A First Step Towards GPU-assisted Query Optimization

Max Heimel
Technische Universität Berlin
Einsteinufer 17
10587 Berlin, Germany
max.heimel@tu-berlin.de

Volker Markl
Technische Universität Berlin
Einsteinufer 17
10587 Berlin, Germany
volker.markl@tu-berlin.de

ABSTRACT

Modern graphics cards bundle high-bandwidth memory with a massively parallel processor, making them an interesting platform for running data-intensive operations. Consequently, several authors have discussed accelerating database operators using graphics cards, often demonstrating promising speed-ups. However, due to limitations stemming from limited device memory and expensive data transfer, GPU-accelerated databases remain a niche technology.

We suggest a novel approach: Using the graphics card as a co-processor during query optimization. Query optimization is a compute-heavy operation that requires only minimal data transfer, making it a well-suited target for GPU offloading. Since existing optimizers are typically very efficient, we do not suggest to simply accelerate them. Instead, we propose to use the additional resources to leverage more computationally involved optimization methods. This approach indirectly accelerates a database by generating better plan quality.

As a first step towards GPU-assisted query optimization, we present a proof-of-concept that uses the graphics card as a statistical co-processor during selectivity estimation. We integrated this GPU-accelerated estimator into the optimizer of PostgreSQL. Based on this proof-of-concept, we demonstrate that a GPU can be efficiently used to improve the quality of selectivity estimates in a relational database system.

1. INTRODUCTION

In recent years, *graphics processing units* (GPUs) matured to fully programmable, highly-parallel co-processors. Modern graphics cards contain up to a few thousand simple, programmable compute cores paired with a few gigabytes of high-bandwidth memory. Several frameworks - like *OpenCL*, *CUDA* and *DirectCompute* - allow for harnessing this massive performance to accelerate general-purpose computations. The usage of graphics cards to accelerate operations in relational database systems has already been extensively studied, often demonstrating promising results. Nevertheless, it remains a niche technology due to severe limitations stemming from limited device memory and large data transfer costs.

We suggest an orthogonal approach towards using graphics cards in a database system: Running query optimization on the GPU. Query optimization is typically compute-bound and requires only minimal data transfer, making it a well-suited target for GPU offloading. Since optimization time is in most cases insignificant compared to the actual query runtime, simply accelerating the optimizer is of little use. Instead, we propose to use the additional compute power of a graphics card to run more involved optimization routines, leading to better plan choices. In essence, this approach indirectly accelerates the database without running operator code on the GPU.

To the best of our knowledge, there is no prior work on using a graphics card during query optimization: All prior research papers focused on using graphics cards to accelerate common database operations. This covers efficient algorithms for sorting [13, 11, 6, 25], hashing [3, 9], indexing [19], compression [8], selections [29, 12], joins [16] and transactions [17]. Most of these papers report promising results, often demonstrating speed-ups of an order of magnitude. However, research has also shown that several open challenges - including GPU-aware query optimization and data placement strategies - remain to be addressed [22]. Our approach avoids these issues and could even help to solve some of them.

We make the following contributions in this paper:

1. We motivate using a graphics card to improve the results of query optimization in a relational database.
2. As a proof-of-concept, we investigate using the graphics card as a *statistical co-processor* during selectivity estimation. In particular, we use the GPU to quickly compute highly accurate estimates of base-table query cardinalities.
3. We provide an implementation of our proof-of-concept, integrating a GPU-assisted selectivity estimator for real-valued range queries based on Kernel Density Estimation into PostgreSQL.
4. We evaluate our estimator, demonstrating that using a GPU can improve estimation quality, in particular for multidimensional data.

The remainder of this paper is structured as follows: In the next section, we give a quick introduction into the archi-

tecture of modern graphics cards and the specifics of general purpose programming on a GPU (GPGPU). In Section 3, we observe some open challenges and motivate using the graphics card during query optimization. Section 4 introduces our proof-of-concept and motivates our design choices. In the following section, we discuss implementation details of the system, including how the estimator has been parallelized and how it has been integrated into PostgreSQL. In Section 6, we present an evaluation of the estimator, demonstrating that using the GPU can improve estimation quality. Finally, Section 7 concludes the paper by discussing possible directions for future work on this topic.

2. PRELIMINARIES

2.1 Graphics Card Architecture

Figure 1 showcases[1] the architecture of a modern computer system with a graphics card. The graphics card - henceforth also called the *device* - is connected to the *host system* via the *PCIExpress bus*. All data transfer between host and device has to pass through this comparably low-bandwidth bus.

The graphics card contains the graphics processing unit (GPU) and a few[2] gigabytes of *device memory*. Typically[3], host and device do not share the same address space, meaning that neither the GPU can directly access the main memory nor the CPU can directly access the device memory.

The GPU itself consists of a few *multiprocessors*, which can be seen as very wide SIMD processing elements. Each multiprocessor packages several *scalar processors* with a few kilobytes of high-bandwidth, on-chip *shared memory* and an interface to the device memory.

2.2 Programming a GPU

Programs that run on a graphics card are written in the so-called *kernel programming model*. Programs in this model consist of *host code* and *kernels*. The host code manages the graphics card, initializing data transfer and scheduling program execution on the device. A kernel is a simplistic program that forms the basic unit of parallelism in the kernel programming model. Kernels are scheduled concurrently on several scalar processors in a SIMD fashion: Each kernel invocation - henceforth called *thread* - executes the same code on its own share of the input. All threads that run on the same multiprocessor are logically grouped into a *workgroup*.

One of the most important performance factors in GPU programming is to avoid data transfer between host and device: All data has to pass across the PCIexpress bus, which is the bottleneck of the architecture. Data transfer to the device might therefore eat up all time savings from running a problem on the GPU. This becomes especially evident for I/O-bound algorithms: Since accessing the main memory is roughly three times faster than sending data across the PCI-express bus, the CPU will have finished execution before the data even arrived on the device.

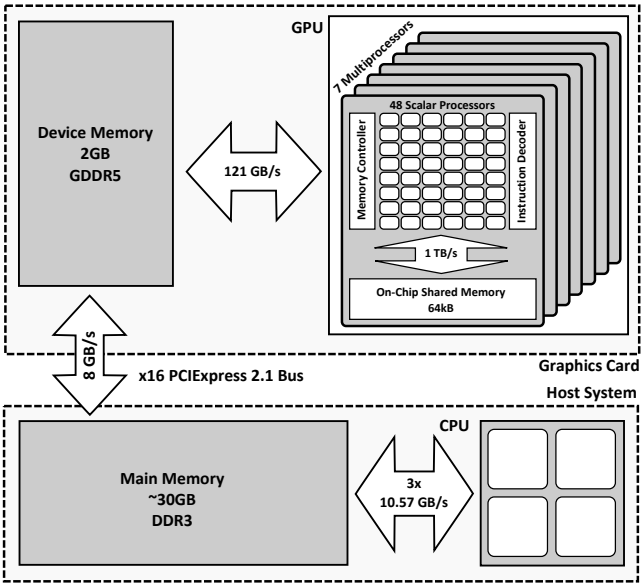

Figure 1: Overview: Exemplary architecture of a system with a graphics card.

Graphics cards achieve high performance through massive parallelism. This means, that a problem should be easy to parallelize to gain most from running on the GPU. Another performance pitfall in GPU programming is caused by divergent code paths. Since each multiprocessor only has a single instruction decoder, all scalar processors execute the same instruction at a time. If some threads in a workgroup diverge, for example due to data-dependent conditionals, the multiprocessor has to serialize the code paths, leading to performance losses. As a general rule, it is therefore recommended to avoid control structures in kernels where possible [7].

