Generalized Database Index Structures on Massively Parallel Processor Architectures

Dissertation
submitted in partial fulfillment of the
requirements for the degree of

Doktor–Ingenieur (Dr.–Ing.)

by

Dipl.–Inf. Felix Beier

Technische Universität Ilmenau
Faculty of Computer Science and Automation

Submission date: 4th of December, 2018
PhD thesis defense: 15th of April, 2019

Reading Committee:

1. Univ.-Prof. Dr.–Ing. habil. Kai-Uwe Sattler
2. Univ.-Prof. Dr.–Ing. habil. Wolfgang Lehner
3. Dr. Götz Graefe

urn:nbn:de:gbv:ilm1-2019000101

Abstract

Index structures are ubiquitous in database management systems as well as in other applications that require efficient access methods in order to identify entries in large data volumes, which match specific search criteria. Height-balanced search trees represent one of the most important classes of indexes. They can be configured with various strategies for structuring the indexed search space for a given data set and for pruning it when different kinds of search queries are answered. In order to facilitate the development of application-specific tree variants, index frameworks, such as GiST (Generalized Search Tree), exist that provide a reusable library of commonly shared tree management functionality. By specializing internal data organization strategies, the framework can be customized to create an index that is efficient for an application's data distribution and data access characteristics. Because the majority of the framework's code can be reused, development and testing efforts are significantly lower, compared to an implementation from scratch. However, none of the existing frameworks supports the execution of index operations on massively parallel processor architectures, such as GPUs.

Enabling the use of such processors for generalized index frameworks is the goal of this thesis. By compiling state-of-the-art techniques from a wide range of CPU- and GPU-optimized indexes, a GiST extension is developed that abstracts the physical execution aspect of generic, tree-based search queries. Tree traversals are broken-down into vectorized processing primitives that can be scheduled to one of the available (co-)processors for execution. Further, a CPU-based implementation is provided as well as a new GPU-based algorithm that, unlike prior art in this area, does not require that the index is fully stored inside a GPU's main memory buffer, which is rather limited in size. The applicability of the extended framework is assessed for image rendering engines and, based on microbenchmarks, the parallelized algorithm performance is compared for different CPU and GPU generations. It will be shown that cases exist, where the GPU clearly outperforms the CPU and vice versa, depending on runtime-specific workload parameters.

In order to leverage the strengths of each processor type for mixed workload scenarios, an adaptive scheduler is presented that can be used to schedule index operations on a hybrid system, where multiple different processor devices exist. After an initial calibration phase that is used to train the scheduler's decision model, it is able to use this model to estimate total execution costs for a given operation and select the best device for it. With the help of a tree traversal simulation that models various query workloads, different scheduling strategies are evaluated. It will be shown that the adaptive scheduler can be used to make near-optimal decisions, increasing query throughputs by an order of magnitude and more, depending on the workload.

Zusammenfassung

Index Strukturen sind allgegenwärtig in Datenbanksystemen und anderen Anwendungen, die eine effiziente Möglichkeit benötigen um in großen Datensätzen nach Einträgen zu suchen, die bestimmte Suchkriterien erfüllen. Suchbäume nehmen dabei einen bedeutende Rolle ein, da sie mit verschiedensten Strategien konfiguriert werden können um den Suchraum zu strukturieren und damit später die für ein Suchergebnis irrelevante Bereiche von der Bearbeitung ausschließen zu können. Die Entwicklung von anwendungsspezifischen Indexen wird durch Frameworks wie GiST unterstützt, welche wiederverwendbare Algorithmen zur Verwaltung von Baumstrukturen zu Verfügung stellen. Diese Frameworks können über interne Schnittstellen spezialisiert werden um Indexe zu modellieren, die speziell auf die Datenverteilung und Zugriffsmuster einer Anwendung zugeschnitten sind. Da dabei ein Großteil des Codes wiederverwendet werden kann, reduziert diese Art der Implementierung Entwicklungs- und Testkosten erheblich im Vergleich zu einer Neuimplementierung. Jedoch unterstützt keines der heute bereits existierenden Frameworks die Verwendung von hochgradig parallelen Prozessorarchitekturen wie GPUs.

Solche Prozessoren für generische Index Frameworks nutzbar zu machen, ist Ziel dieser Arbeit. Dazu werden Techniken aus verschiedensten CPU- und GPU-optimierten Indexen analysiert und für die Entwicklung einer GiST-Erweiterung verwendet, welche die für eine Suche in Suchbäumen nötigen Berechnungen abstrahiert. Traversierungsoperationen werden dabei auf vektorisierte Primitive abgebildet, die auf parallelen Prozessoren implementiert werden können. Die Verwendung dieser Erweiterung wird beispielhaft an einem CPU Algorithmus demonstriert. Weiterhin wird ein neuer GPU-basierter Algorithmus vorgestellt, der im Vergleich zu bisherigen Verfahren, ein dynamisches Nachladen der Index Daten in den Hauptspeicher der GPU unterstützt, da dieser in der Größe sehr beschränkt ist. Die Praktikabilität des erweiterten Frameworks wird am Beispiel von Anwendungen aus der Computergrafik untersucht und die Performanz der verwendeten Algorithmen mit Hilfe eines Benchmarks auf verschiedenen CPU- und GPU-Modellen analysiert. Dabei wird gezeigt, unter welchen Bedingungen die parallele GPU-basierte Ausführung schneller ist als die CPU-basierte Variante – und umgekehrt.

Um die Stärken beider Prozessortypen in einem hybriden System ausnutzen zu können, wird ein Scheduler entwickelt, der nach einer Kalibrierungsphase die erwarteten Ausführungszeiten einer Operation auf allen Prozessoren abschätzen und somit den geeignetsten Prozessor wählen kann. Mit Hilfe eines Simulators für Baumtraversierungen werden verschiedenste Scheduling Strategien verglichen. Dabei wird gezeigt, dass die Entscheidungen des kalibrierten Schedulers kaum vom Optimum abweichen und, abhängig von der simulierten Last, die erzielbaren Durchsätze für die parallele Ausführung mehrerer Suchoperationen durch hybrides Scheduling um eine Größenordnung und mehr erhöht werden können.

Danksagung

An dieser Stelle möchte ich mich ganz herzlich bei allen bedanken, die mich bei der Erstellung der Arbeit unterstützt haben.

Allen voran meinem Doktorvater, Kai-Uwe Sattler, dessen Tür immer offen war um meine Fragen zu besprechen. Er sorgte auch stets für die nötige Motivation neben meiner regulären Arbeit und selbst nach Rückschlägen am Ball zu bleiben und die Dissertation erfolgreich fertigzustellen.

Weiterhin möchte ich auch Goetz Graefe und Ilia Petrov danken, die sich die Zeit genommen haben frühe Versionen meiner Dissertation zu lesen und mir wertvolles Feedback dazu zu geben. Insbesondere Goetz' Anmerkungen haben dafür gesorgt, den Fokus der Arbeit zu schärfen.

Ich möchte auch allen danken, die an zahlreichen Experimenten und Publikationen mitgewirkt haben: Sebastian Breß für die Arbeit am Scheduler, Hannes Rauhe für zahlreiche Experimente mit der GPU und Jens Teubner für seine Beiträge zum Hardware Kapitel.

Auch zahlreiche Studenten standen mir bei meinen Experimenten zur Seite: Torsten Kilias bei der Entwicklung der GiST Erweiterung, Daniel Löber bei der Parallelisierung der Nachbarschaftssuche; Steffen Hirte bei der Integration von GiST in POV Ray; Tafil Kaijtazi bei der Integration von GiST in Ogre3D.

Mein Dank gilt auch meinen ehemaligen Kollegen der TU Ilmenau: Stephan Baumann und Stefan Hagedorn für die vielen fachlichen Diskussionen im Eckbüro; Heiko Betz, Matthias Baag, Martin Sauerbrey und Peter Henkel für die Betreuung der Recheninfrastruktur; Markus Färber und Beat Brüderlin für ihre Unterstützung bei Computergrafik-Fragen; und natürlich den vielen anderen guten Geistern der Uni und des Fachgebietes Datenbanken & Informationssysteme.

Danke auch an meine IBM Kollegen, die mich unterstützt haben: insbesondere an Knut Stolze für die langjährige Zusammenarbeit und die Arbeit an zahlreichen Publikationen und Patenten; Oliver Draese und Jürgen Schimpf für ihre Unterstützung bei meiner Bewerbung um den IBM PhD Fellowship Award; meinen Managerinnen Corinna Peters und Anja Nicolussi, die mir so manches Mal geholfen haben Zeit für die Fertigstellung der Dissertation freizuschaufeln.

Zu guter Letzt gilt mein Dank natürlich meiner Familie und meinen Freunden, die mir im privaten Bereich den stets Rücken freigehalten haben. Insbesondere meine Partnerin, Linda Stocker, hat mir auf den "letzten Metern" sehr geholfen die Arbeit zügig zum Abschluss zu bringen.

Danke!

Contents

Acronyms

ADT	Abstract Data Type
ALM	Adaptive Logical Module
ALU	Arithmetic Logical Unit
AoS	Array of Structures
API	Application Programming Interface
APU	Accelerated/Advanced Processing Unit
ART	Adaptive Radix Tree
AVX	Advanced Vector Extensions
BFS	Breadth-First Search
BVH	Bounding Volume Hierarchy
CAD	Computer-Aided Design
CAM	Computer-Aided Manufacturing
CC-GiST	Cache-Conscious Generalized Search Tree
CPU	Central Processing Unit
CUDA	formerly Compute Unified Device Architecture
DAG	Directed Acyclic Graph
DBMS	Database Management System

DFS	Depth-First Search
dGPU	dedicated Graphics Processing Unit
DMA	Direct Memory Access
DPP	Data-Parallel Processor
DSP	Digital Signal Processor
FAST	Fast Architecture Sensitive Tree
FPGA	Field-Programmable Gate Array
GiST	Generalized Search Tree
GPGPU	General-Purpose computation on Graphics Processing Units
GPU	Graphics Processing Unit
HT	HyperTransport
ICD	Installable Client Driver
iGPU	integrated Graphics Processing Unit
JIT	Just In Time
LLVM	LLVM compiler infrastructure project, formerly Low Level Virtual Machine
MBR	Minimum Bounding Rectangle
MIC	Intel Many Integrated Core Architecture
MPI	Message Passing Interface
NUMA	Non-Uniform Memory Access
Ogre 3D	Object-Oriented Graphics Rendering Engine
OpenCL	Open Computing Language
OpenGL	Open Graphics Library

PCIe Peripheral Component Interconnect Express

POD Plain Old Data Type

POV-Ray Persistence of Vision Raytracer

QPI Intel QuickPath Interconnect

SaIL Spatial Index Library

SHM Shared Memory

SIMD Single Instruction Multiple Data

SIMT Single Instruction Multiple Thread

SM Streaming Multiprocessor, sometimes SMX

SMT Simultaneous Multi-Threading

SoA Structure of Arrays

SoC System on Chip

SP-GiST Space Partitioning-Generalized Search Tree

SPH Space Partitioning Hierarchy

SPMD Single Program Multiple Data

SSE Streaming SIMD Extensions

TD Tag Directory

VPU Vector Processing Unit

VRAM Video RAM

XXL eXtensible and fleXible Library

List of Figures

List of Tables

Algorithms and Program Code

Chapter 1

Introduction

Index structures are mandatory for implementing efficient search operations on large data volumes in traditional relational DBMS (Database Management System) applications. Efficient indexing is also important for applications from other domains, which do not necessarily use traditional DBMSs for storing and processing the available information, for example, computer graphic or other multi-media data applications, and scientific data management.

One challenge in these domains is that mature, well-understood index structures that are ubiquitous in the relational world for supporting various filter operations on potentially multi-dimensional data sets are not always working properly due to certain data characteristics. Therefore, numerous specialized indexes have been developed that are carefully designed for the workload characteristics of the application. Among these indexes, tree-based structures represent the most prominent class. Because a lot of tree management and traversal functionality is common between different instances of this index class, frameworks have been developed that generalize these aspects and provide a common implementation that can be shared between implementing indexes variants. By specializing custom extension points that configure the actual behavior of an index structure, a lot of development efforts can be saved, which fosters rapid prototyping that is required to identify the best structure(s) for a domain-specific use case.

The latest developments in hardware architectures present a new challenge in this context. Due to physical limitations, processor clock frequencies cannot be increased any further, which, from a software perspective, up to that time led to transparent performance gains, without the need to adjust existing code. Instead, hardware manufacturers now focus on increasing the number of processors as well as the bandwidth of interconnecting memory bus systems, providing an infrastructure that can be used by the software to gain speedups through parallelism. This trend has been picked-up by research and industry to develop new, parallelized indexing algorithms that can efficiently utilize new hardware resources, like multi-core CPUs (Central Processing Units) or GPUs (Graphics Processing Units). This hardware combination is of particular interest here, since CPUs and GPUs are already commonly used in multi-media and scientific data management domains, e. g., for rendering visual representations of the data or for running simulations.

The plethora of new hardware-conscious index structures led to a situation, which is comparable to the numerous, logically differing index types, before their implementations have been unified in generic frameworks. Therefore, this thesis addresses the following research questions:

(1) How can parallel (co-)processor support be integrated into index implementations, without re-inventing the wheel for existing tree-based index structures?

(2) How do generic index algorithms need to be adapted so that they can be accelerated by processing them on parallel hardware architectures, CPUs and GPUs in particular?

(3) How can different (co-)processor types be combined in a hybrid system in order to leverage their respective strengths for specific, workload-dependent characteristics?

1.1 Motivation

Section 1.1.1 presents some use cases in more detail to emphasize the importance that these challenging questions are addressed. Typical operations on the application data are discussed as well as their requirements for index structures. Section 1.1.2 highlights the role of index frameworks when it comes to developing actual indexes. The motivation for considering current hardware developments in this context is covered in Section 1.1.3. These dimensions will be combined in this thesis in order to extend the index framework idea for hardware aspects.

1.1.1 Search Operations in Large Data Sets

An effective filtering mechanism is mandatory for many applications that operate on large data sets, because it helps to limit processing to those partitions that are required for executing a certain operation. Index structures are the method of choice in almost every case as they are well-understood and ubiquitous in data management applications such as relational DBMSs. In its simplest form, an index supports search queries for one-dimensional data, e. g., filtering records by a specific attribute. However, there are many applications that operate on more complex structures. CAD (Computer-Aided Design), CAM (Computer-Aided Manufacturing), or geographic/cartographic applications [80], applications that do text processing, genome or image databases [33], and many more were among the first applications that required special search structures for multi-dimensional data.

For example, in the field of computer graphics, 2D images have to be created from 3D models in order to display them on a screen. The source models, usually, consist of billions of triangles as the most common geometric primitive, which are augmented by additional properties, such as surface textures, material properties like opacity or color, and many others. Ray tracing [70, 132] is a well-known technique for this image rendering process that can simulate real-world conditions pretty well, including physical effects, like reflection or refraction. During

this process, the path of light is simulated by rays that are "shot" into the scene. A ray is generated for each pixel in the final image that starts at the virtual camera position. In order to calculate the pixel color, each ray requires a search operation (point query) to determine its intersection point with visible objects in the underlying data model. For each intersection point, another query is required to calculate the incidence of light, originating from simulated light sources in the scene. To obtain photo-realistic images, this process can be repeated recursively to refine the initial results by considering the impact of neighboring objects on the pixel color. For example, additional reflection and refraction rays can be simulated, which originate from each intersection point at the hit object's surface. However, this refinement leads to a potentially exponential growth in the number of search operations.

Another class of rendering algorithms relies on projections of the virtual scene into the camera coordinate system. These algorithms are, usually, employed for interactive CAD/CAM applications [39]. Unlike ray tracing, whose physical simulation of light is rather time consuming and, mostly, calculated offline, CAD software requires user interactions with the scene, e. g., to control the camera or to manipulate objects. Therefore, for each operation, the projections must be calculated in real-time, i. e., only low latencies are acceptable until an image frame is rendered as user feedback. This is a rather challenging task, because the underlying models, such as air planes or power plants, may consist of billions of geometric primitives and additional meta data, like part numbers, part types, material properties, etc. Overall model sizes of tens to hundreds of GB are not uncommon in this domain. In order to reduce the amount of data that needs to be passed through the graphical rendering pipeline, systems implementing these projection-based algorithms rely on visibility-based filtering and incremental refinement of rendering results after each simulation step. Therefore, efficient searches (range queries) are required, which quickly return object candidates that might be visible in the current camera view. Complex occlusion and opacity effects are calculated later, using this significantly reduced data volume as input. Additional filters can be applied in order to reduce data volumes further, for example, distance-based predicates (clipping planes) or predicates that select certain parts in the model, e. g., electric wires, parts that belong to an engine, etc. Such filter combinations lead to n-dimensional predicates, where $n > 3$.

The aforementioned use cases required direct filtering on the underlying base model. Other uses cases, such as collision detection that is an important building blocks for physical simulations [100, 121], require filtering as part of a (spatial) join operation between several models. In order to determine whether two or more 3D models collide, their boundaries need to be tested for intersection. In order to speedup the process, potentially colliding geometric primitives can be filtered within each simulation step by using spatial access methods.

The last related class of operations that shall be discussed here, is the simulation of distance-based interactions between objects in a virtual scene. For example, earth quake simulations [122] can use 3-dimensional geometries with varying data distributions so that certain conditions of the ground, like density, can be modeled. Shock waves are simulated by calculating the impact of a modified, e. g., shifted, primitive upon its neighbors within a certain range. A similar task is required in the *Blue Brain Project* [146] whose objective is to understand how the net of interconnected neurons in a brain works. Neurons, which are, again, modeled by geometric

primitives, like cylindric shapes, are used in simulations to calculate how electric current propagates through synapses as the interconnecting points between them. These simulations require spatial filtering techniques [190, 193] that allow selection of specific primitives in conjunction with filtering of neighboring objects.

Summarizing the main characteristics of these sample use cases, one can clearly see that the aforementioned applications need to cope with huge volumes of high-dimensional data. They require efficient filtering techniques in order to reduce the amount of data that actually needs to be passed to the complex and time-consuming algorithms. Access methods that shall be used therefor need to be flexible in terms of the data elements which shall be handled as well as in terms of supported search space pruning criteria. This flexibility is particularly important, because properties of the analyzed data set might not be known a priori and, sometimes, require explorative analysis with customizable search algorithms [11].

1.1.2 The Need for Index Frameworks

In order to speed up data access and filtering, the applications mentioned in Section 1.1.1, usually, make use of (multi-dimensional) index structures, like search trees. However, the effort for implementing and testing them is quite significant. A common approach to reduce this implementation complexity is to reuse existing implementations in the application's data management layer. Relational general purpose DBMSs provide industrial-strength implementations of common bounding volume data structures, such as B-Trees [20, 48] or R-Trees [81], that can easily be used via declarative interfaces if the application's data model has been mapped to the relational model before. But the latter usually does not happen in practice. Instead, software solutions are built use case-driven, including a custom data management layer that has been tailored towards the needs of the application. For example, partitioning trees like Octrees [150] or k-d-Trees [25] are widely used in computer graphics applications. Thus, each domain, somehow independently, developed its own solutions that work quite well for their requirements. The somehow aged, but nevertheless telling Figure 1.1, compiled by Gaede/Günther [64], illustrates how many index variants exist, each claiming superior performance for a particular workload.

This development is quite natural, but generates a lot of implementation overheads. Even if the code for specific indexes is provided as reusable library, there are often cases where it needs to be adjusted, because of some application-specific workload characteristics that have not been considered in the original library design, effectively leading to a new index variant. For example, the visibility-guided image rendering algorithm in Interviews3D [39] augments nodes in a k-d-Tree [25] by dynamic state information that encodes whether the scene partition covered by that node was visible in the last frame or not, e. g., in case if it has been occluded before. This "visibility band" inside the tree can be leveraged to reduce the overhead for object filter algorithms by limiting them to the changes that occurred since the last frame, which are expected to be rather small compared to a full traversal of the entire scene hierarchy.

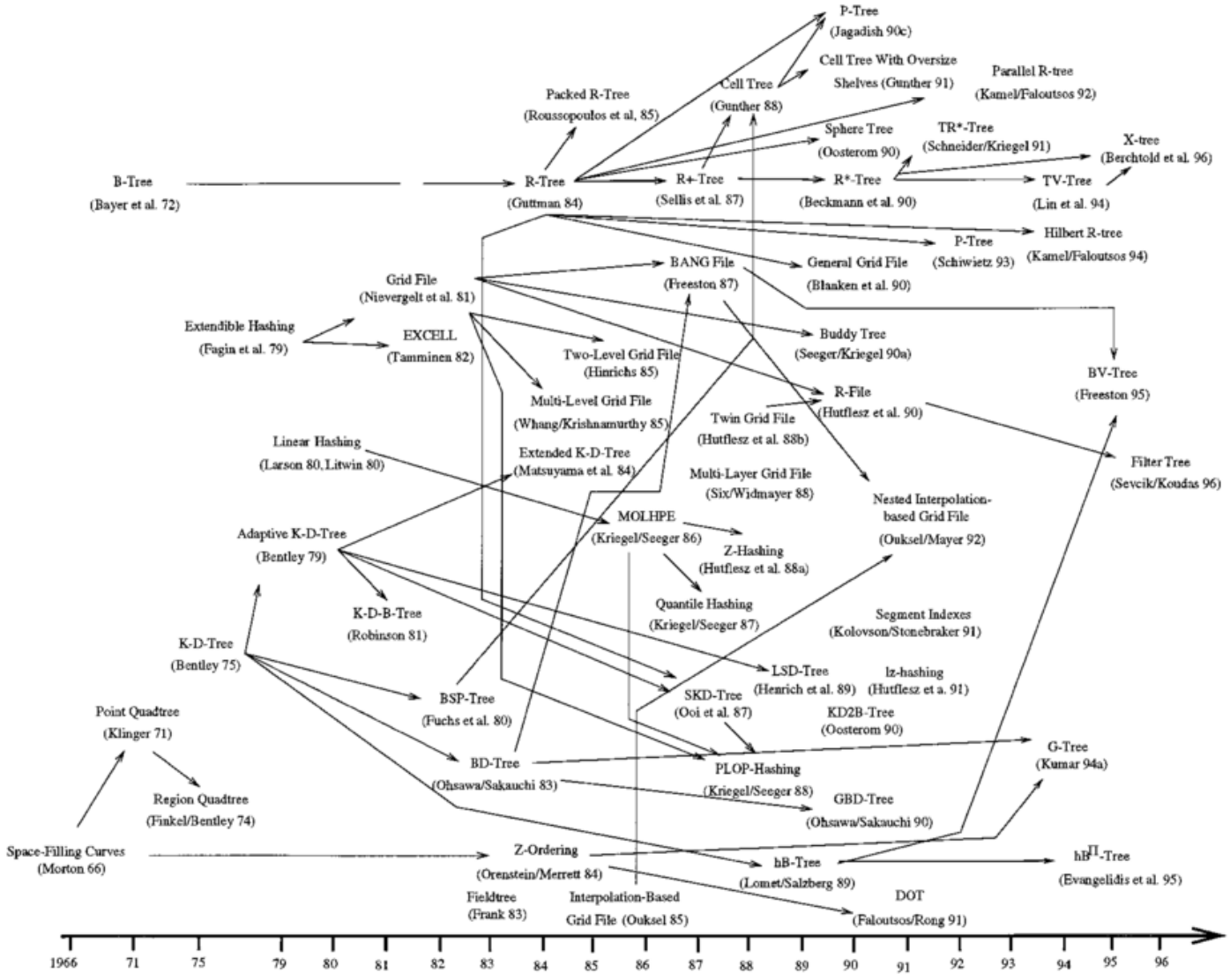


Figure 1.1: History of Multidimensional Access Methods
Source: [64]

A second example is the access method that was developed for the *Blue Brain Project* [11, 190, 193]. The initial approach of using a standard R-Tree implementation failed because of the very dense data distribution that caused a lot of overlaps among the MBRs (Minimum Bounding Rectangles) inside R-Tree nodes, effectively turning each tree traversal into a full scan of the whole data set. As a solution, FLAT [193] has been created, a new specialized index structure that makes use of this density by linking nearby nodes with each other. The tree traversal method has also been modified. An initial tree descending algorithm yields starting points which are then processed in a subsequent neighbor exploring phase to do the final filtering.

There are many other examples where certain data properties, like the "*the curse of dimensionality*" led to the development of new index structures [26, 44, 133]. Such tailor-made solutions work pretty well, at least initially, where the application is in a prototype phase and non-performance related requirements are less important. However, once the application grows and is geared towards commercial use, other functional aspects, like concurrency handling or durability guarantees, need to be addressed, too. While concurrent updates were completely ignored in FLAT, they became particular important for the commercial Interviews3D product. Without correct synchronization mechanisms, concurrent operations executed by multiple end users on a shared data model are not feasible without unintended side effects.

In order to facilitate the development of indexes that correctly implement such complex algorithmic logic, several index frameworks have been proposed [59, 82, 86]. By generalizing common logic that is shared between a majority of index structures and providing it as customizable library code, the use of such frameworks can save development time and testing efforts. Examples for such generalizable index algorithms are:

- clustering of "similar" (key, value)-records inside tree nodes for search space pruning
- management of tree nodes when over- or underflows occur in internal data structures upon record insertion or removal
- tree traversals for answering search queries with correct concurrency control handling
- serialization of in-memory tree data to persistent storage devices with internal buffering mechanisms

Extension points are provided by these libraries to allow injection of index-specific information that distinguishes the various existing index variants from each other. By, for example, defining custom key data types that are stored inside tree nodes, a similarity metric for clustering nearby entries, node visitation strategies for tree traversals, or special query predicates for modeling different kinds of search operations, the aforementioned algorithms can be customized by index developers to parameterize the framework for a specific use case without the need to change any of the shared code.

Summing this up, one can clearly see the need to define custom index structures for different use cases that need to cope with large data sets and non-standard data accesses. Index frameworks are useful to implement such indexes, because they reduce development efforts for complex algorithms that are generalizable and share code between different implementations. A flexible framework design is required to allow efficient prototyping for specific applications, providing extension points that can be used to customize a concrete implementation.

1.1.3 Hardware-Accelerated Search Operations

Many of the motivating applications from Section 1.1.1 run workloads that offer great potential for parallelization. For example, ray tracing generates many independent search queries in each iteration, collision detection requires distance calculations and intersection tests for many geometric primitives that are involved in a spatial join, and physical simulations need to examine many neighboring data elements in order to calculate their mutual interactions in each simulation step. Therefore, exploiting this potential on modern parallel processor architectures represents the second dimension that motivates the research of this thesis.

Parallel processor architectures gained considerable attention recently. The many-cited end of *Moore's Law* [119, 188, 200], caused by physical limitations, such as thermal effects, energy constraints, etc., that prevent a further increase of a processor's clock frequency in order to improve application performance, let to a strong industry focus on the development of processors that are comprised of many independent cores. A multitude of different processor architectures have been developed, each following a slightly different approach of handling parallelism. Multi-core CPUs, for example, are designed for task parallelism and efficiency of sequential steps by embedding complex algorithms, like branch prediction or out-of-order processing, on the chip. GPUs clearly focus on data parallelism with huge arrays of simple processor cores, while FPGAs (Field-Programmable Gate Arrays) can be used to embed any algorithm directly in hardware and handle pipeline parallelism in streaming tasks pretty well. Oftentimes, a parallel processor is used as coprocessor in conjunction with another processor for sequential tasks. Developer toolchains, which are mostly provided by hardware manufactures, simplify the management of these processors by abstracting many low-level details, such as the creation and management of parallel tasks, and support software developers to model, implement, and tune parallel algorithms.

Of course, these new tools have been picked-up by the database community to analyze how the new methodology can be leveraged for the development of index structures that make use of the abundance of available processing resources. Well-known indexes, like B-Trees or R-Trees, have been analyzed to speed up lookups with parallelized search operations [104, 141, 167, 179], to optimize data layouts for efficient memory access [171], or to revise existing structures so that they can be used on different processor device types [111, 194]. Likewise, research in other domains, e. g., computer graphics, has been conducted [62, 90, 143], where, naturally, the use of coprocessors like GPUs is already state of the art since decades.

Parallelizing (indexing) algorithms for modern (co)processor architectures is a challenging task because of the many subtleties that need to be considered when an algorithm is tuned for a specific architecture. Designing efficient data structure layouts for given access patterns is equally important as mapping algorithms to fine-granular, independent units of work that can be scheduled to the available processors. While the former are implementation and tuning tasks that are supported by vendor-provided toolchains, the latter requires a different way of algorithmic thinking [192], which can also be considered the main challenge here. According to *Amdahl's Law* [15], which can also be applied to modern parallel processor architectures [88], the parallelization is an important challenge to solve because the ratio of sequential to parallelized code significantly impacts an application's overall scalability. Therefore, this challenge shall, to some extend, be addressed for the field of tree-based index structures in the following.

1.2 Contributions of this Thesis

The overall goal of this thesis is to develop and evaluate a framework for implementing index structures that utilize the parallel processing resources provided by modern computation hardware. The framework offers extension points that can be specialized by an index developer in order to define different kinds of search trees as basic data structures for custom traversal operations. The framework itself handles tree management operations, automatically parallelizes traversal algorithms, and schedules them to available processor devices in the runtime system. Using the framework bridges the gap between existing generic index structures that are flexible and highly specialized implementations that have been tuned for specific processing devices.

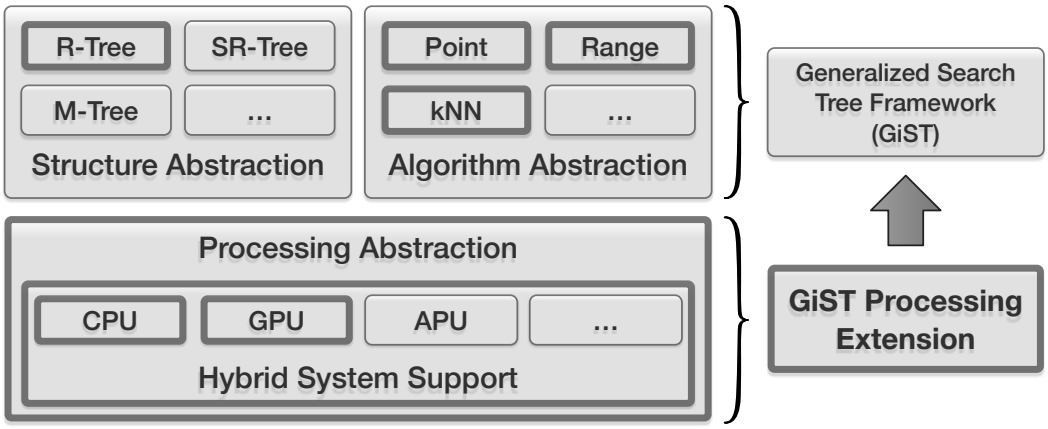


Figure 1.2: Overview of Contributions

More specifically, the following main contributions are made by this thesis (cf. Figure 1.2):

1. The well-known GiST framework for modeling height-balanced search trees is extended by a processing abstraction layer. Based on GiST's generic data structure and algorithm abstraction layers that allow to model many index structures on a logical level, generic index processing primitives are defined that can be executed on arbitrary processor devices.

2. A sample implementation of this physical execution layer is developed that allows to schedule index operations to CPUs and GPUs.

3. A new out-of-core GPU-based algorithm is developed using the CUDA (formerly Compute Unified Device Architecture) programming model. Unlike prior art, the implementation does not require that the index data is fully stored in a GPU's main memory, but can be fetched on-demand.

4. Based on the generic index operation model, an adaptive scheduler is developed that allows to leverage the strengths of CPUs and GPUs in a hybrid system. The scheduler can be calibrated for arbitrary runtime system setups, without changes in the underlying framework implementation.

5. Finally, R-Tree range and nearest neighbor search algorithms are evaluated to assess parallelization benefits for both processor architectures and to compare different scheduling strategies with the original GiST baseline.

1.3 Focus of this Thesis

During the development of the indexing framework, the following assumptions have been made that limit the scope of this thesis:

- The thesis focuses on systems consisting of CPUs and GPUs that are attached as dedicated accelerator cards via PCIe (Peripheral Component Interconnect Express) bus, because such setups are widely used in practice and were also available for experimental evaluations. Integrating other processor types should be simple by choosing an OpenCL-based implementation of the algorithms, because many device types support this standard.
- It is assumed that index structures reside in main memory of the host system that interacts with coprocessor devices, i. e., disk I/O costs are ignored for query processing. In practice, this should not be a strong limitation, because upper tree layers, which are responsible for search space pruning, consume only fractions of an index' data volume and, therefore, can be usually buffered in-memory [75].
- The thesis focuses on query processing only. Tree modification operations and the implementation of concurrency control mechanisms are assumed to be provided by the underlying GiST framework. Therefore, the solution is applicable to rather static indexes that are used for analytic workloads and are refreshed in bulk.
- The thesis focuses rather on generality and extensibility of the resulting index framework than on tuning it for a specific use case. Index structures that that are implemented via the framework will, most likely, be outperformed by fine-tuned implementations. However, considering industrial hardware development cycles, it is rather unlikely that existing software will always be kept up-to-date to fully exploit all new features and capabilities (Wirth's Law [197]). Therefore, lowering development overheads with generic frameworks will likely be a better solution than adapting a large code base of fine-tuned structures in a long-term view.

1.4 Structure of this Thesis

An overview of current parallel processor architectures is given in Chapter 2. Architectural differences are discussed as well as parallel programming models, which are used in the following chapters to parallelize index operations. Chapter 3 covers prior art in this area. Common implementation techniques of parallelized indexes are discussed, as well as existing frameworks that allow to generalize index implementations. Further, the gap between these two research areas is highlighted in order to motivate the case for generalized parallel processing in index frameworks. In Chapter 4, the processing abstraction layer is developed. Further, existing applications from the computer graphics domain are profiled in order to assess potential benefits of parallelized index searches. Tree-based search operations are parallelized in Chapter 5. Moreover, a GPU-based implementation is presented that extends prior art by out-of-core processing capabilities, which is mandatory for scaling indexed data volumes that can be processed, and for leveraging multiple processor devices for the execution. Chapter 6 discusses how different device types can be integrated into the framework and how a scheduler can be implemented that dynamically selects the best processor type for a specific task execution. Chapter 7 summarizes this work and highlights next steps for future research directions.

Chapter 2

Background – Parallel Processors and Coprocessors

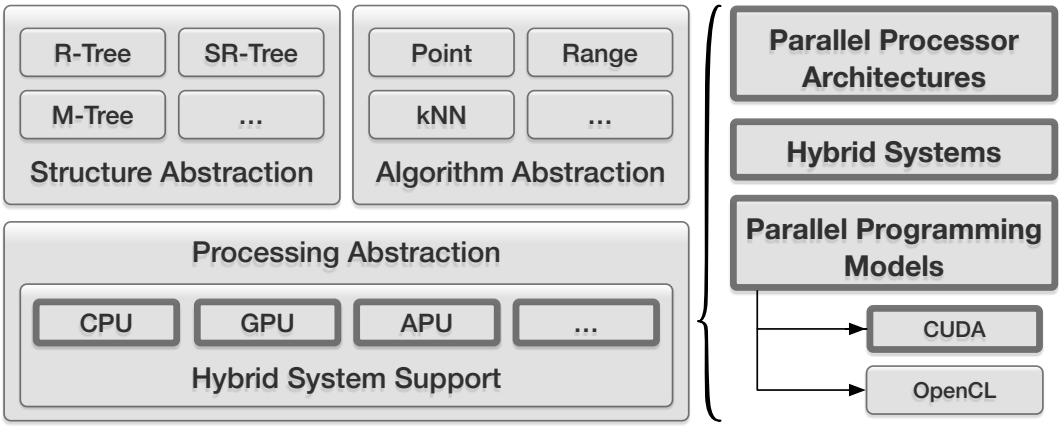


Figure 2.1: Overview of Parallel Processing Background Information

Before deep-diving into the parallelization internals of the indexing framework, this chapter provides some background information on parallel processor architectures (Section 2.1) and hybrid systems that are comprised of multiple different processor types (Section 2.2). This will provide a solid foundation for understanding the rationale behind some algorithmic design decisions that have been made while implementing the framework. Multi-core CPUs and GPUs are in the focus of these sections, because they were used for experimental evaluations.

The second part of this chapter (Section 2.3 and Section 2.4) discusses parallel programming models that provide useful abstractions for low-level hardware details when algorithms shall be implemented for these platforms. The focus in the following clearly lies on the CUDA and OpenCL (Open Computing Language) models, because they currently represent the de facto standards when algorithms for hybrid CPU-GPU systems are developed. CUDA has also been used for implementing the indexing algorithms that are subject of the following chapters.

2.1 Parallel Processor Architectures

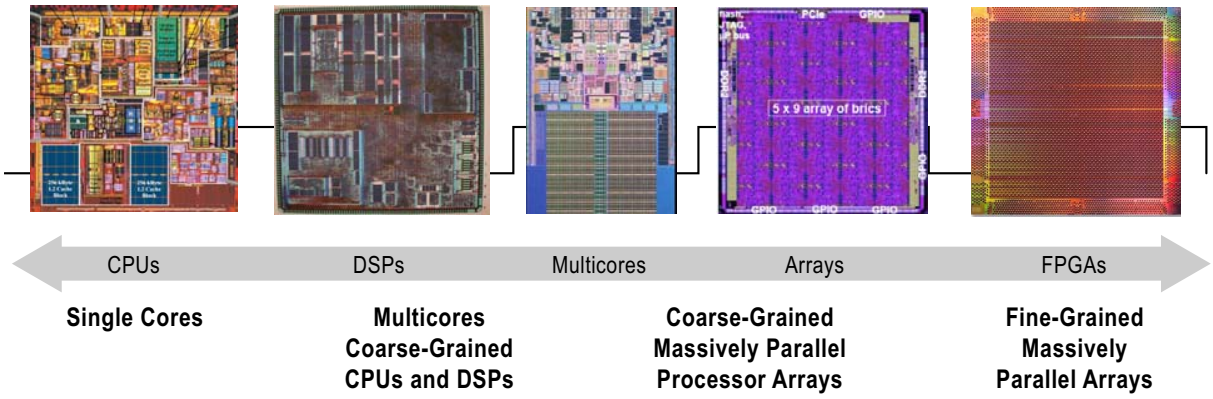


Figure 2.2: Overview of Programmable Parallel Processor Architectures
Source: [185]

It is observable that processors' clock frequencies are no longer increasing since several years. The reason for this is the often-cited *"power wall"* [35, 148, 188, 200], i. e., an exponentially growing energy consumption with increasing frequencies and the thereby arising thermal issues [119, 185, 192]. Instead, it can be observed that the number of cores are increasing in computer systems in order to gain performance by leveraging parallel computing. Further, sizes of caches within systems are increasing, as well as throughputs of interconnects between different kinds of memories and processing units [152] to keep cores busy by fetching data fast enough and avoid stalls through memory access. The latter is also known as the *"memory wall"* in literature [198].

There are various ways how the ever-increasing number of transistors can be wired in order to design processor architectures that fit special problem classes. A coarse-granular overview of some architectures is illustrated in Figure 2.2, which has been taken verbatim from [185]. Some of them are discussed in [129] and in more detail in the following sections, ranging from architecture of modern multi- and many-core CPUs over GPUs to FPGAs. Single-core CPUs are not considered here, because this thesis focuses on parallel hardware. Neither are DSPs (Digital Signal Processors), since these are special-purpose processors, designed for high-performance signal processing workloads that are not considered relevant for the indexing use case.

Multi-core CPUs, which are covered in Section 2.1.1, are commonly available as commodity hardware and, with a higher number of cores, also in the high-end server segment. They resemble the heart of most computer systems and are a good fit for general purpose tasks. A lot of effort was spent by vendors to, e. g., optimize CPU architectures for sequential tasks that involve divergent branches. Most of the currently available processor types also offer some capabilities for data-parallel SIMD (Single Instruction Multiple Data) processing.

The other extreme are processors that are optimized for SIMD tasks, such as GPUs, which are discussed in detail in Section 2.1.2. GPUs consists of large arrays of simple cores and, thus, can be classified as very large vector processors. They are best-suited for computationally intensive tasks that can be subdivided into a lot of fine-granular subtasks, preferably without any branch divergence among them.

Many-core processors, which are briefly broached in Section 2.1.3, can be classified somewhere in between CPUs and GPUs. They are sold as devices having many CPU cores with separate memories and offer a much higher degree of parallelism than CPUs, but are more flexible than simple GPU cores. These devices are comparable to parallel processor arrays and allow to implement a SoC (System on Chip) design where, unlike GPU devices, the entire program runs on the device without the need to synchronize with a host system.

Section 2.1.4 briefly discusses FPGAs, which are important coprocessors in many use cases. For example, they are used in commercial database management systems, such as IBM Netzza [63], to implement (de-)compression and filter algorithms near the data source before the data is streamed into main memory for further processing. FPGAs are freely programmable circuits that allow to embed the circuit corresponding to a specific problem directly on the chip – even logic implemented in CPUs, which also allows a SoC design on these devices. FPGAs are very energy efficient and can benefit from different kinds of parallelism. But they require workloads where many inputs are "streamed" through the device for producing outputs.

2.1.1 Multi-core CPUs

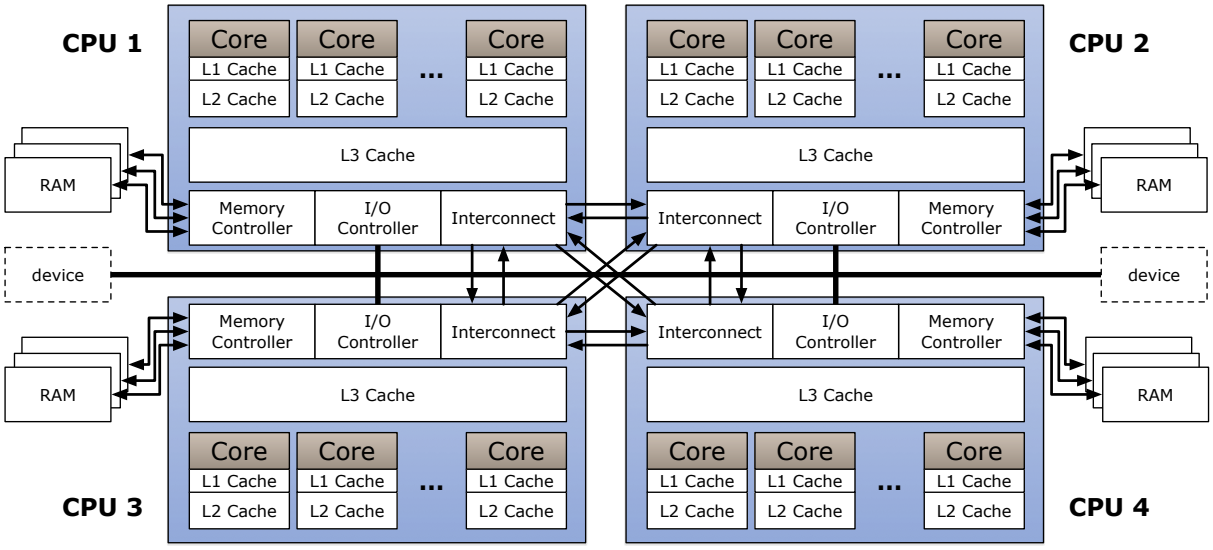


Figure 2.3: Architecture of a modern multi-core CPU

Multi-core CPUs represent the most common parallel processor architecture. A basic overview of their most important components is illustrated in Figure 2.3, which was compiled from many different sources and specifications of the largest vendors of CPU hardware [42, 49, 91, 95, 98, 175, 191, 196]. Of course, these specifications differ between different vendors, CPU families, and models, but the basic concepts remain the same. Those concepts shall be discussed here, including their consequences for algorithm design.

A modern multi-core CPU consists of several processor chips having multiple cores on a single die, ranging from 2 and 4 cores for commodity hardware, like the Intel Core i5 processor, over 12 cores, e. g., in IBM POWER8 processors, to 18 cores in Intel Xeon E7 processors for high-end server hardware. Each core can be managed independently from the others and operates at very high clock rates, usually between 2 and 5 GHz. High frequencies are automatically used for workload peaks to boost computation performance if certain conditions are met, e. g., temperature values comply with the hardware specifications [94]. Depending on the hardware architecture, each core can process multiple hardware threads simultaneously. While Intel refers to this feature as hyper-threading and in its current processor families supports up to two concurrent hardware threads per core [93, 175], IBM names this capability SMT (Simultaneous Multi-Threading) and supports four hardware threads per core in the POWER7 architecture and up to eight in the POWER8 architecture [91, 191].

CPU cores communicate via an internal bus system with all other components on the same die. Communication between different processors is done via an interconnect bus (QPI (Intel QuickPath Interconnect) by Intel [95] and HT (HyperTransport) by AMD, IBM, Apple, and Nvidia [49]). The communication with peripheral devices is implemented by an I/O controller via another bus system, most commonly PCIe [40, 72, 196] or NVLink, which provides even higher connection throughputs [163]. Coprocessors, like dGPUs (dedicated Graphics Processing Units) or FPGAs, are attached via this infrastructure, or, in case of iGPUs (integrated Graphics Processing Units)/ APUs (Accelerated/Advanced Processing Units), reside on the same chip. Both variants will be briefly discussed in Section 2.2. Further, there are usually several other specialized accelerators integrated on a CPU chip, which can be leveraged for dedicated tasks, like video de-/encoding or de/-encryption algorithms. But these accelerators are not considered here, because, unlike CPUs or GPUs, they cannot be programmed for general purpose tasks.

A significant amount of CPU chip space is spent on caches, which are managed by the hardware itself. Typically, L1 and L2 caches are privately accessible per CPU core while the L3, and sometimes a L4 cache, are shared between all cores residing on the same die. These caches form hierarchies, increasing in size and having higher access latencies with each additional layer. Main memory is attached via independent modules that are connected via a memory controller to the available processors. Access latencies and achievable transfer bandwidths from/to these memory modules are better when they are directly accessed from cores that reside on the same chip where the modules are connected to. Indirect access through the processor interconnect shows higher latencies and lower transfer bandwidths, which can have significant impact on application performance. This property is commonly referred to as NUMA (Non-Uniform Memory Access) effect in literature [127], but of less importance in the following.

In addition to caches, registers, and arithmetic-logical function units within the cores, a large number of transistors is used to optimize processing pipelines on each core. CPU cores are implemented as superscalar processors that can execute multiple instructions at once and comprise complex logic to overlap different operation phases (instruction loading, decoding, operand fetch, execution, result writing), predicting a program's control flow with branch prediction, and allowing out-of-order-execution of independent operations. Since these features are purely implemented in hardware, they mostly remain hidden from programmers. But for highly-tuned systems, e. g., database management systems, algorithms are carefully designed to exploit these optimizations in order to deliver maximum performance [34, 169, 212].

