. For the oversubscribed architecture with $d \in \{4, 8\}$, we find that it indeed substantially mitigates the impact of data skew and achieves a stall rate of less than 2%, given a sufficient queue size. For $d \in \{16, 32\}$, the absolute improvement compared to the regular architecture is between 10 and 30% given a sufficient queue size, but we still observe stall rates of more than 25%. Furthermore, the skewed distribution requires a four times larger queue size $qs = 64$ for the stall rates to converge in an oversubscribed architecture.

Based on this analysis, we conclude that dealing with heavy hitters and skewed access to banks requires additional solutions. We thus devise techniques that can exploit the properties of sketching algorithms to counter skew.

5.3 Merging in the Dispatcher

Preventing stalls in a banked architecture means either (1) increasing the number of elements popped from queues in each clock cycle or (2) reducing the number of elements pushed into the queues. In the following, we will discuss how we trade additional logic in the dispatcher for fewer stalls in the presence of heavy hitters and data skew by exploiting temporal locality. The fundamental approach is to merge the updates for input values affecting the same offset before dispatching them to the banks.

As the Select-Update model does not provide an appropriate interface to merge updates, we first establish the theoretical framework by describing the update function in terms of a map and a reduce function. Second, we introduce two mechanisms to reduce stalls by merging updates as shown in Figure 5.5: *Vertical merging* extends the d-collect unit with logic to merge updates from the individual queue heads. *Horizontal merging* buffers updates at each input as a first processing step in the dispatcher.

5.3.1 Map-Reduce Updates

We specify the update function in terms of a *map* and *reduce* function. Intuitively, the *map* function translates the input value to an increment, while the *reduce* function allows us to merge increments either with other increments or the state in the sketch matrix.

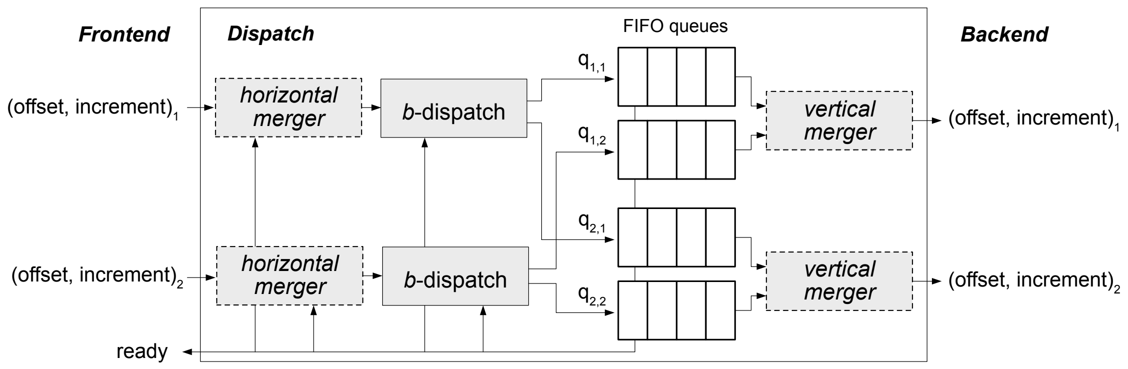


Figure 5.5: Dispatcher architecture. There is one horizontal merger per input and one vertical merger per bank.

Formally, an update is defined as:

$$up_i\,(val, state) = reduce_i\,(map_i\,(val)\,, state) \tag{5.1}$$

In addition to the above definition, we require the reduce function to be associative and commutative. This enables the reduce function to merge increments from any position in the input stream and out of order. Furthermore, we require an identity element to the reduce function that we can assert for inactive inputs.

As pessimistic data parallelism requires mergeability of sketches and the Select-Update model implies that entries in sketches are updated and merged independently, the Map-Reduce model does not impose additional restrictions on our approach.

We evaluate the map function concurrently to the select function, and the computed increment replaces the value as an input to the dispatcher. The update stages in the backend reduce the increment and the current state. However, the properties of the reduce function allow merging any two increments belonging to the same offset, which we exploit to reduce congestion for heavy hitters.

Example (CM): Updates to the CM sketch can be described in terms of a map and reduce function. We define the map function as $map_i : t \mapsto 1$ and the reduce function as $reduce_i : x, y \mapsto x + y$. Thus, every incoming value contributes an increment of one, and the reduce function adds the increment to the state. The identity increment for the reduce function is zero.

The reduce function allows merging increments that affect the same offset as shown in Figure 5.6. The offset of the least recent offset-increment pair is compared with the three following pairs. Increments with matching offsets enter the reduce function to be

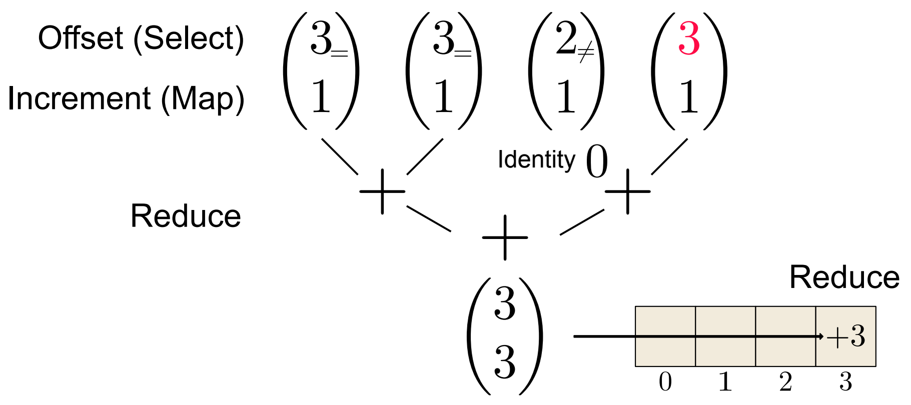


Figure 5.6: Map-Reduce merging for a CM sketch

merged. Non-matching increments contribute the identity element. Finally, we reduce the merged increment with the current state of the matrix at the given offset. Overall, processing three values incurs only one update to the row due to increment merging.

5.3.2 Vertical Merging

Vertical merging pops and merges increments with matching offsets from all queues belonging to the same bank. To this end, the vertical merger enhances the d-collect unit in the banks. As vertical merging occurs behind the FIFO queues, the mergers operate even if the architecture is stalled and contribute to ending stalls.

Figure 5.7 shows the architecture of a vertical merger. In addition to removing queue elements one by one in a round-robin fashion, we also check the heads of all other queues for offsets matching the currently selected one. This functionality is implemented as an additional stage that performs a compare-forward operation based on the offset in the currently selected queue. We pop all queues sharing the selected offset in their head and forward to the next stage. The heads of all other queues remain untouched, and the identity increment is forwarded. A pipelined binary tree of reducers merges the increments in the following stages. After merging, the vertical merger sends the result to its bank.

Adding vertical merging to the architecture requires implementing additional $b(d-1)$ reducers and $b \cdot d$ compare-forwards for each compute unit. The reduction potential for vertical merging depends on the queue heads sharing the current offset. In the best case, all queues have increments with the same offset at their head, and d increments collapse into one increment. In the worst case, all queues are empty or have different offsets at their heads, and vertical merging behaves like the regular d-collect.

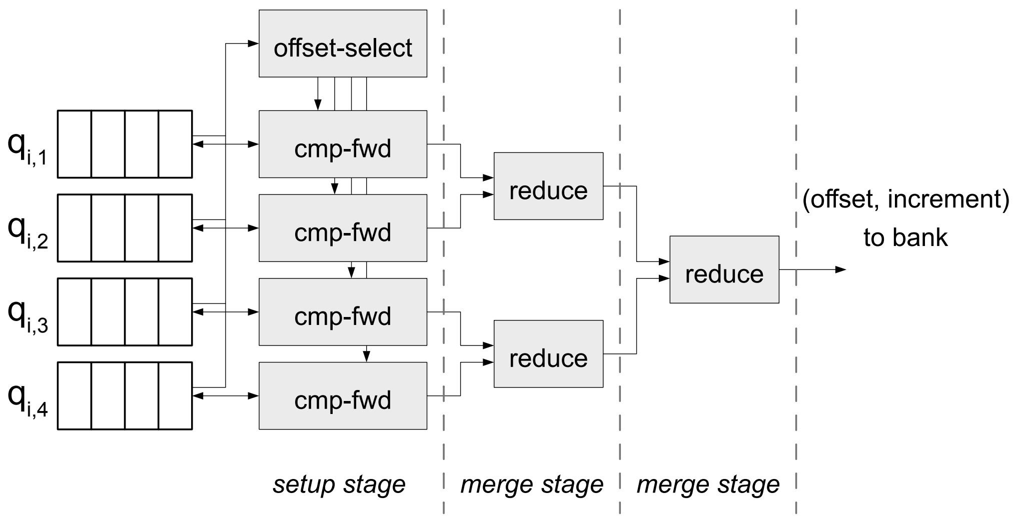


Figure 5.7: Vertical merging with $d = 4$ input values

5.3.3 Horizontal Merging

Horizontal merging buffers increment-offset pairs and merges them with matching offsets. Each of the d parallel inputs has one horizontal merger that directly consumes values arriving in the dispatcher. As horizontal mergers operate in front of the queues, they must obey stalls requested by the architecture. Thus, they help to prevent stalls but cannot contribute to ending stalls.

Figure 5.8 shows the architecture of a horizontal merger. All values pass through a chain of h registers that buffer the last h values as a look-ahead. The merger compares the last offset in the chain with all previous stages in compare-forward logic. If the offsets are equal, the increment in the register is forwarded, while the identity increment replaces the increment in the register. Otherwise, the identity increment is forwarded, and the register remains untouched. In the following stages, a binary tree of reducers merges all increments from the compare-forward units and the increment from the last stage. Finally, the horizontal merger forwards the result to the b-dispatch logic.

Adding horizontal merging to the architecture requires $d \cdot (h - 1)$ additional reducers and $d \cdot (h - 1)$ compare-forwards for a given look-ahead $h \geq 2$. The reduction potential depends on the number of entries in the register chain that share the same offset. In the best case, all entries share the same offset, and h values reduce to a single increment. However, the horizontal merging logic will run with reduced efficiency in the following cycles as the shift register fills with new values. In the worst case, no merge occurs, and all h elements enter the queue sequentially. As the look-ahead h controls the number of

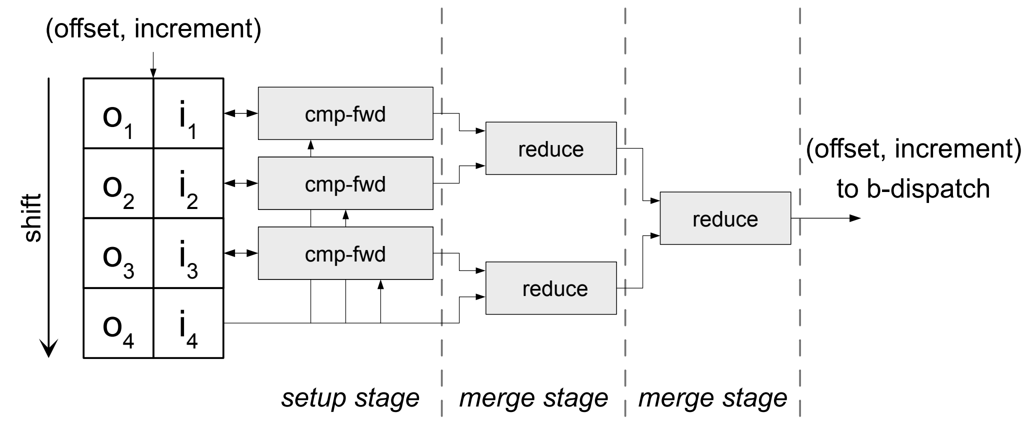


Figure 5.8: Horizontal merging with look-ahead $h = 4$

reducers in horizontal mergers, we can apply horizontal merging in a more fine-grained fashion than vertical merging.

5.3.4 Discussion

Merging techniques aim at avoiding stalls resulting from full queues. To prevent full queues, mergers must ensure that at most $\frac{1}{d}$ values result in accesses to the same bank, as only one value per bank can retire in each clock cycle. If a single heavy hitter appears exclusively on all d inputs, we have to be able to merge at least d such offset-increment pairs per clock cycle on average. Thus, making an optimistic banked sketching architecture resilient against this pathological case requires applying either vertical merging or horizontal merging with a look-ahead $h \geq d$.

Note that merging does not adversely impact stall rates or throughput: Merging optionally allows increment-offset pairs to take effect earlier, but it does not impair other pairs' progress through the pipeline. Furthermore, the associativity and commutativity of the reduce function guarantee the correctness after merging.

While we can establish that banking favors a uniform distribution of accesses to banks and merging effectiveness increases with the frequency of heavy hitters, the actual effect of merging is highly dependent on the distribution of offsets. Thus, we examine the effect of merging techniques in Section 5.7.1, where we provide an extensive evaluation on various real-world and artificial datasets.

5.4 Dispatcher Resource Utilization

The optimistic architecture with banking and merging trades off a d-fold lower state memory consumption for increased resource consumption in the dispatcher. For a sketch with a given matrix shape and sketching functions, architecture parameters have the following impact on resource consumption in the dispatcher:

Parallel inputs d: The number of FIFO queues in a regular and oversubscribed architecture is d^2 and $d^2 \cdot 2$, respectively. Thus, resource consumption for queues and vertical merging in the dispatcher increases quadratically with d. As horizontal mergers cover each of the d inputs, they contribute linearly.

The above shows that the dispatcher eventually becomes the bottleneck of optimistic architectures with an increasing number of inputs d. In these cases, we can build hybrid architectures that maintain r replicas of optimistic sketching units with d inputs. These architectures support $d_{hybrid} = r \cdot d$ inputs.

Queue size qs: Resource consumption in the dispatcher increases linearly with the queue size qs. For BRAM-based FIFO queues, BRAM is the only affected resource.

Vertical merging / Horizontal merger look-ahead h: Enabling vertical merging adds a constant overhead to an optimistic architecture. The number of reducers and compare-forwards in horizontal mergers and, thus, the overall resource consumption increases linearly with the look-ahead h.