3. RUNNING QUERY OPTIMIZATION ON THE GPU

3.1 Challenges in GPU-accelerated Databases

Graphics cards bundle high-bandwidth memory with a massively parallel processor, making them an interesting target for running data-intensive operations. Consequently, there are several papers on accelerating common database operations, often demonstrating promising speed-ups [22]. Nevertheless, there are challenges that remain to be addressed before GPUs can be reasonably used in databases:

- GPU-accelerated databases try to keep relational data cached on the device to avoid data transfer. Since device memory is limited, this is often only possible for a subset of the data. Deciding which part of the data should be offloaded to the GPU - finding a so called *data placement strategy* - is a difficult problem that currently remains unsolved.

- Due to result transfer costs, operators that generate a large result set are often unfit for GPU-offloading. Since the result size of an operation is typically not

[1]The figure shows the architecture of a graphics card from the *Fermi* architecture of NVIDIA. While specific details might be different for other vendors, the general concepts are found in all modern graphic cards.

[2]Typically around 2GB on mainstream cards and up to 8GB on high-end devices.

[3]Beginning with version 4.0, NVIDIA's CUDA framework introduced unified virtual addressing, which tackles this problem. However, this is only a virtual technique aimed at simplifying programming.

known before execution, predicting whether a given operator will benefit from the GPU is a hard problem.

- GPU-accelerated operators are of little use for disk-based database systems, where most time is spent on disk I/O. Since the GPU improves performance only once the data is in main memory, time savings will be small compared to the total query runtime. Furthermore, disk-resident databases are typically very large, making it harder to find an optimal data placement strategy.

- Having the option of running operations on a GPU increases the complexity of query optimization: The plan search space is drastically larger and a cost function that compares runtimes across architectures is required. While there has been some prior work in this direction [15], GPU-aware query optimization remains an open challenge.

3.2 GPU-assisted Query Optimization

The query optimizer of a relational database system generates an execution plan from a query. It picks the plan with lowest estimated cost from the exponentially large space of possible plans for the query. During optimization, multiple candidate plans are generated and cost functions are evaluated on these candidates. Query optimization performs several computations on a comparably small input, generating a small result set. Conceptually, this makes it an interesting problem for GPU offloading.

Based on this observation, we propose to investigate methods for running query optimization on the graphics card. We want to stress that it is not our intention to simply accelerate query optimization: It is already a very efficient procedure and only highly complex queries result in significant optimization overhead. Instead, our focus is on using the additional resources of the GPU to improve the quality of generated plans. This allows us to indirectly accelerate a database system using a GPU while avoiding the issues outlined in the previous section.

One research challenge for GPU-assisted query optimization is to identify components of the optimizer that trade off result quality with computational effort and that are easily parallelized. Such components are prime candidates for offloading to the GPU, using the additional resources to improve quality. Some possible directions are reducing the number of heuristics that prune the search space, using more accurate - but also more expensive - cost functions, and throwing more resources at selectivity estimation to get better estimates.

We would like to stress that our approach is independent of GPU-assisted database operations. Naturally, a DBMS not assisted by GPUs for query processing can still use the graphics card as a co-processor to improve query optimization tasks, as we will showcase in the next section. However, one can also complement GPU-assisted query processing with query optimization on the GPU. In particular, the GPU can be used to handle the additional optimization complexity from using GPU-accelerated operators. We believe that both approaches are important and that they can ultimately complement each other.

As a final point, we would like to note that - in contrast to GPU-accelerated database operators -, our approach also applies to disk-basked systems: Since we aim at providing higher plan quality, we can indirectly accelerate any DBMS, irregardless of its data access strategies.

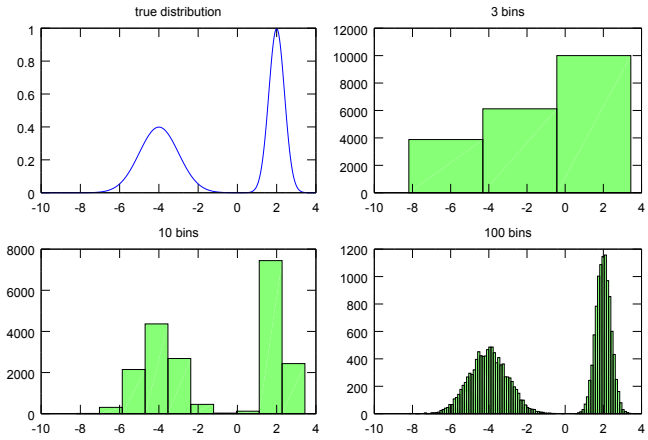

Figure 2: Scaling the accuracy of a histogram by increasing model complexity.

4. A GPU-ASSISTED SELECTIVITY ESTIMATOR

There are many ways how query optimization can benefit from GPUs, from query rewriting to plan enumeration, costing and selection. In the following, we describe, as a first step towards a GPU-assisted query optimizer, a selectivity estimation component that is assisted by a GPU. Using the massive compute power of GPUs, this component can leverage methods that current CPU-based query optimizers do not exploit due to computational complexity and cost.

4.1 The Graphics Card as a Statistical Co-Processor

In order to estimate the cost of candidate plans, the query optimizer needs to know the result sizes of intermediate operations. These sizes are estimated in a process called *selectivity estimation*, which uses data statistics to approximate the fraction of qualifying tuples for an operator. The quality of the selectivity estimates has a direct impact on the plan choice, since modern query optimizers typically react very sensitively to estimation changes [24]. Furthermore, it has been shown that incorrect selectivity estimates - in particular for multi-dimensional queries - are often the cause for bad query performance [21].

Selectivity estimation is a good example for a component that trades quality for computational effort: The estimation quality can be improved almost[4] arbitrarily by refining the data statistics, however, this also leads to more expensive estimates. Figure 2 illustrates this relationship for an equi-width histogram. By increasing the number of bins, we can get arbitrarily close to the true distribution. However, at the same time, computing an estimate gets more expensive, since we have to touch a larger number of bins.

As a proof-of-concept, we will demonstrate how a GPU can be used to improve the accuracy of selectivity estimation

[4]Statistics are typically built from an incomplete sample. With increasing refinement, we eventually overfit to the sample, leading to a loss in estimation quality.

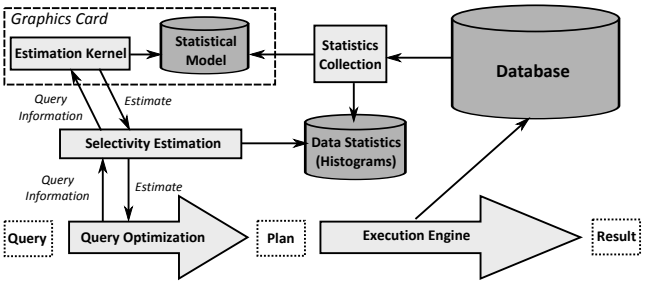

Figure 3: Overview: Using a graphics card as a statistical co-processor.

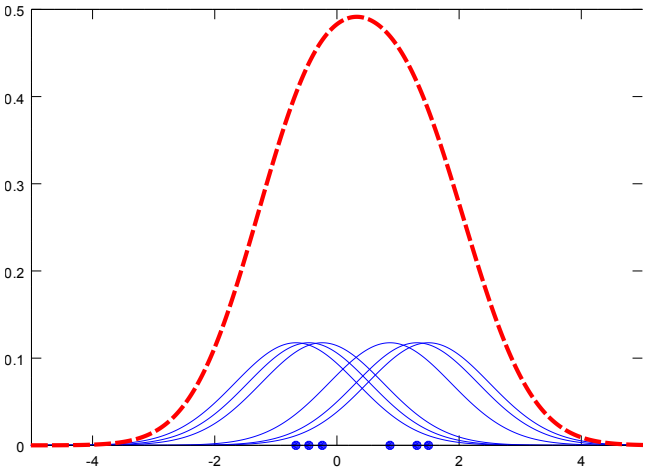

Figure 4: Kernel Density Estimate based on five sample points drawn from a normal distribution.