Another feature that all modern CPU families offer is the capability to execute SIMD operations, i. e., calculate the results of the same operation on multiple input data elements within a single step. Therefore, the input operands must be fetched into wide vector registers before an SIMD operation can be executed, yielding all results at once. The data must be properly aligned in memory for being able to fetch and store them with a single vector read / vector write memory transaction. These SIMD features are made accessible to the programmer via specialized processor instructions. However, each vendor offers different instruction sets for different architectures, which complicates the development of portable algorithms that exploit these features for maximum performance [54]. Some examples for SIMD instruction sets are:

- MMX, different SSE (Streaming SIMD Extensions) versions, or AVX (Advanced Vector Extensions) on x86-based processors
- AltiVec or Cell SPU on the POWER family
- VFP or NEON on ARM

2.1.2 GPU Architecture

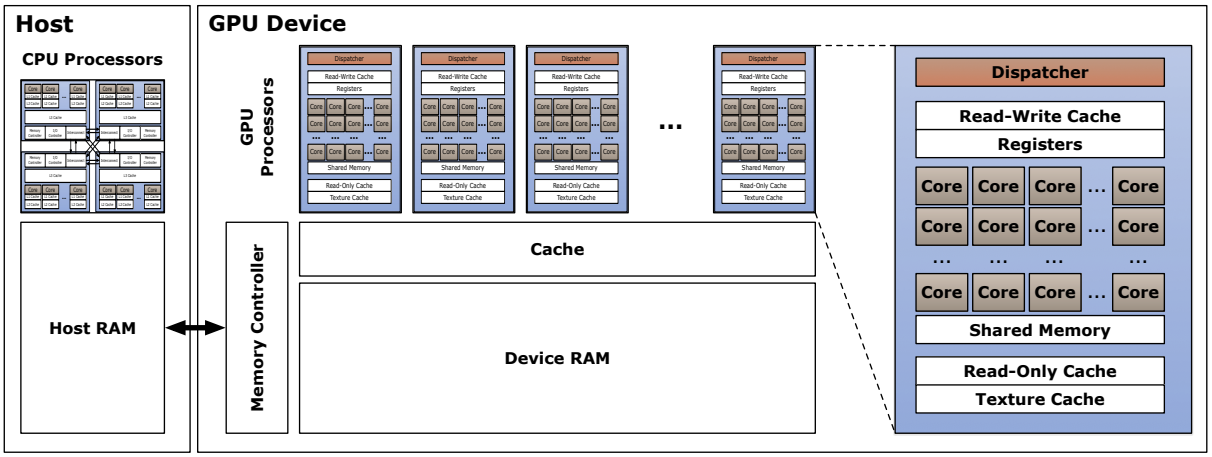


Figure 2.4: Architecture of a dedicated GPU device

Figure 2.4 illustrates a simplified architecture of a dedicated GPU device that is capable to execute general purpose tasks in parallel, i. e., general data-parallel programs which differ from the classic graphic processing pipeline. This model is often referred to as GPGPU (General-Purpose computation on Graphics Processing Units) in literature and by the industry. Current major vendors of such GPU devices are Nvidia and AMD/ATI. Their officially provided references and white papers have been used to compile Figure 2.4 [9, 10, 155, 157, 159, 161, 162]. Of course, some details, like the actual on-/off-chip cache organization, the hardware queuing capabilities, the memory access mechanisms, or the dispatcher behavior, are vendor- and device model/type-specific. But the given overview shall be sufficient to highlight architectural differences compared to a multi-core CPU described in the previous section and their implications on how the programming model has to be adapted for this hardware platform, which will be covered later in Section 2.4.

2.1.2.1 Device Architecture

A dedicated GPU *Device* is attached to the *Host* via an I/O bus system, like a PCIe or NVLink bus. The host uses a device driver to control all operations that are executed by the device, such as resource allocations, memory copy operations, *Kernel* executions, and synchronization barriers. Device operations are sent through the bus as well as all data that is exchanged between those systems. All data that is produced by the host system and acts as input for an offloaded GPU kernel has to be copied to the *Device Memory*. Output data that is generated by GPU kernels and requires further processing on the host has to be copied in the opposite direction.[1] Physically, an PCIe bus link comprises several lanes, which are composed of multiple electrically connected pins that allow full duplex, parallel point-to-point data transfers between the communication endpoints [40]. According to PCIe specifications, 2.5 Gb/s can be transferred via PCIe v1, 5.0 Gb/s via PCIe v2, and 8 Gb/s via PCIe v3 bus links per lane and direction [72]. However, the effective bandwidth for transferred data does also depend on the CPU and chipset which is used [9] and is much lower due to encoding overheads required by the physical interconnect, e. g., PCIe protocol overheads (packets with header information, sequence numbers, error correction codes, package acknowledgements etc.), and other overheads caused by additional communication layers in the application's stack [72]. As later experiments will show (cf. Section 5.3.5), these overheads significantly reduce the achievable transfer bandwidth between a host system and a GPU device, which might become an issue in case of transfer-bound applications. This bottleneck can be mitigated, e. g., by implementing caching techniques to store and reuse data on the device's main memory. In the case of GPUs that are integrated on the CPU die (cf. Section 2.2), CPU host and GPU device share the same main memory and data transfer operations can be omitted. However, memory access still has to be appropriately synchronized through the memory controller, which leads to additional overheads.

2.1.2.2 Processor Architecture

Similar to a CPU processor, a processor on a GPU comprises several cores on a die.[2] Compared to a CPU system, a GPU not only comprises a much higher number of cores per processor, but, usually, also a higher number of processors on the device. For example, Nvidia's Kepler architecture implements 192 CUDA cores per SM (Streaming Multiprocessor, sometimes SMX) [161] and 26 of such SMs are available on a Tesla K80 device [164]. In contrast to CPUs, GPUs are operated at much lower clock frequencies of less than 2 GHz. However, due to the large number of cores, more instructions can be executed simultaneously. Further, GPU processors and cores cannot be operated completely independent from each other. These aspects have to be considered when algorithms are implemented for this architecture.

[1] Note that the requirement for such memory copy operations is dictated by this system architecture. Terms like *"Unified Memory"* or *"Shared Virtual Memory"* that are oftentimes found in programming references or other vendor publications (cf. Section 2.4.3) do only refer to the virtual memory model that is presented to the programmer, i. e., how virtual memory addresses are mapped to the physical host/device address space. Such a model merely simplifies programatic memory management operations but, nevertheless, still requires physical data transfers to be executed implicitly by the device driver under the hood.

[2] Nvidia refers to a GPU processor as Streaming Multiprocessor, sometimes SMX, while a single core is referred to as *CUDA core* [157, 161]. AMD uses the term DPP (Data-Parallel Processor) for a multiprocessor, while a single core is referred to as *compute unit* [10].

A GPU multiprocessor acts as SIMD/SPMD (Single Program Multiple Data) vector-processor for large data-parallel tasks, which are consistently referred to as *GPU Kernel* or simply *Kernel* in literature (cf. Section 2.4.2) and are scheduled to the available cores on the device. While the GPU architecture allows communication and synchronization between the cores of the same multiprocessor, this is not the case for communication between different multiprocessors. This must be considered when *Kernel* algorithms are designed. Unlike CPU threads, which can be scheduled separately to each core, GPU hardware threads are dispatched in *thread groups*, which represent the smallest available dispatch unit on a GPU. Within such a group, also called *Warps* in Nvida terminology [160] and *Wave Fronts* by AMD [9], all threads operate in lockstep mode. That is, they share the instruction counter and, hence, execute the same instruction at the same time, but usually on different data partitions. Which data element shall be processed by a thread is indicated by a unique and possibly multi-dimensional *index*, i. e., a counter that is automatically calculated as offset for each kernel thread. These indexes are stored in special hardware registers and can be queried by each thread through dedicated program instructions.

Differing control flow branches are potentially serialized in this SIMT (Single Instruction Multiple Thread) model (cf. Figure 2.8). If branching conditions differ between threads within a single group, both branches are executed serially, leading to a performance penalty, because threads idle for the currently inactive branch. This effect negatively impacts the overall multiprocessor utilization. Further, the programmer does not have control over the scheduling on the device, i. e., there is no way of assigning a task to a dedicated GPU multiprocessor like it is possible for a CPU thread by assigning it an affinity for a single core, e. g., to avoid NUMA effects. Currently, scheduling on GPU devices is purely implemented in hardware and considers the available resources (number of registers, shared on-chip memory, etc.) when a kernel's subtasks are scheduled. In order to increase a device's utilization in case multiple kernels shall be executed on the same device, recent GPUs allow to execute kernels concurrently [66, 157, 161].

2.1.2.3 Memory Architecture

Current GPU devices offer main memories (often referred to as VRAM (Video RAM)), which are approx. an order of magnitude smaller in size (up to tens of GB) than the available RAM on host systems (tens to hundreds of GB). The VRAM is the only memory region on the GPU that is directly readable and writable by the host through the device driver. Therefore, it is used to store input data as well as result data that is copied back to the host. Further, GPU kernels can use it to store intermediate results, which do not fit into smaller caches. Similar to the cache hierarchy on CPU systems, most GPUs implement several cache layers for VRAM access, which are, usually, less important for GPU algorithms. Since the sweet spot of GPUs are massively data-parallel algorithms, data access locality can only be exploited to a very limited extend. Hence, the available transistors on a GPU are rather used for computing resources and less ones are spent for hardware-controlled Lx caches. As a result of this design, caches are only a few KB up to a few MB in size, depending on the cache level x. Main memory bandwidths are, usually, higher when VRAM is accessed by a GPU core compared to RAM access by a CPU core.[3] Thus, much higher main memory data transfer rates can be achieved on GPUs, which is another reason why Lx caches are of less importance on a GPU device.

[3] For high-end server CPU architectures, comparable memory bandwidths of a few hundred GB/s can be achieved if NUMA effects are properly considered [98, 162].

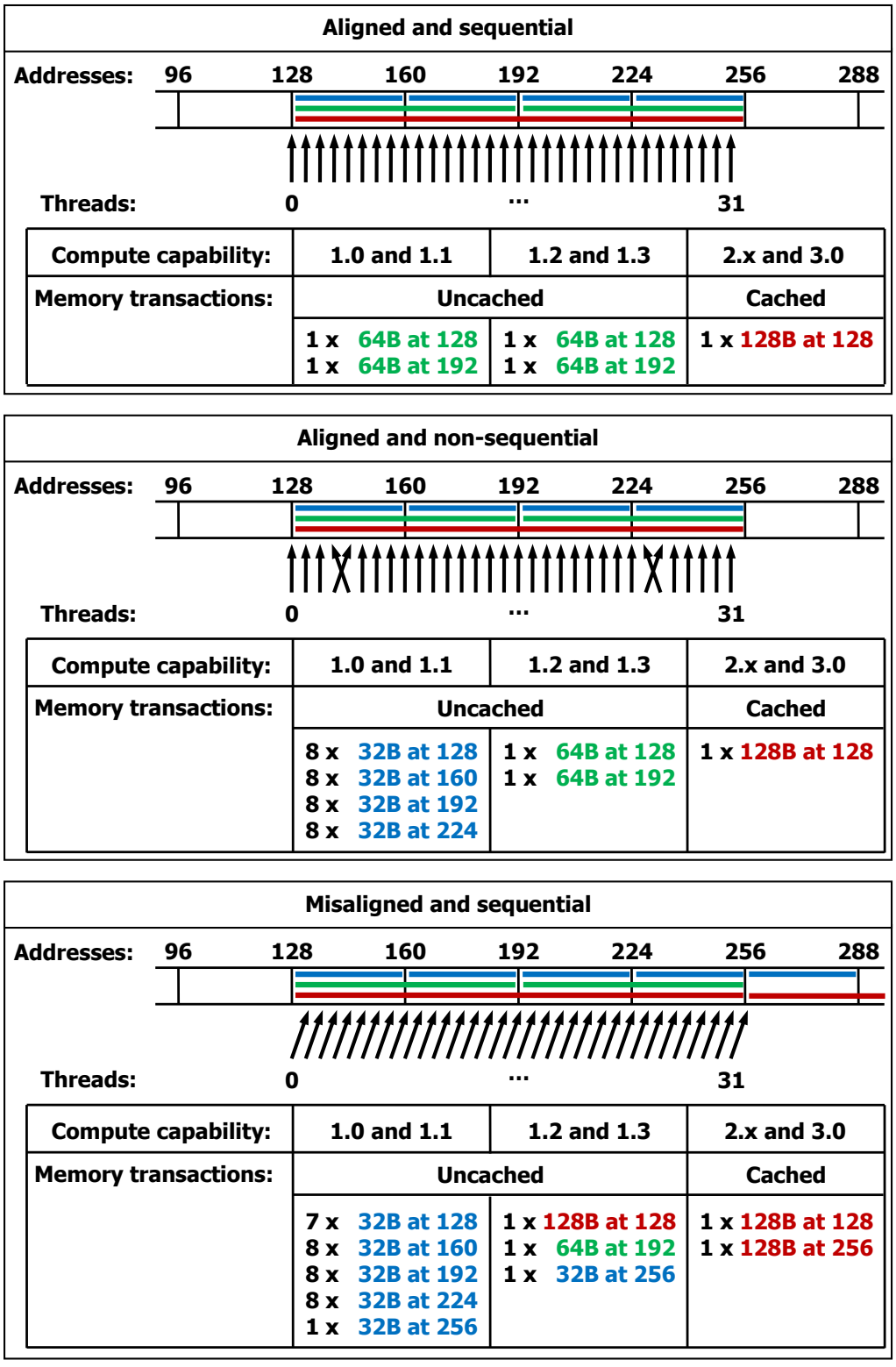


Figure 2.5: Memory Access Patterns on a CUDA-capable GPU
Source: [160]

However, these peak throughputs can only be reached if memory access is properly *coalesced*, i. e., all GPU cores of a scheduled thread group access bytes whose addresses fall into a range that can be transferred using the minimum number of memory transactions issued by the hardware. For an illustration of this property, refer to Figure 2.5, which was taken from the NVIDA CUDA C Programming Guide [160]. But the same effects also occur on AMD GPUs [8]. Memory transactions are comparable to cache line transfers on Lx CPU caches. The term *alignment* refers to the lowest starting address of the memory chunk that shall be read or written simultaneously by all active hardware threads in a group. Threads are active and execute the same memory access instruction (with different offsets) if they are not disabled due to serialized branches. In Figure 2.5, 32 threads belonging to a warp access four bytes each with a memory access instruction (illustrated as arrows numbered with thread index 0 to 31). The offset (the leftmost arrow) should match the starting address of the first required memory transaction (the colored horizontal bars – blue denoting 32-byte, green 64-byte, and red 128-byte transactions), which in all cases starts at address 128. Depending on the hardware version (compute capability), misaligned offsets lead to a different number of differently sized memory transactions that are denoted in the table under the bottom-most figure, including the respective starting addresses. Note that the last issued transaction having the highest address does hardly convey any useful, i. e., accessed, data elements which lowers the effectively achieved bandwidth. Further, for older devices having compute capability 1.0 or 1.1, a separate transaction needs to be issued for each of the threads. The latter does also happen in case the addresses are non-sequential, which is illustrated in the middle figure. Moreover, *striding* might become important, which is not illustrated in Figure 2.5. This term refers to the gap between the addresses accessed by consecutive threads within the same warp which, obviously, leads to more memory transactions being required to transfer the data chunks. In case not all bytes between the requested addresses are actually used, memory bandwidth is again wasted by unnecessary transfers. Therefore, coalescing as hardware-related property should be kept in mind when GPU algorithms are implemented. It must be considered at which offset memory accesses start and how these addresses relate to thread indexes within each thread group.

Once a group cannot continue processing because of such access latencies, the scheduler dispatches another group that is ready to continue and, therefore, hides the latency with overlapping computations – provided that enough hardware resources, like registers, are available to run both groups virtually simultaneously. Therefore, GPU algorithms should divide each kernel into sufficient fine-granular subtasks in order to benefit from this latency-hiding mechanism. Otherwise, transfer-related stalls become observable, leading to memory access-bound kernels.

If data locality can be utilized within a kernel's program logic, GPUs offer dedicated, but rather small (currently a few KB) software-controlled caches. This memory is located on-chip and, therefore, offers a between one and two orders of magnitude lower access latency, compared to VRAM.[4] It is shared between, and read-write accessible by all hardware threads that were scheduled to the same multiprocessor and, thus, can be used for communication between those

[4] On Nvidia GPUs, this shared memory is organized in banks with certain characteristics regarding memory access transactions within warps (cf. [160]). Depending on the addresses of the bytes that shall be read or written by the threads (which implicitly depend on the thread indexes), access might lead to so-called *bank conflicts* that will be serialized under certain circumstances and, thus, can lead to performance penalties. However, this behavior depends on the actual hardware that is deployed and will not be considered any further.

threads. However, since scheduling behavior cannot be controlled by the programmer and the overall progress within a kernel program might differ between the thread groups, access to this shared memory region has to be synchronized with dedicated primitives that introduce barriers in the GPU code. Shared memory can also be used as staging area in case memory access within a kernel would be non-coalesced. In this case, coalesced access is used to fetch data from VRAM into this on-chip cache using a different mapping of thread index to the accessed memory address than required later on by the kernel logic. This engineering trick can mitigate or even completely avoid memory bandwidth bottlenecks.

Depending on the GPU device, other caches might be available for specialized use cases in addition to shared memory. Examples are dedicated read-only caches for constants within the program, which are populated once upon kernel launch and are useful, e. g., for physical or mathematical calculations employing constants like Π. Further, texture caches with special access and clamping semantics are implemented for graphical tasks. However, these caches are not considered relevant for this thesis and will be ignored in the following.

2.1.3 Many-core CPUs

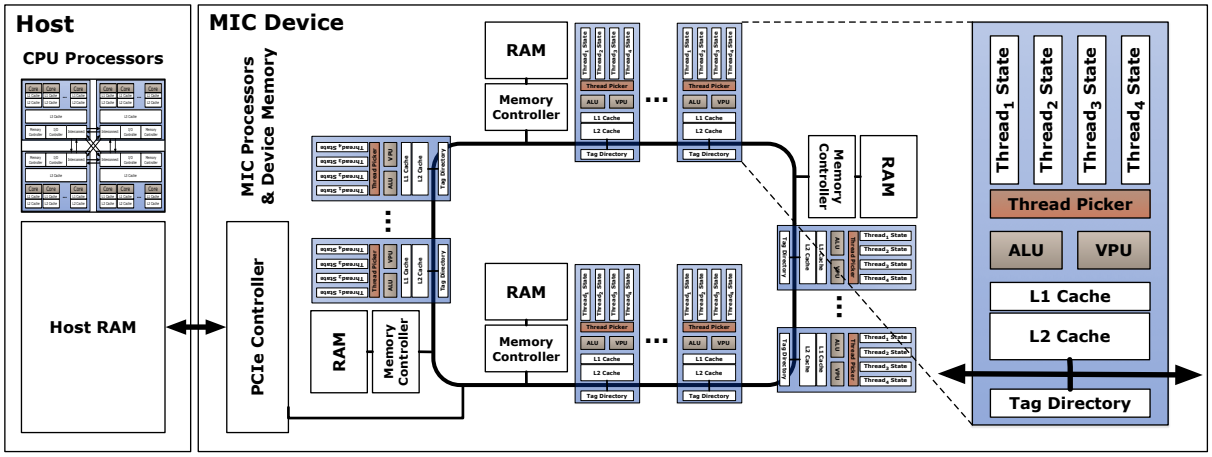


Figure 2.6: Simplified architecture of an Intel MIC device

The MIC (Intel Many Integrated Core Architecture) architecture uses many CPU-like cores on dedicated cards that are offered as so-called Xeon Phi devices. Up to 61 cores are integrated on current devices [96, 97] and future generation devices will comprise even more. Like dedicated GPUs, those (co)processor cards are attached via PCIe bus to a host system. As illustrated in Figure 2.6 (compiled from [46, 97]), the CPU cores on a device are connected via a bi-directional ring bus system that allows high-bandwidth communication among them as well fast data exchange through private on-chip caches, without queueing memory access requests. Those caches are kept coherent through shared TDs (Tag Directories). Each core offers support for 512-bit SIMD instructions by a VPU (Vector Processing Unit) but uses other instruction sets than regular CPUs [97]. Further, it can execute up to four hardware threads in parallel that operate independently from each other and are dispatched by a round robin scheduler (thread-picker), selecting the next thread having instructions ready for execution. Therefore, the threads' internal state, i. e., registers, instruction pointers, etc., are replicated on-chip. Operating with multiple hardware threads at the same time can hide memory access latencies.

Having RAM sizes ranging from 6 to 16 GB, peak memory bandwidths of up to 352 GB/s, and cores operating at a clock frequency of 1.2 MHz, Xeon Phi devices are on par with GPUs regarding these metrics [85, 96, 99]. But MIC cores are more flexible than cores on GPUs, because they do not operate in lockstep mode. Unlike GPU thread groups, hardware threads in the MIC architecture are able to operate independently from each other. So they do not need to be disabled on diverging branch conditions. Thus, compared to GPUs which focus on heavy data parallel workloads, the sweet spot of MIC devices is task parallelism [85]. Further, MIC cores support the x86 instruction set that is often cited as another advantage over GPUs regarding tool support and ease of use in terms of effort required to generate device code. In theory, a simple recompile would be sufficient for the latter. However, as Fang et al. have shown, more tuning effort is required in order to achieve peak performance on the device [55]. For example, in order to utilize all cores, a sufficiently high number of threads and properly vectorized code is required. In order to saturate the available memory bandwidths, streaming stores have to be used and memory access patterns need to be tuned, which is comparable to the tuning steps required for GPUs. Some of these tuning techniques break compatibility with generic x86-based CPUs. So maintaining two code bases for accelerated programs is still necessary. Unlike GPUs, MIC devices offer more flexibility regarding their usage patterns. They might be used in the same way as dedicated coprocessor for parallel tasks, but also allow to the run program entirely on the device without intervention of a host system. They can even be operated in a mixture of both scenarios, depending on the actual workload [97, 99].

Compared to a fully-fledged CPU, MIC devices offer a much higher degree of parallelism but use much simpler cores. Expensive logic, like branch prediction or out-of-order execution, is excluded but, somehow, mitigated by increasing the number of simultaneously active hardware threads. Further, the MIC architecture does not employ shared L3 caches between the cores but achieves much higher memory bus bandwidths compared to commodity CPU systems. Xeon Phi cores operate at much lower clock frequencies than regular multi-core CPUs and they are often claimed to be more energy efficient than competitor GPUs [46].

2.1.4 Field-Programmable Gate Arrays

FPGAs are arrays of configurable circuits that can be programmed in order to embed the logic of user-specific circuits that solve a data processing task by mapping a set of input signals to a set of output signals. Therefore, an FPGA comprises basic building blocks, such as ALMs (Adaptive Logical Modules), DSPs, memory blocks, ALUs (Arithmetic Logical Units) for single or double precision floating point calculations, etc., that can be connected in order to express the actual program logic [14, 186]. Unlike CPUs, whose circuits implement the logic of a fixed instruction set and are able to fetch, decode, and execute arbitrary program code from memory, once an FPGA is programmed, its circuits are “hard-wired” for one task and need to be re-configured once another program logic is required. It is possible to embed CPU logic inside an FPGA [14].

Inputs might be read from and written to different interfaces, such as an PCIe or memory bus. But the therefor necessary controller logic that implements the actual data access, e. g., memory addressing or communication protocols, must be embedded too. Because such building blocks are commonly required for most use cases, FPGA vendors, like Xilinx or Altera, offer devices

with pre-configured components, such as DRAM and PCIe controllers or even CPUs, on FPGA boards. These blocks can be reused by FPGA programmers and integrated in their application-specific circuits, without the need to test and verify this logic again. I addition to these pre-configured "hard" intellectual property circuits, "soft" intellectual property libraries exist for such components that, like libraries comprising binary code for CPUs, can be re-used when an FPGA's freely programmable part is configured.

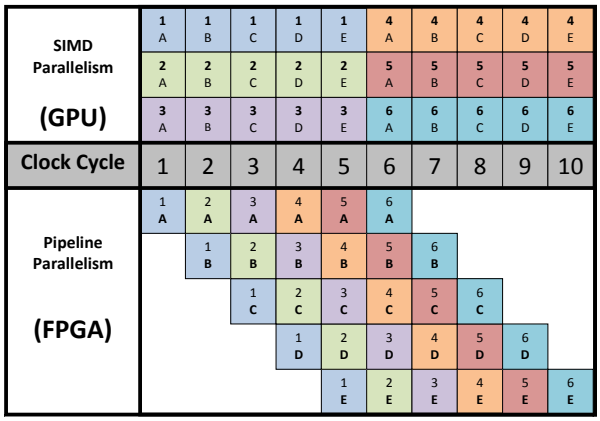


Figure 2.7: Comparison of SIMD and Pipeline Parallelism
Source: [14]

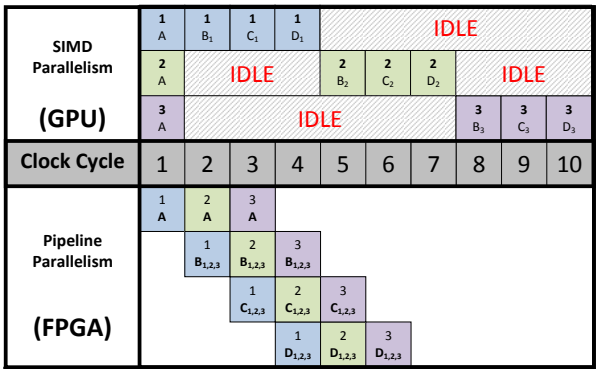


Figure 2.8: Branching in SIMD and Pipeline Parallelism
Source: [14]

FPGAs are well-suited for tasks where streams of many inputs are mapped to respective outputs. Multiple input elements can be processed in parallel by employing *pipeline parallelism* that modern CPUs try to infer dynamically from the incoming stream of instructions for a single program thread by additional control logic. Because instruction steps on FPGAs are "rolled out" in hardware, they do not need to be predicted and such control logic can be omitted. However, signals must be properly synchronized as the inputs "flow" through the circuit. Pipeline parallelism is illustrated in Figure 2.7. Compared to SIMD parallelism on GPUs, where multiple simple cores (three in Figure 2.7) operate the same kernel logic (comprising a sequence of steps $A \rightarrow E$) on multiple data elements (labeled 1 – 6, three at a time) in lockstep mode, all elements would be processed consecutively in an FPGA pipeline. All steps in such a pipeline can be processed truly in parallel once the pipeline is "filled", while lockstep SIMD processing executes them sequentially because corresponding program instructions have to be fetched before each step. Another effect of rolling-out the entire program logic in hardware is illustrated in Figure 2.8. A kernel executing a sequence of steps $A \rightarrow D$, having a common step A and a 3-way branch logic for sub-sequence $(B \rightarrow C \rightarrow D)$, would require 10 cycles on a GPU when all threads within a thread group diverge in their branching conditions, because they need to be serialized. An FPGA circuit can process all branches for each element at once, since they are pre-configured and merely the outputs of matching stages need to be forwarded as input for the next one. Thus, FPGAs can potentially perform better than GPUs for diverging kernels. A second way to achieve parallelism on FPGAs is to replicate the pipeline circuit on the chip, allowing to operate in an SIMD fashion on multiple input element streams, provided that enough "space" in terms of configurable building blocks is available on the die [186].

The logic that should be deployed to FPGAs is, usually, implemented in hardware description languages, such as Verilog or VHDL. Corresponding circuits are generated by synthesis tools and special layout algorithms for the actual FPGA hardware. However, calculating the hardware layout is a computationally intensive process and consumes considerable more time than compiling the program logic for a fixed hardware platform, like a CPU. Recent approaches use CUDA [186] or OpenCL [50, 186] (cf. Section 2.4) for programming FPGAs on a higher level of abstraction than provided by hardware description languages. These approaches do not eliminate the expensive circuit generation step, but relieve developers from the burden of designing parallel pipelines. Using OpenCL and CUDA leverages the programatically specified parallelism through kernels, which are comprised of independent work items that are processed by the same set of instructions. These kernels can be mapped to replicated circuit pipelines that implement this logic. Theoretically, this approach allows to reuse the same, already parallelized code for multicore CPUs or GPUs on FPGAs, letting the OpenCL or CUDA compiler automate the hardware circuit design.

FPGAs are often claimed to be more energy efficient than their CPU or GPU counterparts [14, 154, 186]. One reason is that CPU and GPU cores comprise many functional units that might be unused, depending on the logic implemented by kernels [14]. If the hardware is configured instead, unused portions can be minimized and, thus, the overall efficiency of the entire device is maximized. Another reason why FPGAs consume less energy are the far lower clock rates on these boards, which are in the order of a few 100 MHz only. This requires special care when algorithms are designed in order to avoid suffering from large latencies and low throughputs [154].

2.2 Hybrid Systems

While the previous section discussed specific processor architectures and their traits in an isolated manner, this section focuses on the combination of multiple processor types in a single system, because parallel processors are oftentimes used as *coprocessor* in conjunction with a CPU. While the serial and hard-to-parallelize parts of the program are regularly executed on a CPU, the computationally intensive parts that can be parallelized are offloaded to the coprocessor. Combining different processor architectures for such different workload classes offers the opportunity to utilize their strengths and gain some additional performance benefits for the overall task. Coprocessors are either integrated on the same die, interfacing the same memory and I/O channels as the CPU (e. g., in so-called iGPUs or APUs, which is also possible for integrated FPGAs), or are available as dedicated device cards that are attached over a bus system, like PCIe (which is the common case for dGPUs, Intel Xeon Phi devices, or FPGA cards).

This section will provide some background information about properties of a hybrid system of a CPU and a dGPU device, which will be required in Chapter 6. This commonly used setup is used in this thesis in order to analyze and evaluate the strengths and weaknesses of these devices for the given indexing use case and, based upon these characteristics, derive an architecture that combines the best of both worlds.

2.2.1 Processor Characteristics

The first task that needs to be accomplished when algorithms are designed for a hybrid system is that their logic is analyzed in order to identify parts that fit the nature of the processor architecture on the chip. For example, many transistors are used in CPUs for implementing complex logic that is useful for general purpose tasks, e. g., branch prediction tries to eliminate pipeline stalls and on-chip caches reduce data access latencies for algorithms that work locally on a small data set. Such logic is minimized in GPUs. Instead, most chip space is used for computational units in order to facilitate data-parallel work, and registers in order to lower the costs for thread management. Further, memory access bandwidths are, usually, much higher on GPUs, because this is crucial for keeping the many compute units busy that operate on many different data elements at once. Caches on GPUs are, usually, much smaller compared to CPU caches and are mostly software-controlled, which requires careful algorithm design in case memory access is the bottleneck of the task. In contrast to CPUs, where Lx caches are only implicitly considered by the hardware, they need to be manually configured with dedicated instructions in the GPU case. More details on that will follow in Section 2.4, when programming models are discussed.

As rule of thumb, CPUs are better-suited for complex algorithms involving a lot of branching, etc. on a fairly small amount of data, while GPUs are best-suited for massively data-parallel computations that can be executed on many independent data elements at once without much branch divergence. For the given indexing use case, this means that a major redesign of the available state-of-the-art tree traversal algorithms is required. While the existing algorithms focused on optimizing I/O requests, assuming low costs for predicate evaluations, they have been designed for the CPU model and involve a lot of branches for processing a single query. Under the assumptions of this thesis, i. e., most of the relevant data can be held in main memory and significant portions of the search time is spent on traversing the index tree, this model has to be adapted. In order to identify units of work that fit a (data-)parallel execution model, particularly the one offered by GPUs, it is required to increase the unit of work sizes by introducing a batch-processing interface. Further, as Chapter 5 will show in more detail, different granularities of independent steps are leveraged in order to fully saturate the available processing units. Of course, some tuning is needed in order to optimize memory access patterns and exploit available caches. For the indexing use case, these caches can be used for buffering tree and query data to increase data access locality and to reduce the pressure on the memory bus system.

2.2.2 Memory Bus Systems

The second major property that needs to be considered in a hybrid system is the way how a coprocessor is integrated and which role each processor type plays during the algorithmic control flow. In most cases, the CPU represents the *owner* of the data with direct access to it through the main memory bus. Further, the CPU manages task scheduling to attached coprocessors and synchronizes their execution. Therefore, the CPU-side is mostly denoted as *host*, while coprocessors are usually referred to as *devices*[5]. In most cases, coprocessors, like a GPU or

[5] Mittel and Vetter also provide a overview of common terminology that are widely used in heterogenous computing literature [153].

an FPGA, are attached as *dedicated* devices that are connected over a special bus system, like PCIe or NVLink [163]. But some CPU chips also *integrate* a GPU on the same die, i. e., some transistors are wired as GPU and share the same memory bus with the CPU through the same on-chip interconnect. AMD refers to integrated GPUs as APUs. The fact whether a dGPU or iGPU is used impacts data management strategies significantly and, thus, is crucial for the overall application performance. While costly explicit data movement operations are required for dGPUs in order to copy input data from the device to the host and output data vice versa, this overhead becomes smaller for iGPUs. However, in both cases, it is required to synchronize data accesses in order to guarantee correct results. Therefore, such characteristics must be considered when deciding upon a proper scheduling strategy.

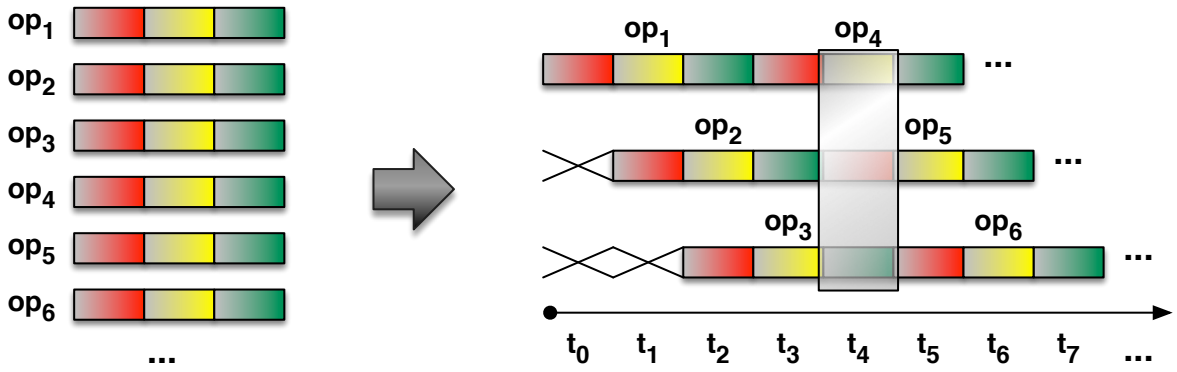


Figure 2.9: Pipelining of multiple Operations on GPUs

In order to avoid, or at least reduce, the impact of data transfers for transfer-bound applications, techniques like pipelining [189] can be used to chunk data that needs to be processed and overlap computational phases with data transfers of subsequent inputs. This technique is illustrated in Figure 2.9. In this example, it is assumed that a large task can be divided into more fine-granular subtasks that can be individually scheduled to a dGPU. Each task (op_i) comprises

- an input transfer operation for copying input data from the host to the device (red bars in Figure 2.9)
- the actual (parallel) computation on the device (yellow bars in Figure 2.9)
- an output transfer operation for copying result data back to the host (green bars in Figure 2.9)

If no pipelining would be implemented, all phases must be executed serially, i. e., all input elements must be copied before the actual computation can be run and, finally, all results can be copied back to the host. With pipelining, these stages can be overlapped as shown on the right in Figure 2.9, leading ideally to a three-fold throughput improvement if all of these phases are equally long. Otherwise, the longest phase dominates the entire pipeline, leading to an overall throughout, which does not fully saturate the interconnect bus bandwidth. In order to pipeline a task it is required that:

- the bus to/from the GPU can be operated in full-duplex mode between the endpoints, i. e., input and output transfers can be issued at the same time (which is the case for PCIe);
- the GPU has multiple memory copy units that can independently schedule input and output copy operations (which depends on the GPU hardware model);
- the GPU can schedule multiple kernels independently and overlap their computations (which also depends on GPU features, i. e., the compute capability in Nvidia terminology);
- the GPU does not inject internal synchronization operations (which might happen in some GPU models). These operations may, for example, prevent that a kernel starts before previously issued copy transactions finish, even if they are independent from each other, like output or input copies for subsequent kernel invocations.

Some libraries implement pipelining techniques under the hood, but, usually, it is up to the application developer to design data processing algorithms in a way that they can be pipelined. That is, these phases as well as required scheduling and synchronization operations need to be managed manually.

For the indexing use case, a CPU/ dGPU hybrid system is analyzed, where the GPU is attached via PCIe bus to the host system. The impact of intermediate data transfers needs to be considered when the algorithms are evaluated. More details on that will follow in Chapter 6. The main challenge is to abstract this system architecture in order to generalize the indexing library for this aspect and allow future evaluations of other system setups, without major code adaptions. A tuning technique, like pipelining, is not integrated, yet, but the library has been designed with that in mind, i. e., different phases are modeled and internal components that are required for buffering data for overlapped, asynchronous processing are implemented, already.

2.2.3 Hardware Limits

In order to fully utilize the available processors on a parallel processing device, an application's task need to be divided into independent subtasks that can be scheduled. The amount of tasks that can be executed in parallel strongly depends on hardware limits, such as:

- the number of processors on the device
- the number of processors per core
- available memory buffer sizes
- the number of available registers for storing results of simultaneously executed operations
- the size of available caches if they are used by the code
- the number of copy units for concurrent data transfers via the device bus

Some of these factors, like the number of available processors, are obvious, while the impact of others, like cache sizes and copy units, is hidden and depends on the actual algorithm that shall be executed on the device. As rule of thumb, it is the best case for any processor to have many fine-granular steps without data dependencies in order to execute them in an SIMD fashion with all available cores. Depending on the hardware limits, these steps can be batched to larger chunks in order to amortize scheduling and task management overheads and minimize synchronization efforts between them. However, for a maximum utilization, it is required to know and model the impact of all these factors for the given workload and tune the code base for the system deployed at runtime. Hardware vendors support application developers by providing tools that calculate device utilizations for a program that was compiled for a specific platform and allow to identify bottlenecks. But this still remains a challenging task that even becomes harder when multiple devices shall be used in a single system and additional dynamic factors, like load balancing between devices, need to be considered.

For the implementation of the indexing framework, these factors have not been considered in full detail. The general goal of the framework is to abstract these details and hide them in the framework itself, trading-in simplicity and ease of use for full performance optimization. For the experiments in the following chapters, major properties, like number of cores, buffer sizes, and cache limits, have been manually configured in a calibration step for the available platform. Device drivers (cf. Section 2.4) allow to query these parameters at runtime, which, in principle, allows to automate the library calibration for the given system. But this engineering task is left for future work and is not in scope of this thesis. Neither is the integration of multiple devices into a single system. While the general framework architecture allows this, the involved complexity of load balancing, etc., prevented further investigation of this additional extension point.

2.2.4 Other Aspects

In addition to the aforementioned, mainly performance-related criteria that need to be considered in a hybrid system, there are others that may have an impact on the decision whether a coprocessor-based design shall be implemented or not. For example, energy efficiency plays an important role for larger deployments [195]. Usually, GPUs consume a lot of energy while CPUs might also be throttled in phases of less utilization in order to minimize the application's energy footprint. As discussed before, FPGAs might even be more efficient, because they mostly eliminate hardware building blocks that are not required for the program that has been configured.

Second, monetary costs might also play an important role when such hybrid systems shall be commercially used. Because specialized coprocessors, like GPU cards, are usually more expensive than CPU boards, this might become an important metric. Further, developing, testing, and maintaining hybrid algorithms is more complex than maintaining algorithms for a single processor type, because special skills are required, which renders the entire development process more costly. Development frameworks, like CUDA or OpenCL (cf. Section 2.4), facilitate this

process by providing an abstraction layer over the actual hardware and automating the deployment to different processor families via code generation. But they also require special skills that are not widely-spread among all developers.

For the indexing use case, such secondary non-performance related criteria have not been considered, although it might be an interesting discussion to model them with an approach like "the five-minute rule" for memory and storage-related metrics [74, 77, 78].

2.3 Parallel Programming Models

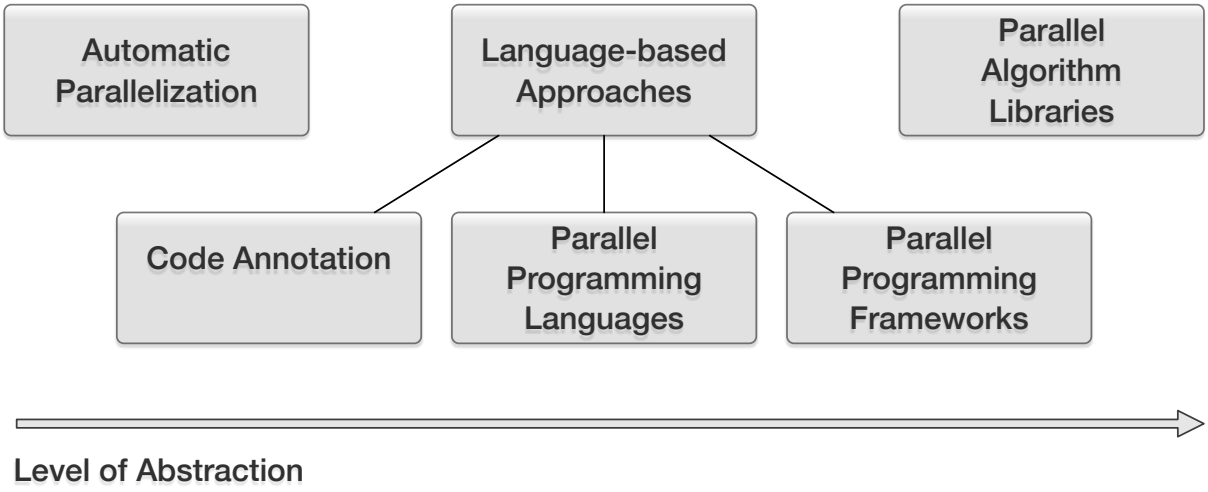


Figure 2.10: Overview Parallel Programming Models

The plethora of available parallel processor architectures described in the previous sections caused the urgent need to handle this abundance of processing resources efficiently during application development. A conceptual shift is required in the way how programs are designed and developed, which is sometimes considered as the "next revolution" in software development [192]. Various techniques exist to assist application programmers in this situation [106, 148, 192], which can be roughly classified as illustrated in Figure 2.10. These techniques are briefly outlined in the following, with an increasing level of abstraction where parallelization models are applied and which interfaces are offered to application developers.[6]

[6] Note that applying these classification criteria on existing real-world tools and products in that area will lead to rather fuzzy results, depending on the point of view. Usually, there is never a standalone specification for a programming model without corresponding implementations that comprise toolchains, including compilers and linkers that (automatically) apply optimizations internally. Moreover, libraries are mostly included in vendor distribution packages for software development kits and ship together with development tools, offering additional higher-level constructs to assist application programmers. Further, other orthogonal classification criteria could be considered, like distinguishing approaches according to the underlying computing model, e. g., shared memory vs. distributed memory [106]. Therefore, the overview, which is provided in the following, is considered far from being complete. But it should provide a good entry point to get started in this huge area that has been widely discussed in literature [19, 76, 148, 149] and will gain significant impact as hardware development evolves.

2.3.1 Automatic Parallelization

The first class covers techniques that *automatically parallelize* code on compiler-level. The core idea here is that compilers can identify portions of the code that could potentially benefit from parallel execution on multiple processors. While, in theory, relieving developers from the burden of parallelizing code themselves, these algorithms can only be applied on very low-level constructs, such as loops, and are, usually, unsuitable for application development in practice. The main issue with auto-parallelization is the lack of knowledge about the application context that could be exploited in order to parallelize the code even further. While it is hard for compilers to derive data dependencies, application developers can specify which operations do not depend on each other and, therefore, may be executed in parallel.

Given the fact that different processor types shall be supported by the index framework and parallelizing tree-traversal algorithms requires explicit definitions of independent operations on a higher level, automatic parallelization is not applicable to this use case.

2.3.2 Parallel Programming Languages

Language-based parallelization approaches offer APIs (Application Programming Interfaces) for letting developers implement expert knowledge in programming languages of their choice. Such APIs specify a control flow and a memory model as well as means for actually implementing independent operations and synchronization operations.

Methods of the first subcategory in this class extend existing programming languages with features for *code annotation*. Similar to auto-parallelization approaches, developers can annotate sequential low-level constructs directly within the code by using meta-primitives that identify parallel regions without data dependencies. The source compiler, in turn, can use this information for translating an annotated source file into a parallel version, injecting necessary data distribution and synchronization routines.

The second subclass covers techniques that model *parallelism directly in programming languages*. Most modern higher languages, such as C++ or Java, specify memory and threading models as well as objects to control state and execution of independent threads. In addition to evolving standards for modeling parallelism in existing languages, plenty new languages have been developed so far [148]. While mostly keeping syntax similar to existing languages (for lowering the required learning curve), these languages require specific toolchains, including compilers, linkers, and runtime drivers. Some of those languages, like CUDA or OpenCL, define models for parallelism that reach beyond the scope of defining parallelism for a single specific processor architecture. Another level of abstraction is introduced that allows mapping of parallel code to multiple, even heterogeneous architectures with different devices.

The last subclass in the language-based category covers *frameworks and libraries* that facilitate the development of parallel algorithms, such as Spark, Hadoop, or MPI (Message Passing Interface). Compared to the previous subclass, parallelism is expressed on higher levels. Developers do not explicitly parallelize operations, but use primitives that can be automatically mapped to parallel runtime versions, which are implemented inside library/framework code. In most cases, developers operate on data structure-level and can fine-tune their algorithms, e. g., by specifying scheduling policies.

For the indexing use case, parallel programming languages are relevant, because they allow to model parallelism on a suitable abstraction layer. While simple code annotation only covers the parallelization of low-level routines, parallelization frameworks rather targets cluster-based systems that consist of multiple compute nodes. CUDA and OpenCL are used in the following, because they are widely spread in the developer communities and allow to maintain a uniform code base for various processing devices. Thus, these models perfectly fit the requirements of an extensible indexing framework and will be covered in more detail in Section 2.4.

2.3.3 Parallel Programming Libraries

The final parallel programming model, which shall be discussed here, covers **parallel algorithm libraries**. Those libraries implement parallel versions of algorithms that solve frequently occurring problem classes on algorithm-level. From an application programmer's point of view, this is the highest possible level of abstraction, because developers do merely program against interfaces provided by the library, while the library code parallelizes those algorithms under the hood. Often, such parallel libraries are developed by hardware vendors, are optimized for their platforms, and are delivered as part of the corresponding software development toolkits. This approach is frequently used in the field of scientific data analytics and completely abstracts from parallel execution models for well-known algorithms. While offering less flexibility for expressing and tuning parallelism in the whole application context, they provide simple means to leverage the available processing power by mapping high-level algorithms to simpler, lower-level primitives and, therefore, aim to increase developers' productivity.

For the indexing use case, developing the framework itself as a parallelized library is the goal of this thesis. Such a reusable library simplifies application development and covers all parallelization efforts internally. Ideally, there shall not be any need to modify library code beyond its pre-defined extension points, unless there is a specific need for that, e. g., because new hardware platforms need to be integrated that require a different calibration of the existing cost models or need special resource management.

2.4 The CUDA/OpenCL Programming Model

Because this thesis focuses on evaluating index algorithms on hybrid CPU and GPU systems, this section gives an overview on the two most prominent programming frameworks for this platform. CUDA, developed by Nvidia, and OpenCL, as standardized counterpart originating from Apple in cooperation with many hard- and software manufacturers, such as AMD, IBM, Intel, Nvidia, and others [107], gained reasonable traction in the developer community and evolved as the de facto standards for such a setup. Both frameworks share very similar concepts, originating from the architectural properties of modern GPUs (cf. Section 2.1.2), that have been generalized and extended further in later releases or by other projects to match other platforms that have been discussed in Section 2.1 as well.

The main idea behind both frameworks is that actual hardware details for specific devices are abstracted and only exposed to programmers via device features. The latter can be leveraged for tuning algorithms for the actual runtime environment by, to some extend, sacrificing platform independence and maintainability. Parallelism is, due to the nature of GPU architectures, mainly expressed by specifying data parallelism within the code, which is automatically mapped to cores in computational back-ends via device drivers. As hardware features evolved, e. g., by integrating multiple copy units for overlapping data transfer operations or the ability to dynamically create and schedule tasks directly on devices, later releases of these frameworks also introduced additional means to express task and pipeline parallelism within the code.

Although having many similarities, slightly different terminology is used in these frameworks, which is explained in this section in conjunction with the underlying programming model that is required for working with (co)processors. The information presented in the following sections has been compiled from many resources. Official guides and references [5, 6, 9, 67, 160] give detailed information and specifications over the programming model, while other books and programming guides [8, 66, 155, 158, 189] focus more on practical aspects for the daily work within this ecosystem and also provide additional hints as well as programming techniques to avoid performance bottlenecks.

2.4.1 Platform Model

The *platform model* describes how a heterogeneous system, comprising regular processors and parallel coprocessors, is structured and how they interact with each other. An overview of the basic concepts is depicted in Figure 2.11. The system part that represents the main processor is referred to has *Host*, while parallel coprocessors, like multi-core CPUs, GPUs, or APUs, are referred to as *Devices*. In most cases, i. e., apart from SoC designs on MIC devices, for example, host code for executing serial program parts and orchestrating the interaction with devices is executed on CPUs, while devices are used for offloading parallel program parts in SIMD or SPMD fashion. Resources for one device are managed by a host thread in a so-called *Device Context*. A single host thread might operate multiple devices and devices might be shared between different contexts.