However, horizontal merging additionally impacts domain value optimization. Consider the CM sketch in Section 2.1.2: Vendor toolchains detect that a single bit is sufficient to encode a $\{+1, 0\}$ increment. This optimization can significantly reduce the resources required for increment handling. In particular, smaller increments decrease the width of queues. As increments in merge trees increase by one bit per level, inflated increments due to increasing h in horizontal merging also affect queues and vertical mergers.

5.5 Limitations

While optimistic architectures consume fewer memory resources, some use cases cannot tolerate processing stalls. Specifically, stalls become an issue if both of the following two properties hold:

Unidirectional communication: The sketching accelerator lacks control communication with the data source or may not request the data source to pause transmitting input data until the sketching unit recovers from stalls. For example, pausing processing data on a hard drive is feasible, while we cannot halt streaming data from network monitoring.

Hard processing guarantees: The application requires that every input value is guaranteed to be processed by the sketching accelerator. For example, the estimates computed by a CM sketch are only hard upper bounds if the sketch observes every input element. An application testing whether specific malicious IP addresses connect to a service may require this guarantee.

Fortunately, hard guarantees for processing at full rate are rarely required. Streaming engines commonly provide soft guarantees as they employ software queues to handle backpressure [24].

Banking and merging exclude sketches (e.g., CM-CU [34, 49]) or applications (e.g., heavy change detection in networks [154]) that require evaluating the sketch on the fly: Due to increments potentially arriving merged and at different clock cycles in each row, monitoring updates to the sketch in real time cannot be trivially translated to an optimistic architecture. Bringing all rows to a consistent state requires waiting for all queue elements to be processed. Thus, optimistic architectures favor applications with a separation between maintaining and evaluating the sketch.

5.6 Application: Approximate Group-By on a Xilinx U250 Accelerator

As optimistic sketching architectures reduce the resources necessary to implement state memory, there are more resources available to implement larger or additional sketches. To show our approach's potential, we approximate the result of a grouped aggregate query as an intuitive application for optimistic FPGA-accelerated sketching. To that end, we use an ensemble of variations of the CM sketch, which we explain in Section 5.6.1. This application can benefit greatly from the additional available resources, as we exploit them to increase accuracy via the sketch size. Furthermore, it tolerates eventual stalls imposed by the optimistic architecture.

We implement sketching on a high-end Xilinx U250 data center accelerator. Section 5.6.2 discusses the architecture. As cloud services, such as AWS or Azure, provide instances with similar hardware, our approach can enable fast insights into large datasets stored in cloud storage or cloud-based data warehouses.

Table 5.1: Sketch ensemble for grouped aggregates

Aggregate	Map	Identity	Reduce	Evaluation	Bound
count(*)	+1	0	+	*min*	Upper
sum(v)	$+v$	0	+	*min*	Upper
min(v)	v	v_{max}	*min*	*max*	Lower
max(v)	v	v_{min}	*max*	*min*	Upper

5.6.1 Sketch Ensemble

We construct an ensemble of four variations of the CM sketch over an input stream s of key-value pairs (k, v). The sketches approximate the result of the following SQL query:

```
select k,count(*),sum(v),min(v),max(v) from s group by k
```

All four sketches use the $H3$ family of hash functions in the select function, but vary in the map, reduce, and evaluation function as well as the quantities they estimate. Table 5.1 lists the functions and estimated quantities.

The CM sketch shown in Figure 2.4a constructed over the keys provides an upper bound on the number of observed tuples $count(*)$ in the group for each key k. Analogously, incrementing by the value v while hashing on the key yields a sketch that provides an upper bound on the sum of values $sum(v)$ grouped by the key k. The idea of scattering aggregate functions using hash functions over multiple rows and columns to mitigate the impact of collisions extends to other algebraic aggregate functions, such as the minimum and maximum. To estimate these quantities, we replace the reduce function with the minimum and maximum function and set the identity element to the highest and lowest possible key, respectively. In the following, we will refer to the sketches by their aggregate (e.g., sum-sketch, max-sketch).

The count-sketch also allows inclusion tests in the set of keys. As the sketch provides an upper bound on the number of tuples with a given key, an estimate of zero provides certainty that the input data did not include the key, and thus querying the sum, min, and max sketches is futile.

Maximizing the size of the sketches is essential to improve the result quality for this application. The expected number of hash collisions on an entry in each row of a CM sketch is $\frac{g}{n}$, g being the number of groups. Thus, the number of collisions reduces inversely proportional to the number of columns. The impact of collisions varies among aggregate functions: While errors for sum and count accumulate with each collision, min and max errors depend only on the most extreme value that hashed to a bucket.

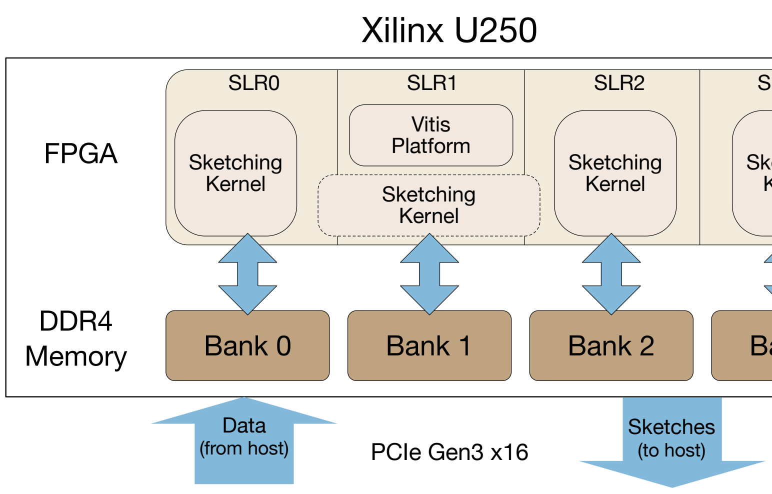


Figure 5.9: Sketching coprocessor for grouped aggregates implemented using Xilinx Vitis.

Each additional row may improve the estimate, as it represents an independent trial with different random collisions.

5.6.2 Accelerator Architecture

We perform sketching on a Xilinx U250 data center accelerator with a high-end UltraScale+ FPGA. We implement sketching as RTL kernels in the Xilinx Vitis framework. Vitis enables portability and reuse of FPGA designs in RTL and high-level languages, implements I/O on the devices, and exposes GPU-like interfaces to host code.

Figure 5.9 shows a schematic of the U250 accelerator card and our sketching implementation on it. We evenly split the input data and transmit the chunks to the four 8GB DDR4 memory banks on the device via PCI Express. The FPGA consists of four Super Logic Regions (SLRs), each directly connecting to a DDR4 memory bank. As connectivity between SLRs is limited, we run independent sketching kernels for each memory bank and lock the kernels operating Bank 0, Bank 2, and Bank 3 inside their respective SLR. We only allow the kernel operating Bank 1 to spread over multiple SLRs because the Vitis platform blocks a significant portion of SLR1 to provide auxiliary functionality.

Each memory bank connects to the kernel via a 512-bit wide interface operating at 300 MHz, which results in a maximum total throughput of 600 gbps for all four memory banks. While processing on the U250 accelerator is limited by the 126 gbps theoretical maximum throughput of PCI Express Gen3 x16, we still provision for maximum throughput when reading from device memory: Existing or future FPGA boards with better connectivity, e.g., multiple 100G Ethernet ports or PCIe Gen4/5, can narrow or close the gap.

UltraScale+ provides two types of on-chip high-density memory elements: Regular BRAM elements provide 18 KB with varying width and depth; UltraRAM (URAM) provides 288 KB with a fixed 72-bit width and depth 4096. As URAM makes up the largest portion of on-chip memory, we map memory segments to URAM. We use the more flexible BRAM for the large FIFO queues in the regular architectures, while we implement the smaller queues in the oversubscribed architectures with logic resources.

The accelerator assumes 32-bit unsigned input values for the key and value. In addition to updating four sketches in parallel, we invest the additional resources provided by the optimistic architecture to implement the count- and sum-sketch with an increased state size of 64 bits. This increase is necessary as the 32-bit state commonly used in sketching implementations [29, 89] is prone to overflows for large skewed datasets.

Overall, we designed the architecture to show the maximum throughput for our application that is practically achievable on the FPGA shipped with the U250. As we integrated our approach with Vitis, our implementation can be adjusted to other boards, cloud-based FPGAs, throughput requirements, and use cases with moderate effort.

5.7 Evaluation

We investigate the performance of our architecture in terms of stall rates, resource consumption, maximum operating frequency, throughput, and accuracy. First, we show the effect of merging on the stall rate for various real-world and artificial datasets to devise the degree of merging required to handle data skew. Second, we show that optimistic architectures are feasible on a modern FPGA, consume less state memory than pessimistic ones, and can provide comparable maximum operating frequencies. Third, we highlight the impact of merging regarding resource consumption and maximum operating frequency. Finally, we show that a modern FPGA accelerator can achieve vast throughput for our approximate group-by application, while optimistic architectures boost accuracy due to larger summary sizes.

We evaluate our approach using various sketching implementations and baselines:

Simulator: A hand-written C++-based software replica of all components in the frontend and dispatcher determining the stall rate. It allows us to validate a broad range of parameters without going through the time-intensive process of compiling or simulating the entire sketching RTL.

Dummy: Sketching unit RTL connected to a minimal dummy I/O template that targets a Xilinx XCVU13P FPGA equivalent to the one on U250 accelerator boards. It allows us to determine the resource consumption and maximum operating frequency of sketching units in isolation and without the overheads of placing and routing with actual I/O. We use the Vivado 2021.2 toolchain.

U250: A complete sketching accelerator for our group-by application based on a Xilinx U250 accelerator card as detailed in Section 5.6. We use the Vitis 2022.1 toolchain with the XDMA 4.1 platform.

EPYC: Sketching for our group-by application performed on two AMD EPYC 7742 CPUs each providing 64 cores with two threads. We use GCC 9.4 with OpenMP for multithreading and compiler intrinsics for vectorization (AVX2). Each software thread constructs an instance of the sketch over a chunk of input data. We evaluate two multithreading strategies: We maintain one sketch (1) per hardware thread to maximize the potential for instruction-level parallelism and (2) per core to reduce cache thrashing.

A100: Sketching for our group-by application performed on an Nvidia A100 GPU using CUDA 11.7. We evaluate two parallelization strategies: (1) All threads cooperate at updating each row to maximize data locality. (2) Each thread applies updates exclusively for one row in a total of r sketch instances. This strategy reduces conflicting atomic operations to memory. We try increasing powers of two for r until the memory required exceeds device limits.

We generate RTL using Scotch [89] as a representative for pessimistic sketching. For optimistic sketching, we adjusted Scotch to include banking, map-reduce updates, and our merging strategies. Based on the simulation results shown in Section 5.2, we select a queue size of $qs = 512$ for the regular and $qs = 64$ for the oversubscribed architecture. As our optimistic architecture addresses a design issue of pessimistic data parallelism, comparisons extend to any pessimistic architecture.

We evaluate our approach using artificial and real-world datasets.

Uniform: Uniformly drawn keys with values fixed to +1.

Zipf(ρ): Keys drawn from a Zipf distribution generated with support $\rho \in \{1.05, 1.5\}$ and values fixed to +1.

Caida: Real-world traces collected from the Equinix Chicago internet exchange [23] in 2011. We use the source IP address as the key and the package size as the value.

Cup'98: Access logs for the 1998 Football World Cup web site [7]. We use the client ID as the key and the answer size as the value.

NYT: Trips of yellow taxis in New York City between 2009 and 2016 [148]. The key encodes the pickup and dropoff location coordinates on a 256x256 grid. The value is the total amount paid for the ride. Data is ordered by the dropoff time.

5.7.1 Impact of Merging on Stall Rates

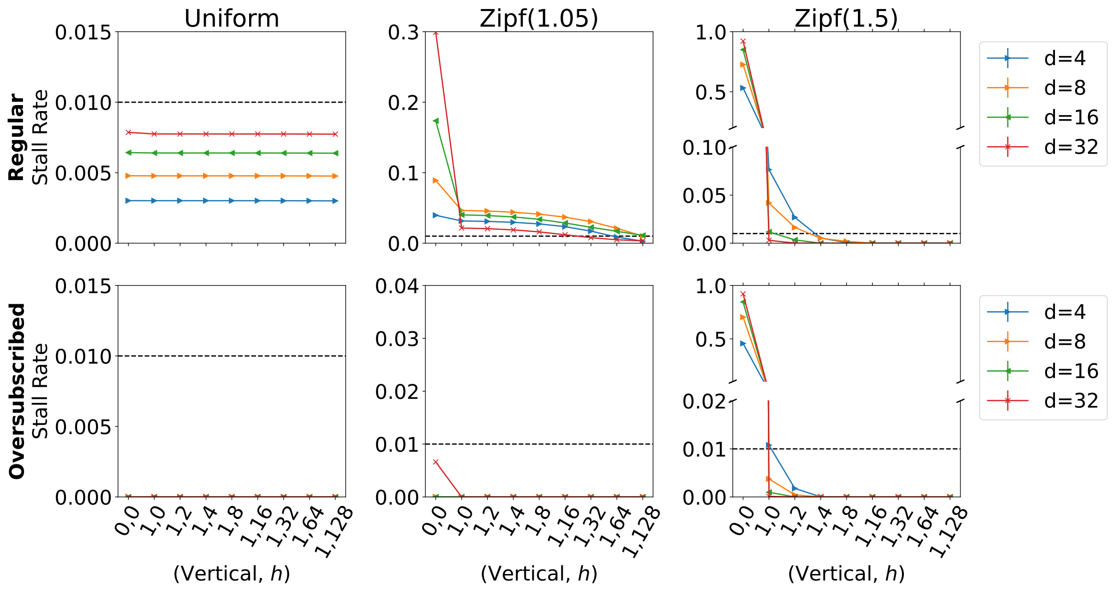
We explore the merging effort necessary to achieve low stall rates based on various real-world and artificial datasets using our simulator. This experiment allows us to select good parameters for FPGA implementations used in the following experiments. In our first experiment, we give an exhaustive overview of how the stall rates vary for datasets and increasing merging effort. We enable vertical merging before gradually increasing the horizontal merging look-ahead h. In a second experiment, we justify this strategy.