- and thus indirectly plan quality. The idea behind this is straightforward: We use the additional computational power of the GPU to run selectivity estimation on a more detailed model, resulting in better estimates. Figure 3 illustrates this idea: We use the graphics card as a *statistical co-processor* during selectivity estimation: A statistical model is stored on the device and used by the GPU to estimate query result sizes.

4.2 GPU-assisted Kernel Density Estimation

We have selected *Kernel Density Estimation* (KDE) as the estimator for our proof of concept. KDE is a non-parametric, data-driven technique for estimating a probability distribution from a data sample [1]. The method uses the sample as supporting points for the estimated distribution: Each sample point distributes some *probability mass* to its neighborhood. Mathematically, this is achieved by centering local probability distributions - so-called *kernels* - around the sample points. The estimated distribution is then computed as the normalized sum of all local distributions. Consequently, KDE assigns high probability to points that lie in the vicinity of sample points. Figure 4 visualizes KDE using a sample of five points drawn from a normal distribution. A more in-depth description of KDE can be found in Appendix Section A or in the corresponding literature [1, 28, 4].

KDE has several advantages over more traditional estimation techniques like histograms and frequent value statistics:

- From a statistical perspective, it has been shown that KDE converges faster to the true underlying distribution than histograms [1]. Thus, when using the same sample size, KDE - in general - generates a smaller estimation error than a histogram.
- One of the major sources of estimation errors in relational databases are incorrect independence assumptions for multidimensional data [5]. There are several techniques for multi-dimensional histograms that solve this problem [14, 5]. However, they are typically complex and costly to construct. KDE on the other hand naturally extends to multidimensional data: As long as the sample is representative, high probability will automatically be assigned to those areas with high data density.
- A KDE model can be easily maintained under changing datasets. While histograms have to be recomputed in regular intervals, KDE can be maintained incrementally by updating the data sample to stay representative. This is a well-understood problem in database research, and several solutions exist [10].

Given its many advantages, KDE has often been suggested as a selectivity estimation method in databases [4, 14]. However, there is a major roadblock, limiting the utility of KDE: Since the complete sample has to be evaluated for each estimation, KDE is a very expensive operation. This is particularly bad during selectivity estimation, which operates on a tight time budget. This forcefully limits KDE to use comparably small sample sizes for selectivity estimation.

We have chosen KDE for our proof-of-concept since it naturally demonstrates how additional compute power can improve estimation quality. KDE can be parallelized very efficiently, making it well suited for GPU-accelerating. This allows us to use a much larger data sample on the graphics card, which in turn leads to improved estimation quality.

5. IMPLEMENTATION DETAILS

Our implementation integrates a GPU-accelerated version of Kernel Density Estimation for real-valued range queries into the query optimizer of PostgreSQL. The estimator is written against the 1.1 specification[5] of OpenCL - an open standard of the kernel programming model. We use the latest stable release 9.1.3 of PostgreSQL.

We have chosen PostgreSQL as our host system for two reasons: First, it is open-source and well-documented, making it easy to integrate our code. Second, it is a disk-based system, allowing us to demonstrate that our approach makes it feasible to accelerate a disk-based database using a graphics card. The choice for OpenCL was mainly made, since it is supported by all major vendors, making our implementation vendor-independent. Furthermore, OpenCL allows us to target other devices than graphics cards, including multi-core processors and FPGAs.

The source code for our implementation can be found at: `https://bitbucket.org/mheimel/gpukde/`.

[5] `www.khronos.org/registry/cl/specs/opencl-1.1.pdf`

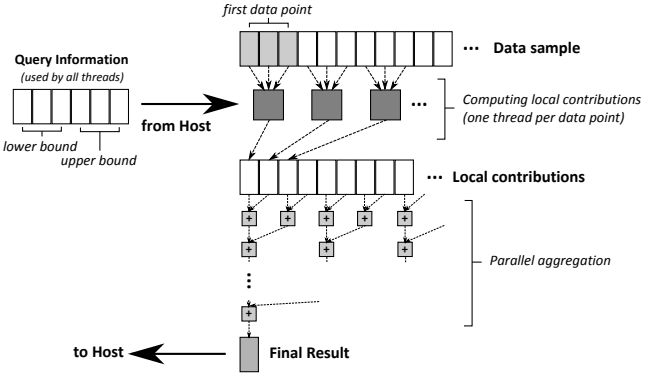

Figure 5: Parallelization Strategy for Running KDE on a GPU (3-D data).

5.1 Running KDE on a GPU

5.1.1 Parallelizing KDE

Computing a range selectivity in KDE requires integrating over the approximated probability distribution. Recall that KDE defines its estimate as the sum over local distributions. Since integration is a linear operator, we can evaluate the integral for each sample point independently and then sum up these *local contributions*. For a more in-depth description of this approach, refer to Appendix Section B.

Computing the local contributions is an embarrassingly parallel operation that can be implemented using a single kernel: Each thread evaluates the integral for one sample point, writing the result to device memory. Afterwards, we use a binary reduction strategy[6] to efficiently aggregate the local contributions in parallel. Figure 5 visualizes this strategy.

5.1.2 Dynamic Kernel Compilation

Computing the local contribution for a d-dimensional data point requires to evaluate all d dimensions[7]. The number of dimensions is unknown while writing the kernel, so we could pass the number as a runtime parameter and use a loop for the computation. However, since control logic in kernels typically leads to a performance penalty on the GPU [7], we want to avoid this strategy.

We use *dynamic kernel compilation* to achieve this goal. In dynamic kernel compilation, we generate and compile kernel code at runtime that exactly matches the problem. In OpenCL, this can for example be achieved by using C preprocessor statements: Once we know the number of dimensions, we pass it as a preprocessor constant to the kernel compiler. This enables the compiler to eliminate the loop via loop unrolling.

We also use dynamic kernel compilation to implement device-specific optimizations. In particular, we introduce the type of device (CPU or GPU) into the kernel as a preprocessor constant and, for instance, pick the optimal memory access pattern based on this constant.

[6]For details, refer to: `http://people.maths.ox.ac.uk/gilesm/cuda/prac4/reduction.pdf`

[7]See equation (16) in Appendix Section B for the corresponding equation.

5.1.3 Picking the Sample Size

The mean integrated square estimation error (MISE) of KDE converges with the sample size n as $O\left(n^{-\frac{4}{5}}\right)$ [1]. This means that the estimation error gets smaller when we increase the sample size. In other words, we should pick the largest sample size that is possible.

There are two factors that limit the possible sample size. First, query optimization - and in particular selectivity estimation - is typically operating on a tight time budget. If the sample size is chosen too large, the estimator will simply take too long. The second limiting factor is the size of the available device memory. Based on our observations, the first factor is irrelevant for graphics cards: Even for the largest possible sample size, the graphics card will take less than 25 milliseconds to compute a density estimate. We thus pick our sample size solely based on the available device memory[8]. For practical reasons, we use an upper bound of 512MB for the sample size, as this is the largest possible size of a singular memory allocation on the NVIDIA graphics card we used for development.

5.1.4 Exact Evaluation

For a small relation, it is possible to copy the complete relation into device memory. In this case, there is no unseen data to which we need to generalize in selectivity estimation: The additional smoothing induced by KDE would lead to incorrect results. Therefore, for small relations, we fall back to exactly counting how many tuples qualify the given query.