For being able to operate on multiple, architecturally different devices, a common abstraction model is required. Parallel device programs are executed on several independent *Multiprocessors* that are comprised of many *Thread Processors* (TP) each[7], which are able to cooperate on a specific subtask. Details for the execution model are covered in Section 2.4.2. *Devices* have dedicated memory and need to synchronize input and output data with the memory in host scope. Details for the memory model are covered in Section 2.4.3.

[7] The terminology between CUDA and OpenCL differs here. An overview of the terms used in each model, including a brief explanation, is provided in Appendix A.

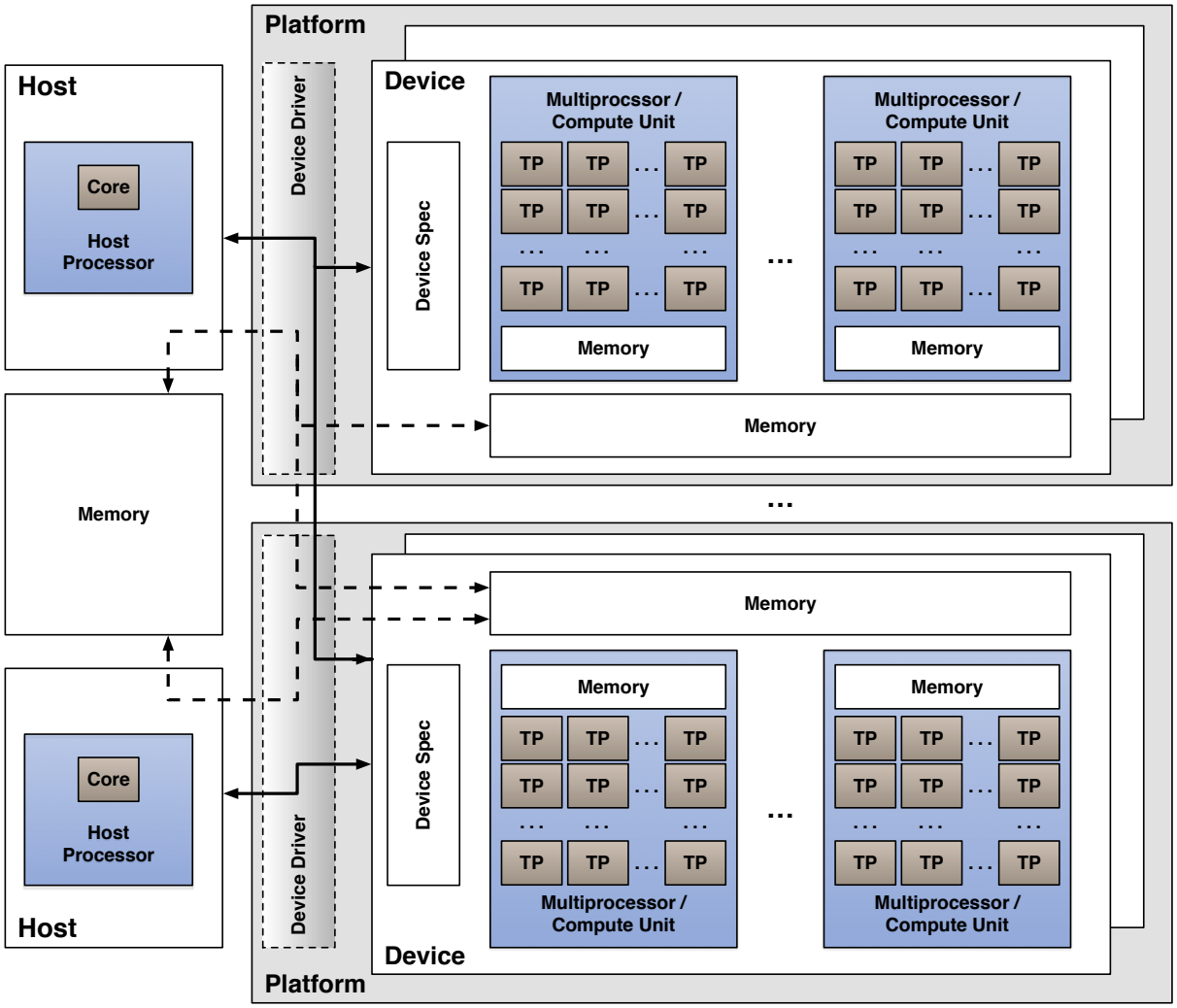


Figure 2.11: Platform Model in OpenCL & CUDA

For interacting with a device from host side, *Device Drivers* are needed, which implement the OpenCL standard[8] and offer a uniform API that abstracts from the hardware implementation (cf. Section 2.4.4). Each driver instance is represented by a *Heterogeneous Platform* object, which offers access to possibly multiple devices. Newer specifications offer means to partition single devices into sub-devices via the device fission extension if that is supported by underlying hardware. This feature is particularly useful for sharing devices between multiple host *Device Contexts* while limiting the number of used cores. Further, some meta information is associated with each device, such as device type, the hardware vendor, supported driver versions, hardware capabilities, supported extensions, and resource limits. These device properties can be queried by the host system at runtime, which enables fine-tuning of algorithms by leveraging available platform capabilities and allows to parameterize the programs with the dynamic information about available resources.

[8] CUDA, of course, only supports drivers for CUDA-enabled devices but the underlying concept is similar.

2.4.2 Execution Model

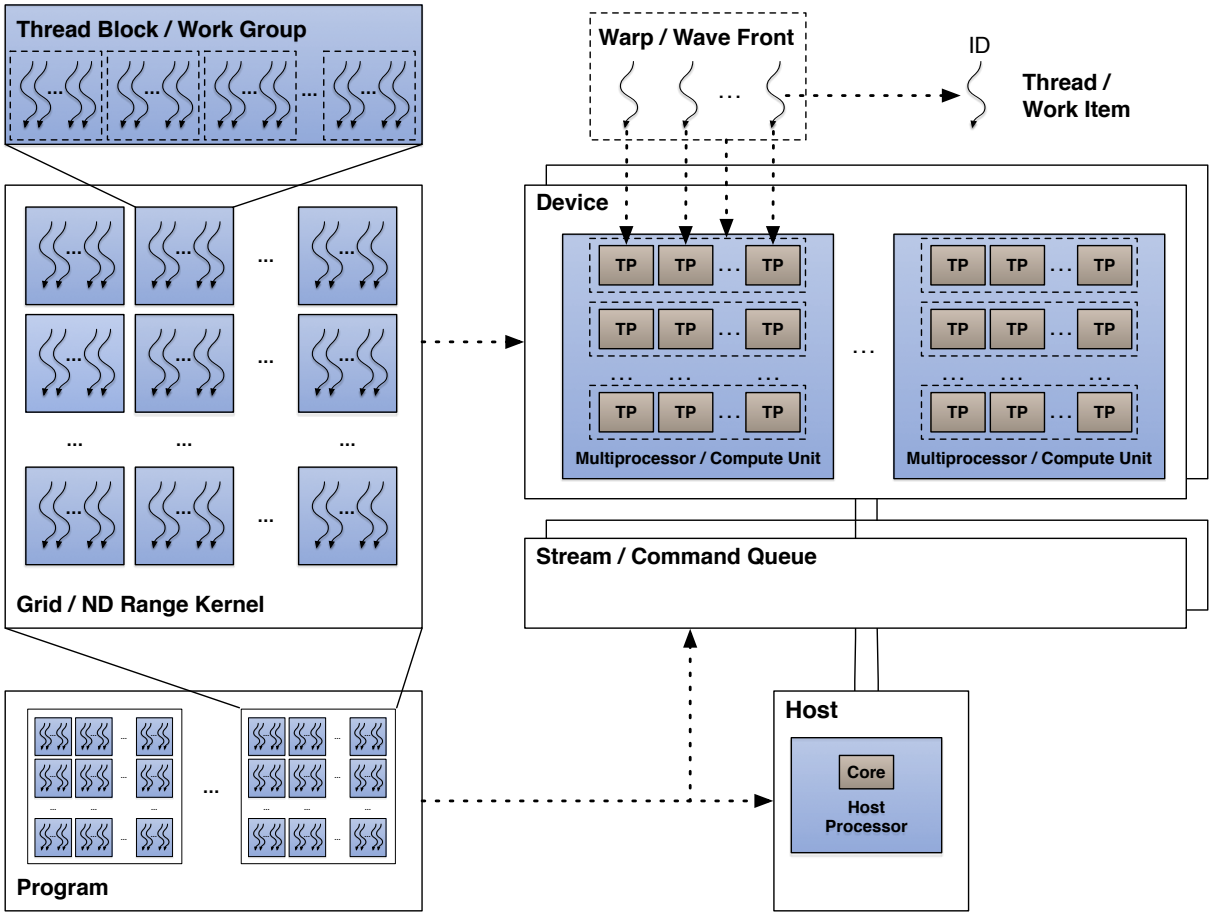


Figure 2.12: Execution Model in OpenCL & CUDA

A simplified overview of the execution model in CUDA and OpenCL is depicted in Figure 2.12. The developer provides sources for parallel *Device Programs* that consists of offloadable parallel parts and sequential operations in between. These program parts are mapped to respective execution hardware as described in Section 2.4.1. *Host* parts are responsible for serial steps and implementing appropriate synchronization between parallel parts that are executed on available *Devices*. The scheduling is done via *Device Command Queues* (one or many per *Device*) in the program *Device Context*, which also allows *Event*-based notifications, e. g., for tracking the execution progress, profiling, etc. All processing and corresponding I/O operations in a queue can either be scheduled synchronously or asynchronously and synchronization points can be injected by using an event API.

Parallel subtasks that represent calculations on *Devices* are called *Kernels*. Each kernel defines a separate index space for addressing single units of work in a data-parallel execution model. This index space is organized hierarchically and allows multi-dimensional addressing of work units via unique IDs. Each level in the hierarchy represents a different task granularity and

defines a model of task (in)dependence. The finest granularity is referred to as *Work Item* in OpenCL or *Thread* in CUDA. It represents a single *Kernel Instance* of the program, having a *Thread ID* assigned to identify the piece of work that shall be processed. Work items are mapped to *Thread Processors* (TPs), which are basically hardware threads on a device core. A work item can have regular control structures, such as branches and loops, i. e., execution paths might differ for different work items. However, within hardware, differing branches might be serialized as described in Section 2.1.2. Because devices, usually, offer SIMD capabilities, work items are grouped in physical units called *Warps*. They are transparent for the developer, since this grouping is implicitly calculated from ID ranges. But they might be considered for tuning algorithms, because these SIMD groups represent the smallest physical scheduling unit on processors and, thus, impact memory operations as described in Section 2.4.3.

On the logical level, *Threads* are organized in *Thread Blocks*, which introduce a more coarse-grained partitioning of the overall index space. While communication, synchronization, and cooperation between the *Threads* within a *Thread Block* is possible, *Thread Blocks* themselves are independent from each other and provide no means for direct synchronization between them. All *Thread Blocks* belonging to same *Kernel* can, therefore, be scheduled independently to the available *Multiprocessors* that are offered by the device, where they might run in parallel if sufficient hardware resources are available. *Kernels* are the highest layer in the OpenCL/ CUDA subtask definition and are scheduled on device level. A *Device Program* might be composed of multiple *Kernels* that, in conjunction with corresponding I/O operations, can be scheduled via *Device Command Queues*, either to different devices or to the same devices for implementing pipeline parallelism as depicted in Figure 2.9.

2.4.3 Memory Model

Figure 2.13 sketches the memory model of OpenCL / CUDA. It shows different kinds of physical memory regions on coprocessor devices that can be controlled by the software. In practice, there might be more memory types (usually in caching hierarchies) on the physical hardware. However, the latter are hardware-managed, device-specific, and, hence, not explicitly represented in a device-independent memory model.

The *Device Memory* represents the available main memory on coprocessor *Devices*. This region can be dynamically allocated and read/written from the *Host* system. It is used as staging area for buffering input and output data as linear collection of data elements. Data elements can be PODs (Plain Old Data Types), i. e., scalar types, user-defined structures, or special types for representing image data. Within a CUDA / OpenCL program, memory transfers between host and device are triggered via explicit copy operations. On dedicated devices that do not share the memory bus with the host, it is possible to *pin* specific memory regions. *Pinned Host Memory* regions are mapped into the address space of the device driver, which prevents that the operating system reuses them. This mechanism allows to schedule data copy operations asynchronously, which can be useful to implement pipelining techniques. In order to simplify memory management when writing accelerated programs, *zero-copy* memory models are offered. This mechanism, which is also referred to as *Unified Memory* in CUDA terminology,

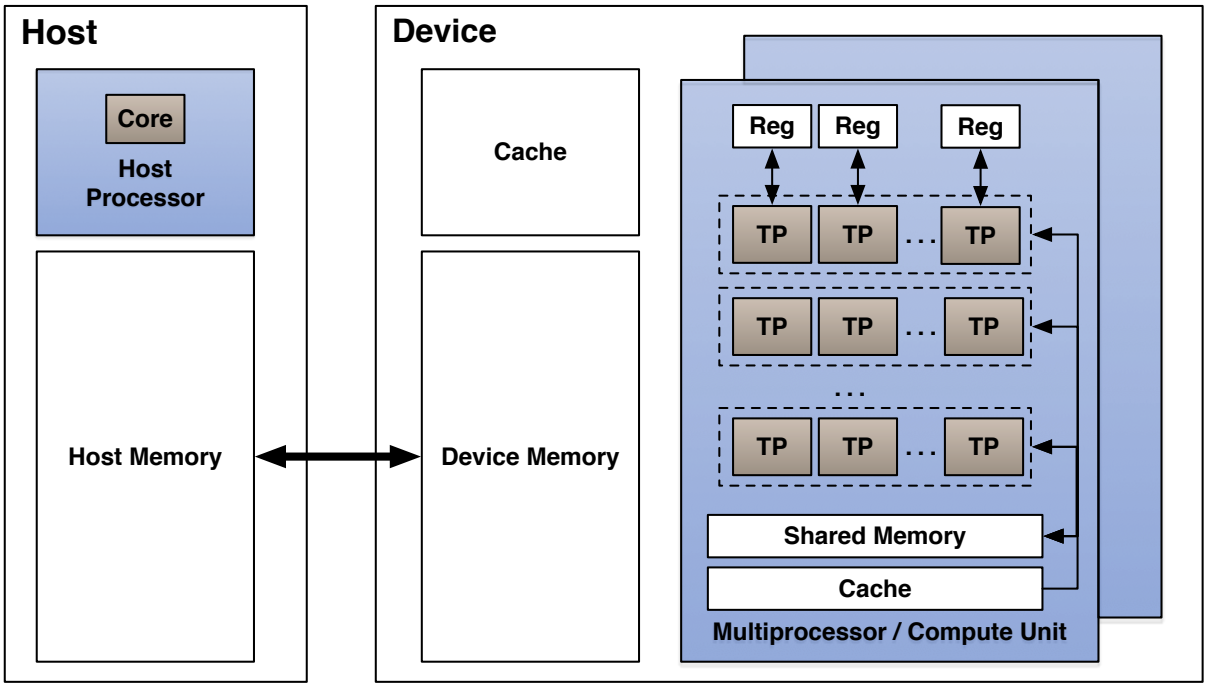


Figure 2.13: Memory Model in OpenCL & CUDA

means that host and device memory is mapped to a common address space and is, therefore, directly accessible from either side. However, this merely represents a logical simplification. For dedicated devices that are attached via a separate bus system, such as dGPUs, physical memory transfers are still required. These transfer operations are scheduled internally by the device driver on-the-fly as soon as memory addresses are accessed that are physically stored on the other side. For devices sharing the same memory bus with the host system, such as iGPUs, those physical copy operations can be omitted.

Device Memory allows shared read/write access from all *Multiprocessors* and their corresponding *Thread Processors*. But no consistency guarantees are given for concurrent accesses from different compute units, i. e., *Thread Blocks*, that are part of the same *Kernel Instance*. Within a *Thread Block*, explicit synchronizations via *Barriers* are possible to allow cooperative processing. Across different work groups, synchronization can solely be implemented via *Events* between separate *Kernel Instances*. The intention behind this model is to avoid synchronized access to *Device Memory* for maximizing throughputs. Therefore, programmers are advised to design memory accesses so that they allow data parallelism by mapping each *Thread ID* to disjoint regions of the input and output data. On hardware level, access operations are grouped in memory transactions per *Warp* that should also be considered when algorithms are designed. As described in Section 2.1.2, requested addresses within a transaction should be coalesced. The necessary parameters, e. g., starting address and transaction width, can be determined via an upfront calibration operation or by querying device properties in order to tune algorithms respectively. Usually, devices cache accessed global memory elements in hardware. But available hardware caches are rather small and no control for them is offered to the programmer.

Software-controlled caching is possible in so-called *Shared Memory* regions. The memory model specifies them as shared across all *Thread Processors* in a *Multiprocessor*, but not across different compute units. In case of Nvidia CUDA devices, this memory is located directly on the chip for reducing access latencies. OpenCL does not make any assumptions here, to offer a more general, device-independent model. *Local Memory* might be on-chip or mapped to *Device Memory* if the device does not support it. Before *Shared Memory* can be used by different *Threads*, it has to be allocated statically in the code or dynamically by the host before a *Kernel Instance* is launched. As this memory type represents a shared resource between all threads, allowing to implement cooperative algorithms, logical access operations can also be synchronized via *Barriers*. Physically, local memory is organized in different memory banks, which should be considered when algorithms are designed in order to avoid any implicit synchronization by the hardware (so-called "bank conflicts" in advanced CUDA programming).

The lowest layer in this memory hierarchy is statically allocated *Private Memory*. This region is accessible only from within the scope of a single *Thread*. Accessing it from the *Host* or by other *Threads*, even if they belong to the same *Thread Block*, is not possible. *Private Memory* is used for storing local variables and is, usually, mapped to hardware registers.

Further, the memory model specifies other regions, which are optimized for certain use cases. They are briefly described here for the sake of completeness, but are not relevant in the following. *Constant Memory* should support mathematical calculations, where a lot of constants are involved. This memory type shares similar properties with the global memory, but only allows read-only access from within a kernel. If available, it can be mapped to special memory on the hardware to utilize caching effects for such read-only accesses. Otherwise, global memory can be used as fallback. CUDA further specifies a *Texture Memory* type, which is useful for image processing tasks as it offers special access operations, e. g., clamping of memory addresses.

2.4.4 Development Model

OpenCL and CUDA strongly evolved in the recent years in terms of libraries and features offered by corresponding toolkits as well as extensions for the entire development environment. A simplified overview on the development model behind those approaches is illustrated in Figure 2.14, which focuses on the core concepts to explain the technology and mandatory steps for developing applications for hybrid platforms.

In general, it is required to develop two different pieces of code:

1. *Host* code, which implements serial parts of the application, high-level synchronization and device management operations, or simple code that, due to overheads, is not worth offloading to coprocessors

2. *Device* code, which is also referred to as *Kernel* code, that implements (data-)parallel (sub)tasks of the entire application and shall be offloaded to parallel (co)processor devices

It is possible to mix these code pieces, e. g., when host functions are defined next to kernel functions within the same source file. During a pre-compilation step that is executed automatically by the toolchain, the code will be split, completely transparent for developers.

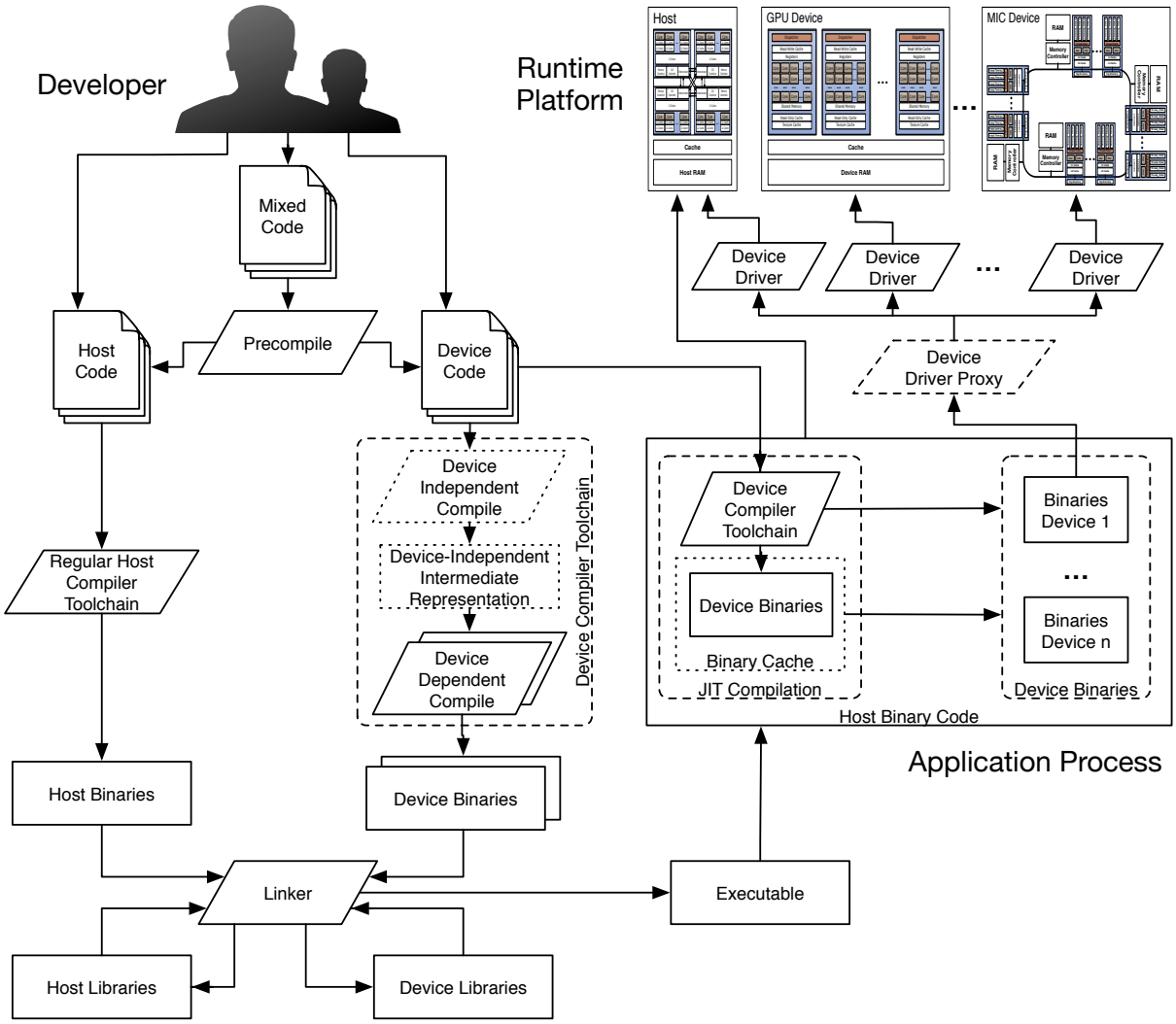


Figure 2.14: Development Model in OpenCL & CUDA

The source code is compiled for respective back-ends afterwards. Therefore, dedicated compilers are used to generate platform-specific binaries, e. g., *host binaries* for x86 CPUs or *device binaries* for GPUs. Those binaries are linked either to executable application code or platform-specific libraries that, in turn, can be linked to other applications. Such libraries can, of course, also be obtained from external sources and some of them are already part of the CUDA / OpenCL development toolkits. Executable application code is always generated for the host platform[9], which can be instantiated by the operating system for creating a new *application process*. It combines all instructions for the host part as well as the interaction with device back-ends.

[9] "Exotic" cases, where this compilation is also integrated into yet another program at runtime are ignored here. This model might be implemented, e. g., for JIT query compilation, but is not considered in this thesis.

In contrast to this *statically precompiled* code for the host, the executable code for devices, usually, provides some more flexibility. While OpenCL focused on a *JIT (Just In Time) compilation* model during runtime, a model of precompiled device binaries has originally been preferred by CUDA. But since CUDA release version 7, runtime kernel compilation is supported, too. JIT compilation enables the implementation of plugin mechanisms for device code. It requires that source code text is either directly embedded in programs or loaded from disk when the application is executed. The respective compiler and linker are also invoked at runtime, usually through toolchain libraries that were linked with the application's executable. Of course, JIT compilation involves overheads that can significantly impact application performance. In order to mitigate these effects, device binaries can be cached by the application to avoid their re-generation in case respective kernels are required multiple times.

Device binaries are loaded into the address space of the already launched process, where parameters are bound to kernels before they are scheduled to available devices in the parallel execution back-end. Those devices are accessed through *Device Drivers* that have also been linked to the application. Multiple of these drivers may exist that support different back-ends and need to be installed on the system for each device that shall be accessed.

Originally, OpenCL as standardized solution has been one step ahead before CUDA in this respect, because drivers for standardized APIs could be easily implemented by various vendors for their platform, either as proprietary drivers, e. g., from Intel, AMD, Nvidia, IBM, Apple, and others (usually supporting their own hardware) or as open-source projects, such as Beignet [92]. In case of OpenCL, where a plurality of many drivers might be available on a single system, drivers for a single device can be linked and accessed directly or an ICD (Installable Client Driver) can be used. The *Installable Client Driver* implements a thin layer on top of actually installed device drivers. Having the same standardized interface, it acts as driver proxy and allows dynamic selection of available back-end devices during runtime, which is useful in heterogeneous environments. Driver calls are dispatched to the actual back-end, completely transparent to the application.

CUDA that originally targeted Nvidia devices only, caught up with OpenCL recently. Instead of solely mapping device code to Nvidia's proprietary PTX format as intermediate representation that abstracts from the actual CUDA-enabled device architecture, a compilation to LLVM (LLVM compiler infrastructure project, formerly Low Level Virtual Machine) is now also supported by the toolkit. This open intermediate representation allows the integration of other compilers that generate code for other execution back-ends, such as x86 or the POWER architecture. Further, community-provided approaches exist that allow the compilation of parallel programs to FPGAs for both development environments, CUDA [165] and OpenCL [50].

2.4.5 Comparison of CUDA and OpenCL

There is an abundance of literature available that compares OpenCL to CUDA according to different criteria, ranging from scientific papers over white papers and technical reports offered by hardware vendors to programing blogs, etc. The base for these comparisons are usually CUDA and OpenCL implementations, which are examined for specific use cases. However, there shall not be too many details repeated here. Instead, some criteria are discussed that have been considered when the actual model was chosen for implementing the algorithms of the index framework.

Anticipating the final result of this discussion, one can say that there are no compelling reasons that clearly favors one approach over the other from a conceptual point of view. Both models share exactly the same core ideas for specifying data-parallel algorithms that can be offloaded to coprocessor hardware. The major differences lie in the terminology that is used [84, 172]. Therefore, a glossary for both APIs is included in Appendix A in order to avoid confusion.

Sometimes, CUDA programs are claimed to yield better application performance, because the CUDA toolchain is implemented by the same vendor that offers compatible hardware and, therefore, is optimally adjusted for the latter. Additionally, more controls to fine-tune algorithms for the underlying CUDA-compatible hardware are provided. Furthermore, additional overheads are expected in OpenCL programs because of the higher generality of the OpenCL specification. But, usually, observed differences are negligible [56, 105]. Finally, talking about "performance" of programming models and corresponding toolchains is not considered an appropriate discussion here. The main reason is that the evolution of both, these software development tools and the available underlying hardware platforms, is assumed to be faster than the evolution of algorithms developed with either of these models. Therefore, the results of performance measurements are expected to be highly fluctuating, having a strong dependency to the way how algorithms are implemented, onto which runtime platform the code is deployed, and which device models are used.

From an algorithm development perspective, which is important for the context of this thesis, other criteria, like programming language support, the availability of different development tools, and the quality of available documentation shall be given more weight in this discussion. The portability as well as the maintainability of developed algorithm code are also deemed to be important, because, in most cases, they outlive tool and platform versions, including built-in optimizers, etc.[10]

In terms of tools, available code samples, and literature, CUDA is considered better than OpenCL. Because the whole CUDA platform development is driven by Nvidia, there is only a single source of available drivers and corresponding tools. They integrate very well with each other and major releases including new features are published frequently [156]. Further, Nvidia invested a lot to form a developer community around CUDA, published tons of training material, offers seminars, and organizes developer conferences. Looking at OpenCL, the landscape

[10] The following discussion represents the author's personal opinion after developing algorithms using both models.

is more heterogeneous. Because OpenCL is merely a standard specification, all hardware vendors have to provide conforming drivers and corresponding tools. Therefore, a lot of different versions exist and the initial hurdles for getting started are higher. Specifically, in heterogeneous systems that use hardware from different vendors, the initial setup revealed to be much more complicated than setting up a CUDA-only environment. Different drivers have to be installed and integrated into the operating system that had different support levels and showed incompatibilities.

In terms of portability and maintainability, OpenCL is considered better than CUDA. As a standardized API, OpenCL allows to specify data- and, to some extend, task-parallel algorithms in a platform-independent manner, while the whole CUDA toolchain is only supported by proprietary Nvidia hardware. While Nvidia clearly focuses on GPUs, OpenCL is supported by a much larger range of platforms, ranging from GPUs over multi- and many-core CPUs to FPGAs. The runtime compilation model that is favored by OpenCL allows to dynamically map specified algorithms to drivers of the actually available hardware. In the CUDA model, there is no need to integrate multiple heterogeneous drivers at runtime. However, this picture changed a little over time. CUDA now also supports runtime compilation and compilation to an intermediate representation (LLVM) that, in turn, can be compiled to different back-ends.

When considering the C++-based programing languages behind OpenCL and CUDA, at the time of writing this thesis, CUDA offered more features, such as C++11/14 and template support. Although OpenCL catches up in this respect and similar features are included in recent language specifications [7], they can hardly be used in practice. Merely standardizing such features is not sufficient here, because many vendors have to adapt and implement them, before they can actually be used by the developer community. Naturally, this adoption process is much slower compared to a homogeneous environment. Looking at the list of available OpenCL drivers [108], it can be clearly seen that most versions are still back-level. While AMD and, to some extend, Intel seems to push support for OpenCL version 2, Nvidia is still merely offering support for OpenCL 1.2. The interpretation of corresponding reasons is left to the reader. However, from a developer's perspective, working in heterogeneous environments, this picture is far away from a standardized "one-approach-fits-all"-solution, which leaves an impression that this will never be achieved on this level.

In summary, both toolsets fit the requirements for this thesis as they share a common model to specify parallel algorithms. OpenCL as standardized variant would, theoretically, be a better choice for implementing the algorithms, because they would be portable to a much broader set of physical platforms. However, CUDA was favored because of better tooling support and documentation. Further, OpenCL offered less support for advanced features, such as template programming, etc. The last, even more practical reason was that Nvidia hardware (Fermi and Kepler architecture) was available in the computer cluster of the graduate school that funded this work. But this decision is not considered a restriction to the applicability of presented concepts. Due to the large number of similarities between both models, tools exist that allow an automatic translation of CUDA programs to OpenCL versions [84, 147]. So, at least in theory, it should be trivial to translate algorithms written for one framework to the other one, although additional fine-tuning for the underlying hardware might be required to achieve good performance [53].

2.5 Summary

There has been a rapid development in parallel processor architectures in the last years. The landscape of parallel processor devices became pretty complex and many platform-specific subtleties need to be considered when their features shall be fully exploited in order to achieve peak performance. Significant tuning efforts are required to optimize processor utilization and memory access patterns, available caches should be leveraged to eliminate bandwidth issues, and algorithms need a design that does not suffer from synchronization overheads.

The presented architectural differences of processor types revealed strengths and weaknesses for specific usage scenarios. Thus, combining their strengths and working around their weaknesses in hybrid environments is a promising approach, but adds additional complexity to this already complicated situation. Developing algorithms for this environment is a challenging task that binds a lot of resources.

Parallel programming models hide many hardware details behind proper layers of abstraction, lowering development costs at the cost of losing the possibility to fully exploit all internals of a specific platform. Different approaches exist that offer different granularities for modeling parallelism, with more or less options for developers to incorporate knowledge about the application context. Programming languages, like CUDA or OpenCL, that are specifically designed for parallel execution models are the method of choice in this thesis. They can be used to specify algorithms that are, to some extend, independent from the actual execution hardware, because corresponding toolchains automatically compile programs to device-specific binaries. As prerequisite, the program of the application domain must be broken down into independent pieces that can be mapped to task-, data-, and pipeline-parallel algorithms. After this has been accomplished, resulting algorithms can be mapped to the corresponding programming constructs that are offered by these languages.

The background information provided in this chapter[11] founds a solid base for the next ones, where tree-based indexing algorithms are parallelized and a framework is designed for integrating several processor types in order to execute these operations on hybrid CPU-GPU systems, without requiring additional modifications of the code. Independent tasks will be identified and mapped to CUDA as one of the most popular programming models for this environment in order to abstract physical execution devices. Further, memory access patterns will be analyzed for minimizing necessary data transfer operations and scheduling algorithms will be developed for deciding which of the available processor types shall be employed for executing a particular operation. This is required as last building block in order to leverage the full power of a hybrid platform. As discussed in the previous chapter, the complexity of these algorithms and involved data structures is hidden behind the well-known facade of the GiST indexing library, which acts as second, logical abstraction layer and simplifies the integration of the framework into application code.

[11] A comprehensive overview of the content of this chapter, including programming models techniques, and examples for database management operations, has also been published in [177].

Chapter 3

Related Work

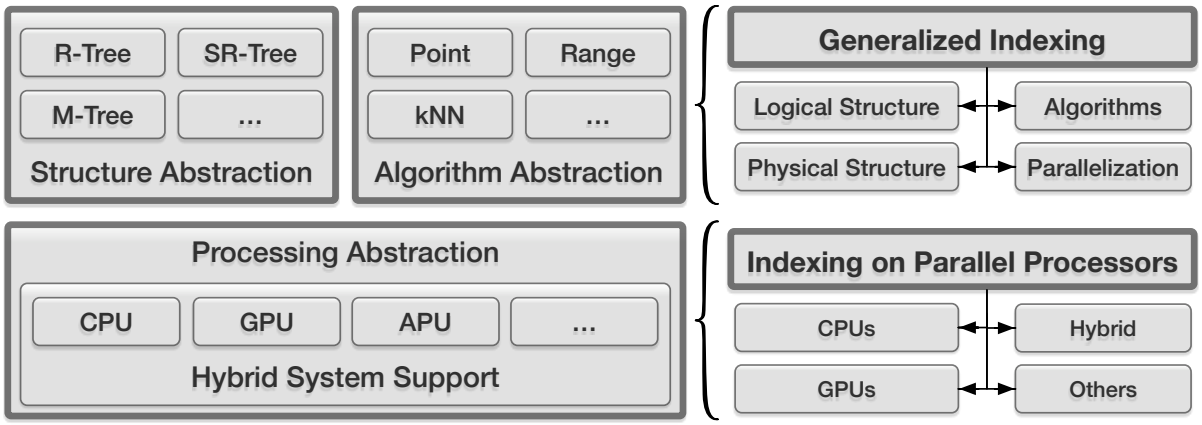


Figure 3.1: Overview of Related Work

In order to enable support for different processor types for any kind of index structure, it is required to abstract both, the logical differences of each index type as well as the the physical execution of index algorithms on specific processor devices. Therefore, prior art that exists in these two areas is analyzed in this chapter and approaches are discussed that can be integrated into a common framework in order to reach this goal. The exiting prior art can be roughly classified into different categories that are illustrated in Figure 3.1.

State-of-the-art techniques for tuning specific index structures for specific processor types as well as a combination of different ones in hybrid systems are studied in Section 3.1. Insights gained from this analysis can be used to identify commonly used techniques that are generalizable for different processor architectures in order to integrate them into the framework's execution layer. Existing frameworks for modeling index structures are subject of Section 3.2. They already provide means to abstract structural and algorithmic differences of many existing index variants. These abstraction layers have to be analyzed carefully in order to identify reusable parts that can be adapted to transparently integrate (co)processor support under the hood of commonly shared framework code. These two discussion threads are joined in Section 3.3 in order to derive strategies for generalizing the processing aspect in generalized indexes. Open tasks are outlined that have not been covered by existing related work so far, which are addressed later in the following chapters.

3.1 A Survey of Index Processing Techniques on Different Processor Types

The optimization of index structures for specific processor architectures is a well-analyzed topic in literature. Many approaches exists, which are primarily categorized here by the processor type that was used as basis for the optimizations and, secondarily, according to the aspect that is improved: the organization of the index data structure, data access patterns during the execution of algorithms, and parallelization approaches for the latter. Of course, several publications address multiple architectures and employed techniques are related in many cases or even depend on each other. For example, in order to use vectorized operations for data-parallel evaluation, the data structures have to be carefully laid out in memory before a single instruction can be executed on many elements at once, no matter whether a CPU or a GPU is used for this. Therefore, some articles might be referenced multiple times in the respective sections. However, identifying such overlapping techniques is exactly the target of this thesis as they represent suitable candidates for generalizing them via a common framework.

Adapting indexes to the evolution of CPU architectures gained the most attention in research so far. The main topic for the majority of publications is the efficient use of the available cache hierarchy as data access revealed to be a major bottleneck in data management [12]. Another big topic is the parallelization of index algorithms on different levels, leveraging multiple cores as well as SIMD operations on a single core. Related work in these areas is subject of Section 3.1.1.

Since the advent of GPUs that can be freely programmed for generic tasks, beyond graphic operations, this computing platform gained reasonable attention by data management research. Indexes as a subcategory of the latter are no exception here. Thus, existing techniques for exploiting the massive parallelism on GPUs for accelerating index operations are subject of Section 3.1.2. Main topics here are the efficient management of data structures in device memory as well as the mapping of tree operations to the data-parallel processing model of a GPU's vector processing units.

Other processor architectures that currently play a niche role for processing index operations are briefly mentioned in Section 3.1.3. However, as these processor types have not been widely used in practice for this use case, yet, it would be hard to identify and generalize well-established techniques. Therefore, they are just included here as reference for potential future extensions in this direction.

Finally, the combination of multiple processor types in a hybrid system is discussed in Section 3.1.4. The focus lies on a combination of CPUs with GPUs as the most commonly used setup in practice. Big topics here are strategies for data sharing between both computing platforms as well as strategies to schedule index operations to the available processing units.

3.1.1 Indexing on CPUs

3.1.1.1 Data Structure Organization

A common technique to optimize the structure of an in-memory index tree for read access is to linearize its nodes, i. e., storing the node data in an array so that they can be addressed directly via offset computation instead of indirect access via pointers. The CSS-Tree [170] is one of the first structure that implements this for the B-Tree index family. Using this technique saves costs for storing pointers and, thus, more search keys can be stored inside a node. However, changes in the tree structure are not possible without rebuilding the memory layout. The CSB$^+$-Tree [171] addresses the update issue of CSS-Trees by applying pointer elimination only partially. Pointers exist to node groups that are itself linearized and can use offset computations within the group. Updates can be handled similar to B$^+$-Trees by updating groups and group pointers instead of single nodes. The node sizes are optimized for caching them in lower-level CPU caches. Upon access all keys are fetched in bulk using a single cache line transfer and can be processed cache-local.

The previous methods were designed for 1-dimensional, ordered domains. The CR-Tree [170] applies the same ideas for multi-dimensional data by proposing a cache-sensitive layout for R-Trees. Additionally, key compression methods are proposed, because n-dimensional MBRs as keys consume more memory than pointers and merely applying pointer elimination does not save much space. Key compression techniques have also been proposed for the B-Tree family [21, 32, 73]. The main idea here is to identify significant bits that distinguish key ranges for binary search. Insignificant prefixes can be reconstructed from traversal paths and, thus, can be removed in lower tree layers.

The CSB$^+$-Tree optimization of tree nodes for caching hierarchies is elaborated further by FAST (Fast Architecture Sensitive Tree) [111, 112]. FAST organizes a binary search tree with a technique called *hierarchical cache line blocking*. Tree nodes are recursively arranged in multiple block layers that match the available layers in the memory hierarchy, from (vector) registers over processor cache lines to blocks for disk-based I/O. The block size of a cache layer matches the transfer unit size of the next upper layer. Prefix compression is also applied by FAST to maximize bandwidth utilization even further. The VAST-Tree [201] extends FAST by a combination of lossy and lossless compression techniques. Lossy compression is used near the root layer, gaining efficient space utilization by sacrificing pruning accuracy. Lear leaf nodes, lossless compression is used for accurate filtering of query results.

While the previous structure optimizations were static, ART (Adaptive Radix Tree) [131] proposes a technique to adapt the node structures of a prefix tree dynamically, based on the data distribution. The main idea is that in sparse data sets not all possible space partitioning splits are used. This can be exploited to reduce node sizes for low fill factors. ART provides different node types that use a combination of direct and indirect addressing of child entries. In addition, some node-specific optimizations can be applied during query evaluation, e. g., the use of SIMD operations. A comparable technique to use different addressing schemes between tree layers has been employed by the KISS-Tree [120]. In addition, the KISS-Tree bounds the tree depth of the underlying prefix tree, which could, in general, grow arbitrarily. The data-dependent tree depth is also addressed by HOT [28] that presents an approach to optimize prefix tree heights.

3.1.1.2 Data Access Patterns

A common optimization technique that can be applied to an algorithm whose data access patterns are known is prefetching. For tree-based indexes, this has, for example, been adopted by the pB$^+$-Tree [45]. The pB$^+$-Tree uses nodes that exceed the size of cache lines. Further, it uses separate arrays for storing search keys and pointers to child nodes, because they will be needed in different phases during the tree traversal. While the first search keys are evaluated, data that is accessed next can already be fetched into the cache so that it is available once the computation finishes. For example, the next set of keys that resides on the next cache line may be fetched or child nodes that are needed for the executing the next traversal step after the evaluation.

While prefetching minimizes execution delays caused by data access latencies of a single operation, batch processing can be applied in order to optimize an algorithm's throughput for many operations. By executing independent operations in batches, the locality of data accesses can be increased and cache thrashing can be avoided, i. e., all operations that would access the same tree data independently during their traversals are executed simultaneously, operating on the data that has been cached once for all of them. Buffer trees implement this for tree operations [18, 211] by associating query buffers with each tree node. These buffers are filled by queries that follow the same paths when traversing the tree structure. For processing the buffers, different strategies are provided that, for example, guarantee query response times. In order to utilize this technique efficiently, it is required that the index workload can be batched, i. e., the application using the index must be able to issue many queries whose results do not depend on each other. This technique has, for example, been successfully applied by a message broker system that batches index probes [124], in SharedDB [68] that implements a database core relying on batch query processing in general in order to utilize caching effects for all operators, and other state-of-the-art database engines that use batch processing on tuple-level [34, 144, 212].

Prefetching and batch processing are general techniques, which can be used to explicitly adapt any algorithm so that it becomes cache conscious. Access linearization of index keys is another, independent optimization technique, which considers the interaction of one specific algorithm with the underlying data structure so that it becomes cache-conscious implicitly. In order to apply it, data accesses made by the algorithm between consecutive iterations or recursive method calls are tracked and rolled-out in memory. That is, accessed elements are reordered so that they are contiguously stored in order of the access sequence. This technique is related to pointer compression of CSS- and CSB$^+$-Trees, where entire tree nodes are rolled-out in level-order so that child pointers can be eliminated. K-ary binary search refines this idea on key-level for arrays [179] and FAST [111, 112] for binary search trees. Linearizing elements according to their access sequence increases access locality, facilitates software-controlled prefetching [13], and allows to apply vector operations, which are discussed in the next section.

Recent research address the placement of data in memory, considering the NUMA architecture of modern CPUs [109, 130, 142] and multi-GPU setups [151]. Access latencies increase significantly with increasing distance of the accessing core from the memory bank that actually stores the data. The main idea behind NUMA-aware algorithms is to track in which NUMA region the data resides, e. g., by partitioning a tree and storing each of them on a different region, and distribute tasks to these regions so that inter-core data accesses are minimized.

3.1.1.3 Parallelization of Index Operations

For parallelizing index operations, mainly two approaches exist, the vectorization of the algorithms so that they can be evaluated by utilizing data-parallel SIMD capabilities of modern processors, and partitioning of the workload into independent parts that can be dispatched to the available cores for parallel execution.

The MASS-Tree [145], a combination of B^+-Tree and prefix tree, shares the tree between all cores and independent operations are scheduled to the available cores for concurrent execution. The parallel buffer tree [187] extends the buffer tree [18, 211], so that multiple cores can cooperate to process the operation sequences that are applied on the tree in batches.

FAST [111, 112] also implements a buffered processing scheme for batched query operations. However, the main focus of FAST is to utilize SIMD operations for speeding-up query evaluations. Based on the linearization of search keys inside the index according to the data accesses that are caused by binary search, the search can be implemented by SIMD processor operations, in a similar way to k-ary binary search on arrays [179, 181]. Many keys from different nodes can be fetched into a single vector register and compared to a (set of) query key(s) at once using a single processor instruction. Comparable vectorization approaches exist, for example, for B^+-Trees [183] and prefix trees [28, 131, 205].

When multiple data elements are processed via vector operations, some of them might not qualify anymore for the evaluation, e. g., because a query has been filtered previously and follows a different tree path than those that are simultaneously evaluated. Common techniques to address this issue is the application of bitmask operations, which filter disabled elements during the operation, and the use of static lookup tables that comprise entries for all possible evaluation results, which can be probed to derive the next tree traversal step. All of the aforementioned SIMD approaches apply some of these techniques to handle branch divergence. However, such diverging branches waste processing resources by executing redundant calculations that do not contribute to the final result anymore. POKER [207] addresses this issue by optimizing query batches so that divergence is minimized. The available batches that need to be processed are analyzed and queries are reshuffled to group those queries that follow the same path in the tree.

3.1.2 Indexing on GPUs

3.1.2.1 Data Structure Organization

One of the major challenges for implementing index operations on GPUs is the mapping of the recursively defined data structures to a memory layout that is suitable for the GPU's massively data-parallel model. A first technique in this context is similar to the CSS-Tree [170] memory layout, where tree nodes are stored in level-order in a contiguous array after it has been bulk-loaded. Almost all tree-based index implementations for GPUs use this technique, e. g., GPU CSS-Trees [57], k-d-Trees [69, 168], k-d-B-Trees [113], B^+-Trees [61, 184] R-Trees [125, 141, 167, 202], and hybrid tree implementations [116]. Another technique for linearizing tree nodes in memory is to enumerate nodes with space filling curves and use a hash table for resolving node addresses. This has been implemented for an Octree GPU port [143].

The second major technique to optimize tree data structures for parallel accesses is to use separate storage locations for different fields inside node structures. In GPU implementations, this technique is often referred to *SoA (Structure of Arrays)* vs. *AoS (Array of Structures)* memory layout and is comparable to a columnar vs. row-wise storage layout of relational database tables. The key idea here is to store search keys and corresponding child pointers in separate, contiguous arrays. Afterwards, they can be accessed using vectorized read operations, without wasting memory bandwidth for transferring data that is not directly needed for computations after the access. The SoA technique is comparable to the CPU cache optimization proposed by the pB$^+$-Tree [45] and is implemented by many GPU indexes [116, 168, 184].

The biggest challenge regarding the memory hierarchy that is involved with GPU-based computing is to optimized data transfers between host and device memory for offloading computations. Almost all GPU index implementations that were discussed so far simply assume that the entire index can be stored in global GPU memory, either after creating it on the CPU-side and transferring it to the GPU, or after bulk-loading it (in parallel) on the GPU. This significantly limits supported indexes sizes compared to CPU-based implementations, because GPU main memory capacities are much smaller. There are a few implementations that do not depend on this assumption and allow out-of-core storage of index data, transferring it dynamically from host memory to the devices. However, they are classified as hybrid approaches, which are covered in the next Section 3.1.4.

Although modern GPUs offer large transfer bandwidths for accessing data in global GPU memory and are very good at hiding access latencies by overlapping blocking transfers with concurrent computations, there are on-chip caches that can be utilized for tuning memory accesses even further. Unlike CPU implementations, hardly any GPU-based index exploits this potential by tuning the data structures for lower-level caches. An R-Tree variant exists that stores index data in SHM (Shared Memory) caches to avoid slow global memory accesses during the tree traversal [125]. However, the proposed solution does not scale for large index sizes.