We report the mean stall rate for 200 iterations of the simulator for a sketch with one row and 2^{22} columns.

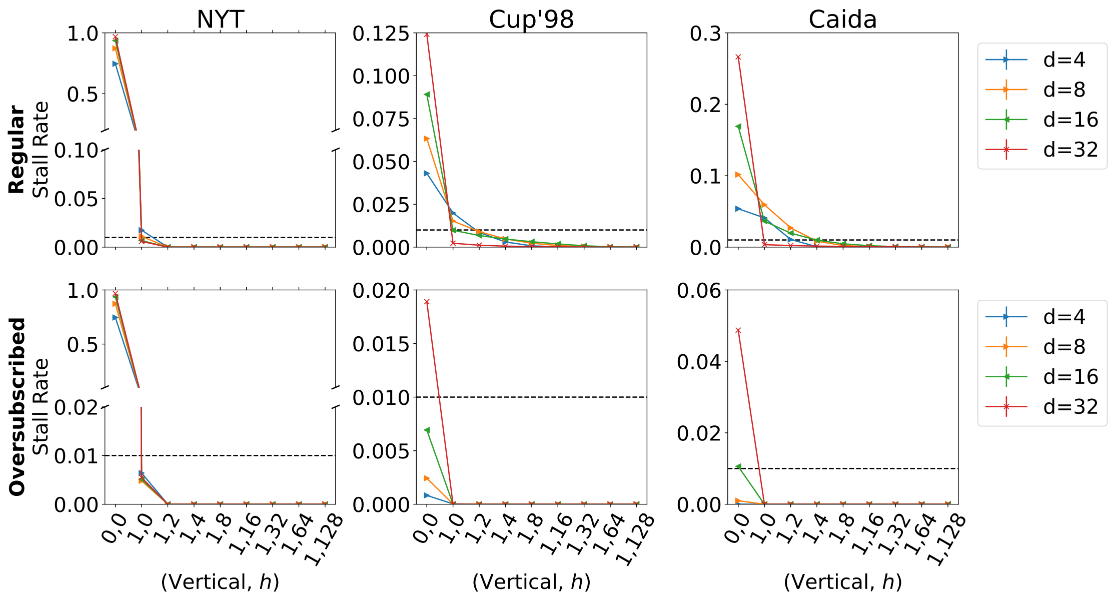
Merger Configuration

Figure 5.10 shows the effect of merging on stall rates for each dataset. As expected, a uniform distribution is not affected by merging, and the banked architecture is sufficient to achieve stall rates below 1%. The Zipfian(1.05) dataset highlights the differences between the regular and oversubscribed architecture: No merging is required for the oversubscribed architecture, as the abundance of banks prevents stalls in almost all cases. However, the regular architecture requires vertical merging and horizontal merging with a look-ahead of $h = 128$ to achieve stall rates below 1% for all values of d. The more skewed distribution in Zipf(1.5) causes the regular architecture to require less merging because the effectiveness of mergers increases. The oversubscribed architecture now requires vertical merging for stall rates below 1%, as the most common value in the distribution constitutes 38% of the dataset and cannot be handled by the banks alone. For the real-world datasets, we observe that the plot for NYT is similar to Zipf(1.5) in that stall rates without merging are above 50% and quickly drop to zero with additional merging. In this sense, Caida and Cup'98 are closer to the less skewed Zipf(1.05) dataset. This observation is intuitive as we expect pairs of taxi pickup and dropoff regions to contain obvious heavy hitters while identifiers in network traffic are less skewed.

Summary: Overall, we see that both architectures require vertical merging to achieve stall rates below 1% on all datasets. The oversubscribed architecture has stall rates close to zero when there is additional horizontal merging with a look-ahead $h = 2$ in our experiments. The regular architecture with $d \in \{4, 8, 16\}$ needs additional horizontal



(a) Artificial data



(b) Real-world data

Figure 5.10: Stall rates for optimistic architectures with increasing merging effort. For Oversubscribed, vertical and horizontal merging with $h = 2$ is sufficient to prevent stalls entirely. For Regular, we need $h = 128$ for stall rates below 1% (dotted line).

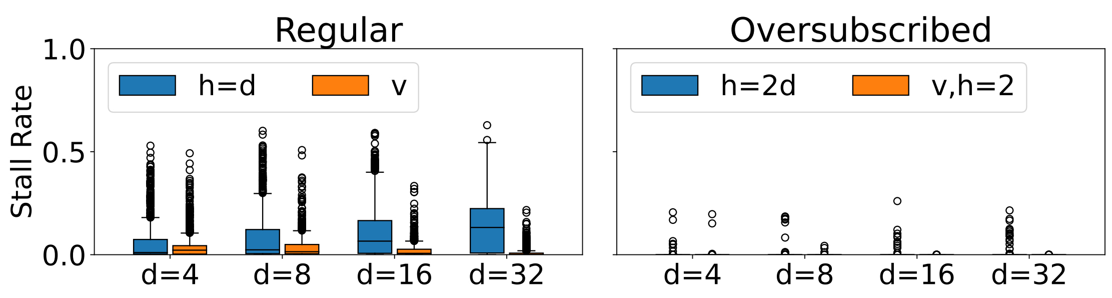


Figure 5.11: Stall rate distribution for architectures that favor vertical (v) or horizontal merging (depth h). Favoring vertical merging is more effective.

merging with $h = 128$ to achieve stall rates below 1% over all datasets. As only one dataset requires such a large look-ahead, we select $h = 64$ as a compromise between merging effort and robustness. For $d = 32$, a look-ahead of $h = 32$ suffices in all cases.

Vertical vs. Horizontal Merging

Next, we justify adding vertical merging before investing resources into the look-ahead h for horizontal merging. Figure 5.11 shows the distribution of stall rates over all datasets and iterations in a boxplot.

For a regular architecture, we compare plain vertical merging and horizontal merging with the same number of reducers ($h = d$). We see that vertical merging improves the median (except for d=4), the upper quantile, and the whiskers (1.5 IQR). The effect increases with d and is most remarkable for $d = 32$ where more than half of runs had a stall rate of 10% and higher for horizontal merging, while vertical merging reaches this value only in outliers.

For an oversubscribed architecture, we evaluate vertical merging with minimal horizontal merging ($h = 2$), as this configuration has been shown to prevent stalls almost entirely in the previous experiment. We compare against horizontal merging with $h = 2d$ resulting in the same number of reducers. While both strategies indeed suffice to prevent stalls almost entirely, favoring vertical merging before horizontal merging prevents several of the remaining outliers.

Summary: Overall, applying vertical merging first leads to more effective estimators for the same number of reducers.

5.7.2 Resource Consumption

Next, we explore the resource consumption of our optimistic architectures for the count-sketch with the dummy I/O template. We set the merging parameters as determined in the previous section. We report the consumed fraction of look-up tables, flipflops, BRAM, and URAM segmented by dispatcher, frontend, backend, and other (e.g., logic to retrieve the state). We also report the maximum operating frequency over five different Vivado implementation strategies to compensate for noise due to randomized algorithms in placement and routing. The operating frequency is likely to be an upper bound for an implementation with I/O, which introduces additional constraints and logic. Note that all reported values are attributes of the implementation and, thus, data-independent.

Optimistic vs. Pessimistic

First, we compare the pessimistic architecture to our optimistic ones. Figure 5.12 shows the pessimistic architecture for $d \in \{4, 8, 16, 32\}$ inputs targeting 80% of the available URAM compared to a regular and oversubscribed optimistic architecture with the same sketch size ($m = 1, n = \frac{4096 \cdot 1024}{d}$). Only for the oversubscribed architecture with $d = 32$ inputs, we report the results for double the number of columns because every bank needs at least one URAM memory segment. Most importantly, we see URAM consumption in the backend decreasing by the expected factor of d when comparing the optimistic and pessimistic architectures. Second, we observe the backend dominating resource consumption for the pessimistic architecture. LUTs and flipflops are between 8 and 9% for all d as the number of memory segments primarily determines the resource consumption of the whole architecture and is kept constant. For our optimistic architectures with $d \in \{4, 8, 16\}$ inputs, we see a reduction by a factor of 2.8 and 3.8 in the LUTs and flipflops consumed as the resource consumption in the backend decreases due to fewer memory segments used overall. For $d \in \{16, 32\}$ inputs, we observe a crucial implication of our optimistic architecture. While the overall resource consumption in the pessimistic architecture increases linearly with d, resource consumption for the FIFO queues and vertical merging grows quadratically for optimistic architectures. Thus, the dispatcher can dominate resource consumption for high values of d. This effect manifests for accelerators $d = 32$ for which the oversubscribed architecture consumes more LUTs than the pessimistic architecture, and the regular architecture takes 38% of available BRAM.

Figure 5.13 compares the pessimistic and optimistic architectures in terms of the maximum operating frequency. First, we observe that the maximum operating frequency for pessimistic architectures remains between 442 and 478 MHz. As the number of

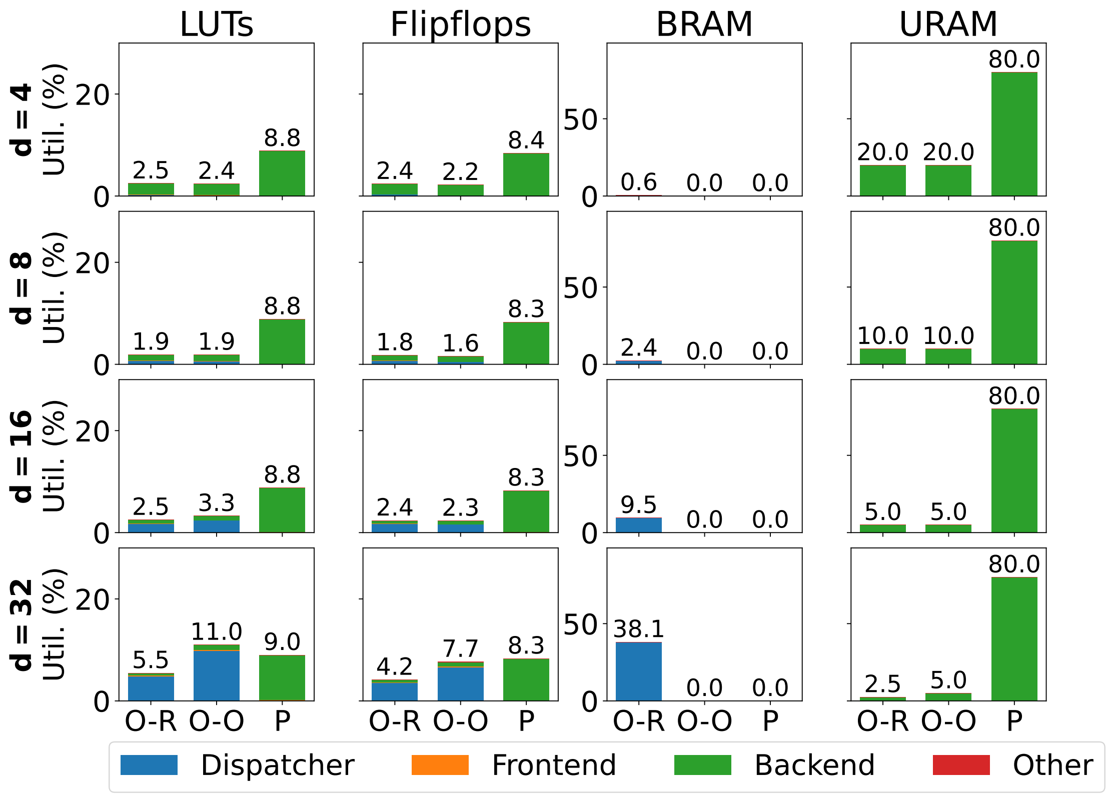


Figure 5.12: Resource utilization for a pessimistic architecture (P) consuming 80% URAM compared to regular (O-R) and oversubscribed (O-O) architectures with the same sketch size. Optimistic architectures reduce URAM utilization.

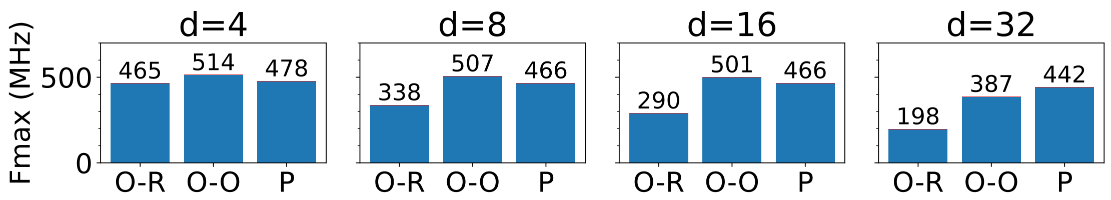


Figure 5.13: Max. operating frequency for a pessimistic architecture with 80% URAM utilization compared to regular and oversubscribed architecture with the same sketch size.

memory segments is kept constant, varying the number of inputs has only a minor impact on the complexity of the overall architecture. For $d \in \{4, 8, 16\}$ inputs, we see that an oversubscribed architecture consistently supports frequencies of more than 500 MHz, while the pessimistic architecture has a 30 to 40 MHz lower operating frequency. Thus, the additional complexity introduced by banking and merging does not outweigh the benefits of reducing the overall resource consumption. This observation turns for $d = 32$: The large dispatcher complicates placement and routing, and the maximum operating frequency for the oversubscribed architecture drops to 387 MHz, while the pessimistic architecture still operates at 442 MHz. The regular architecture does not achieve higher operating frequencies than the pessimistic architecture for any d. While the maximum operating frequency for $d = 4$ inputs is only a few percent lower, it decreases in the order of 100 MHz with every doubling of d. Routing becomes increasingly complex as an additional resource is consumed. For $d = 32$ inputs, BRAM consumption even requires the dispatcher to spread over SLRs.