The parallelization strategy is almost identical to KDE: We schedule a kernel on all data points that writes one to device memory if the point qualifies and zero otherwise. Afterwards, we recursively aggregate all these local contributions to get the exact count of qualifying tuples.

5.2 Integration into PostgreSQL

The integration of our KDE estimator into PostgreSQL requires three major components: *OpenCL context management*, *model management* and *selectivity estimation*. We will now briefly discuss these components.

5.2.1 OpenCL Context Management

The context management initializes the OpenCL runtime and manages access to the OpenCL APIs. It also handles the dynamic compilation of kernels: Kernels are indexed in a kernel library using the injected preprocessor constants as key. This way, we can avoid recompiling kernels that we have already built. When the first request arrives, the context management initializes all required resources and directory structures. To simplify switching between devices, we added a boolean configuration variable ocl_use_gpu to PostgreSQL. The value of this variable is used during initialization to decide whether OpenCL is initialized for CPU or GPU.

5.2.2 Model Management

The model management handles all things related to creating and registering statistical models. We inserted some code into PostgreSQL's ANALYZE command, which is used to trigger statistics collection on a table. When ANALYZE is called for a table, our code checks the table definition for

[8]Taking into account that we want to store samples for multiple tables and that we need some memory on the device to store the local contributions during estimation.

real-valued attributes. If any are found, a new KDE model is automatically created for them. This happens in three steps: First, we collect a random sample of the selected attributes using PostgreSQL's internal sampling routines and transfer it to the device memory. Before shipping the data, we re-scale each attribute to unit variance to achieve better numerical stability [1]. Afterwards, the required kernels are dynamically compiled to match the number of attributes. Finally, we generate a model descriptor and register it within a global registry. The descriptor contains information identifying the scope of the model (table id and attributes) as well as the location of the sample on the device and handles for the estimation kernels.

5.2.3 *Selectivity Estimation*

Selectivity estimation is triggered by code within the query optimizer of PostgreSQL. When a query arrives, the optimizer calls the function clauselist_selectivity to compute the joint selectivity of all predicates defined on a single base table. Our code checks whether a KDE model exists for the given table. If we find a matching model, all range-predicates on real-valued attributes are extracted and handed over to the KDE estimator. The estimator then configures the kernels with the query parameters and starts the estimation. Finally, we pass the computed selectivity back to clauselist_selectivity . PostgreSQL then estimates the selectivity of the remaining predicates using histograms, computing the final result as the product of all estimates.

6. EVALUATION

6.1 Experimental setup

All experiments were run on a custom-built server with the following specifications:

- Intel Xeon E5620, 64-bit, four cores running at 2.4GHz, 12MB Cache.
- 32GB of DDR-3 RAM, clocked at 1333MHz.

The server is equipped with a middle-class NVIDIA GTX460 graphics card, sitting in a PCIexpress 2.1 x16 slot. The graphics card has the following specifications:

- NVIDIA Fermi GF104 core:
  - Seven multiprocessors, each having 48 compute units.
  - 48KB of local device memory, 64KB of constant buffer per compute unit.
- 2GB of DDR4 graphics memory, clocked at 1800MHz.

The experiments were conducted on a 64-bit Scientific Linux 6.2 (Linux kernel 2.6.32-220.7.1.e16.x86_64). The graphics card was controlled with the NVIDIA 295.33 driver for 64-bit Linux systems. All timings were measured using the POSIX function gettimeofday(), which has a precision of one ms.

6.2 Performance Evaluation

In this section, we analyze the performance characteristics of our estimator. In particular, we are interested in the scaling behaviour with increasing data volume and dimensionality, the benefit from evaluating queries on small relations exactly, and the performance boost we get from the GPU.

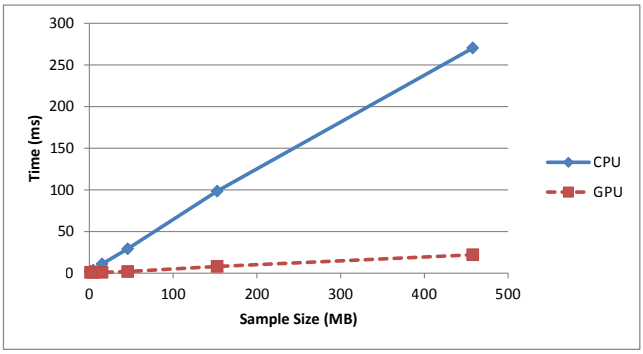

Figure 6: Evaluation: Runtime comparions between GPU and CPU.

6.2.1 *Perfomance Impact of the GPU*

In our first experiment, we investigate how GPU and CPU compare when estimating a selectivity using KDE. For this comparison, we used Intel's OpenCL SDK 1.5, which automatically generates vectorized multi-core code. We believe that this is a reasonable approach, given that on modern multi-core processors, OpenCL typically achieves comparable performance to more traditional techniques like OpenMP [26]. The experiment was run on a synthetic, 800MB dataset of uniformly distributed four-dimensional data points. We ran random, non-empty range queries against this dataset, measuring the average runtime of ten queries on both CPU and GPU. The experiment was repeated multiple times using different sample sizes, allowing us to measure how the estimation time behaves with increasing input size. Figure 6 shows the resulting measurements.

There are a few notable observations: First, the GPU is about an order of magnitude faster than the CPU. On the largest sample size - which is roughly 460MB - the CPU took 270ms, while the GPU took 22ms. Second, the estimator scales linearly with input size. This is the expected behaviour, given that the majority of the work is done while computing the local contributions from all sample points.

Note that the experiment shows that the GPU can produce an estimate roughly 12 times faster than the CPU. Since scaling behaviour is linear, this means that given a fixed time budget, the GPU can use a sample that is 12 times larger than what is possible on the CPU. In KDE, the estimation error converges with the sample size n as $O\left(n^{-\frac{4}{5}}\right)$. Thus, the GPU can generate estimates that are roughly $12^{\frac{4}{5}} \approx 7.3$ times more accurate!

6.2.2 *Scaling with Dimensionality*

In our second experiment, we examine the scaling behaviour of the estimator with increasing dimensionality. For this, we generated random 500MB datasets of uniformly distributed data points with increasing dimensionality. We then measured the average estimation time for five random, non-empty queries on each data set. Figure 7 shows the resulting measurements.

The measurements show a clear minimum in the estimation time for three-dimensional data, with one- and six-dimensional data taking the longest. We can explain this by observing that, since we kept the data volume constant, we have twice as many one-dimensional data points than

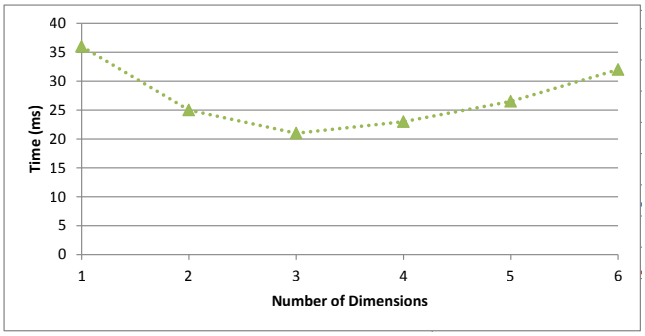

Figure 7: Evaluation: Scaling behaviour with increasing dimensionality.