3.1.2.2 Data Access Patterns

Data access optimization techniques for hiding main memory access latencies, such as prefetching on CPUs, are less important on GPUs, because the aforementioned latency hiding mechanisms, which are directly implemented in hardware by the GPU's thread scheduler, do not require manual software-controlled prefetch instructions anymore. Hiding latencies caused by host $\leftrightarrow$ device transfers is more important. Even if the index can be completely stored in GPU memory, query input and their results need to be copied via the interconnecting bus system before/after offloading index operation kernels. In order to hide these intermediate transfers, pipelining of independent kernels can be applied (cf. Section 2.2.2), which is also referred to as *double buffering* in GPU terminology. The HB$^+$-Tree [184], a GPU-based B$^+$-Tree implementation uses this technique, for example.

A more commonly used technique for optimizing data access patterns is to use batch processing for queries. Unlike CPU implementations that implement batch processing because it increases data access locality and, thus, leverages low-level caches more efficiently, the motivation for applying this technique on a GPU is to increase the amount of data parallelism that can be applied. Many index implementations leverage this optimization technique to maximize query throughputs [57, 61, 69, 71, 90, 184, 202, 206].

A crucial technique for mapping an algorithm to the data-parallel GPU execution model is to linearize memory accesses so that many GPU threads can work independently on the input. In order to coalesce access requests and maximize bandwidth utilization, the offsets of accessed data elements must be consecutive. For tree-based algorithms, mainly two parts need to be modified, which rely on random access in non-vectorized CPU-based implementations: tree traversals iterations and entry access for evaluating tree nodes within a single iteration step.

The SoA memory layout facilitates traversal linearization when nodes and entries are reordered so that the position of input elements increases predictable when traversal proceeds, like proposed by FAST [111, 112]. The next step can be implicitly derived by tracking node offsets locally in each thread. For all other cases, e. g., when multiple traversal patterns should be supported or irregular random accesses, like backtracking, are involved, either recursive methods are needed that manage traversal states on the function call stack, or the state needs to be tracked explicitly in a shared data structure. Recursive function support is very limited on GPUs. Some tree implementation exists that uses the dynamic parallelism feature [113, 167]. The majority of GPU index implementations rely on "stackless" traversals by implementing a shared stack [62, 71] or queue [113, 135, 140, 141, 202] on the GPU. In order to implement data-parallel updates on these structures, the algorithms are partitioned into different phases. First, each thread aggregates the state update locally. Second, disjoint update positions needs to be calculated, e. g., by employing a parallel prefix sum primitive [29, 83]. Third, the update can be executed in parallel without additional synchronization mechanisms. Alternatively, the index can be traversed multiple times, (re)starting at the root node if the regular traversal would backtrack [114]. But this technique requires additional methods to reduce redundant computations, e. g., an enumeration of already visited nodes so that they can be skipped on traversal restart.

Using the SoA memory layout addresses coalesced entry accesses in general. While entry reordering allows to linearize binary search in ordered domains [104, 111, 112], most algorithms do not implement this and rely on the much simpler and generally applicable brute-force evaluation of all keys, which is also referred to as "braided search" in literature [61, 113]. For most indexes, where no ordering between entries exist, braided search needs to be applied anyway [203, 204], if no other techniques, such as deriving an artificial entry order by using space filling curves, are used in addition.

3.1.2.3 Parallelization of Index Operations

Parallelized indexing algorithms on GPUs share many ideas with those indexes that apply SIMD processing on CPUs, because the GPU architecture is inherently data-parallel. Some GPU indexes, such as FAST [111, 112], even use the same algorithms on both platforms. The main difference is that the SIMD width needs to be much larger on GPUs in order to utilize all available processing resources. Compared to their CPU counterparts, usually, larger node fan-outs are used by GPU trees to leverage the full parallelization benefits of braided key evaluations.

As discussed previously, most GPU indexes use batch processing for optimizing query throughputs. The degree of parallelism can be significantly increased when independent queries are processed simultaneously. Further, multiple batches can be scheduled to utilize the available GPU cores. When multiple batches are processed simultaneously, branch divergence is also an issue, like in vectorized CPU implementations that use SIMD instructions. A common technique to address this problem is to reorder queries, just like POKER [207] proposes for CPUs. For GPUs, this has been implemented for k-d-Trees [71].

Search operations in ordered domains can be efficiently parallelized on GPUs by implementing a vectorized k-ary binary search algorithm [104]. Some indexes implement this [111, 112, 206], but most of them rely on braided search that simply evaluates all keys in parallel, because data access patterns are much simpler (cf. Section 3.1.1.2).

Most GPU indexes maximize query throughputs, accepting higher latencies for single queries. They use batch processing to increase the SIMD width for single-node evaluations and apply a parallel BFS (Breadth-First Search) as tree traversal strategy [125, 135, 140, 141, 167, 202]. That is, the tree is scanned in level-order, processing all matching nodes in parallel. The traversal state is tracked by using one of the methods that were described in the previous section.

However, some implementations exist that minimize latency by parallelizing DFS (Depth-First Search). FAST [111, 112] is one of them, because it evaluates multiple levels of binary search in a single vectorized step. The massively parallel restart scanning algorithm [114] traverses a tree in depth-first order and restarts traversals if a backtrack is required. A third DFS implementation is implemented on top of prefix trees [194]. The algorithm speculatively processes nodes of lower tree layers, whose parents have not been evaluated, yet, in parallel to their parent nodes in order to speedup traversals. The idea behind this method is that a GPU offers plenty of processing resources that are hard to fully utilize by an inherently sequential DFS traversal. The DFS can be accelerated by executing potentially redundant evaluations that may not contribute to the final result set and could be avoided in a sequential algorithm, e. g., when a query would be filtered in the parent layer. This approach is comparable to FAST, which also evaluates tree nodes speculatively that are packed into the same vector register. Both algorithms need to merge speculatively computed results along the path that would be evaluated sequentially. Therefore, a lookup table is used that comprises an entry for all possible evaluation results.

RegTT [206] combines BFS and DFS traversal strategies for batch-wise query processing in a hybrid algorithm. RegTT executes BFS for upper tree layers near the root and uses DFS for the lower parts near leaf nodes. The key idea here is that BFS can be used to quickly fan-out searches for the upper tree parts, where it is likely that most of the queries inside a batch share the same paths. The algorithm switches to DFS processing once too much divergence between them is observed that causes many inactive elements in vectorized operations.

3.1.3 Indexing on Other (Co-)Processors

There are a few index implementations for other processing hardware than CPUs or dGPUs. The first one implements an R-Tree directly in hardware circuits [199], using a hardware description language, which could be mapped to an FPGA, for example. For evaluations, a simulation software was used. The R-Tree is encoded as adjacency matrix and traversal algorithms have been modeled as parallel matrix computations.

The CPU implementation of the PALM-Tree [183] can also be deployed on the MIC architecture. Each core uses SIMD operations for query processing. The algorithms are comparable to FAST [111]. Further, operations are processed in batches to utilize all processing resources.

DIDO [208] implements a key-value store, which leverages APUs for query processing, i. e., the index resides in a memory region that is shared between a CPU and an iGPU, which both access the same data via a shared memory bus. DIDO schedules operations dynamically to the processing units at runtime by employing a cost model-based scheduler. The discussion of such hybrid systems is covered in the next section.

APUs are also used for a B^+-Tree implementation [51]. Similar techniques are applied that have already been discussed previously, such as linearizing the tree structure, using a SoA memory layout, tree traversal via offset calculations, batch processing, and reordering search keys in batches to minimize branch divergence.

3.1.4 Indexing on Hybrid Systems

There is an abundance of research on the use of hybrid systems that employ multiple (co)processor types for accelerating specific workloads. For this thesis, combinations of a CPU and a GPU are of particular interest. Mittel and Vetter provide an extensive study [153] about existing algorithms of different problem classes in this area. They classify them according to different criteria, such as scheduling and partitioning strategies, and also provide a broad overview of common programming languages and development frameworks for implementing hybrid algorithms. Further, Breß surveys hybrid processing approaches for database management systems in his Ph. D. thesis [36]. He also proposes an operator scheduling framework, which will be adapted to the indexing domain in Chapter 6.

Existing hybrid CPU-GPU indexing approaches can be roughly classified according to the strategy that is applied in order to decide which tasks should be executed by which processing unit. *Static* approaches always pick one processing unit for specific portions of the hybrid algorithm, e. g., because its architecture suits the requirements of this phase best. *Dynamic* approaches select the best-fitting device for a single task at runtime, e. g., because the same task might either fit one or the other processor type, depending on some workload-specific parameters that cannot be statically selected.

3.1.4.1 Static Scheduling

Static hybrid scheduling has, for example, been implemented by several key-value stores. MegaKV [209,210] implements a pipeline for processing key-value store operations on a hybrid CPU-GPU system. Indexing is a performance-critical part inside this pipeline and is statically assigned to GPUs, while other tasks, such as network processing, preprocessing queries for index access, etc. are executed by the CPU. MegaKV assumes that the entire index, a GPU-optimized cuckoo hash table, can be stored on the GPU device. MemcachedGPU [87] implements a similar mechanism. In addition, it uses DMA (Direct Memory Access) for pushing network processing to the GPU. Both systems rely on batch processing in order to maximize request throughputs by exploiting the available parallelism provided by the GPU.

Another approach in this category is the Buffer k-d-Tree [52, 69], which implements the directory scan on inner nodes with a k-d-Tree on a CPU and executes a brute force scan on the leaf layer that is hosted on a GPU. The directory scan groups queries per leaf node that need to be scanned for filtering the final results. Once enough queries gathered, the buffers are flushed and processed in parallel on the GPU. The idea behind this is that the recursive k-d-Tree traversal for inner nodes involves a lot of branching and random memory accesses, which suits the CPU architecture best, while the batch-oriented braided search is much more efficient on a GPU, because the GPU's high memory bandwidth can be fully utilized. However, the leaf layer must reside in GPU memory in order to avoid a data transfer bottleneck via the host $\leftrightarrow$ device interconnect.

In order to parallelize query processing for a k-d-B-Tree, Kim et al. propose to partition the tree into disjoint subtrees that are owned by separate cores each [113], either by a CPU core or by a GPU multiprocessor core. For query processing, all cores can cooperate to aggregate results locally, which are aggregated by the CPU in order to produce the final result set. The subtrees are always constructed and updated on CPU-side and are simply transferred to the GPU if anything changed for that particular tree partition, which has been assigned to this processing unit. Although the tree partitioning happens dynamically at runtime, the query processing algorithms are statically dispatched.

Liu et al. propose a technique for optimizing multiple query batches that shall be executed on a GPU-based index [136]. The CPU is used to shuffle queries between batches, before they are dispatched to the GPU. The idea is comparable to POKER [207], i. e., by grouping correlated queries, branch divergence should be minimized. In order to determine query correlation, the original batches are scheduled to the GPU and traverse the upper half of the tree, tracking divergence in a node visitation matrix. The latter is used for optimizing the original batches, assuming that the correlation behavior does not change for the lower tree part, whose traversal represents the major work in query processing, because much more nodes reside in lower layers.

3.1.4.2 Dynamic Scheduling

DIDO [208], a key-value store implementation for APUs, uses a dynamic strategy to distribute tasks between between CPU cores and an iGPU. In order to handle load imbalances for these processor types, DIDO uses work stealing [30]. That is, a task queue is associated with each device type and once one of them runs out of work, it takes over some work from other queues.

A different strategy has implemented by the HB$^+$-Tree [184], a hybrid CPU/GPU B$^+$-Tree variant. Unlike DIDO, which partitions entire trees, the HB$^+$-Tree partitions the tree level-wise. Leaf nodes are just stored and evaluated on CPU side, because main memory is a scarce resource on GPU devices and leaf nodes consume most of the memory that is required for storing an index. Inner nodes are mirrored between both processors, so that they are both able to answer queries for the directory scan, before the CPU is used for processing the leaf layer. A load balancing scheme can be implemented on top of that, selecting the GPU for directory traversals if the CPU is busy with evaluating leaf nodes and generating the final results.

The opposite approach has been implemented for a hybrid CPU/GPU R-Tree variant [203, 204], i. e., the leaf layer is offloaded to the GPU to benefit from its increased main memory bandwidth when scanning the entries. The GPU is used as cache for storing frequently accessed leaf nodes of a disk-based R-Tree. In order to determine which leaf nodes can be evaluated on this cache, an additional in-memory tree, the Q-Tree, is used, before the query is processed by the disk-based index. All matching nodes from this phase are answered by scanning the cache in parallel on the GPU. They can be ignored by the CPU R-Tree traversal and, thus, avoid additional disk I/O operations. The remaining matching leaf nodes from the CPU part are fetched and inserted into the GPU cache for subsequent queries, using an eviction strategy on cache overflow.

3.1.5 Summary – Common Implementation Techniques for Indexes on (hybrid) CPU/GPU Systems

In summary, the state-of-the-art index implementation techniques can be classified into the following categories:

- **Data Structure Organization Techniques** that address the question how the index (tree) structure should be organized so that memory consumption is minimized and the data structures are optimally prepared for accesses made by algorithms;
- **Data Access Pattern Techniques** that address the question how the index algorithms should be implemented so that their memory access behavior fits the processor architecture and, thus, yield optimal performance without stalls in the processing pipeline;
- **Parallelization Techniques** that address the question how index algorithms can be parallelized in order to gain performance speedups by collaborative processing of independent data elements or tasks, either on several homogenous processor cores or on different processor devices.

The main techniques that have been described in detail in Section 3.1.1 – Section 3.1.4 can be summarized under these categories as follows:

- **Data Structure Organization Techniques**:
  - **Pointer Compression** reduces overhead for linking tree nodes by linearizing them in memory so that node positions can be directly derived via offsets instead of following indirection via pointers [57, 61, 69, 113, 116, 120, 125, 141, 167, 168, 170, 171, 184, 202].
  - **Key Compression** removes redundancy from keys [21, 32, 73, 111, 112, 115, 120, 131]. Mostly lossless compression schemes are applied so that a compressed key can be fully reconstructed. Some indexes make use of a lossy schemes, which reduce memory consumption even further but just reconstruct the compressed key partially [201]. Therefore, applying a lossy compression reduces search space pruning accuracy and, thus, leads to redundant computations during tree traversals.
  - **Skew Adaption** is implemented by a few indexes to optimize the internal structure based on the data distribution [28, 120, 131]. That is, for sparse search space coverage, a different node implementation is used than for dense distributions. However, most index types dynamically adapt themselves to the data distribution already when the search space is clustered.
  - Using a **Structure of Arrays instead of a Array of Structures memory layout** is a fundamental technique to linearize node data in memory, which is required for vectorizing index algorithms [45,51,116,168,184]. Further, storing fields in separate arrays optimizes memory bandwidth utilization for fetching data that is required in different algorithm phases.
  - **Cache Line Blocking** recursively adapts the data structure layout so that it fits into the available cache hierarchy layers when the data is accessed by specific algorithms, effectively reducing thrashing overheads for the latter [111, 112, 125].
- **Data Access Pattern Techniques**:
  - **Cache-Sensitive Data Placement** adapts algorithms so that they operate locally on memory areas that are located near the processors. Compared to cache line blocking, which adapts the data structure so that it implicitly (via the processor hardware) fits the access patterns of specific algorithms, data placement strategies modify the algorithms (in software) on top of a given data structure so that random memory accesses outside cache regions are minimized. Cache-sensitive data placement strategies are particularly important when different processors cooperate on an index algorithm and need to exchange data, e. g., with NUMA across multiple CPU sockets [109, 130, 142] or memory accesses across the host $\leftrightarrow$ device boundary in hybrid CPU-dGPU systems [151].
  - **Data Access Latency Hiding** techniques minimizes waiting times for inevitable memory accesses across cache boundaries by overlapping the data transfer with computation on already cached elements. Modifying the algorithmic access patterns in software, e. g., by issuing explicit prefetch instructions, is needed if the latency hiding mechanisms implemented in processor hardware do not suffice. For

example, data-dependent random accesses, such as following a pointer, cannot be predicted by the hardware and, thus, need a "hint" in the software. CPU-based indexing algorithms are usually tuned with prefetching instructions to hide cache miss latencies [45, 51, 183]. GPU-based algorithms are using latency hiding techniques to overlap host $\leftrightarrow$ device transfers with concurrent computations [184]. On a GPU, however, hardly any existing index implementation utilizes on-device caches [125]. Instead, they rely on implicit latency hiding in GPU hardware that overlaps blocking transfers in vectorized algorithms.

- **Data Access Linearization** is an important technique for vectorized algorithms. An algorithm's memory accesses are rearranged so that consecutive element ranges are accessed, either sequentially in subsequent steps or via data-parallel instructions. Applying this technique facilitates implicit data access latency hiding and maximizes the bandwidth utilization when data is transferred in chunks / cache lines via the memory bus system. For tree-based indexes, this technique can be applied on node-level, when its entries are evaluated, or on tree-level, when the structure is traversed via one of the "stackless" traversal techniques [62, 71, 104, 111–113, 135, 140, 141, 179, 202].
- **Batch Processing** increases the locality of memory accesses for independent operations. By grouping them and applying them on the same data before continuing with the next algorithmic step, caching effects are shared, i. e., repeated data transfers for each operation via the bus system are avoided [18, 57, 61, 69, 71, 90, 124, 184, 187, 202, 206, 207, 211].

- **Parallelization Techniques**:
  - **Node-level Parallelization** techniques leverage independence on a tree node's entries in order to process them simultaneously by data-parallel instructions. For ordered domains, a vectorized binary search is commonly used [104, 111, 112, 179], while braided search, i. e., brute-force evaluation of all keys, is usually applied in unordered domains [61, 113, 203, 204].
  - **Tree-level Parallelization** techniques leverage independence within a tree's graph data structure. Parallel Breadth-First Search is usually applied to optimize tree traversals that follow multiple paths inside the tree, e. g., via range-based searches or when backtracking is needed [125, 135, 140, 141, 167, 202]. Parallel Depth-First Search algorithms are useful, when just a few paths are traversed. By speculatively evaluating future nodes, the algorithm's latency can be reduced [104, 111, 112, 114, 194]. Of course, there are hybrid versions that combine both strategies [206].
  - **Task-level Parallelization** techniques parallelize on independent operations that are, for example, provided as batches [18, 187, 211]. An important technique in this category is to minimize branch divergence among these operations, because it reduces data access locality [71, 207].
  - **Hybrid System** techniques consider the properties of different processor types in order to maximize an algorithm's performance when it is executed in a hybrid system [52, 69, 87, 113, 136, 184, 203, 204, 208, 208–210]. A hybrid algorithm must consider data placement and task scheduling between the available processors.

There is a lot of overlap between the implementation techniques that are applied for CPUs and GPUs, but their focus differs. While CPU implementations focus on cache optimizations, parallelization techniques and techniques for adapting the algorithms to the data-parallel processing model dominates research for GPU indexes. Recently, data parallelism also gained strong interest in CPU-based index implementations to leverage SIMD features. Many techniques, such as batch processing, latency hiding, and structure layout adaption, are applied by many implementations, independent from the underlying computing platform. However, these techniques are separately re-implemented for each index type. Thus, it makes sense to abstract and reuse them for multiple computing platforms via a common framework, which will be addressed in Chapter 4 and Chapter 5.

For hybrid systems, most state-of-the-art algorithms rely on static partitioning schemes for scheduling heterogeneous algorithmic workloads to the best-fitting processing device. Further, dynamic scheduling strategies exist that address load imbalances at runtime. However, no index implementation exists so far that dynamically schedules operations to the best-fitting processor, based on workload-dependent parameters. Compared to static scheduling strategies, which need to consider the heterogeneous processor architecture up-front for splitting the algorithm into heterogenous phases, i. e., during the design phase of the algorithm, the dynamic selection of the best-fitting processor device does not have this restriction. Applying a workload-dependent scheduling strategy allows to model algorithms uniformly, independent from the runtime-specific processor architecture. This is required to integrate hybrid system support into a common index framework, which is subject of Chapter 6.

3.2 Generalized Indexing

This section discusses related work targeting the development of index frameworks. As index framework, concepts and libraries as the implementation of the latter are considered, which generalize common aspects of index structures, such as organizing the data structure for indexing data of a specific domain, executing generic algorithms on an index, or their integration into the application code under a common interface.

Section 3.2.1 discusses frameworks that allow to model different indexes on a logical level, i. e., frameworks providing a common structure that can be extended to index data from different domains. A logical specialization can customize strategies for organizing the underlying search space and for pruning it when commonly used search algorithms are applied on the index. Section 3.2.2 discusses physical data organization aspects of existing frameworks, targeting the efficiency of the framework implementation as well as the usability of the framework in existing applications. Section 3.2.3 covers frameworks and extensions for modeling different algorithms on index structures and, finally, Section 3.2.4 discusses approaches to guarantee transactional properties of concurrent operations that access the same index instance.

Section 3.3 summarizes these ideas and discusses them with respect to the goal of this thesis, namely to provide an abstraction layer for supporting different processor types inside an index framework. Reusable aspects from prior art are highlighted as well as open tasks that are addressed in the following chapters.

3.2.1 Logical Structure Abstraction

The initial idea of a structure-independent index framework was presented by Hellerstein et al. [86]. The main idea of GiST is to abstract index structures and corresponding algorithms from the underlying *data type* of the elements that are used as search keys. An index is implemented as ADT (Abstract Data Type) that provides a common tree-based data model, including management algorithms for maintaining the structure upon insertion and deletion of records. All that an implementor of a specific index needs to do is to specialize the generalized tree by providing a concrete data type that shall be used as key as well as an implementation of some related operations that define certain properties of these keys. These key-specific properties are used internally to cluster data entries based on their spatial proximity in the overall search space and to bottom-up construct nodes that are linked to a hight-balanced search tree. Therefore, GiST implementations can be classified as "grow-and-post-trees" [138]. Other key operations are used to guide tree traversals when records shall be accessed that match a search condition.

To extend the applicability of GiST to other index classes that are also important for many applications, Aref and Ilyas proposed a similar framework named SP-GiST (Space Partitioning-Generalized Search Tree) [17]. SP-GiST adapts the generalization concepts that were introduced by GiST to model SPHs (Space Partitioning Hierarchies), i. e., tree structures that are not necessarily height-balanced, but recursively refine the search space until either the finest-granular regions cover a minimum number of indexed entries, or a maximum space resolution is reached. *Space-partitioning trees*, such as the k-d-Tree [25], the Quadtree [60], or the Octree [150], are widely used in applications that operate on geometry data, like computer graphics and CAD applications. Compared to the BVHs (Bounding Volume Hierarchies) that can be modeled by GiST, SPHs avoids overlapping search space regions and "dead" indexed search space without any entries by giving up height-balancing of the structure and, consequently, logarithmic algorithm complexity bounds. However, SPHs facilitate certain application-specific optimizations, like level-of-detail filtering. This technique prematurely stops tree traversals when refining the already reached space resolution in the current tree layer by another recursive descending step would not yield any improvements to the final result. For example, increasing the geometric resolution while rendering an image is not useful if it cannot be perceived by human application users anyway.

Another class of indexes that is not originally covered by the previously discussed GiST variants are *metric space indexes*. Unlike vector space indexes, which rely on discrete key values that can be combined to artificial keys for constructing recursive hierarchies, metric spaces indexes merely allow to calculate distances between key entries and organize them based on the resulting spatial proximity. For this class of indexes, which are relevant for multi-media databases and other applications that require very high-dimensional data entries, Chávez et al. described a generic framework on a rather logical level [44]. Many ideas are shared with GiST and its extensions: a common abstract tree-based data model is proposed as well as generic traversal strategies to implement distance-based search operations. There is a lot of similarity between these independently developed frameworks. In fact, Ciaccia et al. exemplified how the M-Tree [47], an important implementation of a metric space index, can be modeled in GiST.

Hadjieleftheriou et al. present SaIL (Spatial Index Library) [82], another generic framework for implementing spatial index structures that can be easily integrated into the application layer. SaIL complements GiST by applying object-oriented software design patterns [65] to reduce the implementation complexity beyond logical extension points. The pattern-oriented represents a simple way for developers to isolate specific implementation aspects from each other and exchange the underlying implementation without impacting the application that uses the framework.

A java-based index library has been published by van den Bercken et al. as part of XXL (eXtensible and fleXible Library) [27]. The authors claim that their approach is more generic than GiST, because it allows to model generic grow-and-post trees [138] and provides additional interfaces for bulk-loading the structures. However, the proposed methods are similar to those that GiST uses to abstract from the actual tree implementation. Besides some additional extensions that are not available in the original GiST publication, the main contribution of XXL lies in the integration of the entire framework into a higher-level library that also provides means to model database query processing functionality beyond index traversals.

An interesting statement made by the authors of XXL is that such commonly available libraries can be used to implement and benchmark various query processing algorithms to make them comparable with each other. This idea has been elaborated by Schäler and Köppen et al. in a project called QuEval [126, 178]. They propose a framework for integrating various index implementations under a common interface with the intention to perform comparable benchmarks, evaluate characteristics of different indexing strategies for different application-specific workload scenarios, and tune internal algorithms on these data structures. Such an evaluation framework is particularly useful, because, although a plethora of different indexing methods have been published so far, only very few comparative studies are available [64].

3.2.2 Physical Structure Abstraction

In addition to the logical abstraction layer for modeling indexing over different domains, GiST also provides abstractions for the physical storage of tree data. GiST implements a page-oriented model, using one page for one tree node, which is used as transfer unit for disk-based I/O. Like in classical indexes used in DBMSs, a buffer pool is provided as intermediate caching layer to avoid I/O latencies. Each node page comprises a set of records that represents node entries. GiST provides extension points for organizing these records on a page and provides two implementations, a generic one that stores keys in no particular order, and another one for ordered domains. Accessing the physical data can be customized, e. g., by applying binary search instead of linear search if keys are ordered, or by transparently injecting (de-)compression algorithms when decoding the data entries.

Another internal GiST implementation detail that has been addressed in literature is the efficient management of nodes that are buffered in-memory in order to utilize different caching levels in the memory hierarchy (cf. Section 3.1.1.1). Kim et al. proposed CC-GiST (Cache-Conscious

Generalized Search Tree) [117, 118], which incorporates the insights gained from research of cache conscious data structures, like the CSB$^+$-Tree [171]. The main ideas that are applied from specialized index structures to the generic framework are the array layout of node entries and pointer compression.

While the work on CC-GiST targets in-memory structures, a paper from Sarwat et al. addresses the next deeper layer in the memory hierarchy [176]. The authors describe how GiST algorithms can be modified to increase their efficiency for the characteristics of flash memory devices, like SSDs. The main contributions of the proposed Flash-Aware Search Tree are specialized flushing policies for main memory buffers that accumulate changes as well as an enhanced crash recovery technique that guarantees durability of transactions operating on the tree.

A generally available indexing library for C++ projects has been published as part of the Boost.Geometry libraries [59]. Although merely an R-Tree implementation is provided at the time of writing this thesis, the template-based framework provides convenient ways for customizing the existing implementation, e. g., by different query types or object clustering methods, and offers standard-conform interfaces for integrating it into application programs.

3.2.3 Abstraction of Index Algorithms

In the initial proposal of GiST, commonly used traversal operations were fully specified inside the framework to implement stateless search operations, i. e., queries that can be independently computed for each key in the index. A result set comprises all leaf entries whose key satisfies the query predicate. Point queries and range predicates can be modeled as stateless searches and were implemented in GiST by using a stack-based traversal algorithm.

However, there are other predicates, such as nearest neighbor searches, that evaluate the relation between multiple keys inside the index, e. g., a distance metric between them. In order to implement them efficiently, traversal operations need to maintain a state between recursive tree descending iterations. By using a branch-and-bound strategy, this state is updated with refined information that is available after processing a single tree node. Afterwards, candidates for the next step are pruned that cannot contribute to the result set any longer. In order to cover such more complex operations, Aoki published an extension for GiST, which also generalizes these *traversal* operations and allows custom specializations via priority queue-based algorithms [16].

The authors of SaIL [82] claim that their framework also allows to implement generalized traversal algorithms by using the visitor pattern. The index is treated as graph structure whose generic nodes types accept any algorithm as visitor, i. e., the algorithm is bound to the actual node implementation at runtime, which can decide how the node's entries shall be processed and which of its neighbors should be visited next. By maintaining a stack or a queue for managing the next nodes to be visited, any traversal strategy can be modeled.

3.2.4 Transaction Support

A rather internal but, nevertheless, very important index implementation aspect has been addressed by Kornacker et al., namely *concurrency control* mechanisms for concurrent operations on GiSTs [123]. The authors demonstrated how the generic GiST traversal operations can be extended by a locking protocol to guarantee logical isolation for multiple transactions that modify and query tree entries in parallel, while ensuring consistent views on the indexed data set. These algorithms have been enhanced by Chakrabarti and Mehrotra, who proposed to use a granular locking protocol instead of the initially proposed predicate locking approach, arguing that granular locking is much simpler to implement and causes lower processing overheads [41].

The work by Kornacker et al. [123] does also address the atomicity and durability properties of transactions operating on GiST. The authors describe a protocol that uses write-ahead logging, page-oriented redo, and logical undo operations to implement the *recovery* aspect, which is very important for applications that need to persist their index structures.

Another generic concurrency control protocol based on granular locking has been published by Lu et al. to address transactional isolation in clipping indexes [139], such as the R^+-Tree [182]. Unlike GiST, where key predicates are used to model BVHs whose hierarchically organized search space regions may overlap and each entry does exactly belong to one tree node, clipping indexes do not allow any overlap between nodes. Therefore, cases, where a single spatially extended key has been clipped to multiple node regions, must be handled separately.

The PALM-Tree [183] addresses concurrency by applying generic bulk-synchronous processing to index structures. That is, it redesigns the processing model so that all operations are executed in a series of batches, which are internally synchronized instead of assuming independent asynchronous operations that need to implement mutual exclusion mechanisms via locks. Each operation batch is processed in different phases, where compatible tasks are executed simultaneously in parallel and pre-defined synchronization barriers are used to join threads before the next phase begins. First, all operations within the batch are analyzed and read-only tasks are executed on the old index state. Second, conflicts between updates are resolved to derive an execution sequence that guarantees serializability. Finally, the reordered update sequence is applied before the next batch is handled.

3.2.5 Parallelization of Index Algorithms

While the previous index generalizations focus on sequential processing on CPUs, some of the techniques covered in Section 3.1 are applicable to generalize parallel processing on index structures on CPUs and / or GPUs.

For example, the parallelized binary search proposed by FAST [111, 112] and its variants can be generally applied to any processor type that support data-parallel SIMD instructions. Although the techniques are not applicable for general index structures and arbitrary traversal strategies, they can be employed as optimization for searching in ordered domain indexes on generic data-parallel processor architectures, such as modern CPUs, GPUs, APUs, or MIC devices.

Generalizable techniques for parallelizing any kind of tree traversal have been proposed by Goldfarb et al. [71]. In the context of a k-d-Tree, the authors discuss autoroping as generic "stackless" traversal technique for vectorizing recursive data structure processing on data-parallel platforms, without re-arranging the physical data structure layout as proposed by FAST. Upon node visitation, ropes, i. e., references to next to-be-visited nodes are managed in a shared data structure that is optimized for data-parallel access, e. g., by many threads on GPUs or other processor types that implement SIMD instructions.

Although parallelized tree traversals have been discussed for specific index implementations on specific processor types (cf. Section 3.1.1.3 and Section 3.1.2.3), the underlying parallel BFS and DFS algorithms are generally applicable to other indexes as well. However, no framework exists, yet, that incorporates this in a generic way for tree-based indexes.

For data-parallel batch processing of index queries, Goldfarb et al. propose an algorithm that can be applied to minimize branch divergence within lockstep traversals [71]. Based on an application-specific penalty metric that indicates divergence between queries in a batch, queries are sorted and regrouped so that divergence is minimized. This can be regarded as generalized branch divergence minimization technique, which has been independently proposed for special index structures [51, 136, 207].

Parallel batch processing of index operations is also in the focus of the PALM-Tree [183]. As discussed previously, the bulk-synchronous processing model addresses concurrency issues when mutating algorithms are applied in conjunction with read operations on a shared index structure. Although the PALM-Tree just provides a B^+-Tree implementation, the proposed technique should be generally applicable to tree-based indexes.

3.3 Discussion – Generalizing "Processing" in Generalized Index Structures

Ideally, these processing aspects would also be generalized inside a common framework so that all index implementations benefit from the underlying techniques. In their vision paper [43], Chavan et al. published a similar idea in a more general context. The authors propose a system architecture where spatial operations and (spatial) indexes, as one important building block for implementing them, are scheduled to a hybrid CPU/GPU processing environment, picking the best device for each task based on a cost model for the latter. However, as Table 3.1 shows, existing index frameworks and specialized index implementations do not achieve this goal, yet.

Table 3.1: Index Frameworks and Extensions

Index Framework	Search Key Types	Search Space Partitioning	Traversal Operations	Transaction Support	Memory Management	Parallelization	Coprocessing
XXL [27]	G	BVH	G	–	G	–	–
SaIL [82]	G	BVH	G	–	G	–	–
GiST [86]	G	BVH	X	–	G	–	–
GiST Searches [16]	(G)	BVH	G	–	(G)	–	–
GiST Transactions [123]	(G)	BVH	(G)	X	(G)	–	–
Flash-Aware GiST [176]	(G)	BVH	(G)	X	X	–	–
CC-GiST [117, 118]	(G)	BVH	(G)	(X)	X	–	–
SP-GiST [17]	G	SPH	–	X	G	–	–
Metric Trees [44, 47]	G	Metric	(G)	(X)	(G)	–	–
GPU Traversals [71]	X	X	G	–	X	G	–
PALM-Tree [183]	X	X	X	X	X	X	–
Specialized Indexes [Section 3.1]	X	X	X	X	X	X	(X)
This Thesis [22, 37, 38, 177]	(G)	(BVH)	(G)	(X)	(G)	G	G
	Logical Aspects			**Implementation Aspects**			

The columns in table Table 3.1 denote different implementation aspects of an index. They are separated into two categories that indicate whether the aspect rather applies to a logical level, i. e., for modeling different index structure types and search operations that utilize search space pruning, or to a physical level, i. e., an internal implementation aspects of data structures and algorithms.

The following values are used inside the cells:

–:	the index (framework) does not consider this aspect
X:	the index (framework) provides a specific implementation without extension points
G:	the index (framework) generalizes this aspect and offers means to extend it for customizing the behavior in a specific implementation
(G):	the index framework inherits this generalized aspect from an underlying base framework that is extended
BVH:	the framework models Bounding Volume Hierarchy indexes over vector spaces
SPH:	the framework models Space Partitioning Hierarchy indexes over vector spaces
metric:	the framework models metric space indexes that can either be implemented as distance-based BVHs or SPHs

SaIL, XXL, and GiST represent base frameworks that mainly address *logical aspects* of BVHs. They generalize ways to index arbitrary data types by defining customizable clustering strategies for building a hierarchy over all entries in the search space. Search queries can be customized by user-defined strategies for search space pruning. Additionally, these frameworks provide extension points to manage the internal data structure layout of tree nodes. GiST is the most prominent framework that has been used as base for several extensions, e. g., for adding transaction support [123], integrating cache-conscious memory layouts by applying pointer and key compression techniques [117, 118], or supporting flash storage devices in the buffer hierarchy [176]. Further, the core ideas of GiST have been reused to address indexing in other search space categories, such as SPHs [17] and metric spaces [44, 47].

Parallelization and other techniques for enhancing search performance in index structures has been addressed by many specialized indexes that were tuned for a specific processor architecture (cf. Section 3.1). But none of them applies these techniques in a generalized way so that other (logical) indexes can be modeled without re-implementing everything from scratch. Further, hardly any specialized implementation leverages co-processing on hybrid CPU/GPU systems. Most indexes either use the CPU or the GPU as sole processing unit or just statically partition the workload between both of them.

However, some techniques that have been proposed for modern CPUs or GPUs are generally applicable. The "stackless" traversal technique, for example, as well as the optimization of independent query batches for data-parallel index traversals can be applied to any processor hardware that supports SIMD instructions. The PALM-Tree [183] also represents a specialized index structure that addresses transaction processing for parallelized operations by applying the bulk-synchronous processing model. Although all these techniques are independent from the logical index type, a generalized implementation does not exist, yet.

The gap between generalized index frameworks and parallelized implementations is addressed in the following. Generalized indexing techniques are combined with techniques from specialized indexes in order to parallelize generic indexing algorithms for CPUs and GPUs. Further, the processing layer is abstracted so that processing on a hybrid system is supported. Neither of the existing frameworks supports such a setup and hardly any specialized implementation leverages co-processing on hybrid CPU/GPU systems. Most indexes either use the CPU or the GPU as sole processing unit or just statically partition the workload between both of them.

Chapter 4

Generalized Processing in Generalized Index Structures

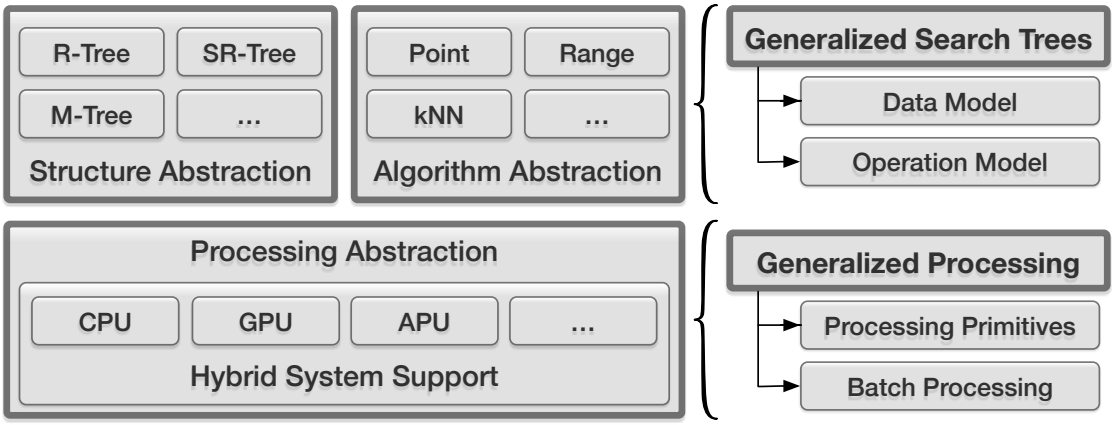


Figure 4.1: Overview of the Index Processing Abstraction Layer

In order to support the implementation of various index variants whose characteristics meet the requirements of applications that operate on multi-dimensional data sets, a generic index model needs to be provided, which allows to customize clustering and search space pruning behavior so that arbitrary logical indexes can be implemented. In this thesis, the GiST framework is reused as base implementation in order to inherit all generic extensions that already exist, such as the generalization of index structures and corresponding search algorithms.

GiST's, data and processing model are briefly summarized in Section 4.1, before its processing-related components are abstracted in Section 4.2 and generic traversal operations are mapped to processing primitives that can be scheduled to arbitrary processing devices. Further, a batch processing interface is introduced in order to increase the degree of parallelism that can be leveraged on massively parallel architectures, like GPUs. Based on these concepts, a processing abstraction layer is developed as GiST extension in Section 4.3. The general applicability of the extended framework is evaluated in Section 4.4. The GiST library is integrated into two computer graphics applications, whose search operations are profiled in order to assess potential benefits that can be gained via a parallel implementation.

4.1 Generalized Search Trees

The work presented in the following is based on GiST [86], because this framework already satisfies the main requirements for modeling the most commonly used index structures in a generic way. This section, therefore, briefly summarizes the key ideas and logical concepts behind it, highlights which of them are reused, and discusses assumptions that are made when developing the generalized processing extension.

The idea for GiST was born when a lot of different strategies were evaluated in order to find the best methods for indexing (multi-dimensional) datasets having different data distributions. Height-balanced tree structures represent one of the most important classes of indexes that have been tested. Different clustering strategies were implemented for them to derive hierarchical structures over the available keys in the underlying vector space. One observation that was made is that all these structures share a lot of code for maintaining, i. e., growing and shrinking, the trees upon insertion and deletion of records and, finally, searching entries inside them that satisfy certain query criteria. Therefore, GiST was proposed as framework for modeling such trees in a generic way, implementing common operations inside the framework, and providing a minimal set of interface methods that depend on the actual key structures used for indexing. In order to model a specific structure extension, such as a B-Tree or an R-Tree, all that an index developer needs to do is to specify these key methods. The rationale behind this idea is that, by using this framework as common code base, a lot of implementation and software testing efforts can be saved, which effectively increases the velocity with whom new indexing strategies can be implemented and evaluated.

The logical data model of GiSTs and their different node types is discussed in Section 4.1.1. Generic algorithms that can be specialized in order to customize query processing on and maintenance of a generalized search tree are subject of Section 4.1.2. Finally, the high-level architecture of the GiST implementation, which acts as the base of the extended framework, is briefly discussed in Section 4.1.3.

4.1.1 Tree Data Model

The conceptual data model of the GiST framework is illustrated in Figure 4.2 (cf. [86]). GiST allows to implement a height-balanced tree structure that can be constructed using the key methods that are described in the next section. The tree data structure is modeled via different *nodes* having multiple *entries*. Each entry consists of an abstract *key* predicate that describes all underlying entries by a logical clustering property, such as a distance, a partition in the logical search space, or anything else that can be used to model a search tree. In addition, each entry has a *pointer / reference* to the next structure in the hierarchy, which might either be a data record, e. g., a row number, a geometric shape, or any other data value encoding the record that has been indexed in the underlying key space, or another node in the search tree that, again, comprises a set of key entries. Like introduced in the original GiST paper, the term *"leaf node"* is used for nodes of the former case, "*data entry*" for their entries, and *"inner node" / "directory entry"* for the latter.

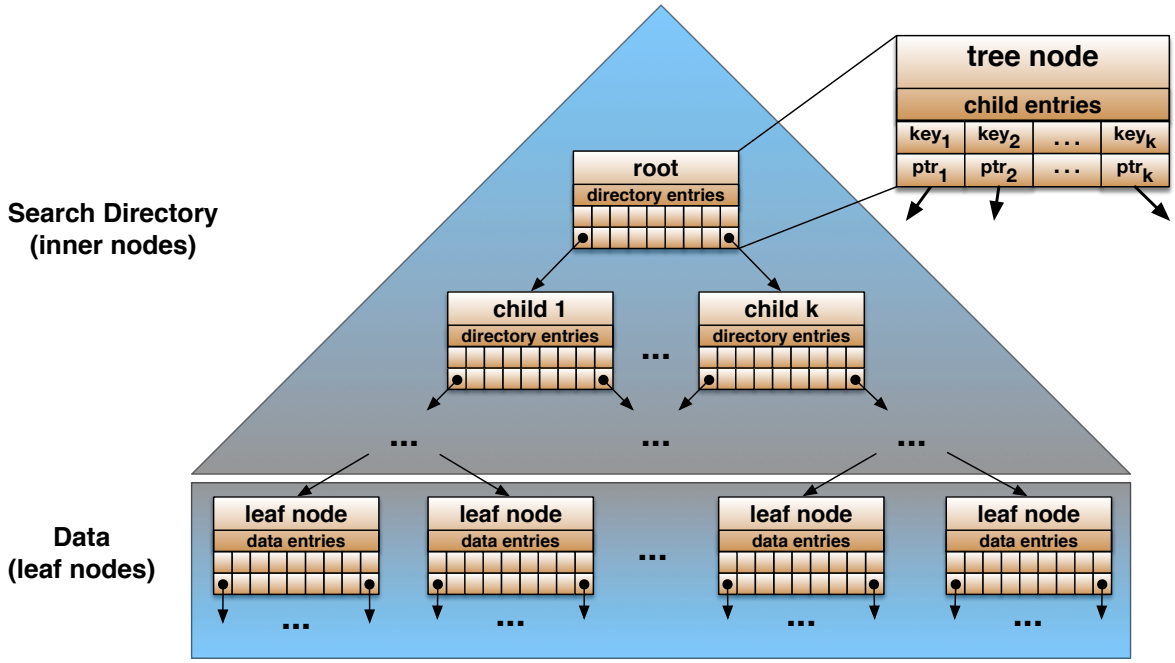


Figure 4.2: Logical Data Model of a Generalized Search Tree

For simplicity, and without loss of generality, it is assumed in the following that those node types are not mixed, i. e., inner nodes do not store any data entries while leaf nodes do not have any directory entries. A mixed node can always be modeled as an inner node having a special direct link to a leaf node that comprises all data entries. However, this would violate the height-balancing property.

The framework extension presented in the following does neither depend on height-balancing nor on a strict tree structure in the mathematical sense. The following algorithms rely on a less restrictive DAG (Directed Acyclic Graph) structure instead and can be applied to recursive index graphs, irrespective of their nesting depth. However, the term *"tree"* will still be used throughout the text in order to use a consistent language with the GiST framework that represents the foundation of this work.

4.1.2 Tree Operations

Table 4.1 lists the most important methods of GiST, which are relevant for this thesis. Methods in regular font represent public interfaces that can be used by applications, which integrate a customized generalized search tree. Customization can be done by specializing abstract *key* methods, which are rendered in bold font in Table 4.1. This thesis focuses on query operations. Corresponding methods for processing stateless and stateful search operations are discussed in Section 4.1.2.1. Operations for maintaining a generalized tree structure are subject of Section 4.1.2.2. Although they are not parallelized in the scope of this thesis and will simply be reused, understanding them is crucial to understand how generalized trees work. Initial ideas how these maintenance operations can be parallelized will be given in the thesis' outlook. All internal key methods, which need to be specialized by developers to customize GiST are discussed in Section 4.1.2.3.

	Operation	Description
Query Operations	`search(N, Q)`	recursively find all entries in node `N` whose predicate is `consistent()` with query `Q`
	`consistent(P, Q)`	predicate evaluation function, returns `true` if the search space covered by predicate `P` might cover an entry that satisfies query predicate `Q`
	`stateInit(N, Q)`	initialize the traversal state for search query `Q` that is applied to the index rooted at node `N`
	`stateConsistent(S, N, Q)`	stateful predicate evaluation, returning all entries of node `N` that are `consistent()` with query `Q` based on the current traversal state `S`
	`stateIter(S, P)`	update traversal state `S` with new search space predicate `P`
	`stateFinal(S)`	finish traversal and compute the final result of the aggregated state `S`
Tree Maintenance	`insert(N, E)`	insert a new entry `E` into the index rooted at node `N`
	`remove(N, Q)`	remove all entries from the index rooted at node `N` that satisfy query predicate `Q`
	`penalty(P1, P2)`	penalty metric indicating the increase of search space coverage if both predicates `P1` and `P2` would be combined to a single predicate via `union()`
	`union(PS)`	combine a set of search space predicates `PS` into single predicate that covers the entire search space of all predicates $P \in PS$
	`pickSplit(ES, k)`	split a set of node entries `ES` into two disjoint subsets of minimum fill factor `k`, usually minimizing a `penalty()` over the `union()` of their predicates

Table 4.1: Generalized Search Tree Operations
Source: [16, 86]

4.1.2.1 Query Operations

Algorithm 4.1 GiST Stateless Search

Input: treeNode, queryPredicate, resultSet
Output: resultSet with all indexed entries that satisfy the queryPredicate

```
1: for (keyPredicate, ptr) ∈ treeNode.entries do
2:   if consistent( keyPredicate, queryPredicate ) then
3:     if treeNode.isLeafNode() then
4:       resultSet.add( keyPredicate, ptr )
5:     else
6:       search( getNode( ptr ), queryPredicate, resultSet )
7:     end if
8:   end if
9: end for
```

Stateless queries on a GiST are answered by recursively traversing the tree and evaluating the query's predicate for all key entries of nodes that are visited (cf. Algorithm 4.1).[1] If such a predicate evaluation via the generic `consistent()` key method yields `false` for a specific key, it is guaranteed that no data entry can be found under the sub-tree rooted at the pointer of the entry that satisfies the query predicate. If a predicate evaluation yields `true` for the key, a matching entry might (but not necessarily have to) be found under the corresponding sub-tree. Matching sub-trees are descended recursively, following the same strategy.