Summary: We confirmed that optimistic architectures reduce state memory consumption by a factor of d while consuming fewer LUTs and flipflops for $d \in \{4, 8, 16\}$. However, we also observe that optimistic architectures do not scale arbitrarily with d. In these cases, hybrid architectures consisting of multiple optimistic replicas can reduce resource consumption for dispatchers. Furthermore, we see the oversubscribed architecture outperforming the regular architecture in terms of maximum operating frequency and overall resource consumption for this sketch and FPGA.

Merging & Reduce

Next, we investigate the impact of merging and more involved reduce functions on resource consumption. We report the resource consumption and set the number of columns to $4096 \cdot 256$ resulting in 20% URAM consumption. Figure 5.14 compares

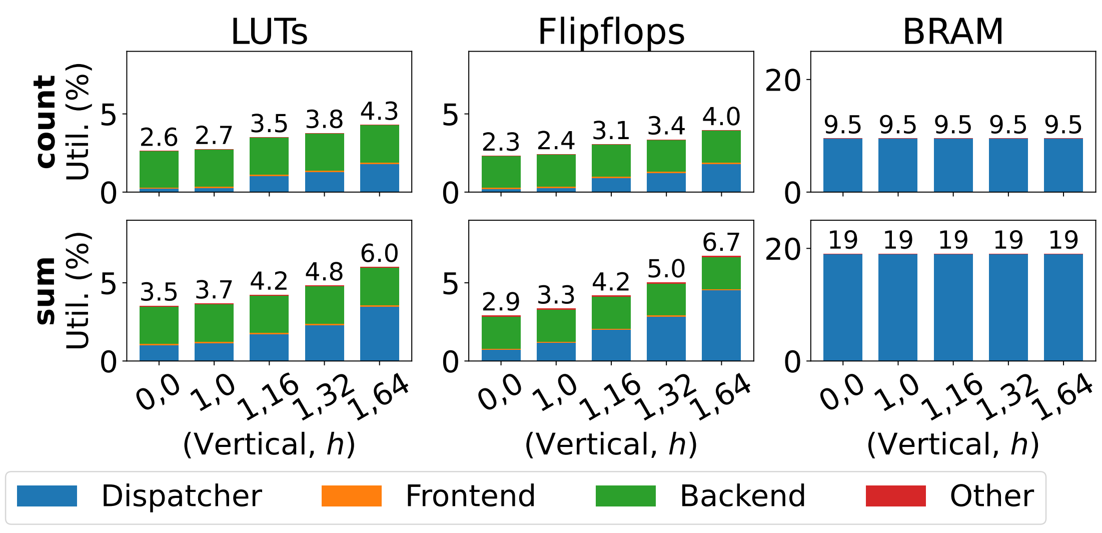


Figure 5.14: Resource utilization for regular and oversubscribed architecture with $d = 16$ utilizing 20% of available URAM for increasing merging effort.

the count-sketch with the sum-sketch for $d = 16$ inputs in a regular architecture. As incrementing state counters by a 32-bit value causes larger increments that need wider adders and FIFO queues, maintaining the sum-sketch requires more resources. We report the resource consumption for no merging, vertical merging only, and vertical merging with horizontal merging and look-ahead $h \in \{16, 32, 64\}$. While BRAM consumption for FIFO queues is twice as high for the sum-sketch, we also observe up to twice as large dispatchers in terms of LUTs and flipflops. This increase shows that the dispatcher is highly affected by the reduce function. Furthermore, we see that the look-ahead h strongly impacts both sketches. Although adding vertical merging increases LUT and flipflop consumption by 5 to 14% of the available resources, adding a horizontal merger with $h = 16$ adds up to 29%. While the added horizontal mergers with $h = 16$ have as many reducers as the vertical merger, the additional merging levels cause an overproportionate demand for resources. This observation also supports our choice to apply vertical before horizontal merging. When further doubling h, the increase is not as high since increments grow by one bit for every level in the tree of reducers.

Summary: We observe that the reduce function has a large impact on resource consumption in the dispatcher. Furthermore, horizontal merging has an overproportionate impact, as growing increments affect all following components.

Table 5.2: Accelerators implemented on U250

Architecture	Replicas per kernel	Rows	Columns
Oversubscribed	1	2	$4096 \cdot 32$
Regular	2	2	$4096 \cdot 16$
Pessimistic	8	2	$4096 \cdot 5$

5.7.3 Application

Finally, we evaluate our group-by application on a Xilinx U250 FPGA data center accelerator. Compared to the dummy I/O template, this implementation contains the logic to interface with a host computer via PCIe and to transfer data between host and device memory. This functionality consumes additional resources and complicates placement and routing of logic on the FPGA. Furthermore, it instantiates four sketching kernels operating on each DDR4 memory bank independently. Table 5.2 lists the accelerators we implemented on the U250 board.

We set the number of rows to $m = 2$ for all architectures as it is the largest number of rows for which the oversubscribed architecture fits the device without additional replication in kernels. The regular accelerator requires a hybrid architecture with two replicas per kernel to fit the device. Naturally, the pessimistic architecture requires eight replicas per kernel. To determine the number of memory segments used, we increase it until compilation fails.

Throughput

We compare regular, oversubscribed, and pessimistic accelerators to optimized data-parallel implementations on two state-of-the-art AMD EPYC CPUs and one Nvidia A100 GPU in sketching throughput. We report the mean throughput over 10 iterations for all datasets and report results for the best implementation strategy for the CPU and GPU baselines. Measurements exclude data transfer to allow for a comparison independent of limitations due to the interconnect. Error bars denote the standard deviation.

Figure 5.15 shows the results. First, we see that all FPGA accelerators achieve around 575 gbps over all datasets which is close to the theoretical optimum of 600 gbps for the 512-bit memory interface running at 300 MHz. The difference is due to the memory interface not providing new data in 4% of clock cycles. These breaks reduce pressure on FIFO queues such that no stall occurs for any dataset. Our FPGA accelerators outperform the baselines by at least 80 gbps; for some datasets, even by more than a factor of 2. The throughput of A100 varies between 320 and 494 gbps, while the EPYC CPU achieves between 223 and 419 gbps. The throughput of our FPGA accelerators is very robust with respect to the distribution of the input data and the sketch size. The performance of the software baselines varies due to caching effects and the costs of atomic operations. This is an advantage of FPGA implementations, given that we report the best implementation strategy for our software baselines and that the best strategy varies for different summary sizes and datasets.

Our FPGA implementations show one to two gbps higher throughput for the larger real-world datasets. While the throughput is independent across data distributions as no stalls occur, there is an overhead for launching kernels over PCIe. Thus, high throughput sketching with on-the-side PCIe accelerators such as our A100 and U250 requires processing data in larger batches to amortize overheads. However, FPGAs famously support processing in the data path, which would avoid such overheads [81].

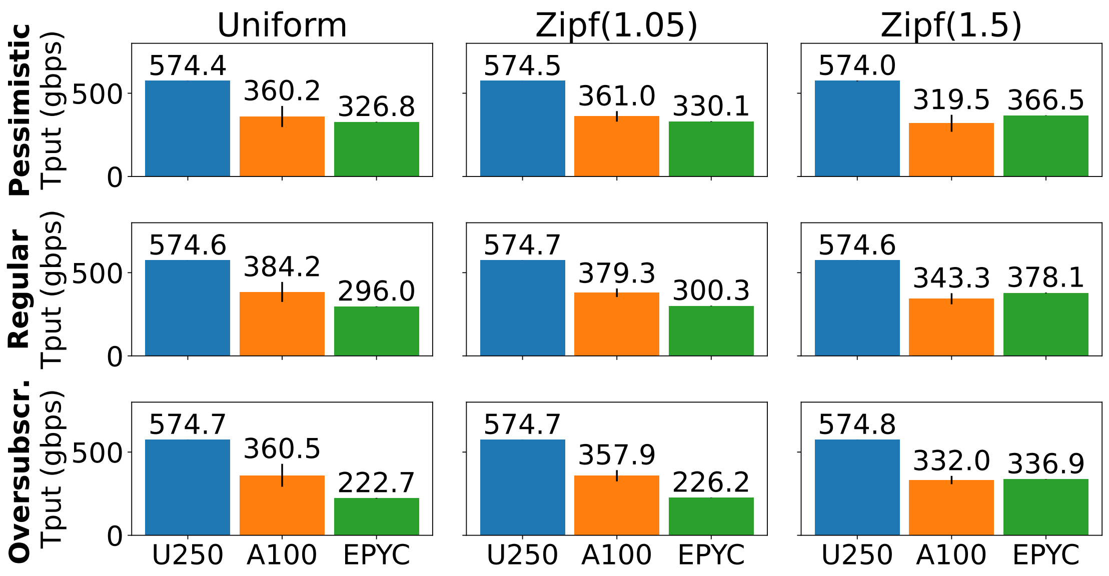
Note that our implementation supports $m > 2$ rows by making multiple passes over the input data, causing a proportionate decrease in throughput. As this behavior is the same for the software baselines, our results also apply to $m > 2$ rows.

Summary: Our optimistic accelerators outperform optimized data-parallel software implementations on a GPU and a CPU by up to 352 gbps. We have shown that our FPGA accelerators practically never stall for various artificial and real-world datasets. Their performance is robust with respect to the sketch size and data distribution, while CPU and GPU throughput varies depending on the implementation strategy, sketch size, and input data.

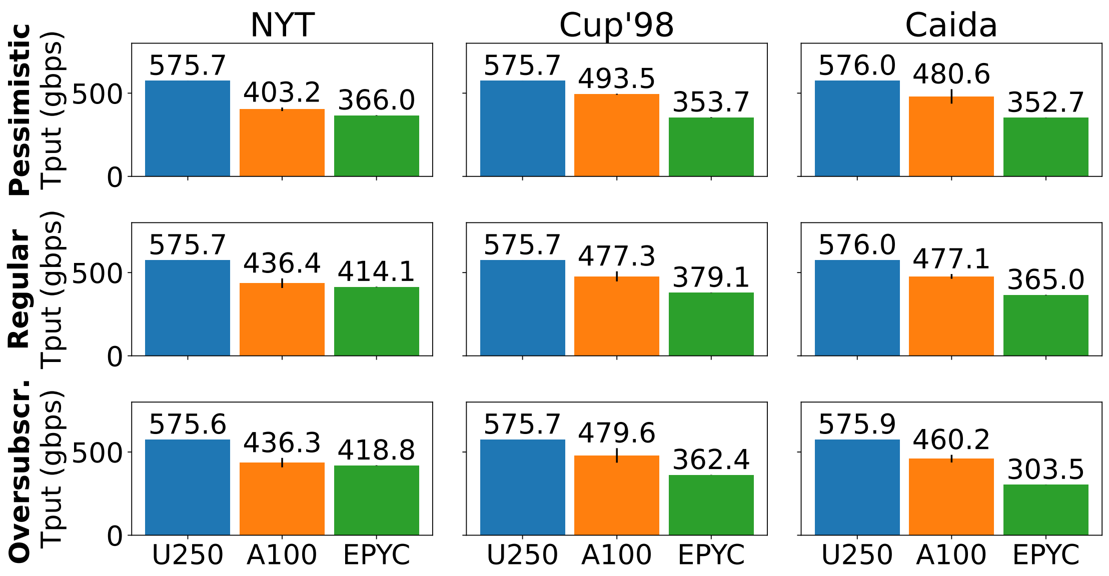
Estimation Error

We compare the accuracy of estimates in our application based on the summary sizes supported by the regular, oversubscribed, and pessimistic accelerator over our real-world datasets. We report the cumulative absolute error over 40 iterations in five scenarios. Error bars denote the standard deviation. First, we evaluate the count-sketch for the entire range of keys (Full Count). As all datasets use only a fraction of the entire domain of keys, this quantifies the ability of the sketch to exclude unseen keys from the result set. The remaining four scenarios evaluate each sketch over the set of keys (Result).

Figure 5.16 shows the results. First, we observe that the increased sketch size translates to increased accuracy in almost all cases. For Full Count, Result Count, and Result Sum, the accuracy increases by up to an order of magnitude, with oversubscribed consistently providing better accuracy. The increase in accuracy for the min and max sketches varies based on the dataset. For NYT, we see increases of an order of magnitude



(a) Artificial data



(b) Real-world data

Figure 5.15: Throughput of regular, oversubscribed, and pessimistic architecture on a U250 accelerator compared to a GPU (A100) and CPU (EPYC). FPGA accelerators provide robust throughput and outperform GPU and CPU baselines in all experiments.

for oversubscribed and regular. For Caida and Cup'98, increased sketch sizes have little impact, and accuracy varies by at most 4%. The different nature of the datasets explains this: For the NYT dataset with its few distinct keys (0.0008%), increasing the sketch size reduces the chance of few extreme groups colliding with others. Caida and WC'98 have more than 0.02% distinct groups, causing thousands of collisions in each entry. As the minimum and maximum of many groups have the same magnitude, these values end up in virtually every bucket of the min- and max-sketch. Thus, reducing the number of collisions does not lead to a significant reduction in the estimation error for min and max estimates for these datasets.

Summary: Increased sketch sizes enabled by optimistic architectures translate to up to an order of magnitude higher accuracy.

5.8 Related Work

Recent work on FPGA-accelerated sketching focuses on data analytics and exploits data parallelism. Scotch generates sketching accelerators for a broad class of matrix sketches based on the Select-Update model [89]. Furthermore, to handle single-column sketches, our original publication on Scotch proposes the Map-Apply model [89] to implement merged updates. To achieve data parallelism, the Map-Apply model requires the user to implement a merger in the apply function manually, hardcoding the degree of data parallelism $d > 1$. In this work, we replace the update function with a map and an associative reduce UDF. Thereby, we obtain a model that is suitable for both column and matrix sketches and allows generating mergers for arbitrary degrees of d. Kulkarni et al. implement the HyperLogLog (HLL) algorithm on an FPGA and parallelize it using multiple concurrent pipelines [92]. Following a similar architecture, Chiosa et al. combine HLL, CM, and FAGMS in a single FPGA accelerator [29]. All of the above techniques implement data parallelism pessimistically via replication. In contrast, our approach maintains a single replica optimistically to save resources.