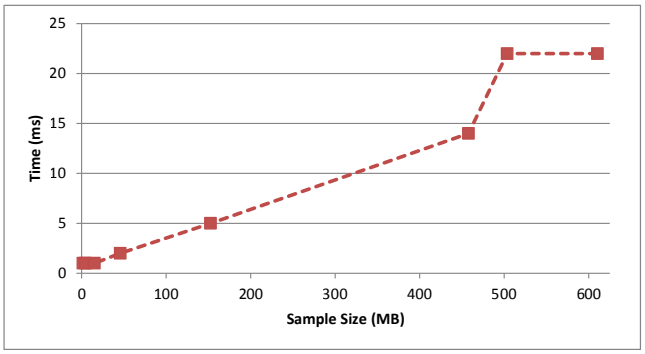

Figure 8: Evaluation: Performance Impact of using exact evaluation on small relations.

two-dimensional ones. Consequently, in the one-dimensional case, twice as many local contributions have to be written to device memory, explaining the higher cost. The increase for higher-dimensional data is likely caused by suboptimal memory access patterns on the GPU. Nevertheless, the experiment still demonstrates that the dimensionality of the data has a much smaller impact on performance than the sample size.

6.2.3 *Impact of Exact Evaluation*

In the final experiment, we investigate the impact on performance when using exact query evaluation on small relations that fit into device memory. We generated datasets of increasing size, consisting of uniformly distributed, four-dimensional data points. Since we fixed the maximum sample size to be 500MB, all datasets smaller than this were evaluated exactly. We then measured the average estimation time of ten random, non-empty range queries against our dataset. Figure 8 shows the resulting measurements.

We can clearly see a sharp increase in runtime for datasets that are larger than 500MB. This increase is the switch-over from exact evaluation to KDE, meaning that exact evaluation is roughly 1.5-times faster than KDE. This is not surprising, given that KDE performs more complex calculations on each sampling point, incurring higher computational cost. Since exact evaluation gives perfect estimates, and is faster than KDE, we conclude that it is a very good mechanism, given the relation fits into device memory.

6.3 Estimation Quality

In this section, we analyze the estimation quality of our estimator. In particular, we investigate how the quality scales with the sample size and the amount of correlation in the data. Also, we compare how our implementation compares against the histogram-based selectivity estimator of PostgreSQL. This comparison is done using both synthetic and real-world datasets. We focus on multidimensional data, given that for single-dimensional data, the histogram approach of PostgreSQL is already sufficient in most cases [5, 14].

Note that we do not perform a full evaluation of KDE against other estimation techniques: Our goal in this paper is to demonstrate that the GPU can be used to improve estimation quality, not to discuss performance characteristics of KDE. Extensive evaluations of KDE as a selectivity estimation technique can be found - for instance - in [14] and [4]. We also won't evaluate the impact of improved selectivity estimates on plan quality. For such experiments and discussions, the reader is referred to existing publications, in particular [23] and [21].

There is one caveat that we want to point out before discussing the results. In its current form, our estimator does not use the "full potential" of KDE. The estimation quality of KDE is determined by the choice of the so-called bandwidth [1], which governs the width of the local probability densities[9]. Picking the optimal bandwidth is a difficult task, and multiple algorithms exist for it [18]. In our implementation, we follow the approach from [14] and use a quick rule-of-thumb formula. While this approach gives a solid first approximation, it is almost always sub-optimal [1]. Consequently, estimation quality could be notably improved by using one of the more involved bandwidth selection algorithms.

6.3.1 *Impact of Sample Size*

In the first experiment, we evaluate the impact of the sample size on estimation quality and demonstrate how our estimator compares on real-world data against PostgreSQL. For the experiment, we use the forest covertype dataset from the UCI Machine Learning Repository[10], a real-world, multi-dimensional dataset that features non-trivial correlations and complex data distributions. The dataset consists of $580,000$ data points, with 54 attributes - out of which ten are numeric. For practical reasons, we restricted ourselves to six of the ten numerical attributes and re-scaled them to the $[0, 1]$-interval. We then generated a workload consisting of $5,000$ random, non-empty range queries against the dataset. The workload was run once on vanilla PostgreSQL and multiple times on our modified version, with each run using a different sample size. Figure 9 shows the results of this experiment.

The measurements confirm the expected behaviour: The estimation error quickly decreases with increasing sample size. This observation is in accordance with the expected $O\left(n^{-\frac{4}{5}}\right)$ error convergence rate. The sharp drop in the estimation error, for a sample size of around $600,000$ data points is caused by the estimator switching to exact evaluation.

[9]See Figure 11 in Appendix Section A for a visualization of how the bandwidth choice impacts estimation quality.

[10]`http://archive.ics.uci.edu/ml/datasets/Covertype`

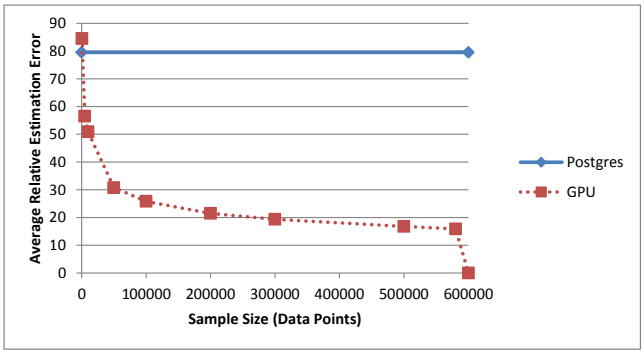

Figure 9: Evaluation: Impact of Sample Size on Estimation Quality.

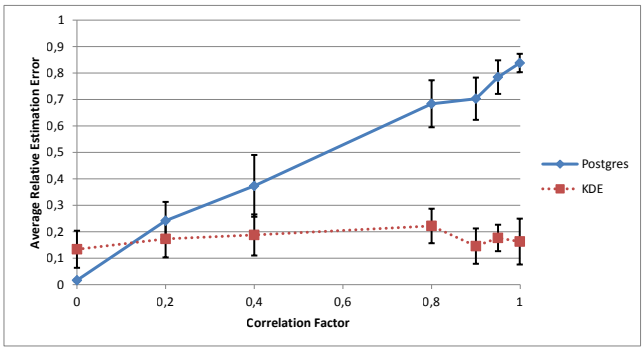

Figure 10: Evaluation: Impact of Correlation on Estimation Quality.

Note that even for small sample sizes - around a few thousand data points - we vastly outperform PostgreSQL on this dataset. This is not surprising, given that the dataset is multidimensional and features non-trivial correlation patterns. PostgreSQL will incorrectly assume independence between the attributes, when combining estimates from multiple single-dimensional histograms. Our KDE estimator on the other hand, directly models the joint distribution, thus avoiding incorrect independence assumptions.

6.3.2 *Impact of Correlation*

In the second experiment, we analyze the impact of correlated data on the estimation quality and demonstrate how our estimator compares to PostgreSQL on synthetic data. For the experiment, we generated multiple random, two-dimensional datasets with increasing correlation between the two attributes. We then ran 500 non-empty, random range queries against this dataset, measuring the average relative estimation error. The experiment was run both on vanilla PostgreSQL and our modified version. For our estimator, we used roughly one third of the data as sample. This size was chosen, since it resulted in the same estimation time - roughly one ms - that vanilla PostgreSQL required using histograms. Figure 10 shows the results of this experiment.

The measurements confirm the result of the previous experiment: In case of correlated data, KDE outperforms PostgreSQL by a huge margin. While the estimation quality of PostgreSQL degrades quickly with increasing correlation, the quality of our estimator remains practically constant. An interesting observation is, that PostgreSQL outperforms our estimator for the case of independent attributes. However, it is likely that, when making a better bandwidth parameter choice, our estimator would achieve similar performance.