In case of inner nodes, this approach guides the "direction" of the search inside the indexed key space and prunes the search space with each tree layer that is processed. Because key predicates are allowed to yield false positive results, i. e., `true` when no matching entry can be found, it is possible to represent all entries of a sub-tree using a coarse-granular predicate that also covers "dead space" but can be represented and evaluated more efficiently, e. g., a bounding box used as key in an R-Tree that simplifies complex geometries.

In case of leaf nodes, matching entries will be added to the query's result set, all others are ignored. The final predicate evaluation in leaf nodes might involve more complex algorithms, compared to those that are applied in inner nodes to calculate the result, e. g., checking intersections of a complex geometry instead of just evaluating intersection tests on bounding boxes in inner nodes. However, due to the search space pruning during directory traversal, the index limits such expensive evaluations to those entries that likely satisfy the query.

[1] In the original GiST paper [86], a special case is discussed for ordered key domains, e. g., when B-Trees are modeled using the framework. In this case, it is not necessary to evaluate all keys inside a node if state-of-the-art algorithms are applied that are well-known from the B-Tree index family, e. g., binary or interpolation search. However, this special case is not examined further in this thesis. Instead, the general case, where all keys inside a node have to be evaluated, is of more interest here. The fact that all tree links are followed whose predicate satisfy a query, is utilized later for implementing a parallel traversal strategy.

Algorithm 4.2 GiST Stateful Search

Input: treeNode, queryPredicate
Output: aggregated result of the stateful traversal

```
1:  state = stateInit( treeNode, queryPredicate )
2:  queue = PriorityQueue()
3:  queue.push( treeNode )
4:  while not queue.empty() do
5:      currentNode = queue.pop()
6:      for (keyPredicate, nextNode) ∈ stateConsistent(state, currentNode, queryPredicate) do
7:          stateIter( state, keyPredicate )
8:          queue.push( nextNode )
9:      end for
10: end while
11: return stateFinal( state )
```

Search space pruning of stateless queries, such as point or range queries, can be executed independently for each tree node by simply checking the query predicate on each key predicate and returning consistent entries. In contrast, the evaluation of stateful queries, such as nearest neighbor searches, depends on the traversal history and yields a more generic result, which could be a set of indexed entries, like in the stateless case, or any other generic aggregate that is updated with each traversal step, like a counter.

Algorithm 4.2 illustrates the high-level stateful search algorithm that was proposed by Aoki as extension for generalized search trees [16]. It implements an iterative "stackless" traversal, using a queue to manage nodes that still have to processed. The queue might be customized with a priority function that can be used to specify the order in which candidate nodes are extracted, e. g., returning the node having the minimum distance to the query point among all remaining candidates in case of nearest neighbor searches. The temporary query result is aggregated in a generic state variable that is initialized upon query start, updated with each consistent entry during the traversal, and finalized once all nodes have been processed. Compared to its `consistent()` counterpart in the stateless version, the `stateConsistent()` key method may, but doesn't have to, consider the current state for search space pruning. Thus, any stateless search may also be modeled as stateful algorithm, which simply ignores the state. The state type and all related functions need to be specified as key methods by the actual GiST implementation.

4.1.2.2 Tree Maintenance Operations

Tree maintenance operations upon insertion and deletion of records are implemented straightforward in the GiST framework, using methods that are state-of-the-art in other height-balanced search trees, such as B-Trees or R-Trees. A simplified algorithm is sketched in Algorithm 4.3, which focuses on the conceptual steps and ignores implementation-specific subtleties from the original GiST publications, e. g., for prematurely stopping traversals and implementing concurrency control mechanisms.

Algorithm 4.3 GiST Insertion

Input: treeNode, newEntry
Output: a new node in case treeNode has been split

```
1:  if treeNode.isLeafNode() then
2:    entryToBeInserted = newEntry
3:  else
4:    find (childKey, childNode) ∈ treeNode.entries,
            where penalty( childKey, newEntry.key ) is minimal
5:    newChildNode = insert( childNode, newEntry )
6:    childKey = union( childNode.predicates )
7:    if newChildNode ≠ none then
8:      entryToBeInserted = (union( newChildNode.predicates ), newChildNode)
9:    else
10:     entryToBeInserted = none
11:   end if
12: end if
13: if entryToBeInserted ≠ none then
14:   if treeNode.numEntries() < MAX_FILL_FACTOR then
15:     treeNode.entries.insert( entryToBeInserted )
16:     newNodeAfterSplit = none
17:   else
18:     (entries1, entries2) = pickSplit( {treeNode.entries, newEntry}, MIN_FILL_FACTOR )
19:     treeNode.entries = entries1
20:     newNodeAfterSplit = createNode( entries2 )
21:   end if
22: end if
23: if newNodeAfterSplit ≠ none and treeNode.isRootNode() then
24:   entry1 = (union( treeNode.predicates ), treeNode)
25:   entry2 = (union( newNodeAfterSplit.predicates ), newNodeAfterSplit)
26:   newRootNode = createNode( {entry1, entry2} )
27:   return newRootNode;
28: else
29:   return newNodeAfterSplit
30: end if
```

When a new $(key, pointer)$-entry shall be inserted into a GiST, the tree is recursively traversed to find the "best" leaf node to accommodate it (lines 1–12). In order to determine the best fit, a key `penalty()` metric (cf. Section 4.1.2.3) is evaluated for all sub-tree keys that are processed during the traversal (line 4). Within each inner node, the subtree having the minimum insertion penalty is selected, i. e., the entry whose predicate already covers the new key entirely or needs the least increase of the covered key space compared to the other candidates. Insertion continues in this path (line 5), installs the entry in the child layer, and updates the extended search space predicate so that the new entry will be found by subsequent search operations (line 6). This extended predicate is calculated using `union()`, another key-specific method. In case the selected child node overflows during the insertion, i. e., the implementation-specific fill rate in number of entries is exceeded, it has to be split into two nodes, the bounding predicates have to be calculated, and need to be installed in the current (parent) node (lines 7–11).

The actual insertion of an entry into a node is common between a leaf node and an inner nodes whose child was split (lines 13–22). If the node can accommodate the new entry, it is simply inserted into the entry set (lines 14–17). The node's parent will care about updating the bounding predicate (line 6). If the node overflows, the entry set is split into two disjoint pieces using the `pickSplit()` key method (line 18). One of them replaces the entries of the current node (line 19), while the other one will be moved to a newly allocated node (line 20) that needs to be propagated to the parent layer (lines 23–30). In case the root node was split, no matter if it was a leaf or an inner node, there is no parent layer that can accommodate the new entry. Thus, a new root node needs to be created that replaces the old one. The new root node has two entries, one for the former root node and one for the newly allocated node from the split (lines 24–27).

Entries to be deleted are searched using the previously described query processing interface. The entries are removed from the leaf layer and key predicates on the traversal path are bottom-up adjusted, like in the insertion case. In case a node underflows, i. e., the number of entries falls below a minimum fill, this can be handled with multiple strategies that are well-known from other trees, such as ignoring the underflow for filling the node with new entries eventually, merging it with neighbor nodes, purging the node and reinserting its remaining entries, etc. All of these strategies try to minimize the amount of dead space that is covered by the remaining key predicates in the tree. But they are not relevant for this thesis.

In the following, it is assumed that an index can be built and maintained by using these methods. Therefore, those and other maintenance methods, like bulk loading an index, will not be examined further. Instead, the focus will clearly be on parallelizing query processing. Another assumption that is made in the following is that concurrent transactions are correctly handled by implementing one of the transaction protocols that have been proposed for GiST (cf. Section 3.2.4). That is, whenever a node is visited, i. e., its keys are processed by a traversal operation like a query, it is assumed that the node is properly latched as long as it is handled by the executing thread in order to prevent concurrent modifications. Further, it is also assumed that traversed nodes and their keys are also properly locked logically in order to prevent anomalies caused by concurrent read and write accesses.

4.1.2.3 Index Specialization

In order to specialize GiST for implementing a specific index structure, a developer needs to provide the key data structures over which the index is defined. Further a set of interface methods needs to be overridden that are used for implementing the previously discussed tree operations. Some key methods are mandatory, some are optional, having a default implementation.

The most important methods are *predicates*, which are defined for *keys*, i. e., elements of the abstract search space, that shall be indexed. Elements and predicates exist independently from an actual index implementation and merely define the index domain in a mathematical sense. When evaluated on a set of elements, the boolean predicate functions describe traits that all these elements have in common, returning `true` if all elements satisfy the predicate, `false` otherwise. A GiST implementation uses predicates for two purposes. First, as *key predicates* for hierarchically clustering index entries via tree nodes. Second, as *query predicates* for describing all entries that shall be returned by a search operation or that shall be deleted from the index.

Predicates are evaluated by the framework during query processing via the `consistent( key_pred, qry_pred )` function. The first predicate parameter is constructed from the key data structure that is stored within a node entry once it is evaluated by the traversal algorithm. The second predicate is provided as user input when a query is triggered.[2] As discussed in Section 4.1.2.1, `consistent()` returns `false` if none of the entries described by the key predicate satisfies the query predicate. Otherwise, i. e., at least one entry might exist for which the query predicate is satisfied, `true` is returned. In other words, specifying `consistent()` specifies the search space pruning strategy that shall be implemented by the index. A GiST extension needs to handle the combination of predicates via the `consistent()` method. If none is available, a default implementation will be used that always returns `true`.

In stateful traversals, predicate evaluations may also consider the current traversal state via the `stateConsistent( state, key_pred, qry_pred )` key method, which is passed as additional argument. The other state-dependent methods are self-explaining: `stateInit()` initializes the state variable on query start, `stateIter( state, consistentEntry )` updates it with the next consistent entry, and `stateFinal()` generates the query result after the query traversal finished.

The remaining key methods are required for managing node structures inside the tree. The `penalty( new_key, existing_key )` method defines a domain-specific metric that is evaluated when a new entry key shall be installed inside a node that is represented by another, already existing key. The numeric return value represents an abstract measure of how much the existing key predicate needs to be increased in order to evaluate to `true` for all the entries. The predicate extension is calculated via a `union( predicate_set )` key method that also needs to be overridden by the index developer. As return value, the function yields a single predicate that satisfies all ones, which have been provided as input.[3]

[2] Note that the same predicate type can be used for both, key and query predicates.

[3] Actually, the union predicate must just yield `true` for all *entries* that are reachable under the sub-trees that belong to the input predicate set. That is, it doesn't necessarily have to be "bounding" all input predicates if they cover dead space. For details, refer to the original GiST paper [86].

Once a node overflows upon insertion of a new entry into its predicate set, it needs to be split into two nodes and the tree structure grows. The partitioning of an overflowing predicate set is calculated via a user-defined `pickSplit( predicate_set, fill_factor )` key method. This method solves the optimization problem of finding a good, ideally the best, partitioning of this set by using the `penalty()` function. Overriding it specifies the index' clustering strategy that is applied by the framework to structure the search space. The additional fill factor argument guarantees that each result partition comprises a minimum number of keys. This is used to balance the number of entries and limit the number of splits asymptotically.

Finally, for the sake of completeness, GiST provides optional `compress()`, `decompress()` key methods that are invoked at (de-)serialization points when a node's keys are accessed. As their names suggest, they apply key (de-)compression schemes on a set of entries to optimize a tree node's storage utilization. But those methods are not relevant in the following.

The concepts and algorithms that are developed in this thesis focus on the predicate evaluation part, i. e., the key types are relevant, the predicates that are defined for them, as well as the `consistent()` method, which represents the main work that is done during query processing. Tree maintenance is ignored for now. In case maintenance methods shall be parallelized too, the `pickSplit()` method should be examined first. It solves an optimization algorithm that, depending on the implemented strategy, needs to evaluate and compare multiple split partitions of a key set. Most of the insertion work is likely performed inside this method, which may benefit from parallel execution.

4.1.3 GiST Architecture

Figure 4.3 provides a high-level overview of the `libgist` library [2] that implements the GiST framework's concepts and acts as base for the following extension.

The index data model is implemented in the `GiST File`, `Page`, and `Record` classes. A whole GiST tree is mapped to a file, which manages its nodes on one page each and stores index data persistently on disk. Accessed nodes are cached in-memory via a buffer pool implementation. Each page stores some meta data about the node and a set of records as raw byte buffers that physically represent the node's entries.

The raw data is logically interpreted by the framework via two data abstraction layers. The first layer abstracts physical storage and access of entries inside node pages. The generic `unordered` layout is relevant for this thesis as it represents the most general implementation, where entries are stored without any particular order, because such an ordering may not exist for an indexed domain. In case an ordering criterion exists that can be leveraged to arrange entries inside tree nodes and employ binary search instead of braided element access, an `ordered` node layout can be used as an optimization thereof. The second data abstraction layer abstracts logical properties of data elements, which are required for implementing the index. The unordered implementation uses the key methods that have been discussed in Section 4.1.2.3 and

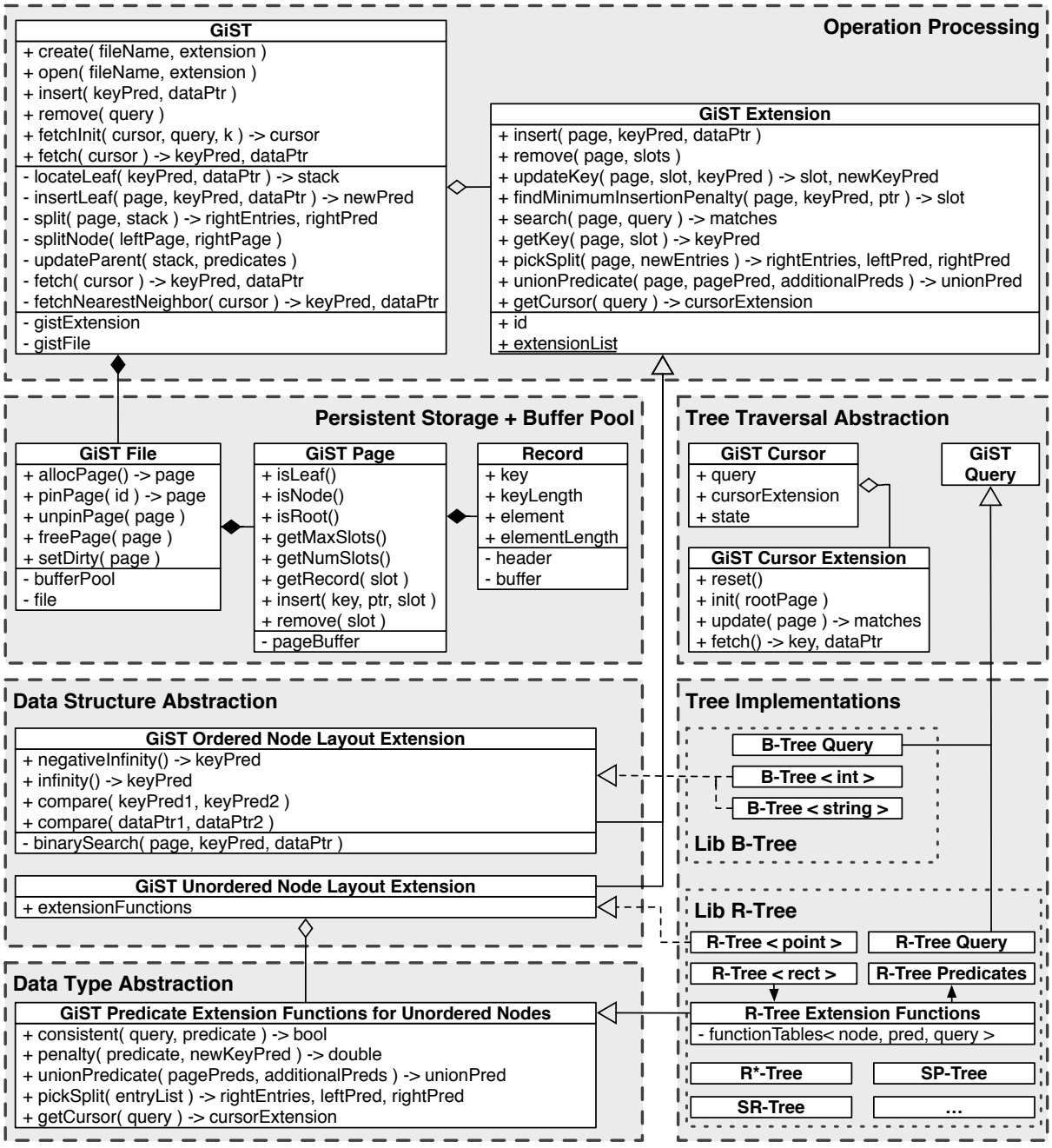


Figure 4.3: GiST Architecture Overview

are provided as global function table in `libgist`. The ordered node extension does not need them to implement the logic of B-Tree family indexes. It just needs to provide a comparison function that defines the ordering of elements as well as special values for representing infinity in open intervals.

The actual indexing logic is implemented on top of the data abstraction layer inside the `GiST` and `GiST Extension` classes. The `GiST` class acts as entry point for the public GiST interface methods that can be integrated into application code. Further, it provides common tree functionality that has been sketched in the previous sections and can be implemented generically for all GiST indexes. Whenever data access is needed by one of these core methods, the `GiST Extension` is used, which breaks down implementation complexity to the aforementioned set of key methods.

Tree traversals are abstracted by `Cursor` implementations. Cursors are updated with each tree node / page and may maintain a traversal state for stateful query variants. The set of nodes that still need to be visited is hard-coded inside the generic tree core functions. Depending on the search strategy, each function either use a stack or a priority queue to manage unprocessed candidate nodes.

The `libgist` library already provides some pre-defined data types as well as GiST extensions for commonly used index structures, such as a B-Tree over integral and character string values as `ordered` variant or various R-Tree implementations for `unordered` domains. The implementation of such a specific GiST instance boils down to the definition of function tables for the extension functions and some type definitions that are plugged together into a new index type.

4.2 Processing Abstraction in Generalized Search Trees

GiST already represents a solid base for modeling search trees on a logical level and implementing many index variants using a unified framework library. However, its original implementation in *libgist* [1, 2] has been designed for single-threaded operations, were each query / insertion / deletion is carried out asynchronously by an independent CPU thread. Each thread maintains its own execution state and may synchronize with others via latching and locking on the tree's data structures.[4] Parallel execution on other processor types are not supported. This is addressed in the following chapters.

The first step that needs to be done is to abstract processing-related components so that they can be mapped to arbitrary execution hardware. Therefore, the data structure layout as well as the processing model of GiST needs to be adapted in order to model tree traversal operations as parallel primitives that can be mapped to modern (co-)processor devices later on. A high-level overview of the modified framework architecture is illustrated in Figure 4.4. The components on the left hand side represent existing GiST modules that can be reused almost unchanged. New or strongly modified components that were introduced as GiST extension in this thesis are shown on the right. Structure-related modifications are discussed in more detail in Section 4.2.1 and algorithmic adaptions are covered in Section 4.2.2.

[4] The publicly available *libgist* library [2] does not support concurrent operations. However, the concepts for addressing concurrency have been proposed in [123].

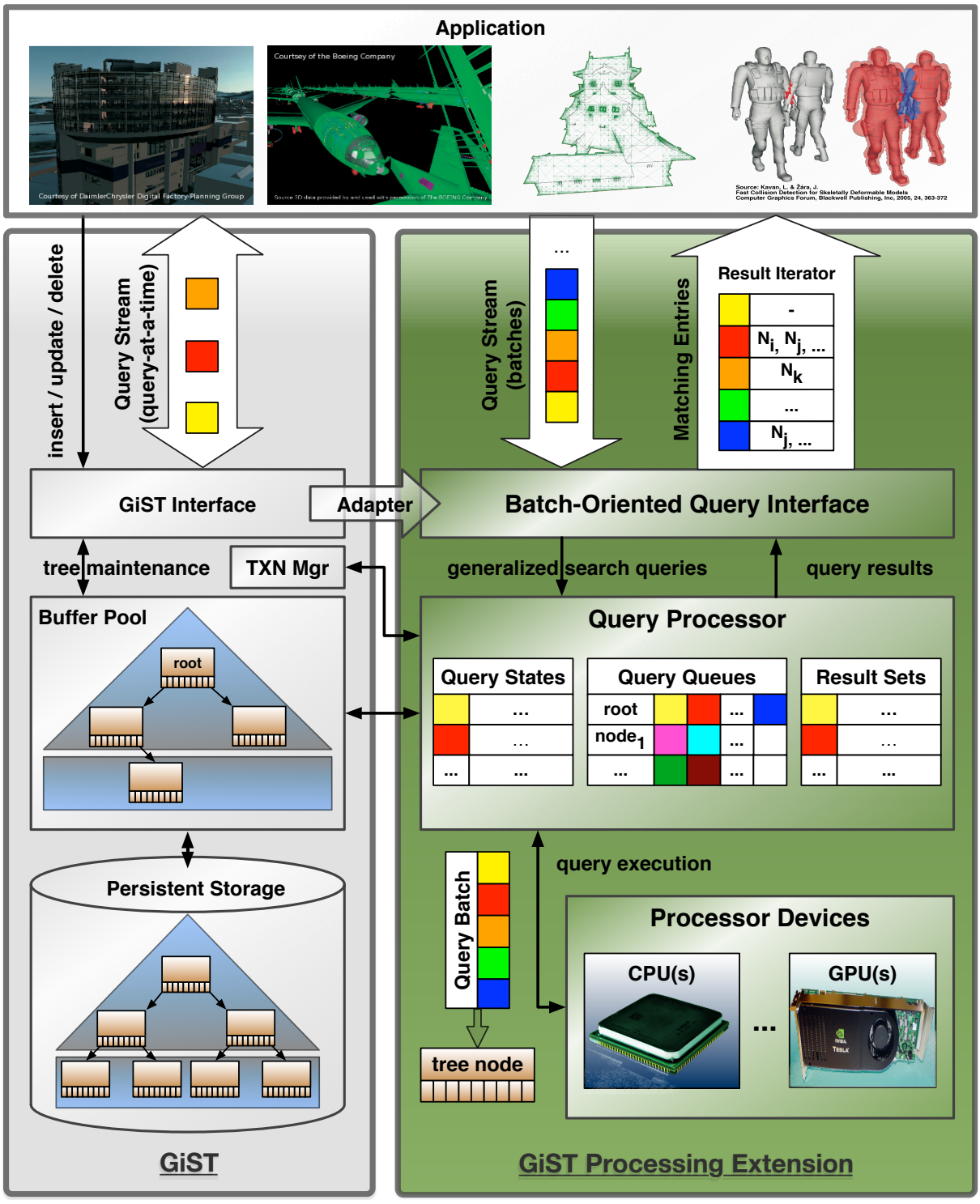


Figure 4.4: Extended Generalized Search Tree Framework

4.2.1 Index Structure Modifications

While entries inside a GiST node represent *(key, pointer)*-pairs from a logical perspective, they do not need to be physically stored next to each other in a contiguous memory chunk, like in libgist. Instead, a *SoA layout* is implemented in the extended framework, which is comparable to the data structures in CC-GiST [117] or other cache-conscious indexes implementations. Like illustrated in Figure 4.2, keys and pointers are stored in a separate contiguous array each, which mainly serves two purposes.

First, it simplifies data transfers between the main memory of a host system and the main memory of a coprocessor device that is attached via a dedicated bus. Using an array layout just requires a single bulk copy transaction without the need to gather all entries from scattered locations or copying a whole node, including irrelevant information for evaluating queries. For the algorithms that will be discussed in Chapter 5, only keys are required for evaluating search predicates, while pointer arrays are just needed for calculating traversal steps on the host system after synchronizing query states. Separating key and pointer arrays, therefore, minimizes data transfer overheads between host and devices.

Second, an array storage is required for processing elements with low-level parallel primitives that process a single instruction on multiple elements (SIMD), using element offsets to access the actual data. SIMD instructions on a CPU work this way as well as the parallel kernel programs that are generated for GPU processors (cf. Chapter 2).

Additionally, these arrays also implement a cache-conscious layout on the host system if they are properly aligned in main memory. A node's keys, for example, will be evaluated in bulk. Thus, accessing a single element in the contiguous array already transfers the following ones to upper cache layers, effectively reducing access latencies for the next elements within the same cache line.

For *node management* within the tree, there are no special requirements for the following algorithms. They just have to be accessible via a single in-memory address for transferring them to coprocessor devices. GiST manages its tree nodes in buffer pools that are mapped to persistent files on disk. This is orthogonal to the following concepts and, thus, the existing implementation can be reused. However, disk I/O operations will not be considered during the evaluations.

4.2.2 Index Operation Modifications

The most significant change in the algorithmic design of the extended GiST framework is that a *batch processing* mode has been introduced for tree traversal operations, which is required to increase the degree of potential parallelism that can be utilized by parallel (co-)processors. That is, applications schedule a set of queries to the index that are considered to be independent and, thus, can be processed in parallel. Of course, scheduling batches of size one is also possible to mimic the original *libgist* implementation that processes each query separately. But, as later experiments will show, this strongly limits parallelization effects.

Internally, *libgist* uses a separate stack or queue for each thread in order to manage all to-be-visited nodes while the tree is traversed. Each thread is also responsible for maintaining its private query state to implement stateful queries, like nearest neighbor searches. This has been modified in the framework extension in order to decouple traversal state management from the executing thread and to vectorize query processing. A central query processor manages the entire traversal state for all active traversal operations in all batches, which allows to let multiple query threads collaborate, no matter whether they have been scheduled to a CPU, a GPU or any other (co-)processing unit. Whenever a batch of queries has been processed for a node, the shared state will be updated and traversal continues with the next candidate node(s). A shared queue is responsible for tracking active queries per node, which, unlike the mapping of nodes to queries that is implemented by *libgist*, facilitates batch query processing. Finishing a single batch of queries for an inner node yields multiple result batches that act as input for the next iteration, one batch of consistent queries for each child node. Once the leaf layer is reached, the node-query mapping needs to be transposed in order to generate result entries for each query.

Result sets represent a special state that is always present for all query types. They collect leaf node entries matching the search predicate(s) of each query and, therefore, need to implement random access semantics via query ID. Like all global state information, a result set is created upon the initial invocation of one of the index' public search interface methods. After scheduling an initial query batch to the index, the result set is returned immediately, offering an iterator-like interface in order to synchronize on the query results. This result set iterator allows to iterate over the set of matching data entries that are returned for the queries using two strategies. Either the iteration preserves the relative order of queries in the batch, i. e., all results of a query have to be available before those of the next query can be returned, or they are iterated without any order constraints, just extracting the next available entry once it is ready. If queries are still in-flight and no new result entries are available yet, the iterator access methods will block the caller thread until new entries have been found and a notification is received. This callback design allows to continue the work in the application layer with independent tasks until the query results are really required in order to proceed. Further, the asynchronous processing model allows to overlap the index operations with other computations, which might help to hide access latencies for the indexed data entries from an end-to-end perspective.

Internal processing of a query batch is decoupled from the public interface in order to achieve full flexibility when scheduling algorithms are implemented and the actual processing logic is dispatched to different devices, such as CPU cores or GPUs. A *(query batch, node)*-pair represents a *processing primitive* that is parallelized later on in Chapter 5 and that is used as scheduling unit for processing in hybrid systems in Chapter 6. Because there might be multiple of such (query batch, node)-pairs active at the same time, they share ownership of the state and update it in a latched, thread-safe context.[5]

[5] As mentioned earlier, concurrency on the logical level, i. e., concurrent queries and updates of the entries within the same tree instance, is ignored in the following. It is assumed that, for example, appropriate locking protocols as proposed in [123] are implemented, which delay processing of a (query batch, node)-pair as long as it might cause transactional anomalies.

4.3 Implementing a Processing Abstraction Layer in GiST

In order to transparently integrate these changes into the GiST framework, the latter needs to be extended. This section gives an overview of required components and discusses extension points of the framework, which are shaded in gray in the following figures. The actual parallelized implementation of processing primitives by mapping them to a parallel programming model is subject of the following chapter.

4.3.1 Architecture Overview

A high-level overview illustrating the main components of the framework extension is given in Figure 4.5. The components can be classified into the following modules:

- **Index Data** components that are used to model an index;
- **Query Data** components that are used to model search operations on these structures;
- **Query Processor** components for orchestrating the query processing on **Processor Devices** via parallelized **Processing Primitives**.

The *Index Data* components are mainly provided by GiST and can be extended by the well-known extension points that have been discussed in Section 4.1. They provide the key structures and predicate definitions for modeling queries. The *Query Data* components were extended in order to provide a batch-processing interface for the external application and handle asynchronously filled result sets. Query processing is decoupled from the application layer via the *Query Processor* in order to transparently inject code for parallel execution. The *Query Processor* is responsible for fetching query batches from the application interface, map them to parallel primitives that can be scheduled to available *Processing Devices*, and synchronize their execution states upon completion. It also fills the result sets once matching leaf entries are found for a query.

Each independent physical processing unit that is available in the system, e. g., a CPU core or a GPU device, is represented by a *Processor Device* adapter, which maps the framework's parallel *Processing Primitives* to low-level device driver APIs and provides monitoring data. In addition to the GiST extension points that affect the logical index structure and its operations, the framework allows to customize the physical processing of tree nodes via the primitive implementations. For each available device type that shall be supported, specific code needs to be provided in order to implement the execution logic for it. In case of a CPU-only execution, e. g., for a search traversal step, a simple function is sufficient. For GPU execution, however, additional code is required in order to implement auxiliary logic, such as data buffering on the device, handling intermediate data transfers upon cache miss, transferring result data, etc. The extended framework library itself provides default implementations for CPUs and GPUs, which will be discussed in more detail in the next chapter. Additional device-specific code can be provided by developers as library extension.

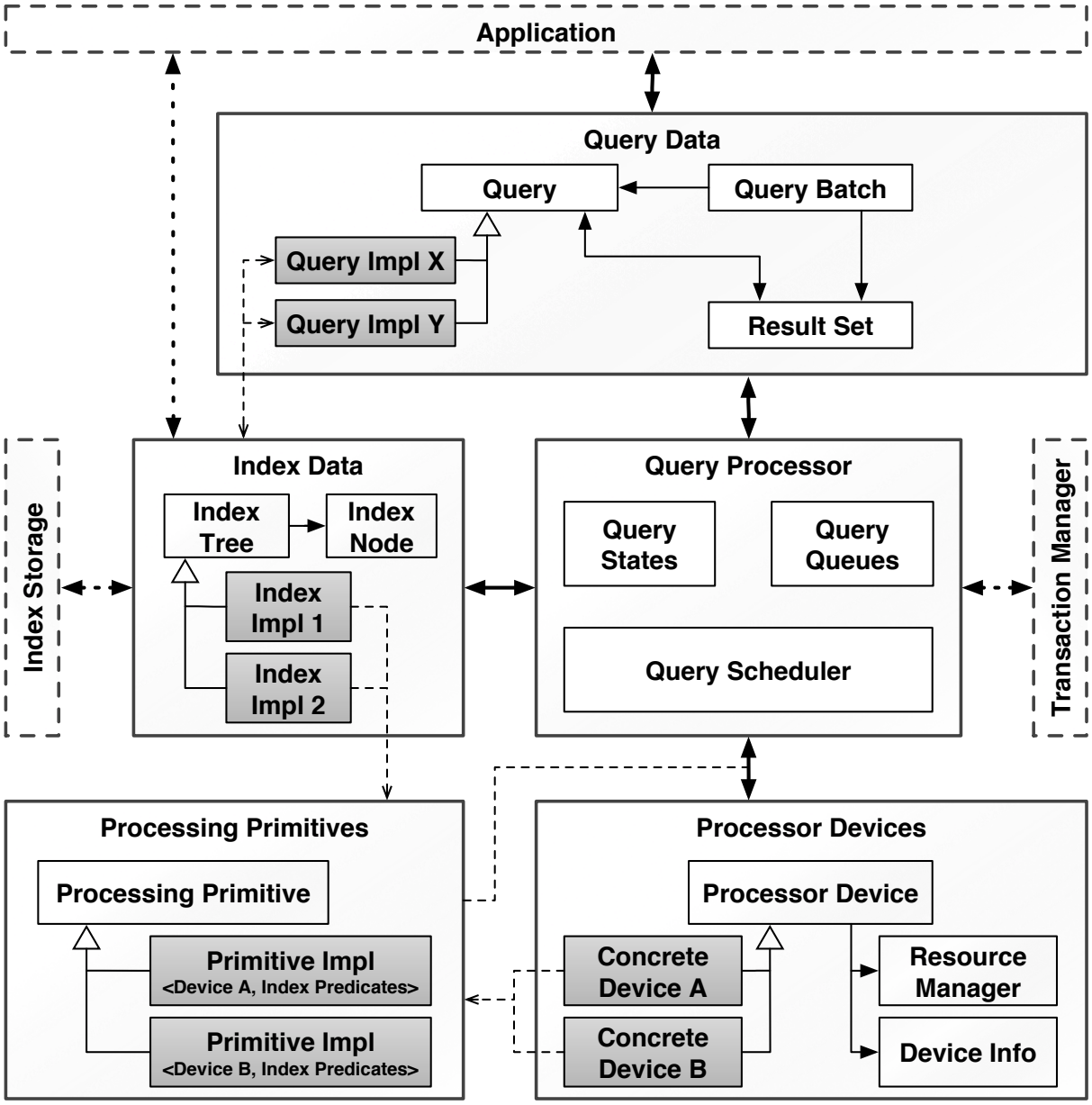


Figure 4.5: Architecture Overview

Interfacing GiST-internal components that are not modified are depicted as dashed boxes. The *Index Storage* abstracts how the tree data is managed in a memory hierarchy. It implements buffer pooling, I/O for persistent storage, etc., and provides random access to an index' nodes. Concurrency control is assumed to be implemented by GiST's *Transaction Manager* that signals whether accessing a node is allowed for a transaction's query and blocks it otherwise.

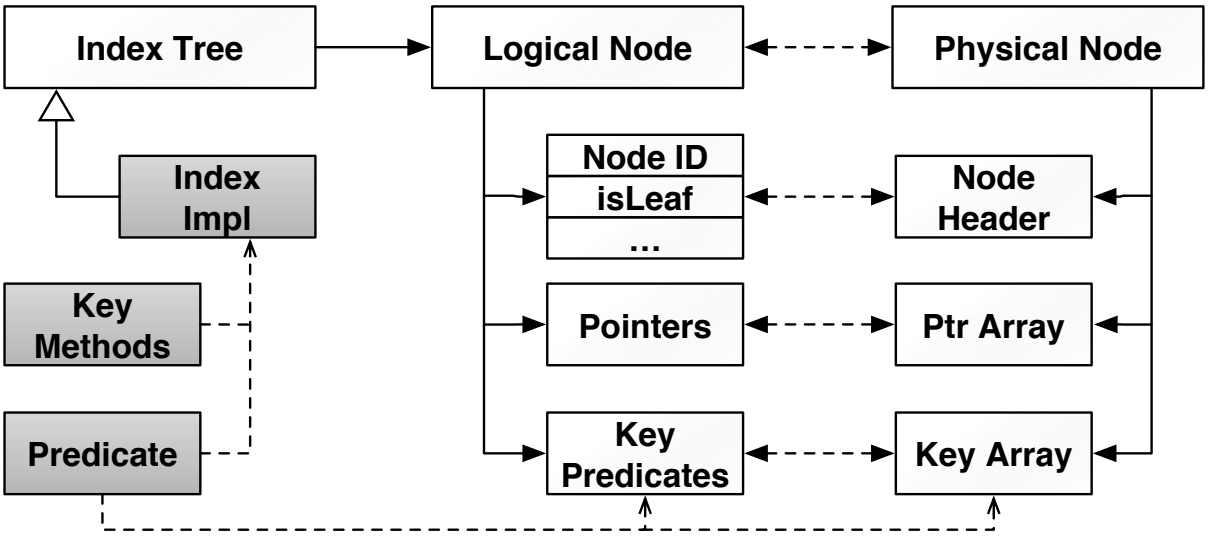


Figure 4.6: Index Structure Components

4.3.2 Index Components

A more detailed view on the components for modeling an index structure is given in Figure 4.6. An index tree represents the entire index structure and provides access to its nodes. An index does not necessarily need to be a tree but it should satisfy the properties of a DAG so that traversal operations terminate in the leaf layer. It might also have multiple root nodes that indicate starting points for traversal operations for all its disconnected components. This allows to, for example, model partitioned trees. In addition to the actual index data, the tree provides GiST key methods and types. Key predicates, in particular, are required to interpret node data and implement the logic for predicate evaluations.

Index data is represented by node structures. Physical nodes represent the raw, serialized data that is accessible via the GiST storage manager and that will be transferred between processing devices later on. Such a node comprises meta information about the node in a header structure as well as the GiST (key, ptr)-pairs. But for performance reasons, these pairs need to be split into two separate arrays, where the pair relationship is reconstructible via the entries' positions within the arrays. The interpretation of the physical data is done via logical nodes, which decode the header information and also construct GiST key predicates from the key structures. This is also the place where the optional `(de-)compress()` GiST methods are applied. As meta-information, a node at least provides its ID for accessing it via the tree as well as a flag that indicates whether a node is a leaf node, whose entries represent actual data elements, or an inner node, whose child pointers refer to other tree nodes.

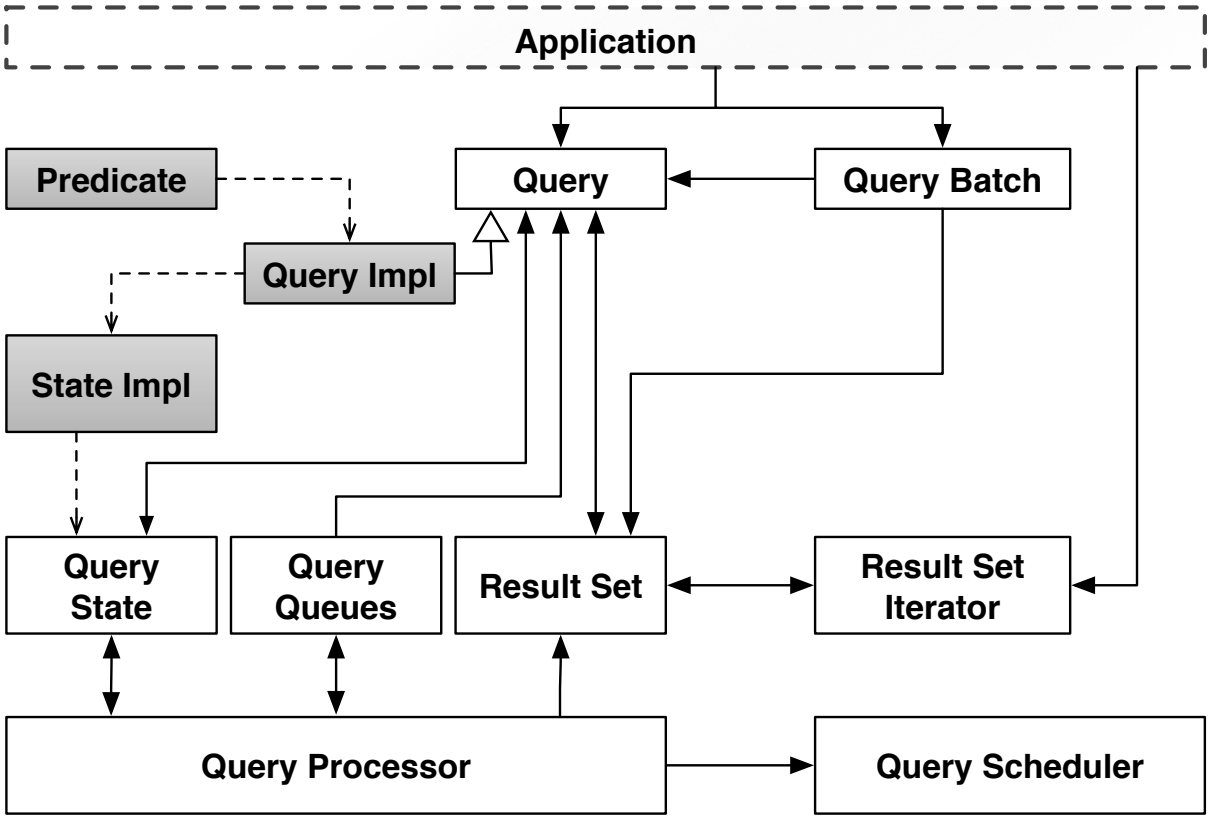


Figure 4.7: Query Components

4.3.3 Query Components

Components that are interacting with queries are depicted in Figure 4.7. Queries are more or less complex predicates that describe all searched index entries via one of GiST's supported predicates, e. g., a `within()`-predicate that is instantiated with a bounding box to implement geometric range search, or a `nearestNeighbor()`-predicate that is instantiated with a specific query point. Which queries are supported depends on the index implementation. Queries are issued via the application, either as batches or as single query stream that is internally converted into batches via an adaptor.

Processing of nodes is handled by a query processor, which manages them inside a queue. The queue stores all active queries per node in order to extract them batch-wise for parallel processing of the traversal step. After each step, the query state is updated and, if an inner node has been processed, matching child node entries are queued for the next iteration. In case of leaf nodes, matching entries are collected in the query's result set. Results can be accessed by the application via iterators that act as synchronization mechanism with the decoupled, asynchronous query processing. If a (future) result is not available, yet, the iterator will block the application until it is provided.

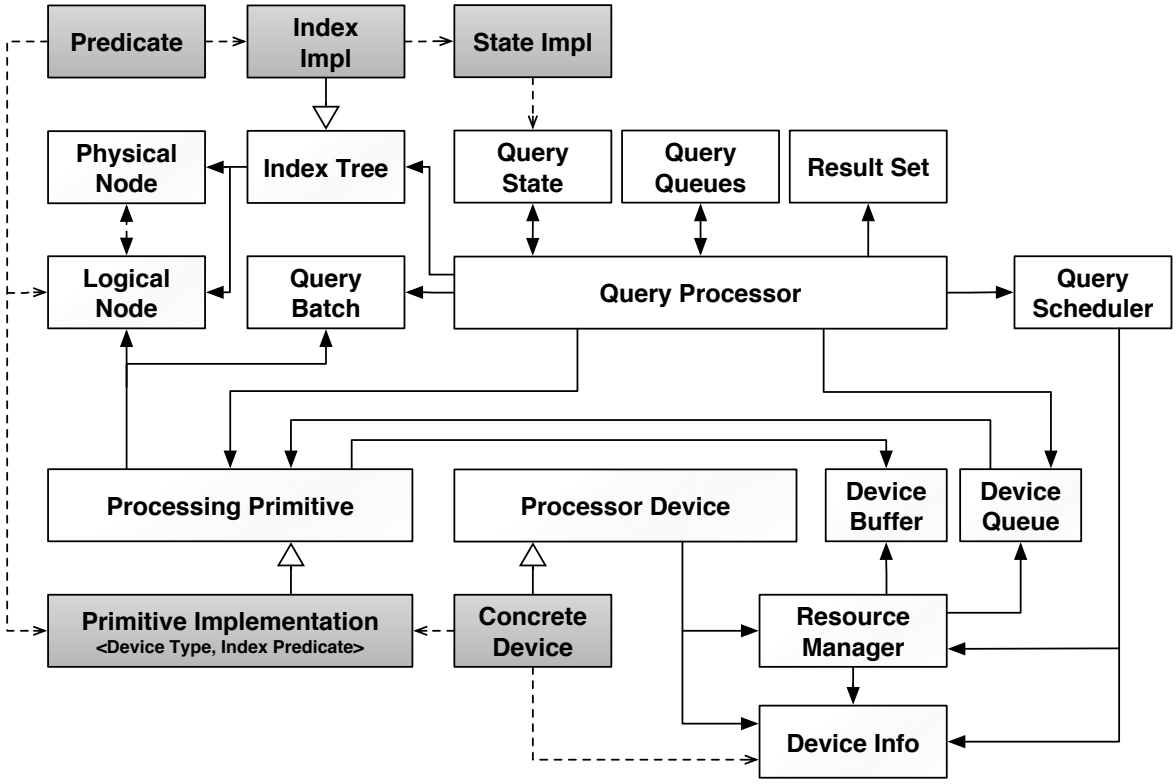


Figure 4.8: Processing Components

4.3.4 Processing Components

All components that are involved in query processing are illustrated in Figure 4.8. After the query processor extracted the next batch of queries from the queue, it fetches the node data from the index and wraps the (node, query batch)-pair into a processing primitive. This primitive acts as a handle for the asynchronous query processing step that can be offloaded to one of the available processing devices. The decision, which device shall be selected, is made by a query scheduler. The scheduler monitors the load on each device and implements a cost model based on the device characteristics that are provided as meta information, such as number of cores, cache sizes, etc. Details on the scheduling component are subject of Chapter 6. Once the decision has been made, the primitive is queued for the selected device and handled asynchronously. Upon completion, the query processor is notified and the query state is updated with refined information after the processing step.

The logic that specifies how a (node, query batch)-pair needs to be processed is encapsulated within device-specific primitive implementation code. Such an implementation exists for each (device type, query predicate)-combination, because special handling might be required for, e. g., injecting intermediate data transfer operations between host and device buffers, or for implementing device-specific optimizations that, for example, leverage the available software-controlled caches on the device. Each primitive implementation depends on the query predicate definition, because merely raw, physical node data can be transferred via the memory bus that connects a device. This data needs to be interpreted and evaluated with functions that are aware of the actual predicate types.

4.4 Evaluation – Applicability of the Extended Indexing Framework in Computer Graphics Applications

The previous discussion illustrated that the proposed index framework based on GiST abstracts actual index structures pretty well, because GiST already covers the main issues. The design of the parallel processing layer as internal extension, which is an orthogonal feature to the already existing GiST extensions (cf. Table 3.1), builds on top of the framework that has already been deployed in many real-world applications, such as the Postgres DBMS and many other projects [174]. Therefore, the new extension should transparently integrate into such applications as well. In this thesis, some experiments have been conducted using POV-Ray (Persistence of Vision Raytracer) [4] and Ogre 3D (Object-Oriented Graphics Rendering Engine) [3, 101] with the intention to evaluate the complexity of integrating the extended framework into application layer code and assess potential benefits of an internal parallelization of index-based search operations.

4.4.1 Framework Integration into Image Rendering Engines

The experiments with the open-source image rendering engine projects revealed that the assumption that the indexing framework can be transparently integrated into application code is only partially true [103]. Integrating the original GiST was quite simple. All that had to be done was

1. identify functions in the code base that access the internal index
2. model existing data structures as keys in the GiST data model
3. for all supported search operations: map hard-coded queries to predicate-based versions
4. replace index access operations with the generalized counterparts

For POV-Ray, a basic 3-dimensional R-Tree was integrated for storing the underlying scene model. Ray intersection tests with the scene that need to be executed for each pixel were mapped to simple 3-dimensional point queries. These modifications could be implemented straightforward. For the Ogre 3D engine, the changes were more complex, because a new spatial index hat to be integrated into an internal storage manager component that filters visible objects within the rendered scene. A modified R-Tree version was used therefor that supports a multi-staged object filtering, which was easy to implement by overriding the GiST key methods. The first stage applies an Ogre 3D-internal *level-of-detail* filter to select the best granularity of the scene model, depending on the camera's distance from the object (cf. Figure 4.9 [134]). The second stage is a *camera frustum culling* operation that, in contrast to the intersection queries

Figure 4.9: Level-of-Detail Models
Source: [134]

LOD	Polygons
0	4336 (44%)
1	2562 (26%)
2	1510 (15%)
3	879 (9%)
4	510 (5%)
total	9797

Table 4.2: Ogre Model Polygons

that have been implemented in POV-Ray, requires a more complex 3-dimensional range query. In order to evaluate whether an object's bounding box intersects the camera frustum, a predicate had to be defined that transforms query coordinates into the camera coordinate system and solves six plane equations during each tree traversal step. The query result comprises currently visible triangles of the object's mesh in the tree's leaf layer. These triangles are filtered in a subsequent *backface culling* step that evaluates the direction of the triangles' surface relative to the camera in order to remove those entries facing away from it. This final post processing evaluates another plane equation predicate for each intermediate result entry. All result triangles are finally transferred to a GPU for the actual image rendering via OpenGL (Open Graphics Library).