Chrysos et al. explore various FPGA implementation strategies for the Exponential CM sketch [32], which maintains exponential histograms instead of regular counters in the sketch matrix. The authors exploit the inherent temporal access patterns in exponential histograms by pipelining updates to the frequently updated first bucket levels. The infrequent updates to the remaining levels are applied iteratively and require stalling the frontend. Furthermore, the authors exploit data parallelism by instantiating multiple backend replicas that operate on the same memory (BRAM and DRAM), which results in stalls for concurrent accesses to the same memory location. To avoid stalls, the

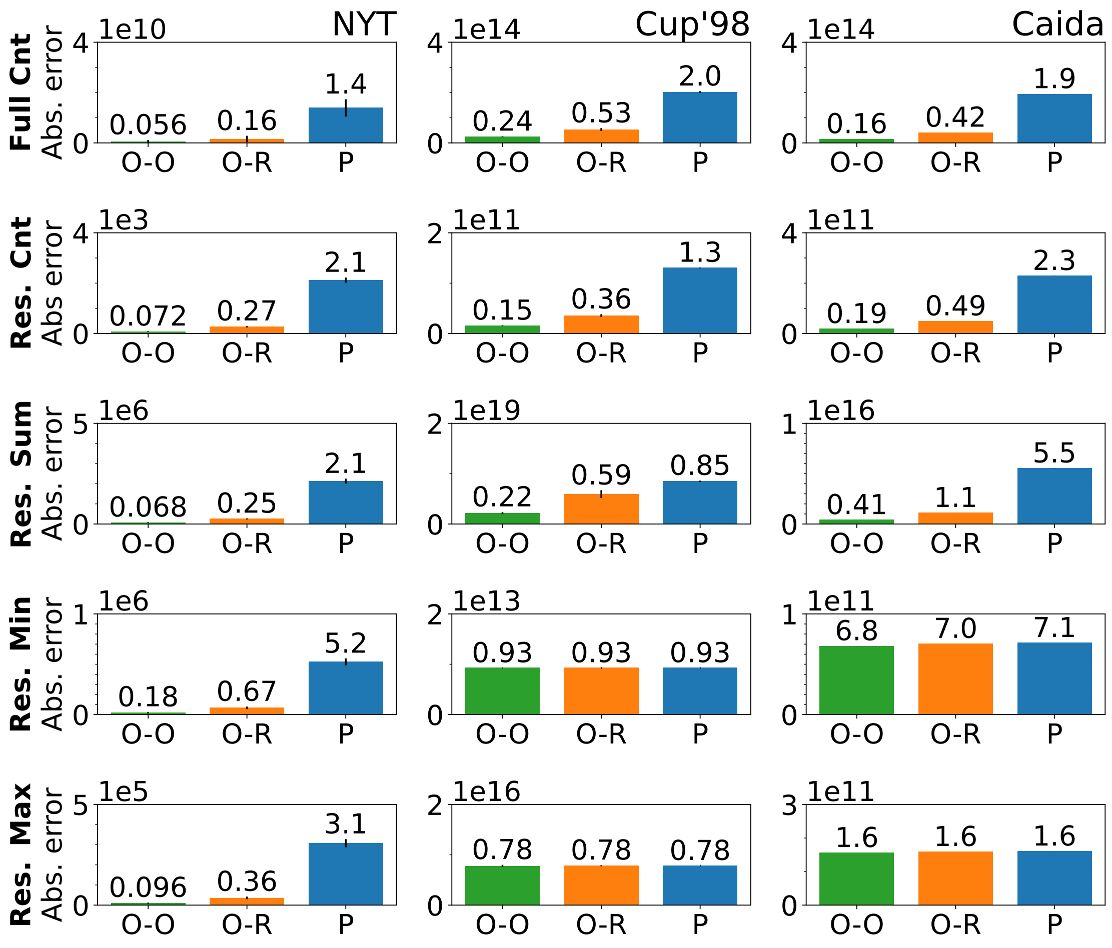


Figure 5.16: Estimation error for approximate group-by queries with summary sizes supported by oversubscribed (O-O), regular (O-R), and pessimistic (P) architectures. The larger summary sizes provided by optimistic architectures result in up to an order of magnitude improved accuracy.

authors detect heavy hitters and route them to a fixed number of additional dedicated replicas. In contrast, the algorithm class that we consider in this work has no inherent access patterns to exploit. In addition, the map and reduce functions are simpler than maintaining an exponential histogram, and the individual states consist of only a few bytes. This allows us to handle heavy hitters more efficiently by pre-partitioning values during dispatching and merging them in pipelines.

To save resources and fit more columns into BRAM, Sateesan et al. replace the simple counters in a CM sketch with approximate ones [132]. This optimization is orthogonal to the ones we propose in this work.

Several authors suggest the Map-Reduce paradigm as a general-purpose programming model for FPGAs [136, 163, 168]. However, a general-purpose Map-Reduce engine cannot assume common algorithm-specific optimizations in sketching accelerators, e.g., streaming updates to a sketch matrix. In contrast, we use map and reduce functions only to replace the update function in the Select-Update model, thereby allowing in-pipeline merging of heavy hitters.

Finally, the network community has also proposed several FPGA-accelerated sketching approaches for network monitoring [133, 140, 141, 153, 154, 169]. These approaches assume that only a few fields of each package need to be processed, and thus it is sufficient to process a single value per clock cycle. As a result, unlike our work, the above approaches do not exploit data parallelism.

5.9 Conclusion

This chapter introduces an optimistic architecture for data-parallel sketching on FPGAs that partitions the state memory in multiple banks available to all inputs. As skew in bank accesses is the Achilles' heel of this architecture and requires stalling, we propose in-pipeline merging of updates to mitigate the impact of heavy hitters by exploiting temporal locality. To enable merging, we introduce a theoretical framework that describes updates in terms of a map and a reduce function.

Compared to the pessimistic state-of-the-art that fully replicates the entire sketching unit for each of the d parallel inputs, our optimistic architecture reduces the consumption of the scarce state memory by up to a factor of d. Furthermore, we apply optimistic sketching to an approximate query processing application, observing robust sketching throughput of 575 gbps over various real-world and artificial datasets, which is up to 352 gbps higher than the throughput of state-of-the-art CPU and GPU implementations. Finally, we show that investing the saved state memory in larger sketch sizes results in better estimates up to an order of magnitude.

Overall, given the widespread availability of FPGA accelerators, our optimistic architecture paves the path towards more resource-efficient and accurate big data analytics.

6 Conclusion

In this thesis, we introduced novel techniques to accelerate approximate data analysis tasks using parallel processors. We have made three main contributions:

GPU-Accelerated KDE for Join Selectivity Estimation: We have strengthened the case for statistical coprocessing on GPUs in relational databases by generalizing feedback-optimized KDE estimators to joins. The proposed estimators support queries consisting of equijoins subject to selections on base tables. The techniques allow for computing crucial selectivity estimates over large databases with small summaries that are easy to construct and learn from query feedback. We have shown that KDE models provide accurate join selectivity estimates. Exploiting GPU acceleration reduces the estimator runtime by up to an order of magnitude and allows for at least four times larger models within the same time budget.

Generating FPGA-Accelerators for Sketching At Line Rate: We proposed a system to automatically generate and optimize FPGA accelerators for sketching with hard guarantees on the processing rate. By keeping an FPGA expert out of the loop in implementing sketching and maximizing the sketch size, the entry barrier to integrating FPGA-accelerated sketching in systems lowers significantly. We have shown that such accelerators match a hand-written implementation. Compared to parallel software implementations, we can improve throughput by up to two orders of magnitude and power consumption by up to 5.6x.

Optimistic FPGA-Accelerated Sketching: We have shown that an optimistic data-parallel sketching architecture significantly reduces the FPGA resources required to implement a broad class of sketching algorithms. While the optimistic architecture potentially requires stalling the architecture due to resource congestion, we have introduced merging techniques that largely avoid such stalls. Furthermore, we have integrated pessimistic and optimistic sketching in an approximate query processing application, showing that a recent FPGA can achieve a throughput of up to 576 GB/s. The larger summary sizes supported by optimistic architectures boost accuracy by more than an order of magnitude.

This thesis shows that parallel processing architectures can significantly improve the trade-off inherent to approximate data analysis. Using a GPU instead of a CPU allows for more accurate selectivity estimates within the same time budget; FPGA-accelerated sketching outperforms CPU and GPU in terms of throughput and energy consumption. Our work on optimistic sketching with banking and mergers improves the resource footprint of FPGA-accelerated sketching and is a compelling showcase for optimizations enabled by custom hardware.

Additionally, our work shows that abstractions and systems allow for hiding the complexity of the underlying hardware architecture from users. GPU-accelerated KDE in the query optimizer inherently hides the complexity of KDE or GPUs to database users; our approaches to FPGA-accelerated sketching lower the entry barrier to implement sketching algorithms on FPGAs for algorithms and software developers by exploiting abstractions for sketching.

6.1 Outlook

We have shown that approximate data analysis combined with parallel processors is a powerful tool to increase the efficiency of data analysis tasks. We believe that we will see more applications exploiting this combination in the future: Besides functionality and saving costs in the presence of evergrowing data volumes [142], there is a trend toward *green computing* focusing on performance per Watt [46, 107]. With the power consumption of IT systems being identified as a significant contributor to CO2 emissions [54], we expect energy efficiency to become even more crucial. Green computing has even manifested in consumer CPUs as Intel offers consumer CPUs both providing *performance cores* optimized for raw performance and *efficient cores* optimized for performance per Watt [45]. Thus, maximizing energy efficiency by performing data analyses approximately and on specialized parallel processors can be an answer to the demands of future data analysis systems. We gladly contribute to this effort with this thesis.

6.1.1 Future Work

We hope our work inspires future research in the area. In the following, we propose three possible directions for future work:

Domain-Specific Abstractions for Data Processing on FPGAs: As shown with our work on FPGA-accelerated sketching [89], generalizing abstractions help reduce implementation effort as frameworks and libraries can provide common functionality. Designing abstractions to accelerate other data processing tasks could help employ FPGAs in data-intensive applications while increasing maintainability, e.g., compression, encoding, encryption, or user-defined transformations in the data path.

Lightweight Kernel Density Estimation: This thesis and previous publications [70, 71, 88] have primarily shown that feedback-driven KDE-based selectivity estimation is viable. While our evaluation shows that GPU acceleration speeds up the evaluation and optimization of our KDE models, the technique could benefit from additional research into reducing computational overhead. If overheads do not dominate the estimators, the error function required for the Gaussian kernel dominates the execution time. Plugging in an approximation for the error function or choosing a different kernel could substantially speed up evaluation and training.

Approximate Stream Processing through Sketching at the Edge: Our experiments on approximate query processing have shown promising results. Thus, building an approximate stream processing system that operates on sketches constructed close to the data sources (also referred to as *edges*) would be interesting. With multi-core CPUs, FPGAs, and GPU-based systems on chip available as edge devices providing high performance per watt, queries could be answered quicker and cheaper with fewer data movement and computational pressure on intermediate nodes. Several exciting research problems arise for such a system, e.g., selecting sketches based on a given query, handling accuracy constraints, implementing windowing efficiently, and switching between dense and sparse representations of a sketch matrix for transfer.

A

Appendix: Derivations for KDE-based Join Selectivity Estimations

A.1 Gaussian Cross Contribution

The Gaussian kernel is a common choice for a continuous kernel in KDE. It is a normal distribution $\mathcal{N}_{s,\delta^2}(x)$ centered on the sample point s and using the bandwidth δ as its standard deviation. In order to compute discretized probability estimates for an integer join key v from this continuous kernel, we simply integrate it over the interval $[v-0.5, v+0.5]$, leaving us with the *discretized Gaussian kernel*. We assume that join attributes are not subject to selections ($A = \mathbb{Z}$) and lift this assumption in Appendix A.2.

A.1.1 Cross Contribution

Substituting the discretized Gaussian kernel into Equation 3.6, yields the cross contribution for the Gaussian kernel:

$$\hat{J}_{i,j} = \sum_{v\in\mathbb{Z}} \int_{v-0.5}^{v+0.5} \mathcal{N}_{t_1^{(i)},\delta_1^2}(x)\, dx \cdot \int_{v-0.5}^{v+0.5} \mathcal{N}_{t_2^{(j)},\delta_2^2}(x)\, dx \tag{A.1}$$

Equation (A.1) does not have a closed-form solution that would allow us to efficiently compute it without summing over all $v \in A$. However, for probability densities $f(x)$, $g(x)$ with values smaller or equal to one, we can approximate $\int_{v-0.5}^{v+0.5} f(x)\, dx \int_{v-0.5}^{v+0.5} g(x)\, dx \approx \int_{v-0.5}^{v+0.5} f(x)\, g(x)\, dx$. We illustrate this approximation in Figure A.1, showing that in this case, the results for first integrating and then multiplying are very similar to first multiplying and then integrating.

Substituting this approximation into Equation (A.1) gives us an approximate closed-form solution:

$$\begin{aligned} \hat{J}_{i,j} &\approx \sum_{v\in\mathbb{Z}} \int_{v-0.5}^{v+0.5} \mathcal{N}_{t_1^{(i)},\delta_1^2} \cdot \mathcal{N}_{t_2^{(j)},\delta_2^2} dx \\ &= \int_{-\infty}^{+\infty} \mathcal{N}_{t_1^{(i)},\delta_1^2}(x) \cdot \mathcal{N}_{t_2^{(j)},\delta_2^2}(x)\, dx \\ &= \mathcal{N}_{t_1^{(i)},(\delta_1^2+\delta_2^2)}\left(t_2^{(j)}\right) \int_{-\infty}^{+\infty} \mathcal{N}_{t',\delta'}(x) \\ &= \mathcal{N}_{t_1^{(i)},(\delta_1^2+\delta_2^2)}\left(t_2^{(j)}\right) \end{aligned} \tag{A.2}$$

The last transformation requires some explanation: The product $\mathcal{N}_{1,2} = \mathcal{N}_1 \cdot \mathcal{N}_2$ of two normal probability density functions is itself a scaled normal probability density function [19]. Its location t' and scale δ' are functions over the parameters of the individual densities. Since we integrate over the full domain, this factor integrates to one, leaving only the scaling factor, which is again given by a normal density function that depends on the mean and bandwidth parameters of the original functions [19].