7. OUTLOOK

This paper motivates the idea of using a graphics card during query optimization. The additional compute power of a GPU can help to improve plan quality, eventually accelerating the database system. We demonstrate the usefulness of this approach via a concrete example: Using a GPU for selectivity estimation. We integrated a GPU-accelerated version of Kernel Density Estimation into PostgreSQL, demonstrating a clear improvement in estimation quality. We will now discuss some possibilities for future work:

A possible next step is to continue investigating ways how to use a graphics card during query optimization. A first step could include identifying expensive, compute-intensive parts of the optimization pipeline that are easily parallelized. In a second step, these parts could then be ported to the GPU and replaced by more expensive methods that improve plan quality. A possible first approach could use existing work on running dynamic programming on a GPU [27] to accelerate join order optimization. It would also be interesting to revisit topics from query optimization literature and evaluate them with regard to search space restrictions that could be relaxed when more processing power is available.

Another possible direction is to improve upon the presented GPU-accelerated selectivity estimator. In particular, it would be interesting to see whether other multidimensional estimation techniques - for example GenHist [14] - could also benefit from running on a GPU. Future work could also focus on extending the KDE estimator to categorical attributes, an extension that would be required to make the estimator useful in a "real-world" database setting. This could build on existing work that deals with extending KDE to discrete [2] and categorical data [20]. Another possible direction is to investigate how to improve the estimation quality of the KDE estimator. In particular, it would be interesting to investigate existing bandwidth-selection algorithms [18] and identify whether they could be efficiently run on the GPU.

A final possible direction for future work is to investigate how GPU-assisted query optimization could be used to handle the additional optimization complexity that stems from GPU-accelerated operators. This could help to resolve some of the issues that currently prevent GPUs from being successfully used in real-world settings. Eventually, this research could merge the two approaches of using graphics cards in a database, with both technologies complementing each other.

8. REFERENCES

[1] *Multivariate Density Estimation - Theory, Practice and Visualization*. John Wiley & Sons, Inc., 1992.

[2] C. Aitken. Kernel methods for the estimation of discrete distributions. *Journal of Statistical Computation and Simulation*, 16(3-4):189–200, 1983.

[3] D. A. Alcantara, A. Sharf, F. Abbasinejad, S. Sengupta, M. Mitzenmacher, J. D. Owens, and N. Amenta. Real-time parallel hashing on the gpu. In *ACM SIGGRAPH Asia 2009 papers*, SIGGRAPH

Asia '09, pages 154:1–154:9, New York, NY, USA, 2009. ACM.

[4] B. Blohsfeld, D. Korus, and B. Seeger. A comparison of selectivity estimators for range queries on metric attributes. *SIGMOD Rec.*, 28(2):239–250, June 1999.

[5] N. Bruno, S. Chaudhuri, and L. Gravano. Stholes: a multidimensional workload-aware histogram. In *Proceedings of the 2001 ACM SIGMOD international conference on Management of data*, SIGMOD '01, pages 211–222, New York, NY, USA, 2001. ACM.

[6] D. Cederman and P. Tsigas. A practical quicksort algorithm for graphics processors. In *Proceedings of the 16th annual European symposium on Algorithms*, ESA '08, pages 246–258, Berlin, Heidelberg, 2008. Springer-Verlag.

[7] N. Corp. Cuda c best practices. Technical report, NVIDIA, 2012.

[8] W. Fang, B. He, and Q. Luo. Database compression on graphics processors. *Proc. VLDB Endow.*, 3(1-2):670–680, Sept. 2010.

[9] I. García, S. Lefebvre, S. Hornus, and A. Lasram. Coherent parallel hashing. In *Proceedings of the 2011 SIGGRAPH Asia Conference*, SA '11, pages 161:1–161:8, New York, NY, USA, 2011. ACM.

[10] R. Gemulla, W. Lehner, and P. J. Haas. Maintaining bounded-size sample synopses of evolving datasets. *The VLDB Journal*, 17(2):173–201, Mar. 2008.

[11] N. Govindaraju, J. Gray, R. Kumar, and D. Manocha. Gputerasort: high performance graphics co-processor sorting for large database management. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, SIGMOD '06, pages 325–336, New York, NY, USA, 2006. ACM.

[12] N. K. Govindaraju, B. Lloyd, W. Wang, M. Lin, and D. Manocha. Fast computation of database operations using graphics processors. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, SIGMOD '04, pages 215–226, New York, NY, USA, 2004. ACM.

[13] A. Greß and G. Zachmann. Gpu-abisort: Optimal parallel sorting on stream architectures. In *Proc. 20th IEEE Int'l Parallel and Distributed Processing Symposium (IPDPS)*, Rhodes Island, Greece, apr 2006.

[14] D. Gunopulos, G. Kollios, J. Tsotras, and C. Domeniconi. Selectivity estimators for multidimensional range queries over real attributes. *The VLDB Journal*, 14(2):137–154, Apr. 2005.

[15] B. He, M. Lu, K. Yang, R. Fang, N. K. Govindaraju, Q. Luo, and P. V. Sander. Relational query coprocessing on graphics processors. *ACM Trans. Database Syst.*, 34(4):21:1–21:39, Dec. 2009.

[16] B. He, K. Yang, R. Fang, M. Lu, N. Govindaraju, Q. Luo, and P. Sander. Relational joins on graphics processors. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, SIGMOD '08, pages 511–524, New York, NY, USA, 2008. ACM.

[17] B. He and J. X. Yu. High-throughput transaction executions on graphics processors. *Proc. VLDB Endow.*, 4(5):314–325, Feb. 2011.

[18] R. L. Hyndman, X. Zhang, and M. L. King. Bandwidth selection for multivariate kernel density estimation using mcmc. Econometric Society 2004 Australasian Meetings 120, Econometric Society, Aug 2004.

[19] C. Kim, J. Chhugani, N. Satish, E. Sedlar, A. D. Nguyen, T. Kaldewey, V. W. Lee, S. A. Brandt, and P. Dubey. Fast: fast architecture sensitive tree search on modern cpus and gpus. In *Proceedings of the 2010 international conference on Management of data*, SIGMOD '10, pages 339–350, New York, NY, USA, 2010. ACM.

[20] Q. Li and J. Racine. Nonparametric estimation of distributions with categorical and continuous data. *Journal of Multivariate Analysis*, 86(2):266–292, August 2003.

[21] V. Markl, P. Haas, M. Kutsch, N. Megiddo, U. Srivastava, and T. Tran. Consistent selectivity estimation via maximum entropy. *The VLDB Journal*, 16:55–76, 2007. 10.1007/s00778-006-0030-1.

[22] M. H. V. M. Michael Saecker, Nikolaj Leischner. Gpu processing in database systems. Technical report, AMD Fusion Developer Summit, 2011.

[23] G. Moerkotte, T. Neumann, and G. Steidl. Preventing bad plans by bounding the impact of cardinality estimation errors. *Proc. VLDB Endow.*, 2(1):982–993, Aug. 2009.

[24] N. Reddy and J. R. Haritsa. Analyzing plan diagrams of database query optimizers. In *Proceedings of the 31st international conference on Very large data bases*, VLDB '05, pages 1228–1239. VLDB Endowment, 2005.

[25] N. Satish, C. Kim, J. Chhugani, A. D. Nguyen, V. W. Lee, D. Kim, and P. Dubey. Fast sort on cpus and gpus: a case for bandwidth oblivious simd sort. In *Proceedings of the 2010 international conference on Management of data*, SIGMOD '10, pages 351–362, New York, NY, USA, 2010. ACM.

[26] S. Seo, G. Jo, and J. Lee. Performance characterization of the nas parallel benchmarks in opencl. In *Workload Characterization (IISWC), 2011 IEEE International Symposium on*, pages 137 –148, nov. 2011.

[27] P. Steffen, R. Giegerich, and M. Giraud. Gpu parallelization of algebraic dynamic programming. In *Proceedings of the 8th international conference on Parallel processing and applied mathematics: Part II*, PPAM'09, pages 290–299, Berlin, Heidelberg, 2010. Springer-Verlag.