Injecting the parallelized version of the indexing API was a lot more difficult than expected. The design of the framework extension requires that the application layer code can cope with its batch-oriented asynchronous execution model in order to fill the index' query queues efficiently, just receiving result data once it is really required. This is the major drawback of the solution proposed in this thesis, when the framework shall be integrated into legacy code bases that implement a synchronous version, following a query-at-a-time processing model. That is, for a high-level operation, such as a rendering call of an image in Ogre 3D, a search is just conducted once and its results are immediately processed after the call as subsequent step inside an internal processing pipeline. The entire pipeline is then executed many times instead of dividing it into fine-granular steps that can be independently vectorized.

Decoupling the generation of new queries from the consumption of query results caused the most issues when the new extended framework should be used by the existing rendering engines. This engineering task could not be addressed in the scope of this thesis. Therefore, porting a whole application like POV-Ray or Ogre 3D is left for future work. Nevertheless, the extended prototype code was sufficient to run some performance measurements in order to profile the impact of search operations on the overall application performance and also assess potential benefits of a parallelized version, which is discussed next.

4.4.2 Profiling of Search Operations

Initial performance benchmarks revealed that within POV-Ray, accessing the spatial index for ray intersection tests required more than 50 % of the overall image rendering time. A similar picture could be observed in the experiments with the Ogre 3D engine for filtering visible scene mesh triangles in order to reduce the amount of data that needs to be transferred to the GPU for the actual image rendering. Two kinds of experiments have been conducted here, comparing the original Ogre 3D rendering algorithm to the modified version that uses the generalized index: an experiment that evaluates the impact of index-assisted filtering based on object visibility, and a scalability experiment to assess benefits for large objects lookups when the index-based search would be parallelized.

For rendering an image, Ogre 3D requires that all triangles to be rendered are present in the main memory of the GPU, where internal culling algorithms are applied on the polygons. With the modified approach, the index reduces the amount of data that needs to be displayed *before* the actual rendering call, transferring only visible portions of the model to the GPU. This approach effectively enables out-of-core rendering in Ogre 3D that could not be done before. However, this comes at additional costs for the *index search* and the dynamic *GPU upload* phases before the actual *image rendering* process. Thus, it is expected that the modified algorithm increases scalability of Ogre 3D for large model sizes but might become slower, due to the overhead compared to a model that is cached on the GPU and completely rendered, ignoring the visibility of different portions of the scene.

Figure 4.10: Ogre Head
Source: [3]

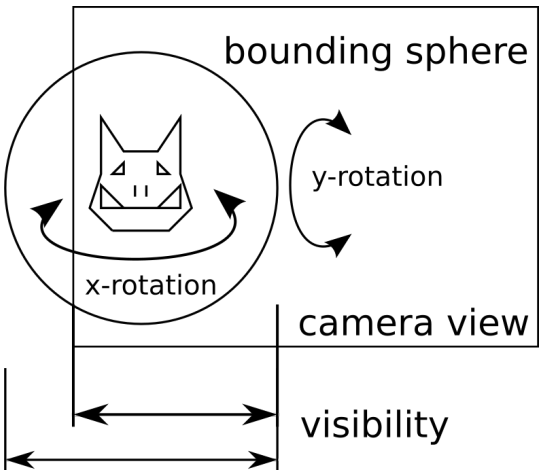


Figure 4.11: Ogre Visibility Experiment

The first experiment analyzes the visibility-based filtering effect of the index on a small sample model, which is illustrated in Figure 4.10. The number of polygons in all level-of-detail instances is listed in Table 4.2. The model is placed in the virtual space, relative to the camera, shifting it partially out of sight as illustrated in Figure 4.11. By increasing this offset, the visibility v can be changed and its effect on the rendering process can been measured, assuming

v % of the model's triangles to be visible if *v* % of its bounding sphere is visible in the camera frustum. Of course, this simplifies measurements significantly as it ignores different densities of the triangle mesh inside the bounding sphere's volume and assumes independence from the cameras perspective relative to the model. In order to minimize the impact of this inaccuracy, the model has been rotated around the x and y axis for each measurement, averaging the number of visible triangles for many camera positions for a single run. The final backface culling algorithm almost halves the result set size, because just the camera-facing side of the model needs to be displayed.

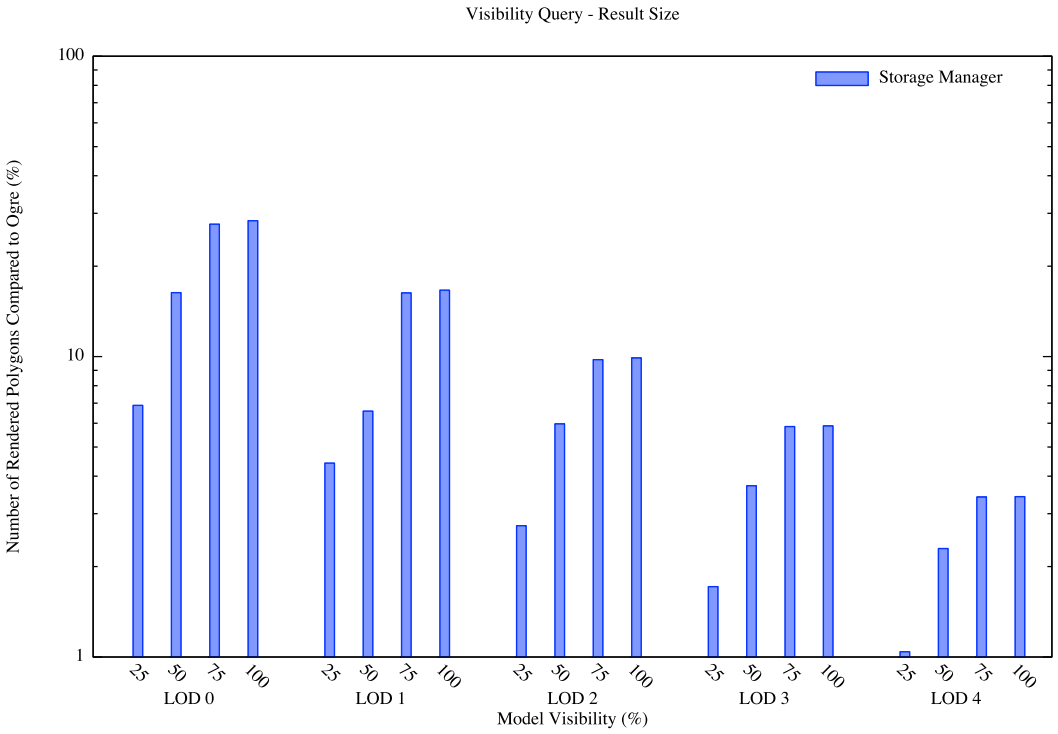


Figure 4.12: Query Result Sizes with Varying Visibility

The result set sizes for the filtering step are illustrated in Figure 4.12, relative to the size of the original model stored in Ogre 3D. The diagram illustrates the result sizes for each of the five level-of-detail representations of the model. Ogre 3D has to store all of them while the modified version transfers only the polygons of the required level. One can clearly see that the pre-filtering using the modified index can reduce data volumes to be rendered by an order of magnitude, even for low selectivities, which is proportional to the visibility parameter. However, this improvement does not reflect in the rendering times, which are illustrated in Figure 4.13 because the rendering time has only a small impact on the overall runtime. The actual image rendering (green bar in Figure 4.13) is comparable to the rendering time required by Ogre 3D (green line in Figure 4.13). For low visibility values, one can see a small improvement for this phase. Most of the time is spent inside the index, traversing the tree and evaluating the chain of search predicates. One bottleneck that has been found by more detailed profiling is that tree traversal cannot be stopped prematurely, even when it is detected that the entire sub-scene represented by a tree node is completely visible. This effect explains the significant increase of

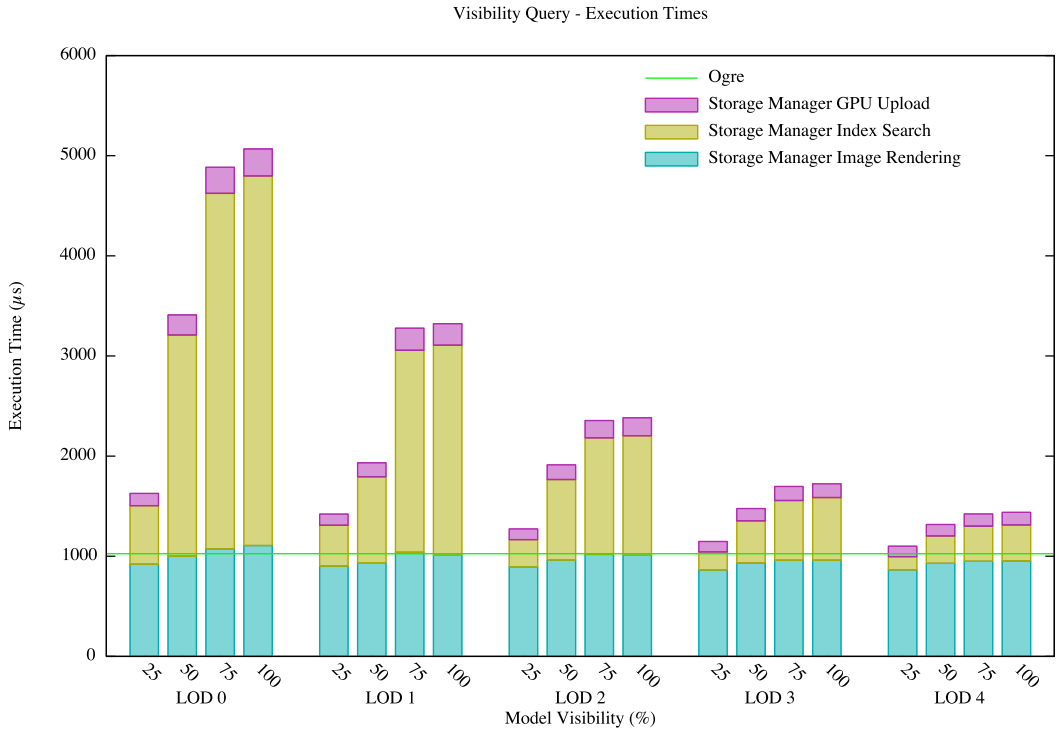


Figure 4.13: Rendering Times with Varying Visibility

search time for large visibility parameters and offers more optimization potential by enhancing the index structure. Switching to a scan and rendering all triangles will speed up these cases. However, the approach followed in this thesis is a different one, targeting to reduce lookup times by parallelizing the traversal.[6]

The second experiment that has been conducted with the Ogre 3D prototype is a scalability experiment with increasing model sizes of a larger open-source model that is illustrated in Figure 4.9. The "*Stanford Bunny*" mesh [134] used for this experiment is comprised of 69,451 polygons. The model has been replicated n times in the same scene in order to verify that the out-of-core approach allows larger model sizes to be rendered and also evaluate the overhead for that. Only a single level-of-detail layer has been used in order to eliminate the effects of the first filtering stage. Further, all replicas were fully visible in the camera frustum in order to eliminate visibility impacts. Thus, only the final backface culling reduces data volumes. The number of polygons that are buffered in GPU memory for the rendering is illustrated in Figure 4.14. By examining the delta between both curves, one can clearly see that Ogre 3D cannot handle larger models when the GPU capacity is reached. The out-of-core approach nearly doubles the manageable model sizes (because of backface culling).

[6] In the Ogre 3D experiment, an image rendering call only represents a single query. In order to increase the effects of batch processing, multiple rendering calls could be collected, e. g., for simulations. For POV-Ray, the parallelization is much simpler because multiple rays can be processed simultaneously.

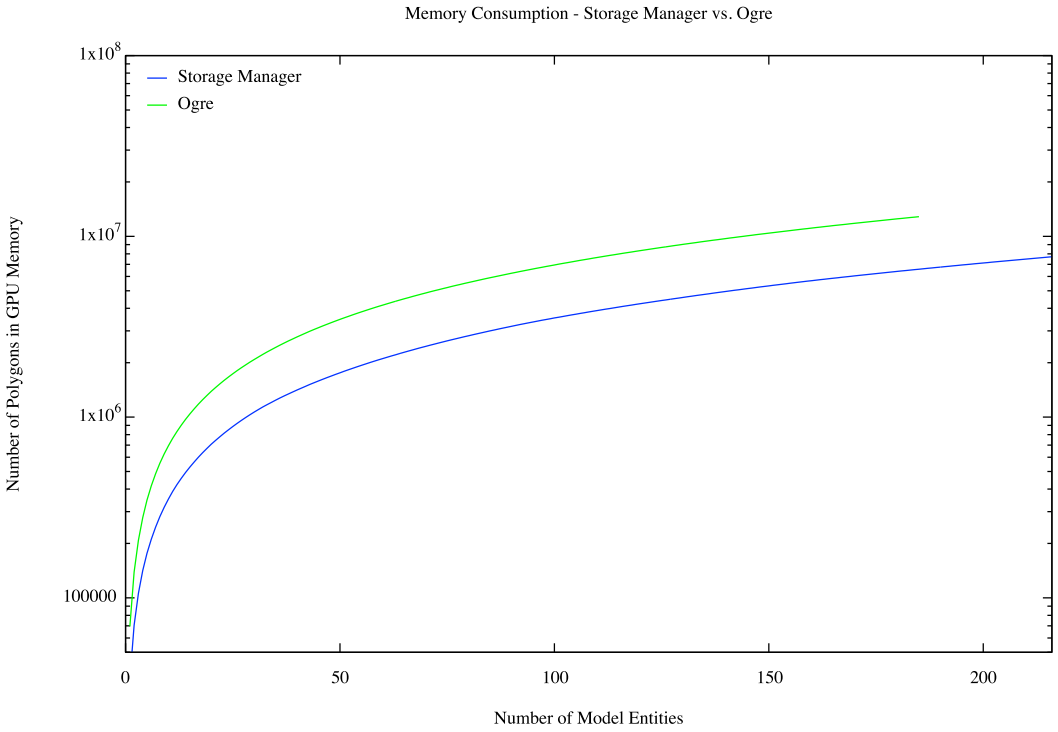


Figure 4.14: Scalability Experiment: Memory Consumption

Execution times for the different algorithm phases are illustrated in Figure 4.15 and Figure 4.16. As expected from the results of the visibility experiments, the actual rendering time only consumes a fraction of the overall runtime. Most of the time is spent in the index and for uploading the result set to the GPU with an increasing impact of the latter with increasing model size. The rendering times fluctuate for very large models when the system reaches its main memory capacity. By comparing the Ogre 3D runtime in Figure 4.15 with the rendering time of the filtered model, one can clearly see that for this phase, unsurprisingly, using the index for reducing data volumes pays off. For this phase, up to 90 % of the time can be saved. So, if the significant overhead caused by the other phases can be reduced, e. g., by parallelization and caching mechanisms, the out-of-core implementation will be a valuable extension to Ogre 3D, even for interactive applications.

In summary, these experiments have shown that a significant portion of the execution times of the examined computer graphics applications is spent for traversing the spatial index structures and, thus, offers great potential for accelerating the overall rendering process by deploying a parallelized version of that. The parallelization proposed in this thesis is an internal one that is directly integrated in the index data structure. Thus, it is orthogonal to other parallelization efforts that operate on a higher level, such as partitioning the input scene of a POV-Ray rendering task beforehand and distributing them to multiple execution units each [58, 166]. Therefore, the following chapters shall present and analyze the potential performance gains of applying an automatic parallelization inside the low-level index access step.

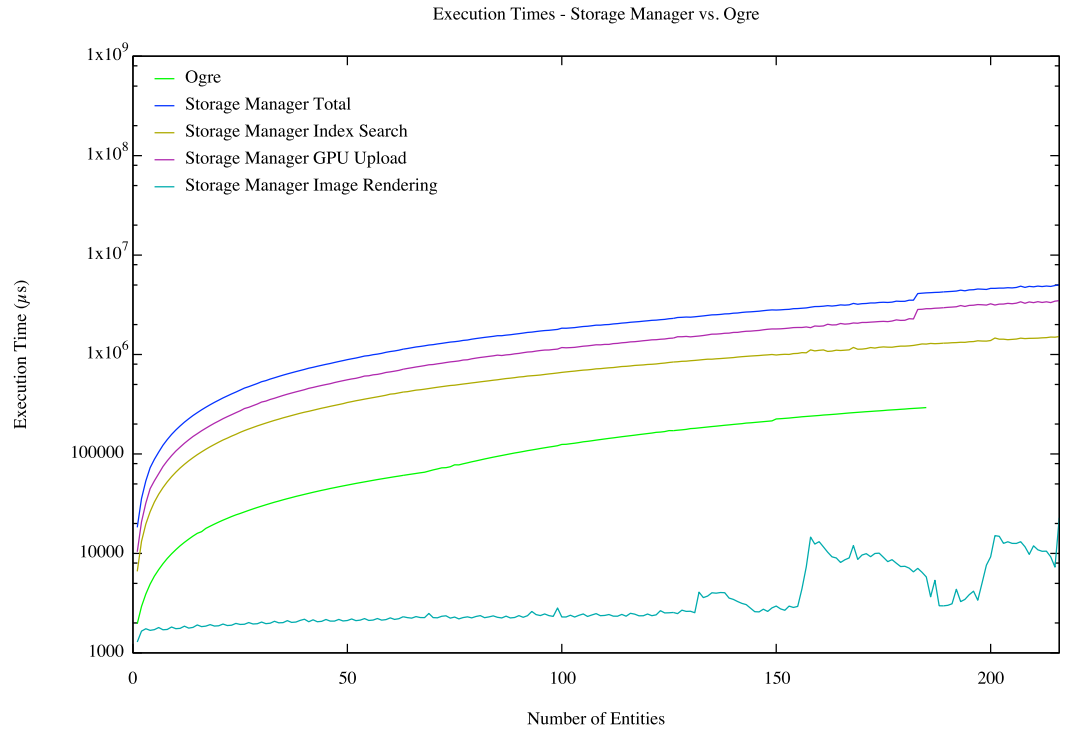


Figure 4.15: Scalability Experiment: Comparison of Absolute Execution Times

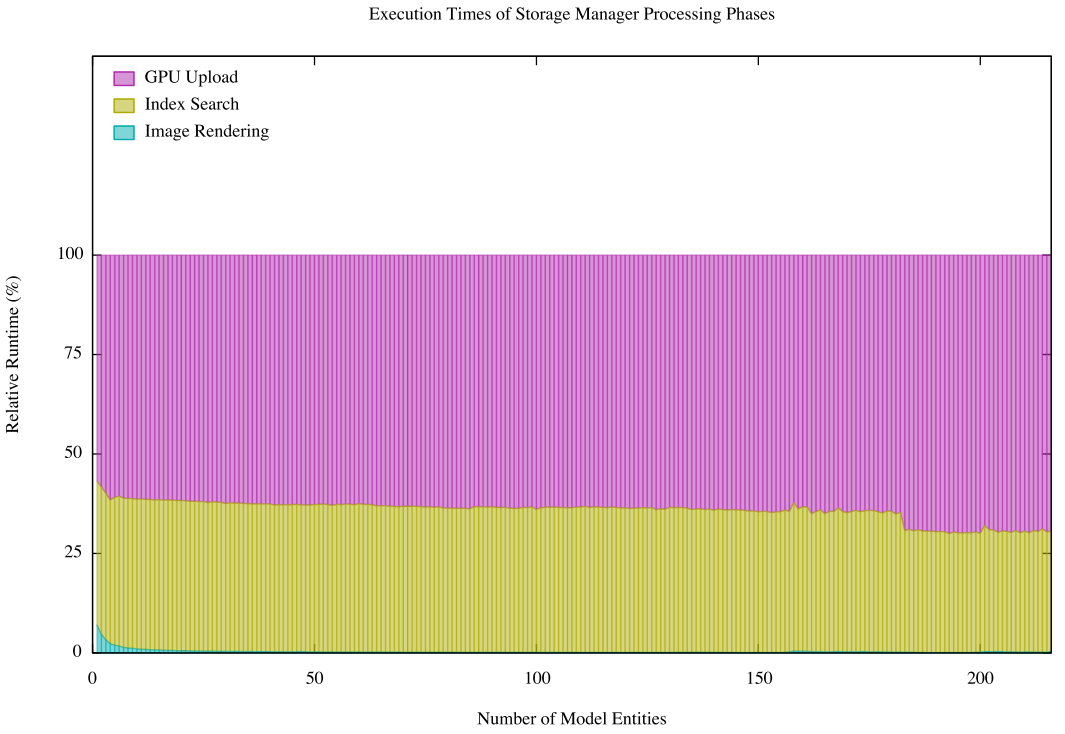


Figure 4.16: Scalability Experiment: Comparison of Relative Execution Times

4.5 Summary

There are many frameworks available that allow to implement different kinds of index structures, whereas search trees represent the most prominent class that has been examined here. By providing commonly shared code in a library, which operates on abstract methods that can be overridden by developers to model a specific index kind, these frameworks save a lot of development costs when an index structure shall be implemented for a given application scenario. Moreover, a lot of extensions have been proposed so far to enhance these structures by implementing support for other functional aspects, e. g., transaction support, interfaces for specifying custom traversal strategies, etc., that are relevant for the practical use of an index in real-world applications.

However, none of these extensions addresses parallelization of the implemented algorithms so far or how up-to-date parallel processor architectures can be leveraged in order to speed up involved calculations. This challenge has been addressed in this thesis by extending the well-known GiST framework by an additional processing abstraction layer that maps index search operations to generic parallel processing primitives, which can be executed by arbitrary processor types. Further, the original GiST framework has been extended by additional batch-oriented interfaces in order to exploit vectorized processing of search queries, which will be discussed in the next chapter.

As experiments with computer graphics applications have shown, parallelizing index lookups is a promising approach, because a significant amount of time is spent in this part of the image rendering pipeline. However, in order to fully leverage parallelization benefits, the processing model of the application layer needs to fit the requirements of the framework extension. That is, it needs to be able to vectorize its search operations and must be able to cope with the asynchronous processing model in order to fill internal processing queues efficiently. Although adopting this pattern in existing legacy code bases causes significant changes, vectorizing operations and exploiting asynchronism is a commonly used pattern in modern software development that should be applied in order to alleviate performance bottlenecks in the age of parallel processing hardware [102, 192].

Chapter 5

A Parallel Execution Model for Generalized Search Trees

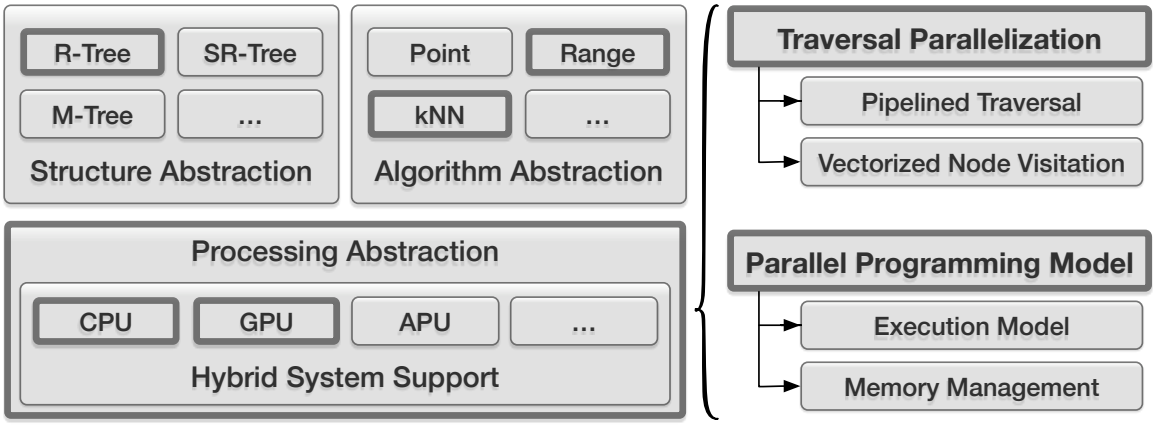


Figure 5.1: Overview of the Parallel Index Execution Model

After it has been shown how different kinds of indexes can be modeled on a logical level and how the physical execution of search operations can be abstracted by generic processing primitives, the latter have to be mapped to actual processor-specific implementations that leverage parallel processing features of the underlying processor architecture. The mandatory steps to accomplish this task are discussed in this chapter.

Generally applicable concepts for parallelizing GiST search algorithms are discussed in Section 5.1. The generic tree traversal algorithm is broken down into independent node-local tasks that can be pipelined and executed on devices that support a vectorized data-parallel execution model. The parallelized algorithms are mapped to a parallel programming model in Section 5.2. Therefore, CUDA has been selected, because it abstracts many device-specific hardware details and allows to seamlessly integrate different GPU generations. Further, a new out-of-core GPU algorithm is presented that does not require full index storage in the GPU's device memory. A prototypical implementation of this algorithm is evaluated in Section 5.3. Based on microbenchmarks, it is analyzed how well the vectorized algorithms perform on a CPU, compared to different generations of GPU devices. The benchmark results will be used in the next chapter to train prediction models for estimating execution times on the available devices in the system in order combine the strengths of both processor types within a hybrid system.

5.1 Parallelizing Search Operations in Generalized Trees

The main idea presented in this thesis is to parallelize generalized search tree traversals by identifying independent operations and map these traversal algorithms to an abstract parallel computation model, which, in turn, can be mapped to various hardware platforms. Therefore, it is required to identify different means of independence when index nodes are processed.

In the original GiST library, search operations were implemented single-threaded, one element at a time, with the following comparison-based tree traversal step: For each key predicate in the current node,[1] identify those children where `consistent( keyPred, queryPred )` yields `true`, store all candidates in a stack/queue for recursively descending inner nodes, otherwise insert them as matching entries into the result set. Continue with the next node from the stack / queue until the latter is empty.

This traversal can be parallelized by mapping it to a braided parallel BFS, which is discussed in Section 5.1.1. Further, processing a single query at a time is not very efficient in terms of data accesses per search, because each key predicate is accessed only once, but access latencies occur for each node whose keys need to be fetched. Therefore, vectorized implementations of node evaluation primitives are discussed for both, stateless and stateful search queries, in Section 5.1.2 and Section 5.1.3.

5.1.1 Parallel Search Tree Traversal

As discussed in Section 4.3, executing search operations in the extended framework can be broken down to processing batches of queries for tree nodes whose key predicates satisfy the query predicates. Candidate nodes that still need to be visited are managed inside a queue, which is comparable to the original GiST traversal algorithm.

This algorithm can be parallelized on node-level, which is illustrated in Figure 5.2. Metaphorically speaking, vectors of query predicate instances "flow" through the tree, starting at the root node. Each query either reaches the final leaf layer, yielding entries for its result set, or is filtered inside an inner node, which removes it from the input of subsequent batches. Each processing primitive, i. e., a pair of a query batch and a to-be-visited node, is independent from the others in the queue and, therefore, can be executed in parallel by one of the available processing units. There might be many of these primitives because of two reasons. First, a single traversal iteration might produce a large set of matching child nodes, depending on the node's fanout and the query predicates' selectivity. Each of them can be processed in parallel, which basically implements a parallel BFS. Second, while an initial node batch is processed asynchronously within the tree, the next one might become ready and is "inserted" at the root node by the application if it implements a streaming model. These primitives are also independent from the others and can be pipelined within the tree.

[1] In case of ordered domains and sorted keys, binary or interpolation search can be used as optimization.

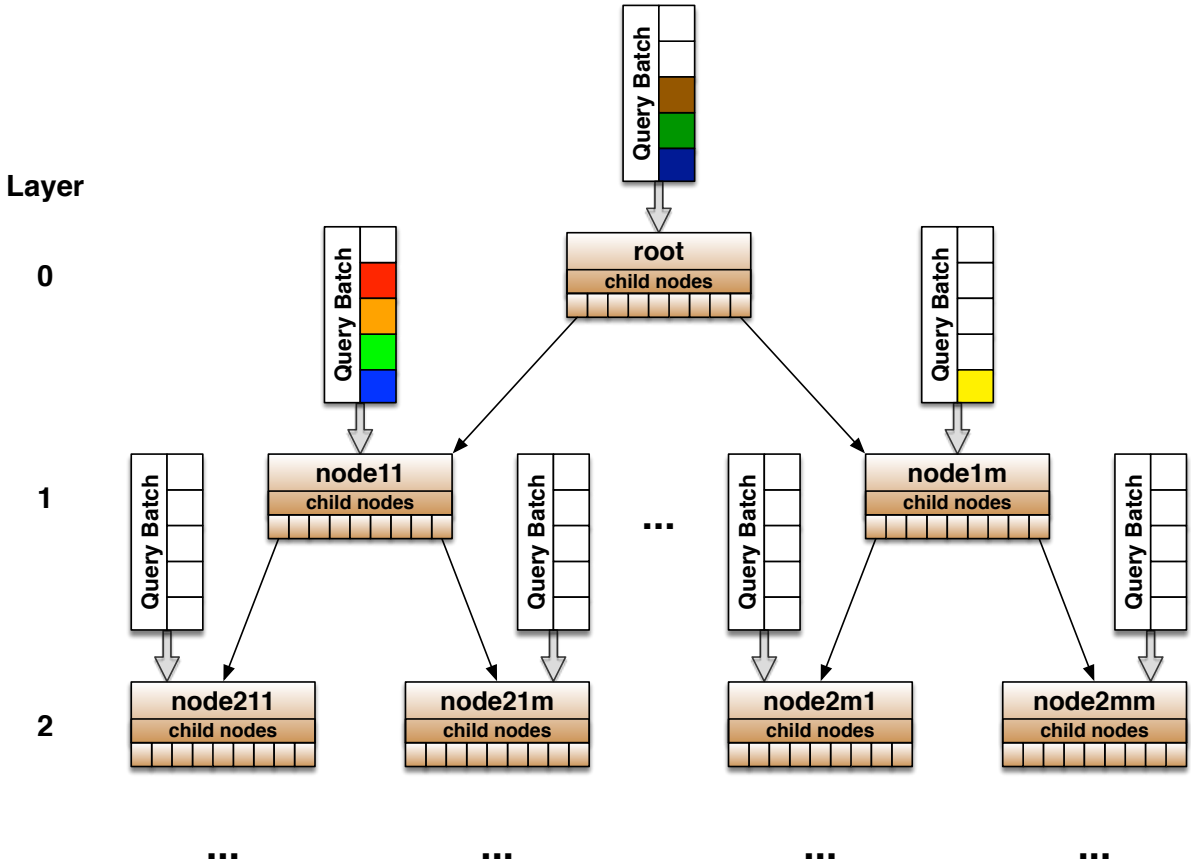


Figure 5.2: Task & Pipeline Parallelism in Index Search

5.1.2 Vectorized Processing of Stateless Queries

The original query-at-a-time processing implemented by GiST causes significant data access overheads with little computation per predicate evaluation - at least for simple predicates, which are common in many indexes. Even if the tree is hosted in-memory, once a node is fetched, it will most likely evict previously accessed ones from low-level caches, because a node's size is usually optimized rather for I/O than for cache access and much larger than cache lines. This can, somehow, be mitigated by implementing a cache-conscious tree layout that hides access latencies, but the idea in this thesis is much simpler and gained considerable traction in the last decade [212]. If a batched search interface is provided for use cases that generate many search queries at once, e. g., for index-nested-loop-joins, *vectorized processing* can be applied in order to traverse the tree simultaneously, which reduces the overall data access overheads. *(node, query batch)-pairs*, that represent a *processing primitive* in the tree traversal, increase the number of operations that can be applied on a node's keys before they are evicted from the cache. The evaluation of such a primitive will be referred to as *"node scan"* in the following, because all query predicates need to be evaluated in a braided search to check whether they are `consistent()` with the key predicates.[2]

[2] Optimizations for ordered domains might be possible.

Algorithm 5.1 Parallel Stateless Node Scan Primitive

Input: queryBatch, keyList
Output: consistent batches for child candidate nodes

```
1: for qry ∈ queryBatch in parallel do
2:     for key ∈ keyList in parallel do
3:         childCandidates[ qry, key ] = consistent( key, qry )
4:     end for
5: end for
6: return childCandidates
```

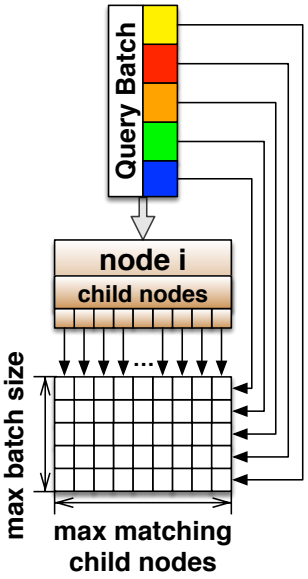


Figure 5.3: SIMD Parallelism in Node Scan Primitives

Further, evaluating multiple key and query predicates at once reveals potential for applying data parallelism, because all keys can be processed independently for each query. The basic algorithm is depicted in Algorithm 5.1 and illustrated in Figure 5.3. Keys act as "guards" for the next tree layer and have to be checked for all query predicates. The cross-product of these checks yields a boolean result matrix whose size can be computed in advance, given the maximum batch size and the node fanout (number of slots). After the node scan, the matrix contains the result for the next traversal step. If an inner node is processed, results are extracted in a columnar manner so that matching predicates are grouped per child node in order to, again, extract query batches as input for the next stage. For leaf nodes, results are extracted row-wise, because matching entries per query predicate have to be gathered. This fine-grained parallelism is best-suited for a vectorized SIMD processing model, which is supported by many processor devices, GPUs in particular.

5.1.3 Vectorized Processing of Stateful Queries

For stateful queries, such as nearest neighbor searches, the evaluation of the pruning predicate depends on the current state that has been aggregated during the tree traversal and will be considered in the stateful `consistent()` predicate evaluation function. Therefore, it is required to maintain a global query state that is updated with the latest information after a node has been processed. In nearest neighbor searches, for example, bounding distances will be refined that can be used to prune entries that are for sure farther away from the query point. The more accurate the state is, i. e., the more index nodes have already been evaluated, the better the search space pruning can be applied, leading to less false positive predicate evaluations for nodes that cannot be excluded from the result candidate list, yet, but do not contain matching entries.

Algorithm 5.2 Sequential Stateful Node Scan Primtive

Input: queryBatch, globalQueryStates, keyList
Output: consistent batches for child candidate nodes

```
1: for qry ∈ queryBatch do
2:     for key ∈ keyList do
3:         state = calculateState( qry, key )
4:         globalQueryStates[ qry ].merge( state )
5:     end for
6: end for
7: return pruneNodes( globalQueryStates, keyList, queryBatch )
```

Algorithm 5.3 Parallel Stateful Node Scan Primtive

Input: queryBatch, globalQueryStates, keyList
Output: consistent batches for child candidate nodes

```
1: localQueryStates = copyAtomic( globalQueryStates )
2: __sync()
3: updateQueryStates( queryBatch, localQueryStates, keyList )
4: __sync()
5: childBatches = pruneNodes( localQueryStates, keyList, queryBatch )
6: __sync()
7: for qry ∈ queryBatch in parallel do
8:     globalQueryStates[ qry ].mergeAtomic( localQueryStates[ qry ] )
9: end for
10: return childBatches
```

Algorithm 5.4 Parallel Query State Update

Input: queryBatch, queryStates, keyList
Output: updated query states

```
1: for qry ∈ queryBatch in parallel do
2:     for key ∈ keyList in parallel do
3:         stateMatrix[ qry, key ] = calculateState( qry, key )
4:     end for
5: end for
6: __sync()
7: for qry ∈ queryBatch in parallel do
8:     queryStates[ qry ] = parallelReduce( stateMatrix[ qry ], stateMergeOp )
9: end for
10: __sync()
11: return queryStates
```

The sequential algorithm for processing a batch of stateful search queries for a single node is depicted in Algorithm 5.2. For parallelizing it, it is needed to resolve some data dependencies. The first part of the parallel version is illustrated in Algorithm 5.3. The algorithm needs to operate on a copy of the global query states that are shared between all node visitations while whole the tree is traversed (line 1). Maintaining this private copy allows to operate on it in parallel, without synchronizing with other primitive executions. The state updates are aggregated locally (line 3) and need to be merged with the global state afterwards (lines 7–9) for maximizing the pruning efficiency for subsequent child node visitations. The set of child batches that acts as input for the next iteration is also aggregated locally to avoid additional synchronizations with other node primitives (line 5).

The initial state copy must be executed atomically, because there might be concurrent updates from other nodes. Further, all SIMD operations need to synchronize between those phases that have data dependencies to the result of the previous one. The search space pruning step in line 5 can be executed using the same SIMD parallelism scheme that has been discussed for the stateless node scan evaluation. It just needs to consider query states when filling the result matrix. The global state merge in line 8 can be parallelized for each query within the batch, because they are independent from each other. But each merge step needs to be atomic in order to serialize concurrent updates of the same query state.

For calculating the local state aggregation in line 3, different parallelization schemes can be applied. The trivial version just parallelizes the aggregation for each query in the batch and sequentially applies the state update for each key, like in Algorithm 5.2. This version does not require additional memory for buffering state evaluation results, apart from the local state copy that is created once a node is visited, but does not fully leverage all parallelization potential.

The second version that also parallelizes the state aggregation is illustrated in Algorithm 5.4. Initially, it applies a pattern that is similar to the previously discussed parallel pruning implementation. A separate local state can be independently calculated for each (query, key)-pair and stored inside a state matrix that acts as buffer for the subsequent parallel aggregation phase (lines 1–5). Such a local state might, for example, be the child node's minimum and maximum distances to the query point for nearest neighbor searches. After this step, the state matrix needs to be transformed into a vector of aggregated state information that, like the input vector, contains a more refined entry for each query. The aggregation can be implemented as parallel *reduce primitive* [29] that applies the state merge as associative binary operation within each step (lines 7–9). This reduce primitive is illustrated in Figure 5.4. The main idea here is to construct a computational tree for each state matrix row that merges independent element pairs into aggregated elements in multiple rounds. The process is repeated with half of the elements of the previous row vector until a single result entry is calculated inside the last column. Afterwards, the last column vector of the matrix stores the aggregated state information for each query.

Compared to the trivial query-wise parallelization, the second algorithm utilizes parallel processing resources more efficiently, but requires more memory for buffering intermediate results. The required buffer size can be calculated in advance. Thus, it is possible to automatically select the best implementation for the characteristics of the available processing device, i. e., memory sizes, number of cores, etc., as well as current workload parameters, i. e., batch sizes, the number of key entries within a node, the size of a single state entry, etc.

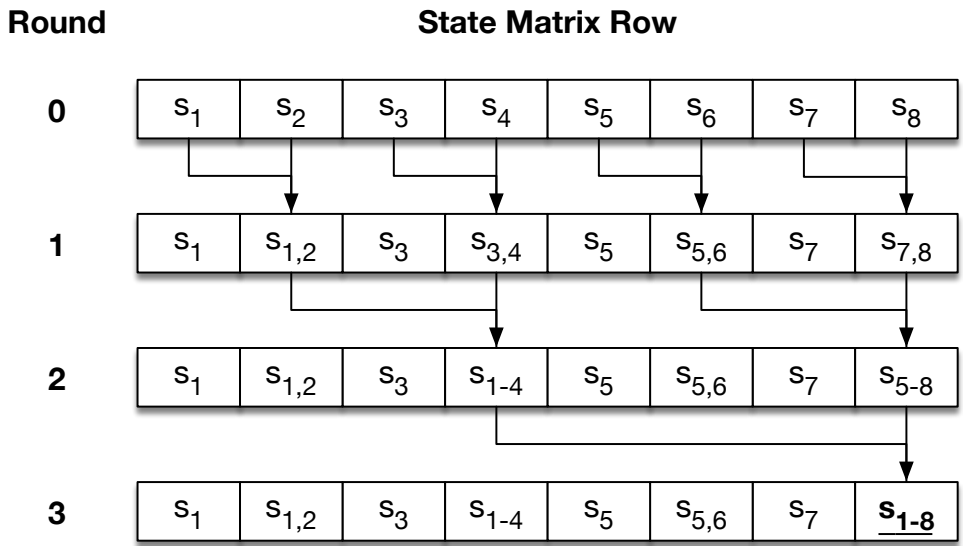


Figure 5.4: Parallel Query State Aggregation

5.2 Parallel GiST Traversals with CUDA

The tree traversal algorithms have to be mapped to a parallel programming model in order to compile it for parallel execution hardware. The algorithms that were prototypically implemented for this thesis use the CUDA model. This mapping, including some tuning techniques for utilizing caches provided by the hardware, are presented in this section.

The competing, vendor-independent OpenCL standard is supported by much more device types, such as many-core CPUs, GPUs from other vendors, or FPGAs. Although it has originally been designed for GPGPU on Nvidia devices only, CUDA has been selected, because Nvidia hardware was available for conducting experiments. However, as discussed in Section 2.4.5, mapping the algorithms from CUDA to OpenCL is possible without major adaptions of the code, because both specifications share a lot of similarities.

5.2.1 Host Execution Model

The host thread orchestrates kernel invocations for each tree traversal iteration and merges query results for the next round. Further, it must assert that all input data is available in global device memory, by injecting data transfer operations if necessary and copying results back to the host. The main logic is depicted in Algorithm 5.5.[3]

[3] The algorithm shows sequential processing of kernels and corresponding data transfers. Overlapping their execution phases for implementing a streaming approach, like discussed in Section 2.2.2, is not considered here. Such a throughput optimization can be implemented using several CUDA streams per device, but is left for future work.

Algorithm 5.5 Host Thread

Input: queryProcessor, device
Output: published query results in query processor

```
1: while queryBatch = queryProcessor.waitForNextBatch( device.getID() ) do
2:    device.copyInputData( queryBatch.getQueryData() )
3:    device.runNodeScanKernel()
4:    queryResults = device.copyResultData()
5:    queryProcessor.mergeQueryResults( queryBatch, queryResults )
6: end while
```

Algorithm 5.6 Query Result Merge

Input: queryBatch, queryResultBatch
Output: updated shared query information in the query processor

```
1: for qry ∈ queryBatch do
2:    queryStates[ qry ].mergeAtomic( queryResultBatch.getQueryState( qry ) )
3: end for
4: if queryBatch.getNode().isInnerNode() then
5:    for (node, queryList) ∈ queryResultBatch.getMatchingEntries() do
6:       if not queryList.empty() then
7:          queryQueues[ node ].insertAtomic( queryList )
8:       end if
9:    end for
10: else
11:    for (qry, nodeList) ∈ queryResultBatch.getMatchingEntries() do
12:       if not nodeList.empty() then
13:          queryResults[ qry ].insertAtomic( nodeList )
14:       end if
15:    end for
16: end if
17: for qry ∈ queryBatch do
18:    nodesLeft = -1
19:    if queryBatch.getNode().isInnerNode() then
20:       nodesLeft += queryResults.getNumMatchingNodesForQuery( qry )
21:    end if
22:    nodesLeft = nodesToProcess[ qry ].incrementAtomic( nodesLeft )
23:    if nodesLeft = 0 then
24:       finalizeQueryResults( queryResults[ qry ], queryStates[ qry ] )
25:    end if
26: end for
```

In order to implement the host logic, a proactor pattern [180] has been applied, because it separates dispatching of tasks (primitive scheduling within the query processor) from their asynchronous processing via task handlers and, thus, allows to provide device-specific implementations. Therefore, a host thread exists for each available processing device, which is registered to the scheduler. Once the query processor schedules a (node, query batch)-primitive to one of the available devices,[4] the host thread will be notified via its queue and starts processing it (line 1).

For processing a task, device-specific data transfer operations are injected via the concrete device adapter (lines 2 and 4) and the kernel is scheduled (line 3). Because these steps depend upon each other and CUDA uses asynchronous communication with device drivers, these methods also include necessary synchronization operations via CUDA events. After query results, i. e., the filtered candidate lists of child node entries for the next round as well as the updated state information in case of stateful queries, have been received from the device (line 4), they need to be merged with the global query states in the query processor (line 5).

The result merge algorithm is depicted in Algorithm 5.6. First, the global query states are updated (lines 1–3).[5] Each state update needs to be synchronized properly, e. g., via latches or a latch-free concurrent data structure implementation, because they might be accessed in parallel by another host thread. In case of a concurrent read, accessing the more accurate state information would be better, because the pruning step can be applied with less false positive matches. Reading the old state version is not an issue, because the final result processing will filter false positives later (line 24).

Second, matching node entries have to be processed (lines 4–16). In case an inner node was scanned, the result is a list of matching queries for each child node entry. These lists are inserted into the node queues for subsequent processing in the next iteration, where they might be scheduled to another processor device (lines 4–9). In case a leaf node was scanned, the result is a list of matching entries for each query that need to be inserted into the queries' result sets (lines 10–16). Again, these queue / list updates need to be serialized to prevent race conditions.

Finally, some bookkeeping needs to be performed in order to finalize query results and properly terminate tree traversals (lines 17–26). For each query, it needs to be tracked how many nodes remain in the traversal backlog, because the query might be filtered before reaching any leaf node, yielding no matching results at all. This tracking needs to be done separately, because query queues are organized per node in order to simplify scheduling. In the merge step, the number of active nodes decreases by one, because one node has been processed by this thread (line 18). For leaf nodes, the traversal terminates in this branch of the index tree. However, others might still be in the backlog. For inner nodes, the traversal continues with matching child nodes that passed the pruning step. Thus, the number of remaining nodes needs to be incremented by the match counts (lines 19–21). These counts are aggregated in a separate vector during the scan, after evaluating the pruning predicates. The increment is applied atomically to

[4] The scheduling itself is subject of the next chapter.

[5] The state update might be parallelized for each query. Further, if a state merge is computationally intensive, offloading that part to a coprocessor might also be beneficial. But none of these optimization has been applied so far.

synchronize with other result merges (line 22), and only if the current merge was the last one for this query (line 23), the result set is finalized (line 24). Because the node counter is updated after all shared data has already been published, this check guarantees that the finalization is applied exactly once with all result candidates provided. During the result finalization phase, the pruning predicate needs to be evaluated with the final query state in order to filter intermediate false positives, before query results can be published via the result sets.[6]

5.2.2 Device Execution Model

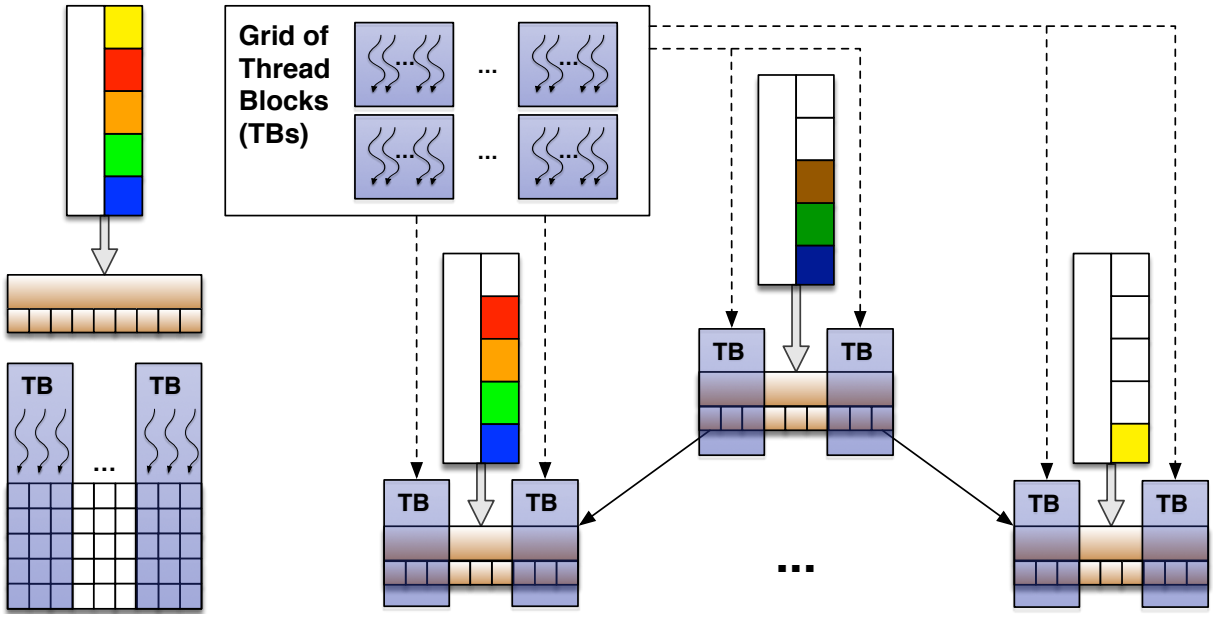


Figure 5.5: Mapping of Index Node Scan Primitive to CUDA Processing Model

The mapping of the node scan primitives to the CUDA processing model is illustrated in Figure 5.5. The predicate evaluation logic for a single query is independent for all keys and can be processed using a SIMD model. Thus, the processing of matrix cells can be mapped to CUDA *Threads*. Logical dependencies between key predicate evaluations, e. g., for stateful queries, are resolved in the previously described synchronized post-processing step after a whole batch has been processed. Physical data dependencies for accessing the same key and query predicates when processing a single batch are resolved by synchronized data accesses and will be utilized for caching, which is discussed in the next section.