Equation (A.2) does not hold for densities with function values larger than one. In particular, when the bandwidth of one of the two Gaussian functions is below $(2\pi)^{-\frac{1}{2}} \approx 0.4$, the error increases drastically. However, since for this value only 1.3% of the probability mass of a Gaussian is located outside of $[\mu - 0.5, \mu + 0.5]$, the estimator is still able to fall back close to sample evaluation.

A.1.2 Generalized Cross Contribution

Plugging the discretized Gaussian kernel into Equation (3.8), we arrive at:

$$\hat{J}_{o_1,\ldots,o_n} = \sum_{v\in\mathbb{Z}} \prod_{i=1}^{n} \int_{v-0.5}^{v+0.5} \mathcal{N}_{t_i^{(o_i)},\delta_i^2}(x)\, dx \tag{A.3}$$

Similar to the single join case, we can plug in the approximation $\prod_i \int_{v-0.5}^{v+0.5} p_i(x)dx \approx \int_{v-0.5}^{v+0.5} \prod_i p_i(x)dx$. The product of k normal densities can be rewritten as a normal density

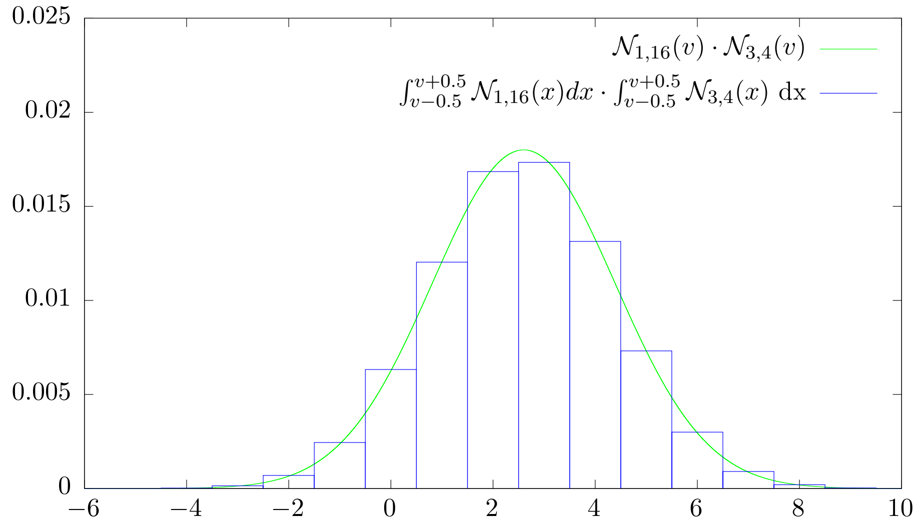


Figure A.1: Approximating the cross contribution for the Gaussian kernel by the multiply-then-integrate approach

and a scale factor that does not depend on the integrand [19].

$$\begin{aligned} \hat{J}_{o_1,\ldots,o_n} &= \int_{-\infty}^{\infty} \prod_{i=1}^{n} \mathcal{N}_{t_i^{(o_i)},\delta_i^2}(x)\, dx \\ &= S_{1\ldots n} \cdot \int_{-\infty}^{\infty} \mathcal{N}_{t_{1\ldots n},\delta_{1\ldots n}^2}(x)dx \\ &= S_{1\ldots n} \end{aligned} \tag{A.4}$$

Since we integrate over the entire domain, the normal density integrates to one, which leaves us with the scale factor $S_{1\ldots n}$ only. The scale factor $S_{1\ldots n}$ is given by:

$$S_{1\ldots n} = \frac{\sqrt{\frac{\delta_{1\ldots n}^2}{\prod_{i=1}^{n} \delta_i^2}}}{(2\pi)^{(n-1)/2}} exp\left(-\frac{1}{2}\left(\sum_{i=1}^{n} \frac{\left(t_i^{(o_i)}\right)^2}{\delta_i^2} - \frac{t_{1\ldots n}^2}{\delta_{1\ldots n}^2}\right)\right) \tag{A.5}$$

Finally, the quantities $\delta^2_{1\ldots n}$ and $t_{1\ldots n}$ can be computed from the individual means and variances of each normal density:

$$\begin{aligned} \delta^2_{1\ldots n} &= \left(\sum_{i=1}^{n} \frac{1}{\delta_i^2} \right)^{-1} \\ t_{1\ldots n} &= \delta^2_{1\ldots n} \sum_{i=1}^{n} \frac{t_i^{(o_i)}}{\delta_i^2} \end{aligned} \tag{A.6}$$

A.2 Selections On Join Attributes

The assumption that join attributes are not subject to selections can be lifted. In general, selections on the join attributes allow us to apply sample pruning for join attributes as well which potentially reduces the number of tuples that we have to consider in the cross pruning step.

Range selections require adjusting the equations for the cross contribution. If a range selection $l \leq A \leq u$ is applied to a join attribute, the normal density function in Equation A.4 has to be considered:

$$\hat{J}_{o_1,\ldots,o_n} = S_{1\ldots n} \cdot \int_l^u \mathcal{N}_{t_{1\ldots n}, \delta^2_{1\ldots n}}(x)dx \tag{A.7}$$

However, we do not have to adjust the maximum distance inequality (Equation 3.10) in cross pruning. As the additionally introduced factor $\int_l^u \mathcal{N}_{t',\delta'}(x)$ is smaller or equal to one, the given inequality remains intact even in case of range predicates on the join attribute.

Point selections are a special case of range selections. As a join attribute subject to a point selection reduces the estimate computation to multiplying estimates for a single selection per table, treating them differently simplifies the required computations: Point selections allow us to handle the joins for an entire equivalence class by local predicates. Furthermore, every table with all join attributes being subject to point selections can even be excluded entirely from the sum over the cross product of all samples.

List of Figures

List of Tables

List of Listings

Bibliography

[1] Swarup Acharya, Phillip B. Gibbons, Viswanath Poosala, and Sridhar Ramaswamy. 1999. The Aqua Approximate Query Answering System. *SIGMOD Record* 28, 2 (1999), 574–576.

[2] Pankaj K. Agarwal, Graham Cormode, Zengfeng Huang, Jeff Phillips, Zhewei Wei, and Ke Yi. 2012. Mergeable Summaries. In *Proceedings of the 31st ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*. 23–34.

[3] Sameer Agarwal, Barzan Mozafari, Aurojit Panda, Henry Milner, Samuel Madden, and Ion Stoica. 2013. BlinkDB: Queries with Bounded Errors and Bounded Response Times on Very Large Data. In *Proceedings of the 8th ACM European Conference on Computer Systems*. 29–42.

[4] Noga Alon, Phillip Gibbons, Yossi Matias, and Mario Szegedy. 1999. Tracking Join and Self-join Sizes in Limited Storage. In *Proceedings of the Eighteenth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*. 10–20.

[5] Noga Alon, Yossi Matias, and Mario Szegedy. 1996. The Space Complexity of Approximating the Frequency Moments. In *Proceedings of the Twenty-Eighth Annual ACM Symposium on Theory of Computing*. 20–29.

[6] Amd. 2022. *UltraScale Architecture and Product Data Sheet: Overview DS890 (v4.2)*. Retrieved December 7, 2022 from `https://www.xilinx.com/content/dam/xilinx/support/documents/data_sheets/ds890-ultrascale-overview.pdf`

[7] M. Arlitt and T. Jin. 2000. A workload characterization study of the 1998 World Cup Web site. *IEEE Network* 14, 3 (2000), 30–37.

[8] Gerassimos Barlas. 2022. *Multicore and GPU programming* (2 ed.). Morgan Kaufmann.

[9] Nathan Bell and Jared Hoberock. 2012. Chapter 26 - Thrust: A Productivity-Oriented Library for CUDA. In *GPU Computing Gems Jade Edition*. Morgan Kaufmann, 359–371.

[10] Bitmain. 2022. *Bitmain*. Retrieved December 7, 2022 from `https://www.bitmain.com/`

[11] Burton H. Bloom. 1970. Space/Time Trade-Offs in Hash Coding with Allowable Errors. *Communications of the ACM* 13, 7 (1970), 422–426.

[12] Mark Bohr. 2007. A 30 year retrospective on Dennard's MOSFET scaling paper. *IEEE Solid-State Circuits Society Newsletter* 12, 1 (2007), 11–13.

[13] Pradip Bose. 2011. *Encyclopedia of Parallel Computing - Power Wall*. Springer US, 1593–1608.

[14] Léon Bottou. 2004. Stochastic Learning. In *Advanced Lectures on Machine Learning*. Springer Verlag, 146–168.

[15] Adrian W. Bowman. 1984. An Alternative Method of Cross-Validation for the Smoothing of Densitfy Estimates. *Biometrika* 71, 2 (1984), 353–360.

[16] Sebastian Breß, Henning Funke, and Jens Teubner. 2016. Robust Query Processing in Co-Processor-accelerated Databases. In *Proceedings of the 2016 International Conference on Management of Data*. 1891–1906.

[17] Sebastian Breß, Bastian Köcher, Henning Funke, Steffen Zeuch, Tilmann Rabl, and Volker Markl. 2018. Generating Custom Code for Efficient Query Execution on Heterogeneous Processors. In *The VLDB Journal*, Vol. 27. 797–822.

[18] A. Z. Broder. 1997. On the Resemblance and Containment of Documents. In *Proceedings of the Compression and Complexity of SEQUENCES 1997*. 21–29.

[19] Paul Bromiley. 2003. Products and Convolutions of Gaussian Probability Density Functions. *Tina-Vision Memo* 3, 4 (2003), 1.

[20] P. Bruel, A. Goldman, S. R. Chalamalasetti, and D. Milojicic. 2017. Autotuning high-level synthesis for FPGAs using OpenTuner and LegUp. In *2017 International Conference on ReConFigurable Computing and FPGAs*. 1–6.

[21] Nicolas Bruno, Surajit Chaudhuri, and Luis Gravano. 2001. STHoles: A Multidimensional Workload-Aware Histogram. *SIGMOD Record* 30, 2 (2001), 211–222.

[22] Fabrizio Tappero Bryan Mealy. 2019. *Free Range VHDL*. Retrieved December 7, 2022 from `https://github.com/fabriziotappero/Free-Range-VHDL-book`

[23] Caida. 2019. *Anonymized Internet Traces 2019*. Retrieved December 7, 2022 from `https://catalog.caida.org/details/dataset/passive_2019_pcap`

[24] Ufuk Celebi. 2015. *How Apache Flink Handles Backpressure*. Retrieved December 7, 2022 from `https://www.ververica.com/blog/how-flink-handles-backpressure`

[25] Kaushik Chakrabarti, Minos Garofalakis, Rajeev Rastogi, and Kyuseok Shim. 2001. Approximate Query Processing using Wavelets. *The VLDB Journal* 10, 2 (2001), 199–223.

[26] Surajit Chaudhuri, Bolin Ding, and Srikanth Kandula. 2017. Approximate Query Processing: No Silver Bullet. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 511–519.

[27] Surajit Chaudhuri, Rajeev Motwani, and Vivek Narasayya. 1999. On Random Sampling over Joins. In *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data*. 263–274.

[28] D. Chen and D. Singh. 2012. Invited paper: Using OpenCL to Evaluate the Efficiency of CPUs, GPUs and FPGAs for Information Filtering. In *22nd International Conference on Field Programmable Logic and Applications*. 5–12.

[29] Monica Chiosa, Thomas Preußer, and Gustavo Alonso. 2021. SKT: A One-Pass Multi-Sketch Data Analytics Accelerator. *Proceedings of the VLDB Endowment* 14, 11 (2021), 2369–2382.

[30] J. M. Cho and K. Choi. 2014. An FPGA Implementation of High-Throughput Key-Value Store using Bloom Filter. In *Technical Papers of 2014 International Symposium on VLSI Design, Automation and Test*. 1–4.

[31] Stavros Christodoulakis. 1984. Implications of Certain Assumptions in Database Performance Evauation. *ACM Transactions on Database Systems* 9, 2 (1984), 163–186.

[32] G. Chrysos, O. Papapetrou, D. Pnevmatikatos, A. Dollas, and M. Garofalakis. 2019. Data Stream Statistics Over Sliding Windows: How to Summarize 150 Million Updates Per Second on a Single Node. In *29th International Conference on Field Programmable Logic and Applications*. 278–285.

[33] Google Cloud. 2022. *Cloud Tensor Processing Units (TPUs)*. Retrieved December 7, 2022 from `https://cloud.google.com/tpu/docs/tpus`

[34] Saar Cohen and Yossi Matias. 2003. Spectral Bloom Filters. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*. 241–252.

[35] Stuart Colville. 2020. *Mozilla Add-ons Community Blog - Introducing a scalable add-ons blocklist*. Retrieved December 7, 2022 from `https://blog.mozilla.org/addons/2020/08/24/introducing-a-scalable-add-ons-blocklist/`

[36] Jason Cong, Jason Lau, Gai Liu, Stephen Neuendorffer, Peichen Pan, Kees Vissers, and Zhiru Zhang. 2022. FPGA HLS Today: Successes, Challenges, and Opportunities. *ACM Transactions on Reconfigurable Technology and Systems* 15, 4 (2022).

[37] Graham Cormode and Minos Garofalakis. 2005. Sketching Streams through the Net: Distributed Approximate Query Tracking. In *Proceedings of the 31st International Conference on Very Large Data Bases*. 13–24.