[28] M. P. Wand and M. C. Jones. Comparison of smoothing parameterizations in bivariate kernel density estimation. *Journal of the American Statistical Association*, 88(422):520–528, June 1993.

[29] R. Wu, B. Zhang, M. Hsu, and Q. Chen. Gpu-accelerated predicate evaluation on column store. In *Proceedings of the 11th international conference on Web-age information management*, WAIM'10, pages 570–581, Berlin, Heidelberg, 2010. Springer-Verlag.

APPENDIX

A. KERNEL DENSITY ESTIMATION

This section serves as a quick introduction into Kernel Density Estimation (KDE), a non-parametric technique for estimating probability densities from a sample. This introduction is based on multiple publications from statistics that describe KDE in more detail [1, 28, 18].

KDE defines an estimator $\hat{p}(\vec{x})$ for the underlying probability distribution $p(\vec{x})$ of a sample of data points. We assume the sample $\vec{x}^{(1)}, \ldots, \vec{x}^{(s)} \in \mathbb{R}^d$ was drawn independently and identically distributed (iid) from $p(\vec{x})$.

A.1 Estimating the Density at a Point

Equation (1) shows the base formula for KDE, defining the density estimate $\hat{p}(\vec{x})$ at a given data point $\hat{x} \in \mathbb{R}^d$ [28]:

$$\hat{p}(\vec{x}) = \frac{1}{s} \sum_{i=1}^{s} K_H\left(\vec{x} - \vec{x}^{(i)}\right) \tag{1}$$

The density estimate for a point $\vec{x}$ is computed as the averaged likelihood from n local probability densities which are centered at the sample points. In KDE, all local probability densities have the same shape and orientation. They are defined by the function $K_H : \mathbb{R}^d \to \mathbb{R}$:

$$K_H(\vec{x}) = \frac{1}{|H|} K\left(H^{-1}\vec{x}\right) \tag{2}$$

There are two components in equation (2), that have to be picked: The *bandwidth matrix* H and the *kernel function* $K(\vec{x})$.

Kernel function The function $K : \mathbb{R}^d \to \mathbb{R}$ defines the shape of the local probability density functions. Any function that defines a symmetric probability density[11] is a valid choice for K. A typical choice is the Gaussian kernel, which centers a normal probability distribution around each sample:

$$K_G(\vec{x}) = (2\pi)^{-\frac{d}{2}} \exp\left(-\frac{1}{2}\vec{x}^T\vec{x}\right) \tag{3}$$

A different possible choice is the Epanechnikov kernel, which uses a truncated quadratic function as the local distribution:

$$K_E(\vec{x}) = \left(\frac{3}{4}\right)^d \cdot \prod_{i=1}^{d} \left(1 - x_i^2\right) \cdot \mathbb{1}_{|x_i| \le 1} \tag{4}$$

Here $\mathbb{1}_p$ is the indicator function, which takes the value one if the predicate p holds and zero otherwise. [1]

Bandwidth matrix The bandwidth matrix $H \in \mathbb{R}^{d \times d}$ determines the strength and orientation of the local density function. In order to be a valid bandwidth-matrix, H has to be symmetric and positive definite.

The actual choice of kernel function K does not play a substantial role in the estimation quality of KDE [1]. Therefore, it is common practice to choose a kernel function that simplifies computation or derivation. The choice of the bandwidth matrix H on the other hand is crucial to the estimation quality [18]. If H is chosen "too small", the estimator won't be smoothed enough, resulting in a very spiky estimator that overfits the sample. On the other hand, if H is chosen "too large", the estimator will be smoothed to strongly, losing much of the local information and underfitting the distribution. Figure 11 demonstrates the influence of the choice of bandwidth on the estimation quality of KDE.

[11] This means that the function K has to observe the following two properties:

1. $\int_{\mathbb{R}^d} K(\vec{x})\, d\vec{x} = 1$
2. $\forall \vec{x} \in \mathbb{R}^d : K(\vec{x}) = K(-\vec{x})$

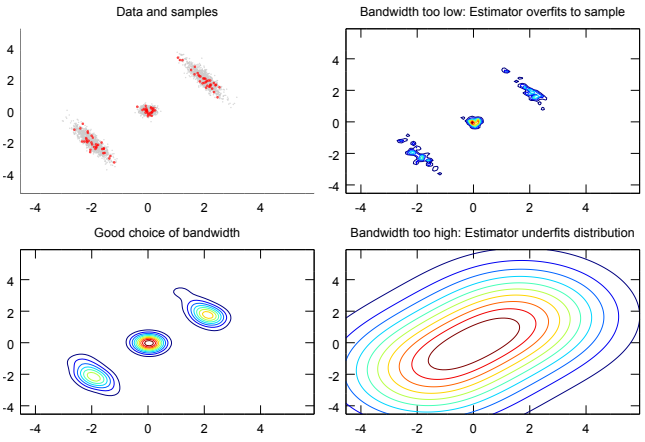

Figure 11: Impact of the bandwidth on estimation quality.

A.2 Density Estimation for a Region

KDE can also be used to compute the probability mass $p(\Omega)$ that is enclosed in a region $\Omega \subseteq \mathbb{R}^d$. Mathematically, this amounts to integrating (1) over all points within the region:

$$\begin{aligned}\hat{p}(\Omega) &= \int_\Omega \underbrace{\hat{p}(\vec{x})}_{(1)}\, d\vec{x} \\ &= \frac{1}{s}\sum_{i=1}^{s} \int_\Omega K_H\left(\vec{x} - \vec{x}^{(i)}\right)\end{aligned} \tag{5}$$

We can give a simple closed-form expression of (5) under the following two assumptions:

1. We assume that Ω is a hyperrectangle, i.e., it is the Cartesian product of intervals within the d dimensions: $\Omega = [l_1, u_1] \times \ldots \times [l_d, u_d] \subseteq \mathbb{R}^d$.
2. We assume that H is a diagonal matrix, such that $H = \operatorname{diag}(h_1, \ldots, h_n)$.

Under those assumptions, we can show[12] that (5) can be expressed for the Epanechnikov kernel as:

$$\hat{p}(\Omega) = \frac{1}{s}\sum_{i=1}^{s}\prod_{j=1}^{d} \frac{3 \cdot \mathbb{1}_{l'_j \le u'_j}}{4 \cdot h_j^3} \left[\left(h_j^2 - x_j^{(i)2}\right)\left(u'_j - l'_j\right) - \frac{u'^3_j - l'^3_j}{3} + x_j^{(i)}\left(u'^2_j - l'^2_j\right)\right] \tag{6}$$

In this formula, the region boundaries of Ω were adjusted to the support of the local density around the sample points. The adjusted boundaries are: $l'_j = \max\left(l_j, x_j^{(i)} - h_j\right)$ and $u'_j = \min\left(x_j^{(i)} + h_j, u_j\right)$.

[12] For the derivation, refer to Appendix Section B.

A.3 Bandwidth Selection

Selecting the optimal bandwidth is the most important parameter choice for a KDE estimator [18]. The bandwidth is typically chosen to minimize some error measure. In statistics literature, the most commonly chosen error metric is the *Mean Integrated Square Error* (MISE). Since computing the MISE requires knowledge of the exact underlying distribution, it is custom to use techniques that approximate MISE from the data sample. These approximations give rise to typical bandwidth selection algorithms like cross-validation and plug-in methods [18].