The evaluation of a (node, query)-batch, i. e., the whole node-scan matrix, can be mapped to at least one *Thread Block* (TB). Each block is responsible for processing a partition of the matrix, having one column for each node key and one row for each query predicate, which is illustrated

[6] For stateless queries, results might be published prematurely, because there cannot be false positives due to inaccurate state information.

on the left hand side in Figure 5.5. The matrix should be partitioned in a way that each partition covers a number of cells that is an integer multiple of the device's physical scheduling unit size, i. e., the warp size, in order to avoid that any thread processors idle during the execution.

The number of blocks that are used for processing a single scan matrix is a parameter that can be used for fine-tuning the algorithms for a specific hardware platform. Increasing the number of blocks makes sense for devices having a lot of available cores in case the whole processing is compute-bound. But increasing the block count also increases pressure on the device's main memory bus for accessing data elements, because separate thread blocks might be scheduled to separate multi-processors on the device and, thus, provide no means to synchronize data accesses among each other. Further, each additional thread block causes some overhead for scheduling and post-processing. Another fine-tuning parameter is the number of matrix cells that are processed by a single thread. Increasing this parameter increases the unit of work that is executed serially per thread, which might be useful for algorithm instances that are bound by the main memory access bandwidth on the device. The more cells are processed by a single thread, the less threads and thread blocks are required for processing all cells within the matrix. For large node and batch sizes in particular, this may significantly reduce data element accesses after the data has been cached. For simplicity, it is assumed that each thread processes exactly on cell in the following.

All node scans can be processed independently from each other. This perfectly fits the CUDA *Grid* model, which is illustrated on the right hand side in Figure 5.5. A grid is executed by one node scan *Kernel* instance and represents a single iteration that advances all queries within the grid by one layer in their overall tree traversal. The main steps of the kernel algorithm are sketched in Algorithm 5.7. The kernel function is invoked with pointers to device memory buffers, where input and result data is stored as large contiguous arrays (cf. Section 5.2.4). Further, it gets built-in values for the thread and block IDs within the grid that can be used to calculate the unique *Thread ID*, which is used to calculate offsets for the data elements that shall be processed by the kernel instance (line 1).

This ID is used next to fetch corresponding input data from global device memory into shared on-chip memory (lines 2–6). Since node and query data is cached on the device, the actual input offsets for kernel data transfers need to be calculated indirectly via a descriptor entry that is passed as row in a node scan table (line 2). This table is prepared in another kernel before the actual scan kernel is invoked.[7] The data elements are fetched in parallel, using coalesced reads for each structure as illustrated in Figure 5.6. Each thread fetches some bytes of the data elements at pre-defined offsets, so that a consecutive range of data is read from memory in order to maximize bus transfer throughputs. The actual interpretation of the structure is done later by casting the raw byte array. Coalesced writes for storing result data are handled in the opposite way. Node data can be fetched with a single coalesced read, because all keys are stored as arrays and only a single node is required for a scan kernel. For a single query, multiple read calls might be required, depending on its key and state structure layout. For fetching all queries in the batch, a gather operation is needed, i. e., multiple read transactions with different offsets are issued, because each query might be buffered at a different location on the device.

[7] Details on descriptor entries of the scan table are discussed in Section 5.2.4, when caching is explained in more detail.

Algorithm 5.7 Node Scan Kernel (Stateless Queries)

Input: nodeScanTable, nodeBuffer, queryBuffer, resultBuffer, predicateFn
Output: Updated search space pruning results in resultBuffer.

```
1:  TID = ( threadIdx, blockDim, blockIdx, gridDim )
2:  shared desc = fetchDescriptor( nodeScanTable, TID )
3:  shared qryVec = fetchQueryData( queryBuffer, desc, TID )
4:  shared keyVec = fetchData( nodeBuffer[ desc.nodeID ], desc.numKeys, TID )
5:  shared matrix = allocateResultMatrix( desc.numQueries, desc.numKeys )
6:  __sync()
7:  qryIdx = calcMatrixRow( desc.numQueries, desc.numKeys, TID )
8:  keyIdx = calcMatrixCol( desc.numQueries, desc.numKeys, TID )
9:  resultIdx = calcMatrixCell( desc.numQueries, desc.numKeys, TID )
10: if resultIdx > 0 then
11:     matrix[ resultIdx ] = predicateFn( keyVec[ keyIdx ], qryVec[ qryIdx ] )
12: end if
13: __sync()
14: compressResultData( matrix, TID )
15: __sync()
16: writeData( resultBuffer[ desc.resultOffset ], matrix, TID )
```

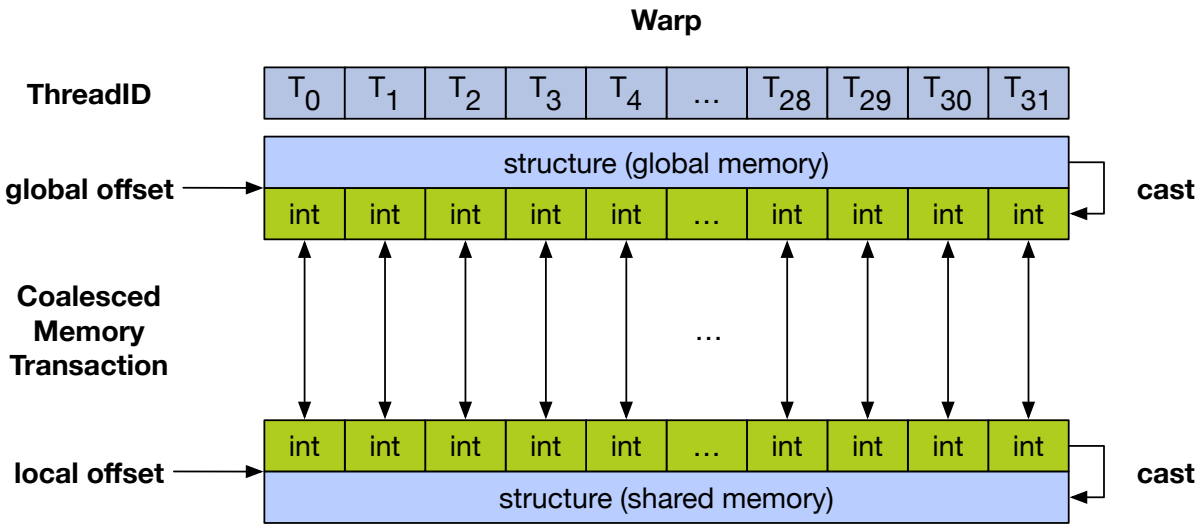


Figure 5.6: Coalesced Memory Transactions between Global and Shared Memory

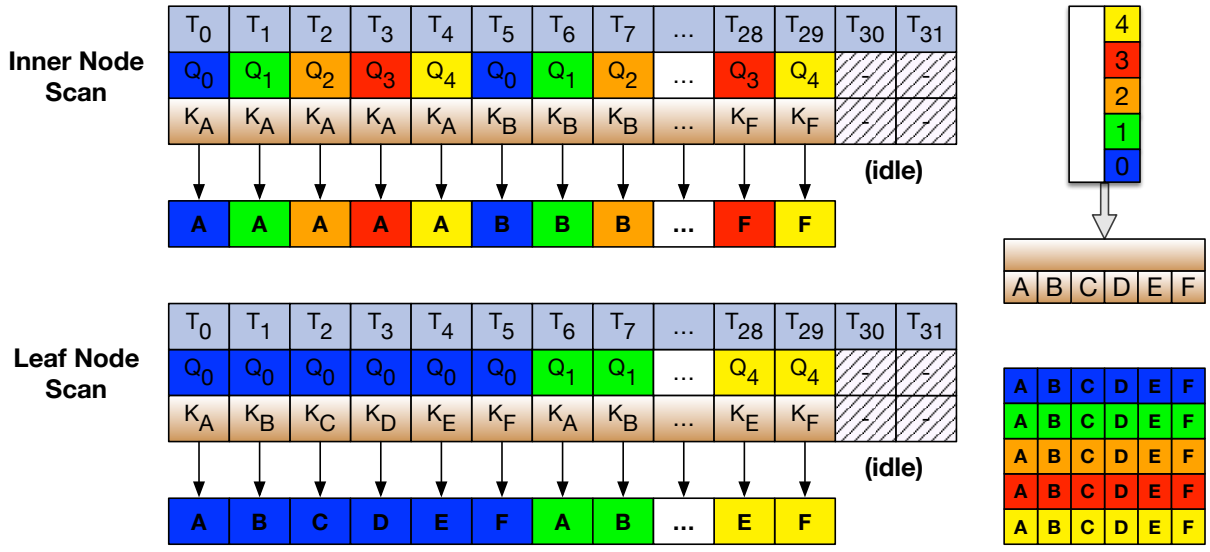


Figure 5.7: Result Matrix Offsets for Inner and Leaf Node Scans

Next, the scan matrix is pre-allocated in a local temporary buffer (line 5), because it has to be post-processed later, before results are written back to the device memory. After these setup steps, all threads need to synchronize before they can continue to evaluate the pruning predicates in parallel (lines 7–13). Based on the node type, i. e., whether it is an inner node or a leaf node, and the *Thread ID*, some offsets need to be calculated for reading the elements that should be processed by the current thread and for storing predicate evaluation results (lines 7–9). Because there might be more threads in the block than data elements to be processed, e. g., for cases where there the number of cells is not an integer multiple of the *Warp* size, only those threads participate in the predicate evaluation that have not been deactivated by a special, negative offset value (line 10). The results of the pruning predicates are stored in the matrix (line 11) and all threads need to synchronize before the matrix can be post-processed (line 14) and written back to the result buffer (line 16).

The result offsets within the matrix are calculated like illustrated in Figure 5.7. For inner nodes, a key-major layout in the linearly addressed buffer is used (column-major matrix), while a query-major layout is preferred for leaf nodes (row-major matrix). The reason for that is that consecutive arrays of elements are needed in memory for the result post-processing step. Therefor, the matrix needs to be scanned again in parallel in order to produce vectors of matching queries for each inner node and vectors of matching key entries for each leaf node, before they can be copied back into the global memory using coalesced write transactions (line 16). These vectors might be sparse, i. e., they have gaps for all entries whose corresponding matrix cells did not pass the pruning predicate evaluation. Therefore, a stream compression algorithm, based on a parallel prefix sum primitive [83], is used to remove all these gaps.

The compression algorithm is illustrated in Figure 5.8. For each entry in the vector that needs to be compressed, an integer value representing the predicate evaluation result is stored (`1` for `true`, `0` for `false`). The parallel prefix sum calculates the sum over those numbers up to

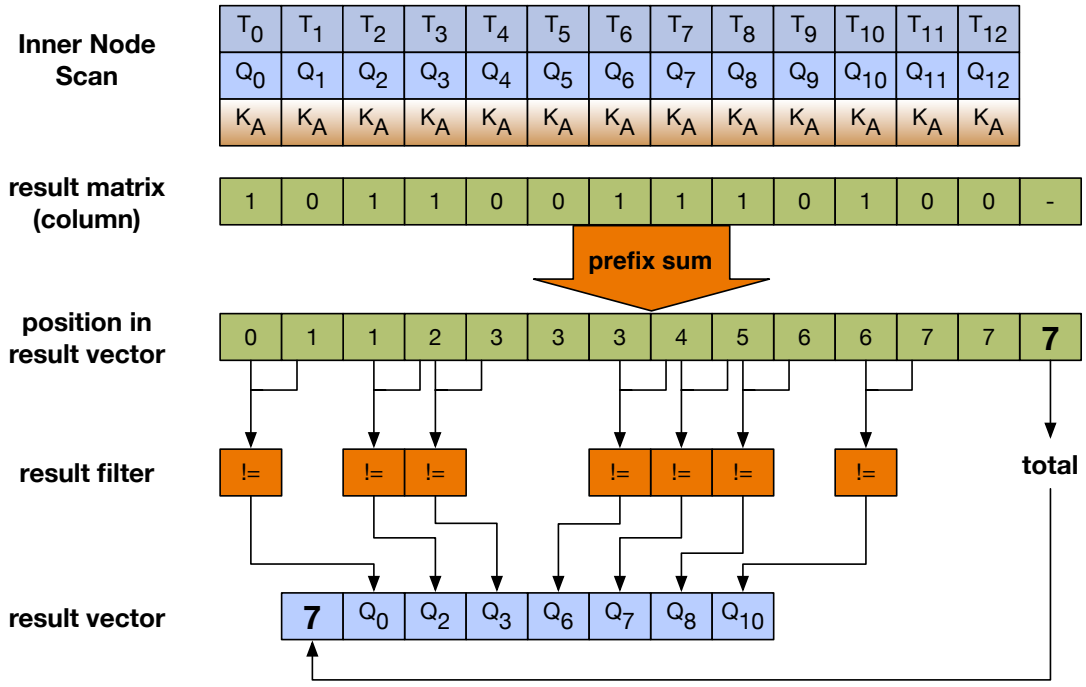


Figure 5.8: Result Post-Processing (Inner Node)

that position in the vector by applying a pattern that is similar to the parallel reduction, which has been described in Section 5.1.3. The main difference here is that, after the initial reduce phase, the aggregates are also propagated downwards in the computational tree so that all vector entries are updated. The prefix sum number indicates the offset of the element in the result array. Afterwards, neighboring elements are compared with each other and only those entries are copied into the result vector, where the following number differs from the current one. These elements only differ when the predicate evaluation returned `true`.

As its last element, the prefix sum also yields the total number of matches, which is required to track the number of active nodes that still need to be traversed for a query. Note that the `total` in Figure 5.8 denotes the total number of queries for node having key K_A. In order to get the number of nodes for each query Q_i, the prefix sum needs to be calculated again, now over each matrix column. Although these matrix accesses are not coalesced for the second calculation, this is not an issue, because random element access is more efficient in the shared memory buffer that is used to store intermediate results (cf. Section 5.2.4). Coalesced writes are only required when the node count vector is written back to global memory afterwards.

The discussion of stateful queries has been omitted here, because the processing pattern is similar to the one described above. First, state information has to be fetched into an intermediate buffer using coalesced reads, local states need to be calculated for each key using the state function, and the result needs to be aggregated for each query using one of the previously discussed reduce methods, before the updated state can be passed as additional argument for predicate evaluation. Finally, the vector of updated state information needs to be copied back to device memory using coalesced writes.

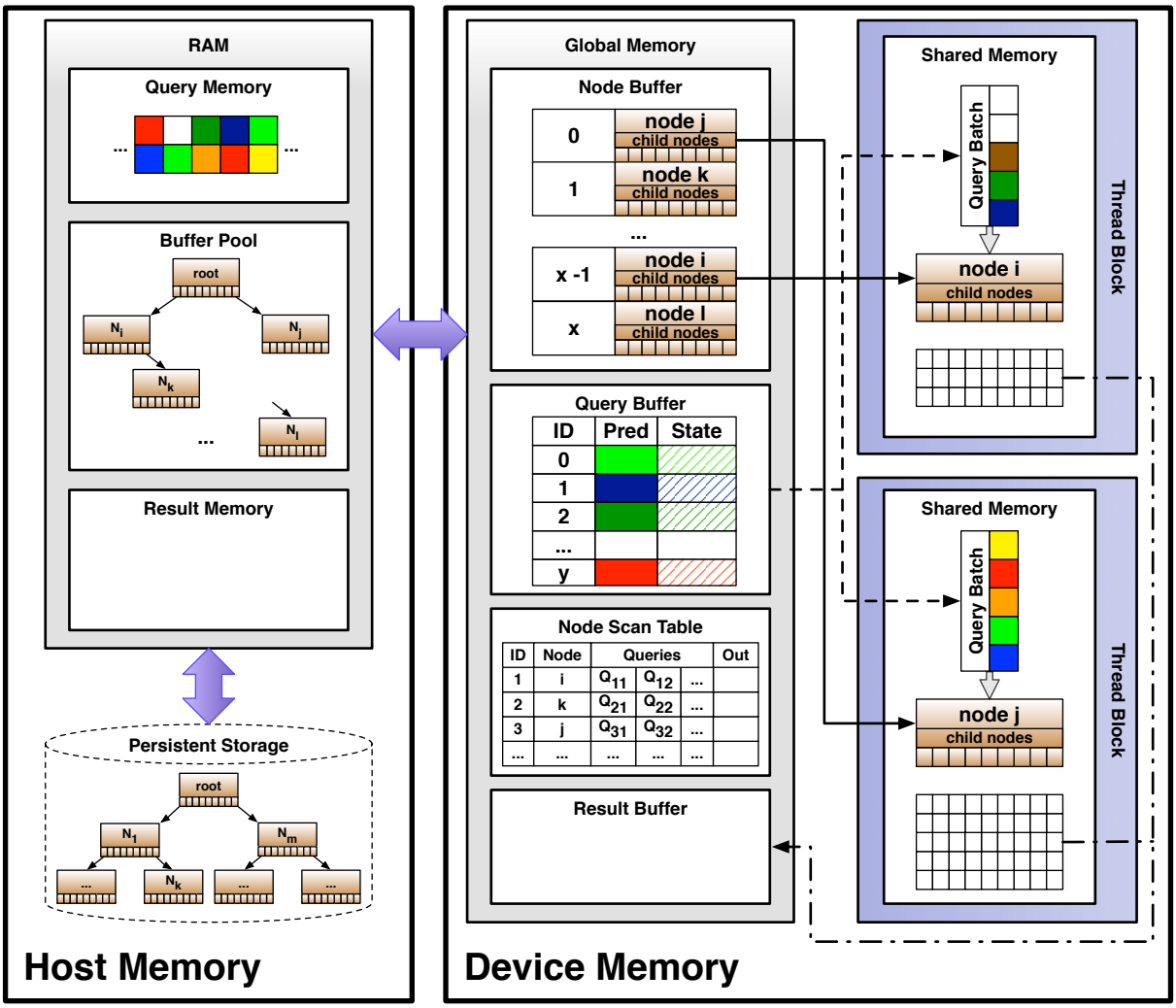


Figure 5.9: Index Data Management on Host and Device

5.2.3 Host Memory Management

The host and device memory management was designed in a similar way like buffer pools in DBMSs. This out-of-core technique has been selected, because GPU memories are usually orders of magnitude smaller than main memories on the host. This design also facilitates potential device sharing for other tasks when not all resources are reserved exclusively. For the host side, an assumption for now is that the tree is memory-resident, or at least most parts of it. This assumption is usually true in real-world applications (cf. "B-trees vs. hash indexes" in [75]) and prevents that external storage transfer costs dominate the process, hiding benefits of the additional computing power.

As shown in Figure 5.9, a buffer pool of tree nodes is stored on the host side, reusing the GiST implementation for synchronizing tree data with their copies on persistent storage. Since the host is responsible for copying node and query data to device buffers, it also needs to maintain

some meta information about what data is already buffered in order to omit unnecessary transfer operations.[8] Buffered nodes can be reused for pipelined processing, which is particularly useful for upper tree layers near the root, because they are accessed more frequently than lower ones, where queries "spread" over the search space. Data for query predicates can be reused between subsequent scan iterations. Caching effects increase if predicates have a high selectivity, replicating themselves for many child nodes per layer.

5.2.4 Device Memory Management

When a node scan kernel launches, it needs to know which data shall be processed and where its results should be stored that can be fetched by the host thread afterwards. In order to separate data transfers from kernel execution and implement data buffering for subsequent scan iterations, the actual node and query data, i. e., predicate keys as well as state information, is copied to dedicated device buffers during the input copy phase. Data that is already available on the device can be skipped, which reduces transfer overheads via the host $\leftrightarrow$ device interconnect. Device buffer management is subject of Section 5.2.4.1.

A scan primitive, which represents the kernel parameter, does not convey any user data, but merely comprises a (node ID, query ID vector)-pair that needs to be translated into buffer offsets where kernel threads can access it. Each device buffer needs to be able to arbitrarily evict unused data entries for replacing them with new ones. Therefore, buffer entries are stored without any particular order that could be used to derive such offsets implicitly. Thus, it is required to explicitly calculate offsets for all kernel blocks, which is implemented in a parallel scan preparation phase that is discussed in Section 5.2.4.2.

When data is processed by the kernel, it needs to be transferred from global device memory to each core via the main memory bus on the device. Compared to on-chip data accesses, such transfers are quite slow. Further, a special data layout is required for coalesced memory transactions, which does not necessarily match the kernel's data-parallel access patterns in all phases. In order to optimize global memory accesses, the on-chip shared memory region can be used as intermediate buffer, which discussed in more detail in Section 5.2.4.3.

5.2.4.1 Device Buffers

The main memory buffers on the device are illustrated in Figure 5.10.[9] They are managed as arrays of fixed-size entries that are implicitly addressed by element index. Variable-sized data, such as a varying number of node keys or queries inside a query batch, is handled by padding the

[8] Updating node data is not in the focus of this thesis. In case, it should be implemented later, this would also be the place where device caches need to be invalidated and refreshed.

[9] The rationale behind this design is discussed in the following. There might be other implementations that require other mechanisms in order to achieve efficient memory access patterns.

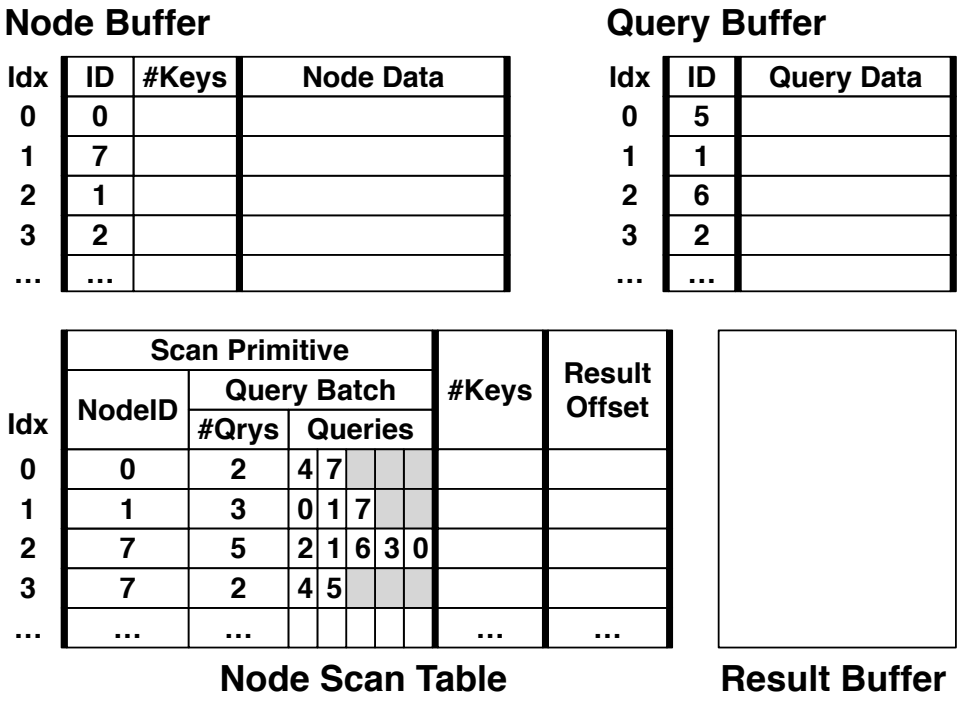


Figure 5.10: Device Buffers

structures to a pre-configured maximum size and storing the actual number of entries separately. Each structure is assumed to be contiguous in memory in order to coalesce data accesses. Each buffer is split into multiple arrays, which are separated via thick borders in Figure 5.10. Each array is required for specific operations that do not need the other data elements. Using such a "columnar" SoA memory layout (cf. Section 3.1.5) for buffer tables allows vectorized data access, without transferring unused entries via the memory bus system .

The *Node Buffer* and *Query Buffer* store data entries and ID values separately, because the actual data is just required by the scan kernels, while IDs represent header information that is merely required for translating scan primitives into buffer offsets. For host $\leftrightarrow$ device transfers, this layout requires multiple transfer operations, one for each entry that is copied, and one for synchronizing the ID header. As later experiments will show, executing many small-sized transfers are inefficient, due to the overheads that are involved (cf. Section 5.3.5). This overhead can, somehow, be mitigated by managing host data in pinned memory regions and overlapping many asynchronous copy operations. But a general advice is to reduce the number of transfers while increasing their chunk size. Thus, transferring many IDs at once for synchronizing the header leads to better bus utilizations, compared to many random access transfers. Node entries are assumed to be quite large, because key arrays are also persisted on disk, which requires larger chunks for efficient disk I/O operations in the buffer pool, anyway. Thus, transferring nodes separately should suffice to utilize the available host $\leftrightarrow$ device memory bandwidth. Query entries, however, just comprise information of a single predicate key and, potentially, some state data. These structures are, usually, much smaller than index nodes and might require a different buffering strategy. But for simplicity, the current implementation just re-uses the same mechanisms that have been implemented for the node buffer.

Scan primitives are stored as array in a *Node Scan Table*. They are transferred from the host to the device via vectorized data copies. Additional data that is required to fully parameterize a scan kernel block is stored in separate arrays. They are populated directly on the device during the scan preparation phase, which is discussed in the next section. Each row in the table is called *Descriptor* in the following.

Each kernel needs to store its scan results in a *Result Buffer*, which is organized as one large pre-allocated memory region. Depending on the number of keys and queries in a primitive, each kernel yields a different number of result cells in the scan matrix. This needs to be handled before the kernel is launched, in order to partition the buffer into disjoint regions, where each thread can materialize its result. Because there are no synchronization mechanisms between threads of different blocks, there is no simple way to calculate result offsets dynamically.

A first strategy is to manage the result buffer as statically pre-partitioned memory region, assuming that all matrices are of the same, maximum possible size of the maximum number of keys per node $\times$ the maximum number of queries per batch. Each scan matrix can be implicitly addressed by its descriptor position inside the table. Using this scheme requires no overheads for calculating result addresses, but might lead to wasted data transfer bandwidth, because irrelevant cells are transferred with a single result transfer operation that did not have a corresponding input (query, node key)-pair.

A second strategy dynamically structures the result buffer during the preparation phase. Therefore, the number of keys within a node is needed as separate array inside the scan table in order to compute offsets where a kernel should store its predicate evaluation results, relative to the other blocks in the grid. This approach requires two result transfers, one for the offset vector and another one for the result block. If there are many cells that do not contribute any results, this approach can significantly reduce the transferred data volume for the overall result set.

The second version is selected in the following, because the host $\leftrightarrow$ device interconnect is assumed to be a bottleneck in the processing pipeline, which is a frequently discussed critical point in several, many-cited publications [79, 128].

5.2.4.2 Scan Preparation

The scan preparation phase is sketched in Algorithm 5.8 and Figure 5.11. A separate kernel is required for this, because descriptors need to be prepared as parameter before the scan can be executed and there are no synchronization mechanisms between kernels.

The prescan kernel mainly scans through the ID vectors stored in each descriptor and replaces each node and query ID within corresponding offsets in the data buffers. The first operation calculates the unique thread ID that is address the data-parallel units of work (line 1). The partitioning that is applied is illustrated in Figure 5.11. Each thread block is responsible for a horizontal scan table partition, i. e., a set of descriptor rows, which are fetched with coalesced reads (lines 2–4).

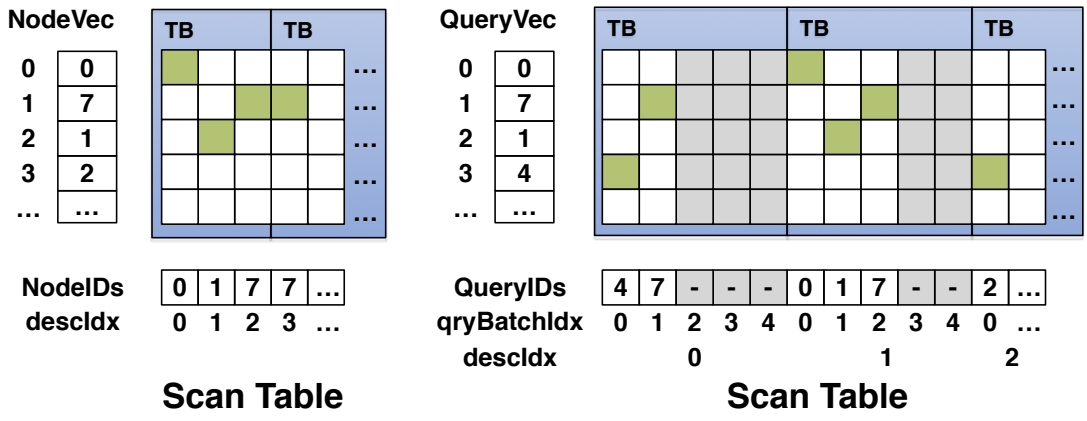


Figure 5.11: Scan Preparation Phase

Algorithm 5.8 Scan Preparation

Input: nodeScanTable, nodeBuffer, nodeBufferSize, qryBuffer, qryBufferSize
Output: Updated offsets in nodeScanTable for scan kernel.

```
1:  TID = ( threadIdx, blockDim, blockIdx, gridDim )
2:  numDescriptors = calculateDescriptorsPerBlock( TID )
3:  shared descVec = fetchDescriptors( nodeScanTable, numDescriptors, TID )
4:  __sync()
5:  shared nodeVec = fetchData( nodeBuffer, nodeBufferSize, TID )
6:  __sync()
7:  nodeIdx = calcMatrixRow( nodeBufferSize, numDescriptors, TID )
8:  descIdx = calcMatrixCol( nodeBufferSize, numDescriptors, TID )
9:  if descVec[ nodeDescIdx ].nodeID == nodeVec[ nodeIdx ].nodeID then
10:     descVec[ descIdx ].nodeID = nodeIdx
11:     numKeys = nodeVec[ nodeIdx ].numKeys
12:     numQueries = descVec[ nodeDescIdx ].numQueries
13:     descVec[ descIdx ].numKeys = numKeys
14:     descVec[ descIdx ].resultOffset = numQueries × numKeys
15: end if
16: shared qryIDs = fetchData( qryBuffer.IDs, qryBufferSize, TID )
17: __sync()
18: (numQrys, descIdx, qryBatchIdx) = calcQryParms( qryIDs, descVec, TID )
19: qryIdx = calcMatrixRow( qryBufferSize, numQrys, TID )
20: if descVec[ descIdx ].queries[ qryBatchIdx ] == qryIDs[ qryIdx ] then
21:     descVec[ descIdx ].queries[ qryBatchIdx ] = qryIdx
22: end if
23: __sync()
24: writeDescriptors( nodeScanTable, descriptors, TID )
```

Node IDs and query IDs could be translated independently in parallel. However, in Algorithm 5.8, these phases are processed sequentially, node ID translation in lines 5–15 and query ID translation in lines 16–22. Using sequential processing here increases locality on the descriptor rows, which do only need to be fetched, updated, and written back to global memory once by a single kernel. Separating these phases would require a different memory layout that separates node ID storage from query vectors in order to avoid conflicting descriptor row updates when translated IDs are written back to global memory (line 24). Further, such a separate layout would require multiple input transfers for scan primitives as well as separate handling of node IDs and query batches on the host. Thus, the first, sequential version has been implemented. It can use coalesced memory transactions, without transferring irrelevant data during the input copy and allows to materialize updated descriptors en bloc, without additional kernel synchronization.

The ID-to-offset-translation phases are similar for both ID types. First, the vector of IDs is fetched from corresponding device buffers (lines 5–6 / 16–17). Second, each entry is compared to the ID specified in the descriptor (lines 9 / 20) and the offset is written back by the thread that found a match (lines 10 / 21). The offset replaces the original ID that is not required anymore after this point in time. Assuming that each ID is used only once inside the buffer, there is exactly one thread that executes this write operation and no write conflicts occur. The ID vector comparisons can, again, be interpreted as matrices as illustrated in Figure 5.11. Each thread inside a block is responsible for one cell, whose row and column indices can be calculated like in the scan kernel (lines 7–8 / 18–19).

The difference between processing node IDs and query IDs is the calculation of the column index, because a query batch might comprise empty entries if it is not fully populated (shaded grey in Figure 5.11). Node ID vectors might only have empty entries if not all descriptor rows are used, e. g., for the last thread block. Threads that are responsible for empty cells can either be disabled, e. g., by writing an invalid offset, like `-1`, into the vector, or they are filtered upfront so that no empty cells exist. The strategy for implementing this is represented by the function in line 18. Disabling threads wastes processor resources during branching, but simplifies index calculations, because row and column indices can be directly derived from the thread ID and the maximum number of queries. Avoiding idle threads can be achieved by calculating a prefix sum over the number of queries, so that each thread can derive its offset afterwards. Only the first version has been implemented in the prototype for this thesis, because the algorithms are optimized for large batches, where only a low number of idle threads are expected and less overhead is required for the additional prefix sum.

If the result compression strategy is implemented (cf. Section 5.2.4.1), a descriptor's result offset can be initialized with the total number cells in the scan matrix, i. e., the actual number of queries in the batch $\times$ the number of keys in the node. Result offsets and the number of keys do only need to be written once for each node. Thus, they are materialized by the thread having a matching node ID (lines 11–14). In order to calculate the final result offsets for each descriptor, a separate prefix sum needs to be calculated over the initial vector in a subsequent kernel.

5.2.4.3 Shared Memory Buffering

Threads in the prescan and scan kernels access data entries multiple times in order to evaluate the matrices that are spanned by the input vectors. Therefore, it is useful to cache them inside the on-chip shared memory, which has significantly lower access latencies, compared to global device memory. Input vectors are cached initially by each thread block before threads start operating on them, followed by an internal thread synchronization primitive that acts as barrier.

The block-local shared memory regions can be pre-allocated for each kernel, using the maximum vector size, which is known in advance and can be statically configured for each device. While the number of keys inside a node is a static property of the index and, usually, configured for optimized buffer pool I/O, it is assumed to be fixed and represents the upper bound for the node vector size. However, key vectors can be partitioned, like discussed in Section 5.2.2, and processed by multiple thread blocks. But in the following, one block per node is assumed for initial performance evaluations. The number of queries per block can be configured arbitrarily per device. The upper bound for both parameters is determined by the maximum amount of shared memory that is offered by the device type, which can be queried via the device driver.

Since matrix cells have to be post-processed in the scan kernel, all entries need to fit into the temporary shared memory buffer. Directly storing each cell in global device memory would lead to additional memory transactions, which should be avoided for latency and bandwidth reasons. The latter is particularly important, because the matrices are accessed with different patterns (row-wise vs. column-wise) that would lead to non-coalesced read and write transactions. The shared memory minimizes such penalties, because it shows better behavior for random access. Therefore, it can be used as staging area between coalesced global device memory accesses.

5.3 Evaluation – Parallel Search Tree Traversals on CPUs and GPUs

The algorithms presented in the previous sections have been implemented with CUDA for GPUs and as C++ code for CPUs.[10] The objective of this section is to asses under which conditions using a parallelized GPU search outperforms the serial CPU counterpart and vice versa, while considering potential overheads, e. g., for data transfers. Therefore, a micro-benchmark has been implemented, which varies important input parameters that describe a scan primitive, executes the algorithm for both platforms, and compares execution times with each other. The experiments have been conducted on 2011 and 2015 hardware in order to evaluate the impacts of recent hardware changes, thereby verifying the applicability of the framework's hardware abstraction layer.

[10] The CPU algorithm is not in the focus of this thesis. Here, the primitives have been simply implemented like in the original GiST library [2] for unordered domains, i. e., as a plain function that uses nested loops to iterate over scan matrix elements and processes them sequentially. Compared to GiST, just another loop for each query in the batch has been added.

After the benchmark has been described in more detail in Section 5.3.1, potential speedups of GPU over CPU execution for different workloads are discussed in Section 5.3.2. Stateless range queries and stateful nearest neighbor searches are analyzed in this section. Next, query throughputs for different workload parameters are analyzed in Section 5.3.3 in order to determine optimal configurations for executing stateless queries on each processor type. Section 5.3.4 analyzes the framework internally by evaluating different execution phases and their contribution to the overall query processing time to identify future tuning potential. Finally, Section 5.3.5 covers an experiment that analyzes data transfers between a host system and an attached GPU device, because this phase consumes significant portions of the overall runtime for certain workloads.

5.3.1 Setup – Node Scan Benchmark

For evaluating the potential benefits of a vectorized GPU-based search over a CPU-based implementation, a micro-benchmark has been developed, which explores the relevant parameter space for a node scan primitive. An overview of the benchmark, its input parameters, and all its phases that are tracked is illustrated in Figure 5.12. For each parameter combination, a scan primitive is generated and executed on the available devices. The common initialization, i. e., allocation of memory buffers, etc. is not measured. The benchmark simply assumes that all node and query data is already present in host memory. Execution times for all phases are measured, including any algorithmic overheads for the initial scan preparation phase as well as the final merge of scan results. In order to gain a realistic picture, a cold-cache setup is used, i. e., data transfer times for all host $\leftrightarrow$ device copy operations are included, which already proved to be relevant in many other places [79, 128]. In order to compare CPU and GPU execution with each other, a single *GPU-speedup* metric s is calculated, which is defined as $s = \frac{t_{CPU}}{t_{GPU}}$, i. e., the ratio of the total execution times that have been measured for each device type. For $s < 1$, the CPU algorithm was faster by factor $\frac{1}{s}$ and for $s > 1$, the GPU implementation outperformed the CPU by factor s. In case $s \approx 1$, both algorithms are on par, which represents a break-even point for guiding scheduling decisions that are subject of Chapter 6.

Important dimensions for parameterizing a single scan primitive are:

- the number of keys per node (*slots*)
- the number of query predicates in a search batch (*queries per task*)
- the size of a single slot
- the size of a single query predicate
- the *selectivity* of each query predicate in the batch
- the *correlation* between the queries in the batch

	Model	Year	Cores	Proc. Freq.	Mem. Bandwidth	
CPU	Intel Xeon X5690	2011	6	3.4 – 3.7 GHz	RAM	25 GB/s
CPU	Intel Core i7-6700	2015	4	3.4 – 4.0 GHz	RAM	34 GB/s
GPU	Nvidia Tesla C2050	2011	448	1.15 GHz	RAM	144 GB/s
					PCIe2	5.7 GB/s
GPU	Nvidia GeForce GTX TITAN	2015	2688	0.88 GHz	RAM	480 GB/s
					PCIe3	11.6 GB/s

Table 5.1: Hardware Specification for Microbenchmarks

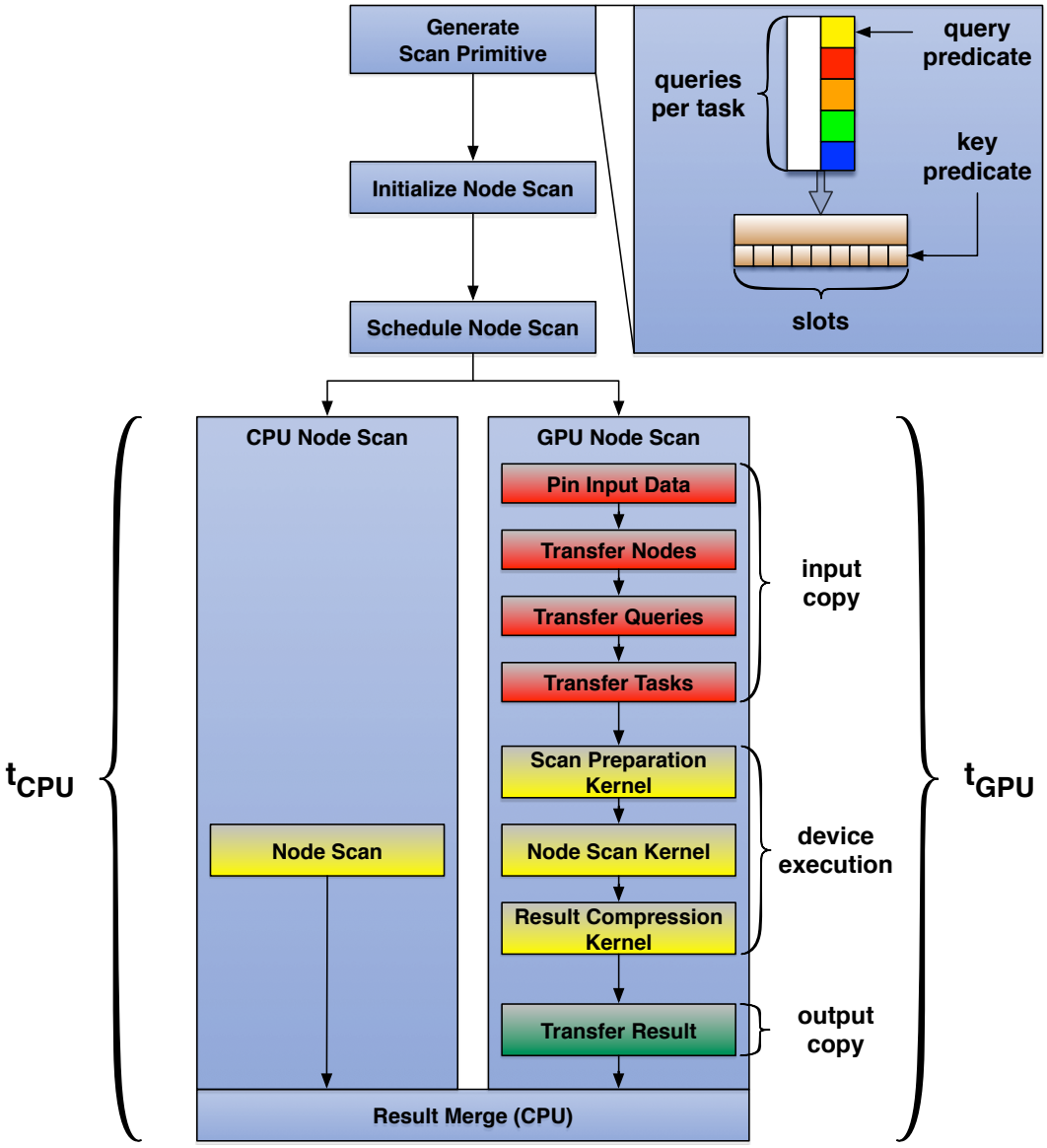


Figure 5.12: Overview Node Scan Benchmark

The first two parameters directly impact the parallelization degree by spanning the matrix of independent predicate evaluation cells. Parameters describing the key type and supported predicates depend on the actual index implementation. For the prototypical evaluation in the following sections, a three-dimensional R-Tree has been implemented with the framework [110, 137]. That is, each node key is represented by two 3D-point coordinates, which encode the minimum and maximum corners of the MBR that completely covers the underlying search vector space.

For stateless queries in Section 5.3.2.1, a range query was implemented that is encoded as MBR. In this case, the predicate evaluation simply represents an MBR intersection test. For stateful queries in Section 5.3.2.2, a nearest neighbor search was implemented that is encoded by the 3D-coordinates of a single query point. For evaluating the query, a state-of-the-art branch-and-bound algorithm for nearest neighbor searches in R-Trees [173] was implemented, using the Euclidean distance function. As query state, the algorithm aggregates the minimum distance between a candidate entry and the given query point, i. e., the actual distance for a leaf node or the minimum distance to the borders of an MBR for inner nodes. Second, a distance is calculated that represents the maximum distance between a point within an MBR and the query point, where at least one entry must exist, because the MBR would have been resized otherwise. The state-based pruning calculates the distances between all candidates, aggregates them, updates the state with refined bounding values, and prunes those entries that, based on the currently aggregated state, are for sure farther away from the query point than all remaining candidates.

The last two parameters are required for modeling different end-to-end scenarios, where queries flow through an entire index tree. The *selectivity* denotes how many child *slots* match the query predicate (in %). The *correlation* denotes which of the child *slots* should match, compared to the other queries in the batch, i. e., an overlap (in %) for the subsequent scan iteration on the next level. Because *selectivity* and *correlation* are merely relevant for evaluating the impact of different scheduling strategies, they are discussed in more detail in the next chapter.

Of course, the execution also depends on the hardware that is used to process the scan. For the experiments in this thesis, two different CPUs and GPUs were available (cf. Table 5.1). The same benchmark has been evaluated for the combination of the 2011 devices and for the combination of the 2015 devices, in order to asses how hardware developments impact the generalization aspect for a single, unified framework implementation. Node and query vector sizes were varied up to the maximum value that could be used on a GPU to fill the shared memory with cached entries.

5.3.2 CPU / GPU Speedup Analysis

5.3.2.1 Comparison of Stateless Search Operations

When tasks are scheduled to a GPU, it is expected that the parallelization effect becomes visible for large workloads that can fully utilize the available processors on the device. For node scan primitives, this means a large number of queries in a batch and a large number of slots per node as the number of matrix cells increases by their multiple. In these cases, the GPU should clearly outperform sequential execution on a CPU. However, executing the node scan on a GPU

involves overheads for intermediate data transfers, the scan preparation, and scheduling the kernels through the device driver that also requires additional communication via the PCIe bus system. Therefore, it is expected that a minimum workload size is required in order to outweigh these overheads by the performance gains through an increased parallelization. Thus, for a small number of slots and/or a small number of queries per batch, the CPU execution should be faster, because it doesn't cause such overheads at all.

Considering the hardware development over time, it is expected that, due to the abstraction layer provided by CUDA, these expectations are still met when the speedups are compared between the 2011 and the 2015 hardware configurations. As the algorithmic patterns that are implemented by the index framework, i. e., the mapping of tasks to the underlying execution model as well as involved memory access patterns, are independent from the runtime platform, the overall shape of the speedup curve shall be the same, maybe having different amplitudes and break-even points.

The results of the micro-benchmark for stateless range queries are illustrated in Figure 5.13. Each 3D-plot shows the speedup values that were measured over two dimensions of the parameter space, varying the number of slots and queries per batch on 2011 and 2015 hardware. For the experiments in the two topmost figures, one thread block was used to process a scan primitive task. For the last experiment in the bottom figure, a more fine-granular grid configuration was used to work around an inefficient memory access pattern in the prototypical implementation that is discussed below. Multiple thread blocks were used in this case for processing a single node, partitioning its child entries into chunks of 8 slots per block (cf. Figure 5.5). Further, as more detailed profiling of the node scan phases revealed (cf. Section 5.3.4), the initial implementation of the prescan algorithm for on-device caching also suffered from device main memory access inefficiencies that would require additional tuning of the prototype. But since the impact of input data transfers was not as high as expected, these efforts were not spent. Instead, the scan preparation was omitted for the experiments on 2015 hardware, assuming direct addressing by block ID within the grid for calculating input and output offsets. Thus, the two lower figures just comprise runtime measurements for host $\leftrightarrow$ device data transfers, the scan kernel, and the result compression, while the upper figure comprises the full stack.

It can clearly be seen that for workloads with low parallelism, i. e., small batch and node sizes, the CPU implementation outperforms the GPU on both system configurations. Data transfers and kernel overheads were significant in these cases. For workloads with a high degree of data parallelism, the GPU was faster than the CPU, up to $2\times$ for the old system and up to $10\times$ for the newer one. Comparing the 2011 and 2015 results with regard to the absolute speedup factors, the differences between CPU and GPU execution emphasized significantly for low/high parallelism cases. For high parallelism workloads, the increased number of available processing resources on the modern GPU can be efficiently used, leading to much higher speedups. For low parallelism workloads, the modern CPU was up to $200\times$ faster than the GPU. Thus, the impact of the overheads became more important as hardware evolved. However, the break-even points, i. e., where execution on CPU and GPU are equally fast, hardly changed for both platforms. The break-even points of $s = 1$ and some additional discrete speedup values are illustrated as contour lines in Figure 5.13. Knowing the $s \approx 1$ configurations is particularly important for guiding scheduling decisions on hybrid platforms, which is subject of the next chapter.