[38] Graham Cormode, Minos Garofalakis, Peter J. Haas, and Chris Jermaine. 2012. Synopses for Massive Data: Samples, Histograms, Wavelets, Sketches. *Foundations and Trends in Databases* 4, 1–3 (2012), 1–294.

[39] Graham Cormode and S. Muthukrishnan. 2005. An Improved Data Stream Summary: The Count-Min Sketch and Its Applications. *Journal of Algorithms* 55, 1 (2005), 58–75.

[40] Oracle Corporation. 2022. *Oracle's SPARC T8 and SPARC M8 Server Architecture*. Technical Report. Oracle Corporation.

[41] T. S. Czajkowski, U. Aydonat, D. Denisenko, J. Freeman, M. Kinsner, D. Neto, J. Wong, P. Yiannacouras, and D. P. Singh. 2012. From OpenCL to High-Performance Hardware on FPGAs. In *22nd International Conference on Field Programmable Logic and Applications*. 531–534.

[42] Robert H Dennard, Fritz H Gaensslen, Hwa-Nien Yu, V Leo Rideout, Ernest Bassous, and Andre R LeBlanc. 1974. Design of Ion-Implanted MOSFET's with very Small Physical Dimensions. *IEEE Journal of Solid-State Circuits* 9, 5 (1974), 256–268.

[43] Alin Dobra, Minos Garofalakis, Johannes Gehrke, and Rajeev Rastogi. 2016. *Sketch-Based Multi-Query Processing over Data Streams*. 241–261.

[44] Tarn Duong and Martin L. Hazelton. 2005. Cross-Validation Bandwidth Matrices for Multivariate Kernel Density Estimation. *Scandinavian Journal of Statistics* 32, 3 (2005), 485–506.

[45] IBM Cloud Education. 2022. *How 13th Gen Intel® Core™ Processors Work?* Retrieved December 7, 2022 from `https://www.intel.com/content/www/us/en/gaming/resources/how-hybrid-design-works.html`

[46] IBM Cloud Education. 2022. *What Is Green Computing?* Retrieved December 7, 2022 from `https://blogs.nvidia.com/blog/2022/10/12/what-is-green-computing/`

[47] V. A. Epanechnikov. 1969. Non-Parametric Estimation of a Multivariate Probability Density. *Theory of Probability & Its Applications* 14, 1 (1969), 153–158.

[48] C. Estan and J.F. Naughton. 2006. End-biased Samples for Join Cardinality Estimation. In *22nd International Conference on Data Engineering (ICDE'06)*.

[49] Cristian Estan and George Varghese. 2003. New Directions in Traffic Measurement and Accounting: Focusing on the Elephants, Ignoring the Mice. *ACM Transactions on Computer Systems* 21, 3 (2003), 270–313.

[50] Joan Feigenbaum, Sampath Kannan, Martin J. Strauss, and Mahesh Viswanathan. 2003. An Approximate L1-Difference Algorithm for Massive Data Streams. *SIAM Journal on Computing* 32, 1 (2003), 131–151.

[51] Tom Feist. 2012. Vivado Design Suite. *Xilinx White Paper* 5 (2012), 30.

[52] Philippe Flajolet, Éric Fusy, Olivier Gandouet, and Frédéric Meunier. 2007. Hyperloglog: The Analysis of a Near-Optimal Cardinality Estimation Algorithm. In *Proceedings of the 2007 Conference on Analysis of Algorithms*. 127–146.

[53] Philippe Flajolet and G Nigel Martin. 1985. Probabilistic Counting Algorithms for Data Base Applications. *Journal of Computer and System Sciences*, 31, 2 (1985), 182–209.

[54] Charlotte Freitag, Mike Berners-Lee, Kelly Widdicks, Bran Knowles, Gordon S. Blair, and Adrian Friday. 2021. The Real Climate and Transformative Impact of ICT: A Critique of Estimates, Trends, and Regulations. *Patterns* 2, 9 (2021).

[55] Edward Gan, Jialin Ding, Kai Sheng Tai, Vatsal Sharan, and Peter Bailis. 2018. Moment-based Quantile Sketches for Efficient High Cardinality Aggregation Queries. *Proceedings of the VLDB Endowment* 11, 11 (2018), 1647–1660.

[56] Sumit Ganguly, Phillip Gibbons, Yossi Matias, and Avi Silberschatz. 1996. Bifocal Sampling for Skew-Resistant Join Size Estimation. *SIGMOD Record* 25, 2 (1996), 271–281.

[57] Hector Garcia-Molina, Jeffrey D Ullman, and Jennifer Widom. 2008. *Database Systems* (2 ed.). Pearson.

[58] Minos Garofalakis, Johannes Gehrke, and Rajeev Rastogi (Eds.). 2018. *Data Stream Management*. Springer.

[59] Lise Getoor, Benjamin Taskar, and Daphne Koller. 2001. Selectivity Estimation Using Probabilistic Models. *SIGMOD Record* 30, 2 (2001), 461–472.

[60] Amit Goyal, Hal Daumé, and Graham Cormode. 2012. Sketch Algorithms for Estimating Point Queries in NLP. In *Proceedings of the 2012 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning*. 1093–1103.

[61] Khronos OpenCL Working Group. 2022. *The OpenCL Specification Version 3.0*. Retrieved December 7, 2022 from `https://registry.khronos.org/OpenCL/specs/3.0-unified/pdf/OpenCL_API.pdf`

[62] Dimitrios Gunopulos, George Kollios, J Tsotras, and Carlotta Domeniconi. 2005. Selectivity Estimators for Multidimensional Range Queries over Real Atributes. *The VLDB Journal* 14, 2 (2005), 137–154.

[63] Dimitrios Gunopulos, George Kollios, Vassilis J. Tsotras, and Carlotta Domeniconi. 2000. Approximating Multi-Dimensional Aggregate Range Queries over Real Attributes. *SIGMOD Record* 29, 2 (2000), 463–474.

[64] Sumit Gupta. 2015. *What Is NVLink? And How Will It Make the World's Fastest Computers Possible?* Nvidia. Retrieved December 7, 2022 from `https://blogs.nvidia.com/blog/2014/11/14/what-is-nvlink/`

[65] Bala Gurumurthy, David Broneske, Tobias Drewes, Thilo Pionteck, and Gunter Saake. 2018. Cooking DBMS Operations using Granular Primitives. *Datenbank-Spektrum* 18, 3 (2018), 183–193.

[66] Peter Haas, Jeffrey Naughton, and Arun Swami. 1994. On the Relative Cost of Sampling for Join Selectivity Estimation. In *Proceedings of the Thirteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*. 14–24.

[67] Bingsheng He, Mian Lu, Ke Yang, Rui Fang, Naga K. Govindaraju, Qiong Luo, and Pedro V. Sander. 2009. Relational Query Coprocessing on Graphics Processors. *ACM Transactions on Database Systems* 34, 4 (2009).

[68] Bingsheng He, Ke Yang, Rui Fang, Mian Lu, Naga Govindaraju, Qiong Luo, and Pedro Sander. 2008. Relational Joins on Graphics Processors. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*. 511–524.

[69] Max Heimel. 2018. *Data Processing on Heterogeneous Hardware*. Ph. D. Dissertation. Technische Universität Berlin.

[70] Max Heimel, Martin Kiefer, and Volker Markl. 2015. Self-Tuning, GPU-Accelerated Kernel Density Models for Multidimensional Selectivity Estimation. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. 1477–1492.

[71] Max Heimel and Volker Markl. 2012. A First Step Towards GPU-assisted Query Optimization. In *International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures*. 33–44.

[72] Max Heimel, Michael Saecker, Holger Pirk, Stefan Manegold, and Volker Markl. 2013. Hardware-Oblivious Parallelism for In-Memory Column-Stores. *Proceedings of the VLDB Endowment* 6, 9 (2013), 709–720.

[73] Christoph Heinz. 2007. *Density Estimation Over Data Streams*. Ph. D. Dissertation. Philipps-Universität Marburg.

[74] John L Hennessy and David A Patterson. 2011. *Computer Architecture* (5 ed.). Morgan Kaufmann.

[75] Ihab Ilyas, Volker Markl, Peter Haas, Paul Brown, and Ashraf Aboulnaga. 2004. CORDS: Automatic Discovery of Correlations and Soft Functional Dependencies. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*. 647–658.

[76] Intel. 2022. *FPGA Optimization Guide for Intel® oneAPI Toolkits (Rev. 13)*. Retrieved December 7, 2022 from `https://www.intel.com/content/dam/develop/external/us/en/documents/oneapi-dpcpp-fpga-optimization-guide.pdf`

[77] Intel. 2022. *Intel FPGA Acceleration Card Solutions*. Retrieved December 7, 2022 from `https://www.xilinx.com/products/boards-and-kits/alveo.html`

[78] Intel. 2022. *Intel® Agilex™ Logic Array Blocks and Adaptive Logic Modules User Guide (2022.05.24)*. Retrieved December 7, 2022 from `https://www.intel.com/programmable/technical-pdfs/683577.pdf`

[79] Intel. 2022. *OneAPI Specification 1.2*. Retrieved December 7, 2022 from `https://spec.oneapi.io/versions/1.2-rev-1/`

[80] Yannis Ioannidis and Stavros Christodoulakis. 1991. On the Propagation of Errors in the Size of Join Results. In *Proceedings of the 1991 ACM SIGMOD International Conference on Management of Data*. 268–277.

[81] Zsolt Istvan, Kaan Kara, and David Sidler. 2020. *FPGA-Accelerated Analytics*. now.

[82] Zsolt Istvan, Louis Woods, and Gustavo Alonso. 2014. Histograms as a Side Effect of Data Movement for Big Data. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*. 1567–1578.

[83] Yesdaulet Izenov, Asoke Datta, Florin Rusu, and Jun Hyung Shin. 2021. COMPASS: Online Sketch-Based Query Optimization for In-Memory Databases. In *Proceedings of the 2021 International Conference on Management of Data*. 804–816.

[84] H. V. Jagadish, Nick Koudas, S. Muthukrishnan, Viswanath Poosala, Kenneth C. Sevcik, and Torsten Suel. 1998. Optimal Histograms with Quality Guarantees. In *Proceedings of the 24rd International Conference on Very Large Data Bases*. 275–286.

[85] Steven G. Johnson. 2014. *The NLopt nonlinear-optimization package*. Retrieved December 7, 2022 from `http://ab-initio.mit.edu/nlopt`

[86] Vinod Kathail. 2020. Xilinx Vitis Unified Software Platform. In *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 173–174.

[87] Martin Kiefer, Max Heimel, Sebastian Breß, and Volker Markl. 2017. Estimating Join Selectivities Using Bandwidth-Optimized Kernel Density Models. *Proceedings of the VLDB Endowment* 10, 13 (2017), 2085–2096.

[88] Martin Kiefer, Max Heimel, and Volker Markl. 2015. Demonstrating Transfer-Efficient Sample Maintenance on Graphics Cards. In *Proceedings of the 18th International Conference on Extending Database Technology*. 513–516.

[89] Martin Kiefer, Ilias Poulakis, Sebastian Breß, and Volker Markl. 2020. Scotch: Generating FPGA-Accelerators for Sketching at Line Rate. *Proceedings of the VLDB Endowment* 14, 3 (2020), 281–293.

[90] Martin Kiefer, Ilias Poulakis, Eleni Tzirita Zacharatou, and Volker Markl. 2023. Optimistic Data Parallelism for FPGA-Accelerated Sketching. *Proceedings of the VLDB Endowment* 16, 5 (2023), 1113–1125.

[91] Colian Ian King. 2020. *Powerstat*. Retrieved December 7, 2022 from `https://github.com/ColinIanKing/powerstat`

[92] Amit Kulkarni, Monica Chiosa, Thomas B. Preußer, Kaan Kara, David Sidler, and Gustavo Alonso. 2020. HyperLogLog Sketch Acceleration on FPGA. In *2020 30th International Conference on Field-Programmable Logic and Applications (FPL)*. 47–56.

[93] M. Kurek, M. P. Deisenroth, W. Luk, and T. Todman. 2016. Knowledge Transfer in Automatic Optimisation of Reconfigurable Designs. In *IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines*. 84–87.

[94] Per-Ake Larson, Wolfgang Lehner, Jingren Zhou, and Peter Zabback. 2007. Cardinality Estimation using Sample Views with Quality Assurance. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*. 175–186.

[95] Hongrae Lee, Raymond Ng, and Kyuseok Shim. 2011. Similarity Join Size Estimation using Locality Sensitive Hashing. *Proceedings of the VLDB Endowment* 4, 6 (2011), 338–349.

[96] Steven Leibson. 2022. FPGAs vs ASICs: Choose Your Path Carefully. Retrieved December 7, 2022 from `https://www.eejournal.com/article/fpgas-vs-asics-choose-your-path-carefully/`

[97] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2015. How Good Are Query Optimizers, Really? *Proceedings of the VLDB Endowment* 9, 3 (2015), 204–215.

[98] Viktor Leis, Bernhard Radke, Andrey Gubichev, Alfons Kemper, and Thomas Neumann. 2017. Cardinality Estimation Done Right: Index-Based Join Sampling. In *8th Biennial Conference on Innovative Data Systems Research*.

[99] Ping Li, Anshumali Shrivastava, Joshua Moore, and Arnd Christian König. 2011. Hashing Algorithms for Large-Scale Learning. In *Proceedings of the 24th International Conference on Neural Information Processing Systems*. 2672–2680.

[100] Qing Liu. 2009. *Encyclopedia of Database Systems - Approximate Query Processing*. Springer, 113–119.

[101] Guy Lohman. 2014. Is Query Optimization a "Solved" Problem? SIGMOD Blog.

[102] 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 *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 1633–1649.

[103] Volker Markl, Peter Haas, Marcel Kutsch, Nimrod Megiddo, Utkarsh Srivastava, and Tam Tran. 2007. Consistent Selectivity Estimation via Maximum Entropy. *Proceedings of the VLDB Endowment* 16, 1 (2007), 55–76.

[104] Volker Markl, Vijayshankar Raman, David Simmen, Guy Lohman, Hamid Pirahesh, and Miso Cilimdzic. 2004. Robust Query Processing Through Progressive Optimization. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*. 659–670.