A solid first approximation for the bandwidth $\hat{h_i}$ in dimension i can be computed by assuming that the underlying probability distribution is normal. Under this assumption, it has been shown [1] that the optimal bandwidth for the Epanechnikov kernel is given by:

$$\hat{h_i} = \sqrt{5} \cdot n^{-\frac{1}{d+4}} \cdot \hat{\sigma_i} \tag{7}$$

B. DERIVATION OF THE INTEGRATION FORMULA

In this section, we derive a closed-form expression for the KDE range estimator. The derivation is adapted from the ones found in [4, 14]. We begin with the definition of the range estimator in equation (5):

$$\hat{p}(\Omega) = \frac{1}{s} \sum_{i=1}^{s} \int_{\Omega} K_H \left(\vec{x} - \vec{x}^{(i)} \right)$$

We can further expand this definition, by plugging in the definition of K_H from equation (2), giving us the following:

$$\hat{p}(\Omega) = \frac{1}{s \cdot |H|} \sum_{i=1}^{s} \int_{\Omega} \underbrace{K \left(H^{-1} \left[\vec{x} - \vec{x}^{(i)} \right] \right)}_{=K_{\Omega}^{(i)}}$$

The term $K_{\Omega}^{(i)}$ in equation (8) denotes the fraction of probability mass from the local density centered around sample point $\vec{x}^{(i)}$, that lies within Ω. The final estimate is the normalized sum of the local contributions from all sample points.

We will now derive a closed form expression for computing the local density contributions $K_{\Omega}^{(i)}$, assuming we use the Epanechnikov kernel from (4). First, we make use of our assumption of a hyperrectangular region $\Omega = [l_1, u_1] \times \ldots \times [l_d, u_d] \subseteq \mathbb{R}^d$, which allows us to evaluate the integral in the definition of $K_{\Omega}^{(i)}$ from (8) for each dimension seperately:

$$\begin{aligned} K_{\Omega}^{(i)} &= \int_{\Omega} K \left(H^{-1} \left[\vec{x} - \vec{x}^{(i)} \right] \right) d\vec{x} \\ &= \int_{l_1}^{u_1} \ldots \int_{l_d}^{u_d} K \left(H^{-1} \left[\vec{x} - \vec{x}^{(i)} \right] \right) dx_d \ldots dx_1 \end{aligned} \tag{8}$$

In order to simplify the formula further, we use the following two observations:

1. The assumption of a diagonal bandwidth matrix $H = \text{diag}(h_1, \ldots, h_d)$ allows us to express the matrix-vector product within equation (8) as:

$$H^{-1} \left[\vec{x} - \vec{x}^{(i)} \right] = \begin{pmatrix} \frac{x_1 - x_1^{(i)}}{h_1} \\ \vdots \\ \frac{x_d - x_d^{(i)}}{h_d} \end{pmatrix} \tag{9}$$

2. The multivariate Epanechnikov kernel is - like the multivariate Gaussian kernel - a so-called product kernel, meaning it can be expressed as a product of single-dimensional kernels:

$$K(\vec{x}) = \prod_{i=1}^{d} \hat{K}(x_i) \tag{10}$$

Plugging both (10) and (9) back into (8), we arrive at:

$$K_{\Omega}^{(i)} = \int_{l_1}^{u_1} \ldots \int_{l_d}^{u_d} \prod_{j=1}^{d} \hat{K} \left(\frac{x_j - x_j^{(i)}}{h_j} \right) dx_d \ldots dx_1 \tag{11}$$

In equation (11), each of the d factors within the product depends on exactly one of the integration variables. This allows us to push the integration into the product, expressing the original d-dimensional integration as the product of d one-dimensional integrals:

$$K_{\Omega}^{(i)} = \prod_{j=1}^{d} \int_{l_j}^{u_j} \hat{K} \left(\frac{x_j - x_j^{(i)}}{h_j} \right) dx_j \tag{12}$$

We will now plug the definition of the Epanechnikov kernel K_E from equation (4) into equation (12). For the sake of notational convenience, we will however only observe one of the d single-dimensional integrals from (12).

$$\begin{aligned} &\int_{l_j}^{u_j} \hat{K}_E \left(\frac{x_j - x_j^{(i)}}{h_j} \right) dx_j = \\ &\int_{l_j}^{u_j} \frac{3}{4} \left(1 - \left[\frac{x_j - x_j^{(i)}}{h_j} \right]^2 \right) \cdot \mathbb{1}_{\left| \frac{x_j - x_j^{(i)}}{h_j} \right| \leq 1} dx_j \end{aligned} \tag{13}$$

The only part in equation (13) that is not trivially integrable is the indicator function. The effect of the indicator is, that the integrand becomes zero for all values of the integration variable x_j that are not within the interval $\left[x_j^{(i)} - h_j, x_j^{(i)} + h_j \right]$ - also called the *support*. This means we can get rid of the indicator function by adjusting the integration bounds to the support:

$$\begin{aligned} &\int_{l_j}^{u_j} K_E \left(\frac{x_j - x_j^{(i)}}{h_j} \right) dx_j = \\ &\mathbb{1}_{l_j' \leq u_j'} \cdot \frac{3}{4} \cdot \int_{l_j'}^{u_j'} \left(1 - \left[\frac{x_j - x_j^{(i)}}{h_j} \right]^2 \right) dx_j \end{aligned} \tag{14}$$

In equation (14), we use the modified integration bounds $l_j' = \max \left(l_j, x_j^{(i)} - h_j \right)$ and $u_j' = \min \left(x_j^{(i)} + h_j, u_j \right)$, which are adjusted to use the intersection of integration bound and support. Finally, we multiply the whole term by a new indicator function to guarantee that the integral becomes zero if the integration bound and the support are disjoint, i.e., if $l_j' > u_j'$. The remaining integrand in (14) is now a

quadratic function that is easy to integrate:

$$\begin{aligned}
&\int_{l'_j}^{u'_j} \left(1 - \left[\frac{x_j - x_j^{(i)}}{h_j}\right]^2\right) dx_j \\
&= \int_{l'_j}^{u'_j} \left(1 - \frac{x_j^2 - 2 \cdot x_j \cdot x_j^{(i)} + x_j^{(i)2}}{h_j^2}\right) dx_j \\
&= \frac{1}{h_j^2} \cdot \int_{l'_j}^{u'_j} \left(h_j^2 - x_j^2 + 2 \cdot x_j \cdot x_j^{(i)} - x_j^{(i)2}\right) dx_j \qquad (15)
\end{aligned}$$

The antiderivative of the integrand in equation (15) is given by the following polynomial: $-\frac{x_j^3}{3} + x_j^{(i)} x_j^2 + \left(h_j^2 - x_j^{(i)2}\right) x_j$. Plugging this back into equation (12) gives us the final integration formula for the local range density around point i:

$$\begin{aligned}
K_\Omega^{(i)} &= \prod_{j=1}^{d} \frac{3 \cdot \mathbb{1}_{l'_j \leq u'_j}}{4 \cdot h_j^2} \\
&\left[\left(h_j^2 - x_j^{(i)2}\right)\left(u'_j - l'_j\right) - \frac{u_j'^3 - l_j'^3}{3} + x_j^{(i)}\left(u_j'^2 - l_j'^2\right)\right] \qquad (16)
\end{aligned}$$

Plugging (16) back into (8) and observing that for a diagonal H: $|H| = \prod_{i=1}^{d} h_i$, we arrive at the final estimation formula:

$$\begin{aligned}
\hat{p}\left(\Omega\right) &= \frac{1}{s} \sum_{i=1}^{s} \prod_{j=1}^{d} \frac{3 \cdot \mathbb{1}_{l'_j \leq u'_j}}{4 \cdot h_j^3} \\
&\left[\left(h_j^2 - x_j^{(i)2}\right)\left(u'_j - l'_j\right) - \frac{u_j'^3 - l_j'^3}{3} + x_j^{(i)}\left(u_j'^2 - l_j'^2\right)\right]
\end{aligned}$$