[105] Charles Masson, Jee E. Rim, and Homin K. Lee. 2019. DDSketch: A Fast and Fully-Mergeable Quantile Sketch with Relative-Error Guarantees. *Proceedings of the VLDB Endowment* 12, 12 (2019), 2195–2205.

[106] Yossi Matias, Jeffrey Scott Vitter, and Min Wang. 1998. Wavelet-based Histograms for Selectivity Estimation. *SIGMOD Record* 27, 2 (1998), 448–459.

[107] Rick Merritt. 2022. *What Is Green Computing?* Nvidia. Retrieved December 7, 2022 from `https://blogs.nvidia.com/blog/2022/10/12/what-is-green-computing/`

[108] Ahmed Metwally, Divyakant Agrawal, and Amr El Abbadi. 2005. Efficient Computation of Frequent and Top-k Elements in Data Streams. In *Proceedings of the 10th International Conference on Database Theory*. 398–412.

[109] Jayadev Misra and David Gries. 1982. Finding Repeated Elements. *Science of Computer Programming* 2, 2 (1982), 143–152.

[110] Guido Moerkotte, Thomas Neumann, and Gabriele Steidl. 2009. Preventing Bad Plans by Bounding the Impact of Cardinality Estimation Errors. *Proceedings of the VLDB Endowment* 2, 1 (2009), 982–993.

[111] Ethernity Networks. 2022. *Ethernity Networks - FPGA SmartNICs for Network Acceleration*. Retrieved December 7, 2022 from `https://ethernitynet.com/cornerstones/fpga-smartnics-for-network-acceleration/`

[112] Thomas Neumann. 2011. Efficiently Compiling Efficient Query Plans for Modern Hardware. *Proceedings of the VLDB Endowment* 4, 9 (2011), 539–550.

[113] Jorge Nocedal. 1980. Updating Quasi-Newton Matrices with Limited Storage. *Mathematics of Computation* 35, 151 (1980), 773–782.

[114] Nvidia. 2022. *CUDA C++ Programming Guide - Design Guide*. Retrieved December 7, 2022 from `https://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf`

[115] Nvidia. 2022. *Nvidia A100 Tensor Core GPU - Data Sheet*. Retrieved Decemember 7, 2022 from `https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/a100/pdf/nvidia-a100-datasheet-us-nvidia-1758950-r4-web.pdf` PG-02829-001_v11.7.0.

[116] Frank Olken. 1993. *Random Sampling from Databases*. Ph. D. Dissertation. University of California, Berkeley.

[117] F. Olken and D. Rotem. 1992. Maintenance of Materialized Views of Sampling queries. In *Eighth International Conference on Data Engineering*. 632–641.

[118] Odysseas Papapetrou, Minos Garofalakis, and Antonios Deligiannakis. 2012. Sketch-Based Querying of Distributed Sliding-Window Data Streams. *Proceedings of the VLDB Endowment* 5, 10 (2012), 992–1003.

[119] Emanuel Parzen. 1962. On Estimation of a Probability Density Function and Mode. *The Annals of Mathematical Statistics* 33, 3 (1962), 1065 – 1076.

[120] Evaggelia Pitoura. 2018. *Encyclopedia of Database Systems - Selectivity Estimation*. Springer, 3371–3372.

[121] M. Powell. 1994. A Direct Search Optimization Method That Models the Objective and Constraint Functions by Linear Interpolation. In *Advances in Optimization and Numerical Analysis*. 51–67.

[122] Murad Qasaimeh, Kristof Denolf, Jack Lo, Kees A. Vissers, Joseph Zambreno, and Phillip H. Jones. 2019. Comparing Energy Efficiency of CPU, GPU and FPGA Implementations for Vision Kernels. In *15th IEEE International Conference on Embedded Software and Systems*. 1–8.

[123] M. V. Ramakrishna, E. Fu, and E. Bahcekapili. 1997. Efficient Hardware Hashing Functions for High Performance Computers. *IEEE Transactions on Computers* 46, 12 (1997), 1378–1381.

[124] Naveen Reddy and Jayant Haritsa. 2005. Analyzing Plan Diagrams of Database Query Optimizers. In *Proceedings of the VLDB Endowment*. 1228–1239.

[125] Reflexces. 2022. *Reflexces - A world of opportunities*. Retrieved December 7, 2022 from `https://www.reflexces.com/`

[126] Murray Rosenblatt. 1956. Remarks on Some Nonparametric Estimates of a Density Function. *The Annals of Mathematical Statistics* 27, 3 (1956), 832 – 837.

[127] Florin Rusu. 2009. *Sketches for Aggregate Estimations over Data Streams*. Ph. D. Dissertation. University of Florida.

[128] Florin Rusu and Alin Dobra. 2007. Pseudo-Random Number Generation for Sketch-Based Estimations. *ACM Transactions on Database Systems* 32, 2 (2007).

[129] Florin Rusu and Alin Dobra. 2008. Sketches for Size of Join Estimation. *ACM Transactions on Database Systems* 33, 3 (2008).

[130] A. Saavedra, C. Hernandez, and M. Figueroa. 2018. Heavy-Hitter Detection Using a Hardware Sketch with the Countmin-CU Algorithm. In *21st Euromicro Conference on Digital System Design*. 38–45.

[131] Stephan R. Sain, Keith A. Baggerly, and David W. Scott. 1994. Cross-Validation of Multivariate Densities. *J. Amer. Statist. Assoc.* 89, 427 (1994), 807–817.

[132] Arish Sateesan, Jo Vliegen, Simon Scherrer, Hsu-Chun Hsiao, Adrian Perrig, and Nele Mentens. 2021. Speed Records in Network Flow Measurement on FPGA. In *31st International Conference on Field-Programmable Logic and Applications (FPL)*. 219–224.

[133] Robert Schweller, Yan Chen, Elliot Parsons, Ashish Gupta, Gokhan Memik, and Yin Zhang. 2004. Reverse Hashing for Sketch-Based Change Detection on High-Speed Networks. In *Proceedings of ACM/USENIX Internet Measurement Conference'04*.

[134] David Scott. 2015. *Multivariate Density Estimation - Theory, Practice, and Visualization* (2nd ed.). John Wiley & Sons.

[135] P Selinger, Morton Astrahan, Donald Chamberlin, Raymond Lorie, and Thomas Price. 1979. Access Path Selection in a Relational Database Management System. In *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data*. 23–34.

[136] Yi Shan, Bo Wang, Jing Yan, Yu Wang, Ningyi Xu, and Huazhong Yang. 2010. FPMR: MapReduce Framework on FPGA. In *Proceedings of the 18th Annual ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. 93–102.

[137] S. J. Sheather and M. C. Jones. 1991. A Reliable Data-Based Bandwidth Selection Method for Kernel Density Estimation. *Journal of the Royal Statistical Society. Series B (Methodological)* 53, 3 (1991), 683–690.

[138] A. Skodras, C. Christopoulos, and T. Ebrahimi. 2001. The JPEG 2000 Still Image Compression standard. *IEEE Signal Processing Magazine* 18, 5 (2001), 36–58.

[139] Javier Soto, Thomas Krohmer, Cecilia Hernandez, and Miguel Figueroa. 2019. Hardware Acceleration of k-Mer Clustering using Locality-Sensitive Hashing. In *22nd Euromicro Conference on Digital System Design*. 659–662.

[140] Javier E. Soto, Paulo Ubisse, Yaime Fernández, Cecilia Hernández, and Miguel Figueroa. 2021. A High-Throughput Hardware Accelerator for Network Entropy Estimation Using Sketches. *IEEE Access* 9 (2021), 85823–85838.

[141] Javier E. Soto, Paulo Ubisse, Cecilia Hernández, and Miguel Figueroa. 2020. A Hardware Accelerator for Entropy Estimation using the Top-k Most Frequent Elements. In *23rd Euromicro Conference on Digital System Design (DSD)*. 141–148.

[142] Statista. 2022. *Volume of data/information created, captured, copied, and consumed worldwide from 2010 to 2020, with forecasts from 2021 to 2025*. Retrieved December 7, 2022 from `https://www.statista.com/statistics/871513/worldwide-data-created/`

[143] Michael Stillger, Guy Lohman, Volker Markl, and Mokhtar Kandil. 2001. LEO - DB2's Learning Optimizer. In *Proceedings of the VLDB Endowment*. 19–28.

[144] Herb Sutter. 2005. The Free Lunch is Over: A Fundamental Turn Toward Concurrency in Software. *Dr. Dobb's Journal* 30, 3 (2005), 202–210.

[145] Krister Svanberg. 1987. The Method of Moving Asymptotes — A New Method for Structural Optimization. *International Journal for Numerical Methods in Engineering* 24, 2 (1987), 359–373.

[146] Arun Swami and K Schiefer. 1994. On the Estimation of Join Result Sizes. In *Advances in Database Technology — EDBT '94*. 287–300.

[147] Jakub Szuppe. 2016. Boost.Compute: A Parallel Computing Library for C++ Based on OpenCL. In *Proceedings of the 4th International Workshop on OpenCL*.

[148] NYC Taxi and Limousine Commission. 2009-2016. *TLC Trip Record Data*. Retrieved December 7, 2022 from `https://www1.nyc.gov/site/tlc/about/tlc-trip-record-data.page`

[149] Charles C. Taylor. 1989. Bootstrap Choice of the Smoothing Parameter in Kernel Density Estimation. *Biometrika* 76, 4 (1989), 705–712.

[150] J. Teubner, R. Mueller, and G. Alonso. 2010. FPGA Acceleration for the Frequent Item Problem. In *IEEE 26th International Conference on Data Engineering*. 669–680.

[151] Jens Teubner and Louis Woods. 2013. Data Processing on FPGAs. *Synthesis Lectures on Data Management* 5, 2 (2013), 1–118.

[152] Mikkel Thorup and Yin Zhang. 2004. Tabulation Based 4-Universal Hashing with Applications to Second Moment Estimation. In *Proceedings of the Fifteenth Annual ACM-SIAM Symposium on Discrete Algorithms*. 615–624.

[153] Da Tong and Viktor K. Prasanna. 2015. High-Throughput Sketch-Based Online Heavy Change Detection on FPGA. In *International Conference on ReConFigurable Computing and FPGAs*. 1–8.

[154] D. Tong and V. K. Prasanna. 2018. Sketch Acceleration on FPGA and its Applications in Network Anomaly Detection. *IEEE Transactions on Parallel and Distributed Systems* 29, 4 (2018), 929–942.

[155] Kostas Tzoumas, Amol Deshpande, and Christian Jensen. 2011. Lightweight Graphical Models for Selectivity Estimation Without Independence Assumptions. *Proceedings of the VLDB Endowment* 4, 11 (2011), 852–863.

[156] Kostas Tzoumas, Amol Deshpande, and Christian S. Jensen. 2013. Efficiently Adapting Graphical Models for Selectivity Estimation. *The VLDB Journal* 22, 1 (2013), 3–27.

[157] David Vengerov, Andre Menck, Mohamed Zait, and Sunil Chakkappen. 2015. Join Size Estimation Subject to Filter Conditions. *Proceedings of the VLDB Endowment* 8, 12 (2015), 1530–1541.

[158] Jeffrey Vitter. 1985. Random Sampling with a Reservoir. *ACM Transactions on Mathematical Software* 11, 1 (1985), 37–57.

[159] Vasily Volkov. 2016. *Understanding Latency Hiding on GPUs*. University of California, Berkeley.

[160] Matt P Wand and M Chris Jones. 1994. Multivariate Plug-In Bandwidth Selection. *Computational Statistics* 9, 2 (1994), 97–116.

[161] Zeke Wang, Bingsheng He, and Wei Zhang. 2015. A Study of Data Partitioning on OpenCL-Based FPGAs. In *25th International Conference on Field Programmable Logic and Applications*. 1–8.

[162] Zeke Wang, Johns Paul, Hui Yan Cheah, Bingsheng He, and Wei Zhang. 2016. Relational Query Processing on OpenCL-Based FPGAs. In *26th International Conference on Field Programmable Logic and Applications*. 1–10.

[163] Zeke Wang, Shuhao Zhang, Bingsheng He, and Wei Zhang. 2016. Melia: A MapReduce Framework on OpenCL-Based FPGAs. *IEEE Transactions on Parallel and Distributed Systems* 27, 12 (2016), 3547–3560.

[164] Loring Wirbel. 2014. Xilinx SDAccel: A Unified Development Environment for Tomorrow's Data Center. *The Linley Group Inc* (2014).

[165] Xilinx. 2022. *Adaptable Accelerator Cards for Data Center Workloads*. Retrieved December 7, 2022 from `https://www.xilinx.com/products/boards-and-kits/alveo.html`

[166] Xilinx. 2022. *High Performance Computing*. Retrieved December 7, 2022 from `https://www.xilinx.com/applications/data-center/high-performance-computing.html`

[167] Xilinx. 2022. *Vivado Design Suite User Guide - Design Flows Overview (Version 2022.2)*. Retrieved December 7, 2022 from `https://docs.xilinx.com/r/en-US/ug892-vivado-design-flows-overview/Revision-History`

[168] Jackson H.C. Yeung, C.C. Tsang, K.H. Tsoi, Bill S.H. Kwan, Chris C.C. Cheung, Anthony P.C. Chan, and Philip H.W. Leong. 2008. Map-Reduce as a Programming Model for Custom Computing Machines. In *2008 16th International Symposium on Field-Programmable Custom Computing Machines*. 149–159.

[169] Jose Fernando Zazo, Sergio Lopez-Buedo, Mario Ruiz, and Gustavo Sutter. 2017. A Single-FPGA Architecture for Detecting Heavy Hitters in 100 Gbit/s Ethernet Links. In *2017 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*. 1–6.

[170] Erkang Zhu, Fatemeh Nargesian, Ken Q Pu, and Renée J Miller. 2016. LSH Ensemble: Internet-scale Domain Search. *Proceedings of the VLDB Endowment* 9, 12 (2016), 1185–1196.