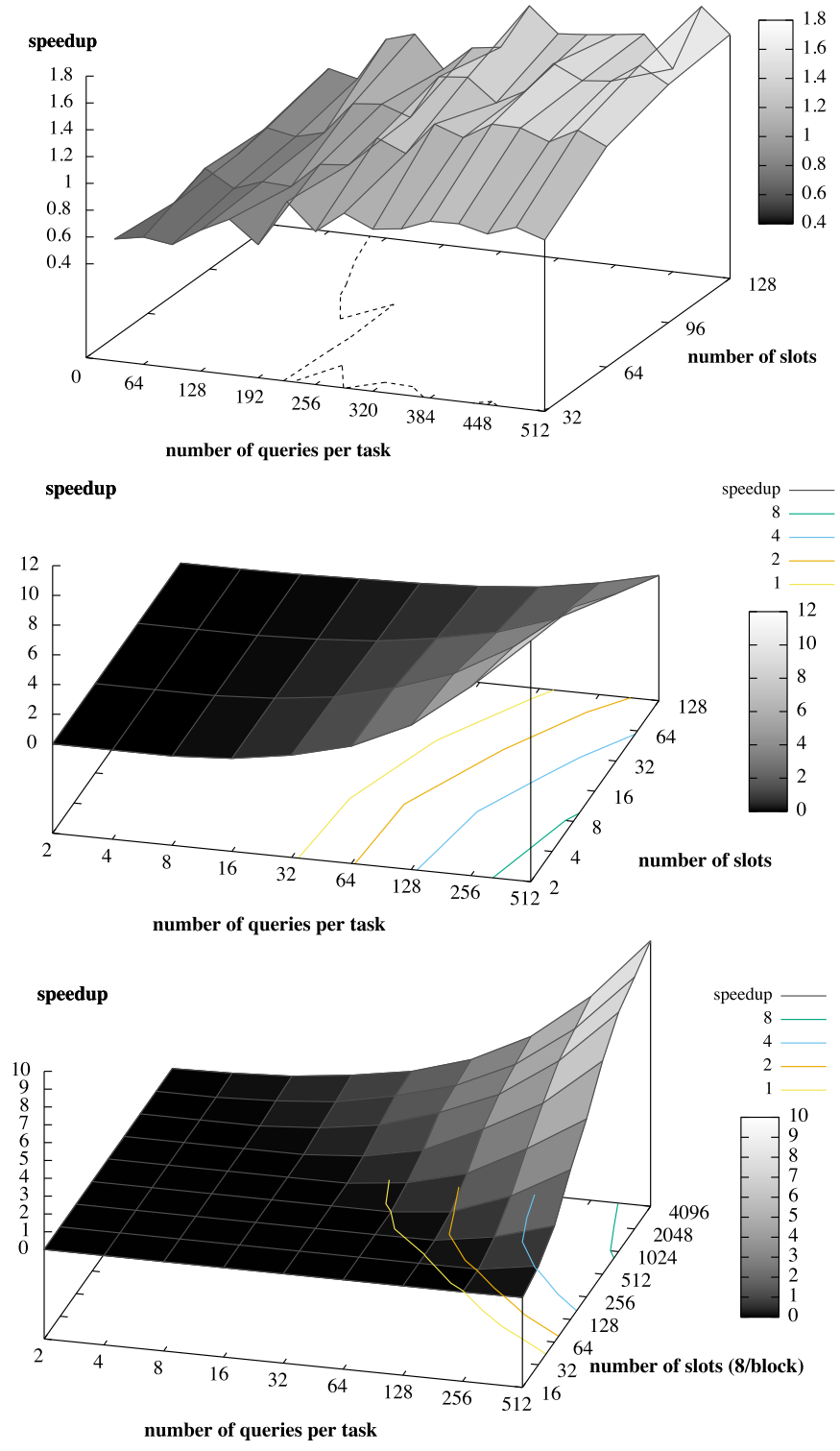


Figure 5.13: Comparison of R-Tree Range Predicate Speedups on CPU and GPU
2011 Hardware (top), 2015 Hardware (middle),
2015 Hardware with Finer-Granular Grid Configuration (bottom)

One interesting observation can be made by looking at the results of the 2015 hardware: contrary to the assumption that, due to the higher degree of parallelism, the speedup increases with an increasing number of slots per task, the exact opposite was the case. This effect was not visible for the 2011 hardware configuration and was caused by an inefficient access pattern of GPU threads to global device memory. In the initial implementation, the threads fetched multiple elements from device buffers in a loop to reduce the number of blocks that need to be instantiated for a kernel, assuming that access latencies would be hidden by overlapped memory transactions of different warps. In order to verify this hypothesis, another experiment has been conducted that increases the number of blocks in the grid, using multiple blocks per node so that the number of loop iterations decreases. The number of slots per block was kept constant. The results of this experiment are illustrated in the plot at the bottom of Figure 5.13. With an increasing number of active blocks, the GPU can schedule more memory transactions for all in-flight threads and is able to compensate this latency effect, up to a point where the device is fully saturated. On the one hand, this identifies additional tuning potential, but on the other hand, to some extent, relativizes the idea of generalized hardware-independent algorithms. A certain amount of tuning is required for new platforms as the architectural differences might reveal effects that were not observable before. After this workaround, the speedup curve shows the expected shape again.

5.3.2.2 Comparison of Stateful Search Operations

The speedups for stateful predicates are expected to be comparable to the stateless counterpart. That is, the overall shape of the speedup plots should not change, showing a better CPU performance for low-parallelism workloads and a better GPU performance for high-parallelism cases. However, offset and slope of the curve might be different, i. e., break-even points might be shifted and the absolute speedup values may differ, because the predicate evaluation logic changed. The nearest-neighbor search predicate that has been implemented for this thesis is much more complex compared to the range predicate version, because the former requires multiple Euclidean distance calculations and a distance aggregation, while the latter merely needs to compare some coordinates for the MBR intersection test.

The results of the micro-benchmark for a single iteration of stateful nearest-neighbor searches on 2011 hardware are illustrated in Figure 5.14. Like in the range predicate evaluation, the experiments were conducted without on-device data caching as it proved to cause to much overhead in the prototypical implementation (cf. Sections 5.3.2.1 and 5.3.4). The 3D-plot shows the expected shape, having speedups $s < 1$ for a lower number of slots and queries, and $s > 1$ for high workload parameter configurations. Compared to the range predicate evaluation, the maximum speedups are much higher, because there are much more calculations involved for each element of the scan matrix. While in the stateless predicate case, the ratio of (on-device) data transfers to calculation operations clearly emphasizes (intermediate) data accesses, this changes because of the many distance functions that need to be evaluated for the nearest neighbor search. Therefore, the GPUs computational power can be fully leveraged, once a sufficiently large number of queries need to be answered per batch.

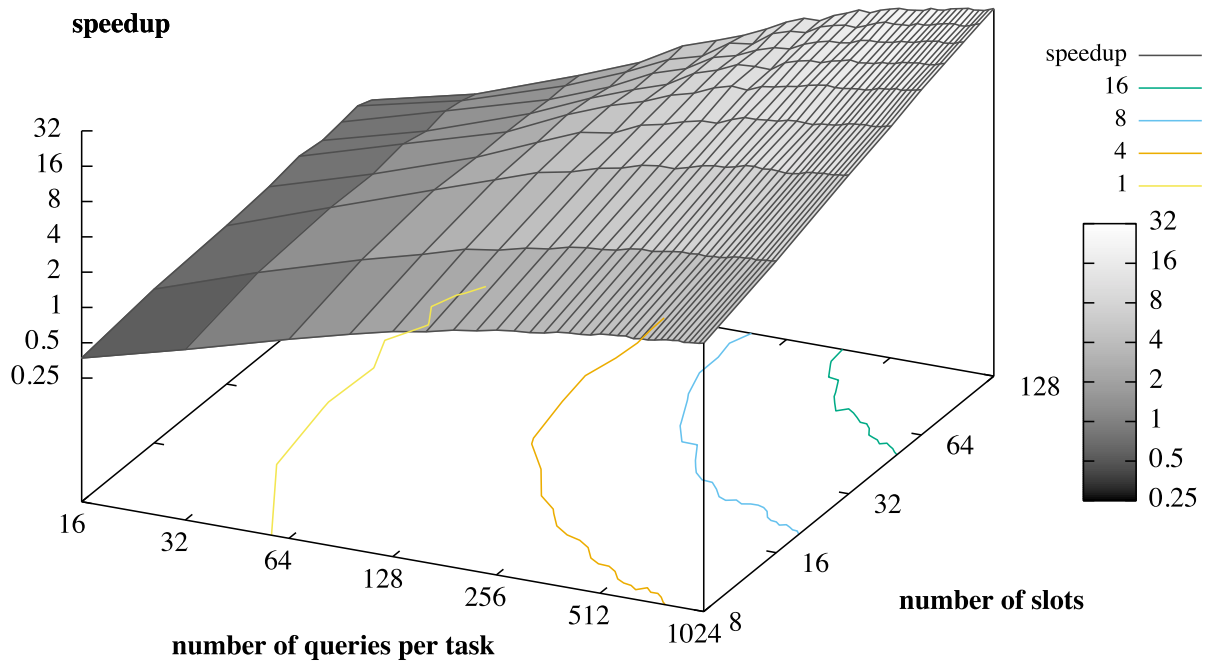


Figure 5.14: Comparison of R-Tree Nearest Neighbor Search on CPU and GPU (2011 Hardware)

The experiments have not been re-evaluated for the 2015 hardware, because, as the re-evaluation of the range-predicate scan has shown, it is not expected that the overall shape of the speedup curve changes. There will also be cases, where $s < 1$ for low-parallelism workloads and $s > 1$ for high-parallelism scenarios. Likely, the break even points shift and the curve's slope will be increasing, because the newer hardware provides more processing resources than the old GPU.

5.3.3 Query Throughput Analysis

The speedup analysis suggests promising results, i. e., there are many cases, where the GPU outperforms the CPU if workload parameters are properly chosen. However, a single speedup value only denotes the ratio of CPU and GPU execution times. That is, each value compares these metrics just for a single parameter configuration. It does not make any statements about the optimal configuration for each device type. This gap is closed in this section by examining different workload configurations for different processor types separately.

In this experiment, absolute query throughputs have been analyzed for a range of possible query batch sizes, assuming nodes with fixed size of 96 and 2048 slots, which are evaluated using the R-Tree range predicate. This corresponds to a drill-down into one slice of the corresponding speedup curve that have been discussed previously. Query throughputs are calculated as $\frac{\text{number of queries in the batch}}{\text{processing time for single node}}$ and are expected to scale almost linearly with increasing workload size

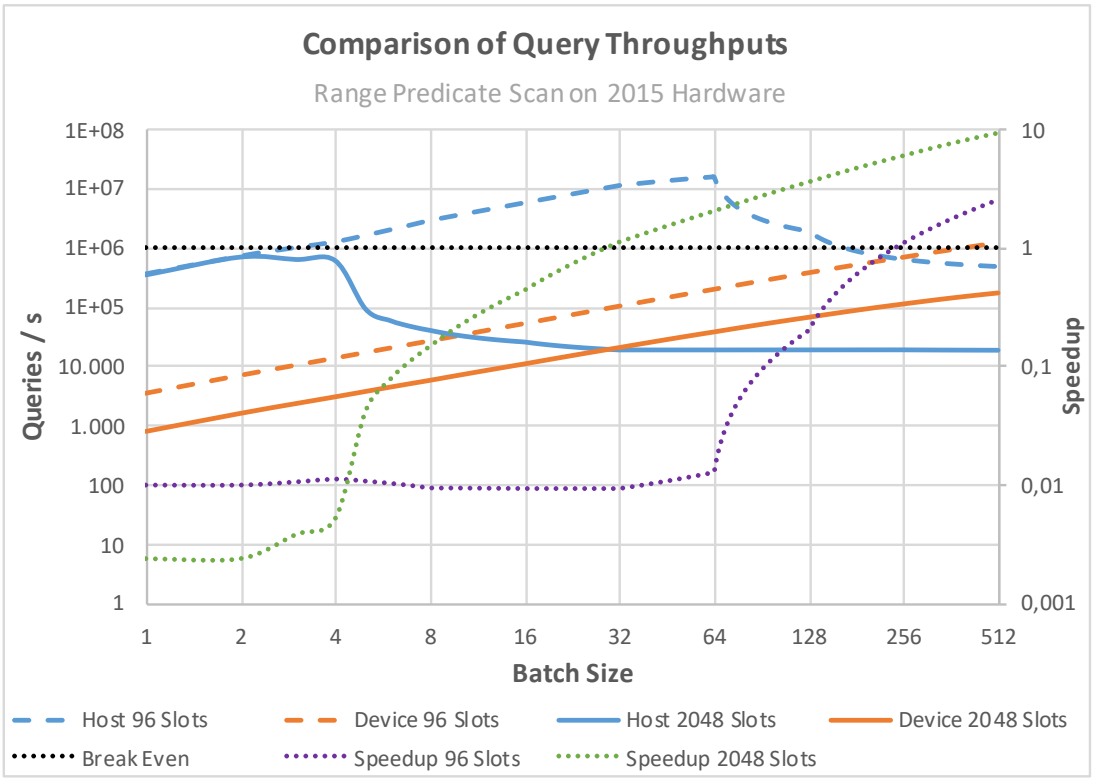


Figure 5.15: Comparison of R-Tree Range Predicate Scan Throughputs on 2015 Hardware

until full processor utilization is reached. After this point, throughputs should flatten. Further, it is expected that once the break even point for GPU processing is reached (where the speedup is greater than one), the GPU throughput should also be larger than the CPU throughput.

The evaluation results are plotted in Figure 5.15, showing throughput values on the primary and, for reference, respective speedup values on the secondary y-axis. Of course, throughputs for the large node are lower, because more predicate evaluations need to be performed. The CPU throughputs behave as expected. They scale linearly until the saturation point at 64 queries (small node) / 4 queries (large node) is reached. For larger input sizes, the throughput stays constant after dropping by more than an order of magnitude. This drop can be explained by caching effects, where storing evaluation results lead to thrashing in one of the lower-level caches once the point is exceeded where everything can be handled cache-local. GPU throughputs increase linearly over the entire parameter range. That is, even for the maximum batch size, the GPU is not fully utilized, yet.

Comparing absolute throughputs for a single node size shows that the sole speedup value can be misleading. Even if the latter indicates that the GPU is faster, the CPU exceeds GPU peak throughputs significantly for lower batch sizes. For the small node, the maximum GPU throughput exceeds the minimum CPU throughput for a batch size of one. For the large node, not even this happens. Speedup values are only observed because of the CPU performance drop for medium-sized query batches. This is another indicator for an inefficient memory access behavior on the GPU, because throughputs should be higher if more result matrix cells can be evaluated in parallel by the device.

With respect to the framework these results mean:

- Using the current GPU implementation is not optimal for evaluating the analyzed predicate for a single node. The predicate, which basically implements just some floating point comparisons, seems to be too simple, in terms of computational complexity, in order to gain notable performance gains via parallelization. A CPU-optimized implementation can yield much better performance because less overheads are involved.
- The vectorized batch processing approach implemented in this framework is useful in this respect, i. e., by selecting a good value for the maximum batch size, cache locality can be exploited. Compared to the query-at-a-time processing, which would be implemented by GiST, this may result in an order-of-magnitude throughput enhancement for processing a single tree node. The question to what extend this also speeds-up end-to-end tree traversals is answered in Section 6.3.4.

Of course, additional tuning can be applied in order to shift any of these curves. The GPU is also not fully utilized, yet, which is not surprising for a single-node evaluation. Once multiple nodes would be scheduled at the same time using a single kernel invocation, the aggregated throughput values should also increase further until all multiprocessor cores are saturated. Further, if the predicate's complexity is increased, the GPU might also excel due to its massively parallel processing capabilities. But analyzing this deeper is left for future work.

5.3.4 Profiling Node Scan Phases

As the experiments of the previous sections have shown, offloading the parallelized index scan operation to coprocessor devices is beneficial for certain kinds of workloads. This section analyzes these conditions by profiling the phases that are involved during the execution on a GPU device. The algorithmic stack for a GPU-based execution is significantly more complex compared to a CPU-based search (cf. Figure 5.12). Thus, it is expected that it causes a lot more overheads for those phases, which are not required by the CPU implementation. In general, as the speedup results of the previous evaluations suggest, the impact of these overheads shall become visible for low-parallelism cases and their relative runtime in the entire stack shall decrease for high-parallelism cases.

A second expectation is that the host $\leftrightarrow$ device I/O operations consume a significant portion of the total runtime, because the PCIe interconnect has a very low transfer bandwidth, compared to the other memory transfer bandwidths for accessing on-device RAM and on-chip shared memory within a kernel. Therefore, on-device caching should be beneficial and the additional overheads caused by prescan kernel are expected to be negligible.

The profiling results of the GPU-based range predicate execution are illustrated in Figure 5.16. A setup was used that employs the 2015 GPU and includes the workaround for the inefficient memory access pattern, i. e., each kernel launches multiple thread blocks for processing a single node (cf. Section 5.3.2.1). Six scenarios are shown here, one for each combination of two parameters.

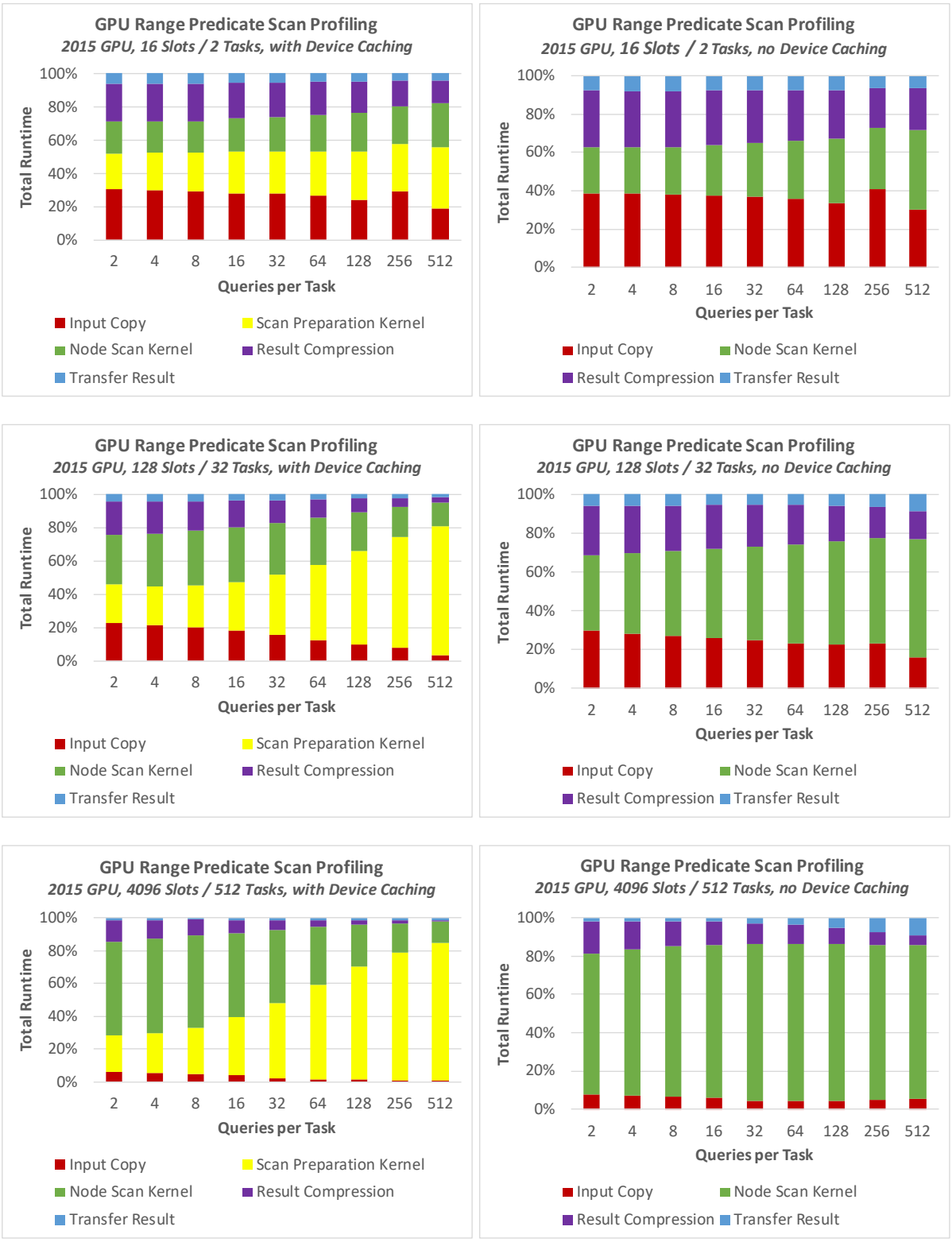


Figure 5.16: Profiling GPU R-Tree Range Predicate Evaluation (2015 Hardware)
Low / Medium / High Parallelism Workloads (top / middle / bottom),
With / Without on-Device Scan Data Caching (left / right)

All diagrams on the left hand side include the *caching* algorithm. All plots on the right hand side do not include the prescan phase, simulating how direct addressing by block ID would behave, where no ID-to-buffer-offset translation is required. Because a cold-cache setup is used, data transfers are included in all measurements. Implementing the on-device cache may significantly reduce input transfer times upon the next kernel invocation if node and query data would already be present on the device, because just the scan descriptors need to be provided. In the current implementation, the host synchronizes result sets after each kernel invocation for guiding the next scheduling decision (cf. Chapter 6) and, thus, output transfers are not impacted by caching.

The *workload class* describes the amount of parallelism (from top to bottom: *low* / *medium* / *high*) that was defined by the slot and batch size parameters of the node scan operation. Each profiling test series corresponds to the GPU runtime part of the 2-dimensional (queries, speedup)-plane in the bottom plot of Figure 5.13. That is, the number of queries has been varied in Figure 5.16 to illustrate the effects of an increasing parallel workload, while the number of slots were fixed to some discrete values that impact the slope of the speedup curve:

- *low* workload: 16 slots
- *medium* workload: 128 slots
- *high* workload: 4096 slots

By looking at the plots on the left that use the on-device caching algorithm, one can see that the scan preparation kernel consumes a significant portion of the total node scan runtime. While the topmost plot illustrates that the prescan runtime increases moderately with an increasing number of queries, comparing the plots from top to bottom shows that the runtime is heavily impacted by increasing the number of slots. The reason for that is, most likely, an inefficient memory access pattern, when node IDs are resolved. The time required for the scan preparation is even larger than the input copy phase and, as expected, the impact of the input copy itself is only significant for the low-parallelism cases. Therefore spending some additional engineering efforts for tuning the caching algorithm would only be beneficial for low-parallelism workloads that are not the sweet spot for a GPU-based execution, anyway. Tuning this phase might shift the break-even points towards a GPU-based execution, but was not implemented in the scope of this theses, because the general behavior of the speedups are not expected to change at all. In summary, the expectation that the proposed input data caching pays-off could not be validated for the current implementation of the prototype, because the algorithms are bound by the kernel runtimes.

The plots on the right, however, show that the initial expectation regarding the overheads of additional phases that are not related to the actual predicate scan were fully met. That is, with an increasing number of scan matrix cells, the relative runtime portions of the I/O and result compression phases decrease significantly and the scan parallelization fully pays off, compared to a CPU-based execution that consumed more time for executing the scan ($s > 0$).

The results in Figure 5.17 illustrates the profiling measures of the stateful nearest neighbor search on 2011 hardware. For this experiment that was conducted in early stages of the development, neither caching nor result compression was implemented. The diagram at the top illustrates the impact of an increasing parallelism by increasing the number of slots for a fixed number of 512 queries. The profiling test series, therefore, corresponds to the GPU runtime part of the 2-dimensional (slots, speedup)-plane in Figure 5.14. Like in the stateless experiment and as expected, the impact of the I/O transfer phases decreases with an increasing number of matrix cells. The scan quickly becomes compute-bound, which is a perfect fit for using the GPU here instead of to the CPU-based counterpart, because the speedup values are significant.

In order to evaluate the complexity of the predicate evaluation, a second experiment has been conducted for the nearest neighbor search, varying the number of the R-Tree dimensions. Those results are illustrated in the bottom diagram of Figure 5.17, keeping the number of queries and slots fixed. It can be seen that the complexity for evaluating additional dimension coordinates in the distance function impacts the relative kernel runtimes just slightly. The input copy phase is hardly impacted, i. e., its runtime increases similar to the kernel. The reason for that is the additional data that needs to be transferred to represent the multi-dimensional coordinates. Since the Euclidean distance function is only slightly impacted by an increasing dimensionality, the observable effects here are negligible. But this might be different for other stateful query types, where calculating the distances is the dominating factor, e. g., for index-assisted queries in highly-dimensional metric spaces [44, 89].

5.3.5 CPU / GPU Data Transfers

As the previous experiments have shown, intermediate transfers between host and device memory may consume a considerable amount of the overall processing time. This factor is even more important when no data caching is implemented. This section, therefore, analyzes the input copy phase and how block sizes impact transfer rates, in order to derive recommendations for future tuning the framework's coprocessor execution path.

During the experiment, multiple (1,000) chunks of data are simply transferred between host and device memory via the interconnecting PCIe 2.0 bus system. Transfer times are measured for different copy operations that are supported by the device driver, namely:

- *Synchronous copy* that copies each block serially via a separate API invocation. No additional host-device synchronization is needed, because it is already handled by the device driver.
- *Asynchronous copy* where all copy requests are scheduled to the device and the device driver handles them internally via multiple copy units. Handling additional synchronization events is needed to block the host thread until a transfer completes.

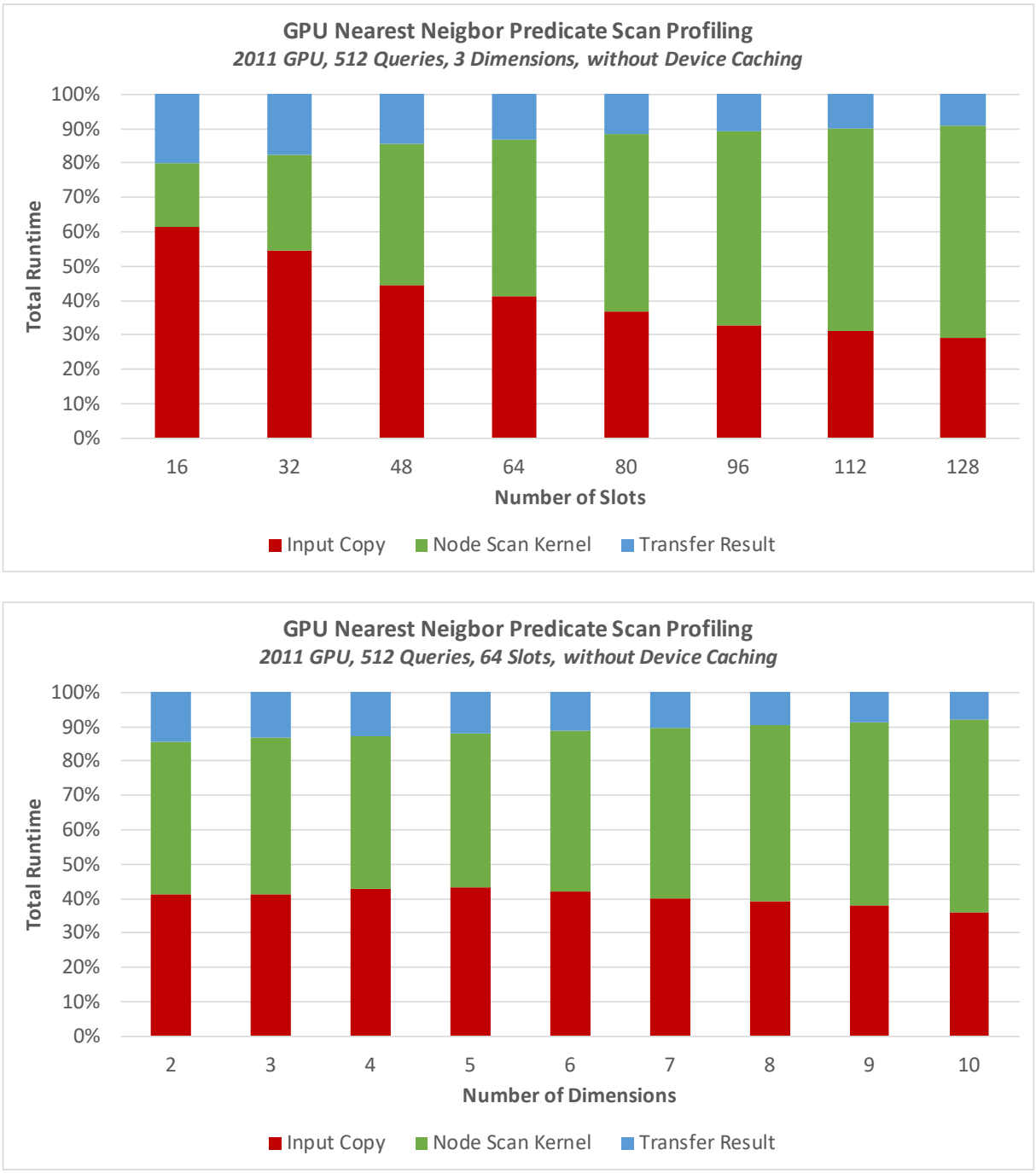


Figure 5.17: Profiling GPU R-Tree Nearest-Neighbor Predicate (2015 Hardware)
With Increasing Number of Slots per Task (top)
With Increasing Tree Dimensionality (bottom)

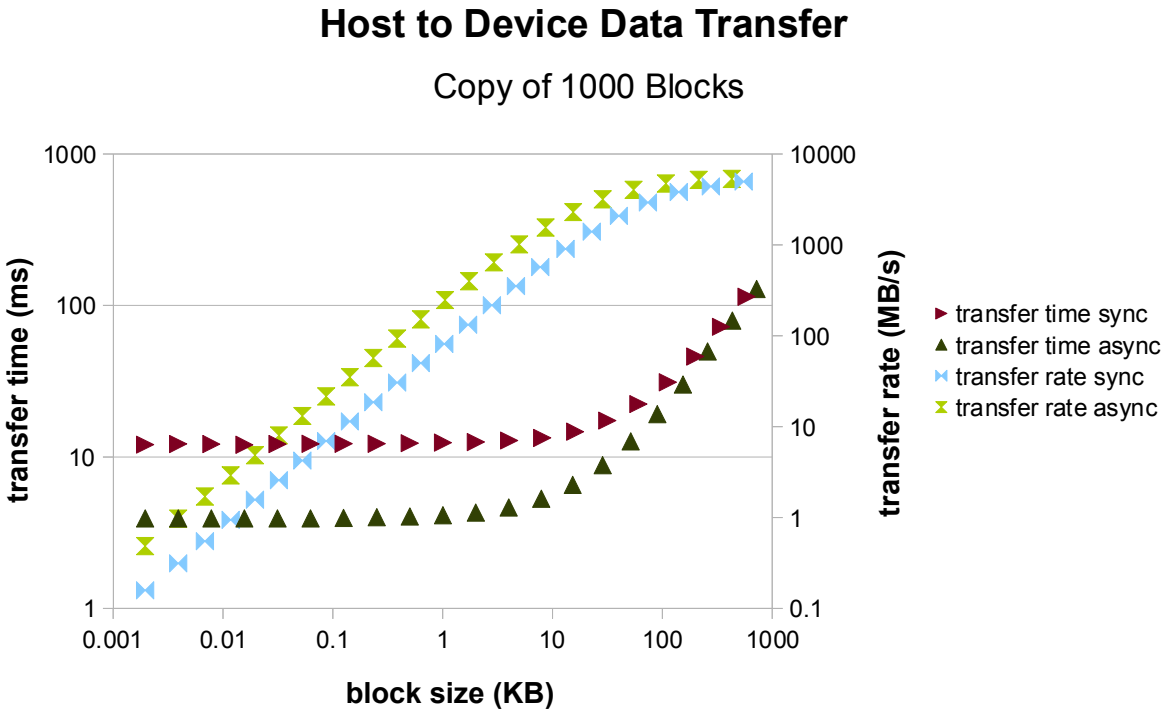


Figure 5.18: Host → Device Data Transfer Benchmark for PCIe 2.0 Bus

The experiment has been repeated for different chunk sizes and the overall transfer times as well as achieved data throughputs have been measured in order to assess where peak throughputs can be reached.

Since protocol overheads are associated with each copy operation, it is expected that a certain data volume is needed in order to fully saturate the bus system with useful payload. For small chunk sizes, involved overheads should be more significant, leading to lower overall throughputs. Using the asynchronous API for multiple chunks should be beneficial, because it enables the device driver to optimize several copy operations internally. Compared to a sequential synchronous transfer, this should result in higher overall transfer rates for transferring the entire chunk set.

The results of the experiment are plotted in Figure 5.18. Total transfer times are plotted on the primary y-axis and corresponding transfer rates, i. e., $\frac{1{,}000 \text{ blocks} \times \text{block size}}{\text{transfer time}}$, is plotted on the secondary y-axis. All initial expectations could be validated. It can be seen that peak transfer rates are only reached for larger blocks having a size between 10 and 30 KB, depending on the employed transfer strategy. Using the asynchronous transfer for smaller-sized blocks is beneficial as it almost triples the transfer rate and, thereby, reduces the data volume that is needed to fully saturate the interconnect. Of course, these results depends on the bus system and these numbers will change for PCIe 3.0, for example. But the general behavior for different block sizes is not expected to change as there will always be protocol overheads, no matter which interconnect is used.

In summary, the following recommendations can be derived:

- If each chunk already exceeds the size where peak throughput can be observed, the bus system can be fully utilized with payload data.
- Below this size, the time to complete separate synchronous copy calls is almost independent from the block size. That is, the latency to copy some data does not change, but the bus utilization with useful data increases linearly with the chunk size.
- This utilization can be improved if many chunks are copied asynchronously.

In context of the indexing framework, this means:

- Nodes should be sufficiently large if synchronous transfers are used. Otherwise, many node copies should be grouped via asynchronous operations. For example, in case of a 3-dimensional R-Tree, i. e., 6 double precision coordinates à 8 bytes per key for storing child node MBRs, 640 slots are needed in order to reach the 30 KB chunk size for a single-node transfer. Copying multiple nodes asynchronously that are ready for processing by the GPU can reduce this size.
- The same applies to query batches, whose physical size depend on the predicate type. Grouping many transfers for all batches that shall be processed in the current iteration is also useful here.
- Result matrices that are transferred back to the host system for post-processing should always exceed the critical chunk size as their number of entries scales quadratically. If additional compression techniques are employed here, e. g., for reducing the number of bits to represent a result cell, this might change, however.
- If multiple copy operations should be grouped via the async driver API, the host memory that acts as source / target of the transferred data needs to be pinned, i. e., managed by the device driver. This has to be considered in the optimized framework design, when data locations are determined, in order to avoid intermediate host-to-pinned-host memory transfers.

5.4 Summary

In this chapter, it has been demonstrated how generalized search trees can be prepared for leveraging the computational power of modern processor devices by applying techniques that have merely been used for specialized index implementations before. The algorithms that have been developed for stateful and stateless query processing exemplified how generic search tree traversals can be parallelized and mapped to a state-of-the-art programming model for generating device-specific code. The ideas and concepts behind the parallelization have been published in various articles and other publications [22, 23, 110, 137].

In particular, it has been shown how

1. sequential tree traversals in GiST can be revised in order to resolve data dependencies and unlock different forms of parallelism;
2. these search algorithms can be mapped to a batch-oriented, braided BFS that exploits coarse-granular task and pipeline parallelism as well as fine-granular data parallelism in vectorized processing primitives;
3. the algorithms can be mapped to the CUDA programming model in order to implement them on any GPU device that is supported by corresponding device drivers;
4. limited main memory capacities on GPU devices can be addressed by applying an out-of-core algorithm design, where only parts of the index needs to be available on the GPU.

The evaluation of the parallelized index algorithms has shown that the use of a GPU may lead to significant speedups, compared to a CPU-based implementation – but only when the workload is sufficiently large to utilize the available processing units. Otherwise, overheads for the more complex algorithmic stack completely eliminate any performance gains of the actual query evaluation phase and can even slow down the GPU-based execution path, compared to the much simpler, but sequential single-threaded execution on a CPU. That is, there must be enough queries and index entries inside a processing primitive so that the data-parallel algorithms on many, but slow thread processors are faster than their sequential counterpart on a single, but fast processor that does not need additional data preparation steps, such as data transfers, offset calculations to isolate threads, etc.

Further, deeper analysis of achievable query throughputs on different processor types revealed that the proposed batch-processing extension for GiST renders query processing algorithms cache friendly. By just applying this optimization, query throughputs on a CPU could be achieved that exceeded any observed GPU throughput for the analyzed index implementation. Additional tuning of the GPU part is likely to change this, but not in scope of this thesis and left for future work.

The main observation that will be followed-up next is that query performance on each of the available device types strongly depends on runtime-specific workload parameters. That is, there are regions in the parameter space, where the CPU outperforms the GPU and vice versa. This is likely to be true for most hardware combinations, because the underlying processor architectures are fundamentally different and designed for different workload classes. Therefore, hybrid processing is analyzed in the next chapter, with the intention to leverage different architectures in a hybrid system to maximize overall performance for mixed workloads. Those mixed scenarios are much more likely to occur in real-world applications than the artificial ones that have been used under laboratory conditions in the evaluation microbenchmarks.

Chapter 6

Scheduling in Hybrid CPU-GPU Architectures

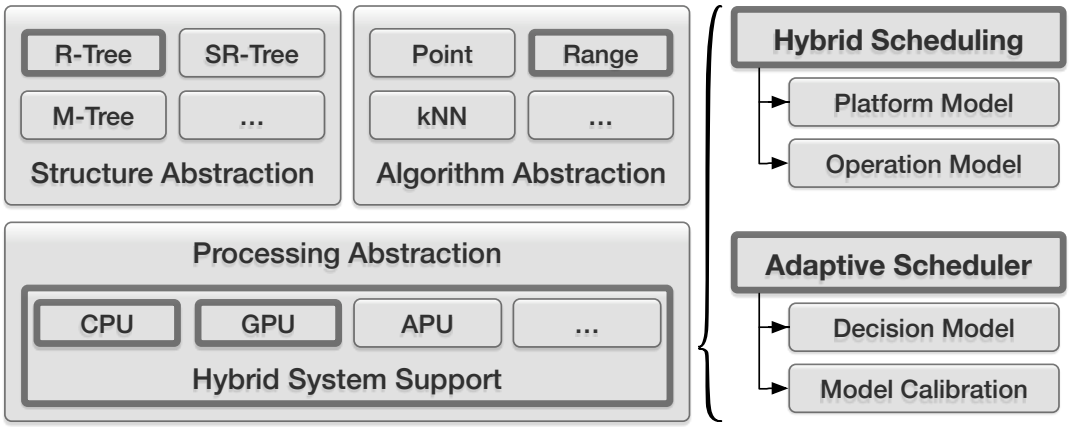


Figure 6.1: Overview of the Hybrid System Scheduler

After support for different processors has been added to the framework via the generalized execution layer, it is only natural to combine different processor types in a hybrid system to optimize query performance. While massively parallel devices, like GPUs, may accelerate workloads where many data elements can be processed independently, the use of CPUs is better-suited for smaller workloads, where the computational speedup cannot compensate overheads for, e. g., additional data transfers. In order to fully leverage the strengths of each processor type, a scheduler needs to be provided that estimates the execution costs on each available device and dispatches operations accordingly. This task is addressed in more detail in this chapter.

Section 6.1 demonstrates how different processors can be abstracted under a generic platform model and how the processing primitives from the previous chapter can be mapped to generic operators whose execution costs may be estimated for dynamic workload parameters. On top of this system abstraction, a scheduler can be integrated into the framework's processing layer,

which dynamically selects the best execution unit at runtime. An adaptive scheduler implementation is presented in Section 6.2, which can be calibrated to build a decision model for arbitrary hybrid systems. The applicability of this approach on a hybrid CPU/GPU system is evaluated in Section 6.3. A simulator for R-Tree range predicates is developed that allows to compare different scheduling strategies for various index workload parameters in order to assess the benefits of hybrid processor use for end-to-end query performance.

6.1 An Index Operation Scheduler for Hybrid CPU / GPU Systems

The index operation scheduler is integrated into the framework's execution layer. The scheduler is consulted for each processing primitive in order to decide which of the available processing devices is likely the best choice for executing it. For guiding this decision, a cost model has to be built that abstracts internals of the underlying hardware and also requires an abstraction over the various primitive implementations that could be provided by the framework. These abstractions allow to implement it as generalized, reusable component inside the framework itself that does not require any changes once new user-defined query predicate types are specified through the framework's extension points or when the code is deployed on different hardware platforms.

The platform model for abstracting details of the underlying hardware and the interactions with it are discussed in Section 6.1.1. The generic model for representing an index operation that can be executed on different devices are subject of Section 6.1.2. The scheduler component and its integration into the framework's execution layer is discussed in Section 6.1.3, before a specific implementation is presented in the following section.

6.1.1 Platform Model

The execution hardware can be abstracted using a hybrid system model, like the platform model of CUDA or OpenCL (cf. Section 2.4.1). That is, multiple execution *devices* can be used to execute asynchronous computation operations and a *host* is responsible for coordinating operations between them. Devices own dedicated memory regions for buffering input, intermediate, and output data and those memory regions can be accessed via a bus system.

The *bus system* is responsible for exchanging data between host and device via well-defined transfer APIs. Devices may either be connected via a dedicated bus system, such as PCIe, or share the same memory bus with the host. In the latter case, e. g., if the CPU is used as execution unit or an iGPU, a data transfer is a zero-copy operation, i. e., the data is only virtually transferred by reference and doesn't need to be physically moved. In general, a transfer operation can be characterized via latency and bandwidth parameters that, usually, depend on the amount of data that is transferred (cf. Section 5.3.5). Further, it might offer features, such as full duplex support, that can be utilized to model pipelined operations (cf. Section 2.2).

Device memory regions are modeled as two distinct types. First, there is the *(global) buffer memory* that can be accessed from the host system for storing input and fetching output data for each scheduled operation. Second, there is a *(private) cache memory* that can be used for software-controlled caching in the operations' implementation. For the scheduler, mainly the sizes of the global memory buffers are relevant to determine whether the data of an operation does fit onto a device or not. Other workload-specific parameters, like access latencies, etc., could also be modeled, but this level of detail is not applied in the following. Information about the private cache memory is important for configuring the operation instances, but also ignored for guiding scheduling decisions for now.

From a processing perspective, a device represents a set of vector processors that are capable of executing *independent SIMD tasks* without communication between them. That is, the scheduling unit for a whole device is a set of such tasks and there is no control over on-device scheduling. Each processing primitive represents one SIMD task that is executed in parallel by many threads in lockstep mode. Those threads operate on disjoint global buffer regions and might communicate over the private cache memory. But as this is only relevant for specific algorithm implementations, these internals are ignored by the scheduler. For guiding scheduling decisions the scheduler just needs to be aware of the number of processors on the device.

Operation queues represent the scheduler's abstraction layer to determine the load on the device. They might implement different strategies, e. g., buffering a certain amount of tasks until they are scheduled to the device at once, overlapping execution phases in a pipelined manner, etc. Queues can be queried to estimate whether processing a primitive task would be delayed.

6.1.2 Index Operation Model

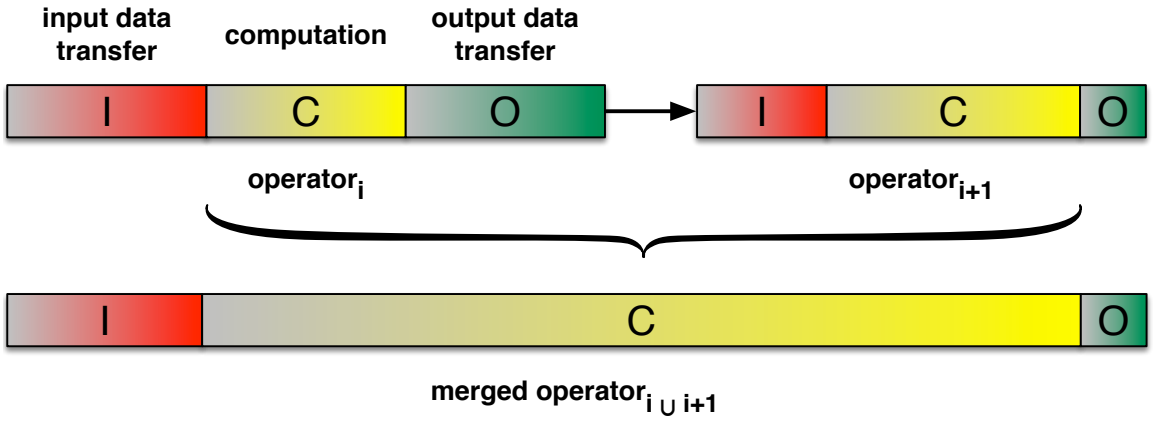


Figure 6.2: Abstract Operator Model for Single Processing Primitives

The model for an abstract operation that can be scheduled to a specific execution device is illustrated in Figure 6.2. A simple model has been selected to abstract as many operation-internal implementation details as possible for representing to all kinds of processing primitives that the framework might provide. The bare minimum that an operator must provide for the scheduler is information about its processing phases that are involved in hybrid system setups.

When an operator should be executed on a specific device, it must assert that all input data is available in global device buffers so that the actual computation operation can access it. Therefore, each operator models an *input data transfer phase* that hides all internals from the scheduler, e. g., whether and how on-device data caching is implemented or not. For a single primitive instance that shall be scheduled, the scheduler needs to be able to estimate the total input transfer time. In case caching is implemented, this will be different for each primitive and depends on whether there is a cache hit or an explicit transfer would be needed. For the actual transfer, the operation needs to provide information on the size(s) of the data chunk(s) that need to be copied via the memory bus system before the computation can start. For the node scan primitive, the input transfer is impacted by the query predicate and key entry vectors. The scheduler can join this information with the bus system's characteristics in order to estimate the required transfer time.

The *computation phase* represents the total execution time that is required to transform the operation's input data into output data that will be transferred back to the host upon completion. Because this phase heavily depends on the actual operation that shall be executed, the scheduler is assumed to build a model for each operation type that is supported by the framework in order to estimate its execution time. All supported operation instances are statically provided by the framework, but the cost model might employ additional runtime information to calculate the estimations based on parameters describing the workload. For the node scan primitive, the calculation effort mainly depends on the size of the input matrix.

Similar to the input transfer phase, the *output data transfer* phase is finally responsible for copying result data from the device back to the host system. The size of the output depends on the operation that is applied. Usually, it can be calculated upfront, because this is required anyway for pre-allocating the output buffer before the operation is launched. In case of the node scan primitive, the output size is determined by the size of the scan matrix.

In case an operation consists of a sequence of sub-operations that shall be scheduled as a single unit of work, it can be simplified to a merged operator whose intermediate I/O phases are hidden. This is illustrated at the bottom of Figure 6.2. The intermediate phases are merged into a larger computation phase, because, usually, in such a sequence the subsequent operator is applied to the output of the previous one, eliminating the need for intermediate I/O. For the node scan primitive, such a simplification makes sense for the intermediate kernel invocation that is required in case on-device caching is implemented, hiding the scan preparation inside a merged (prescan $\rightarrow$ scan)-operator that is scheduled as a whole.

Applying this abstract model definition to the stateless GPU node scan operation from Section 5.2 yields multiple operator variants that are illustrated in Figure 6.3. The first two variants model the operator including the on-device node caching implementation. That is, the operator's host thread checks whether a node is already present on the device and skips the node data transfer on a hit (variant *A*) or transparently injects the copy on cache miss (variant *B*). Variant *C*, which has also been evaluated in Section 5.3.4 to work around the prescan implementation inefficiency, always copies the node data but may skip the prescan phase afterwards. In any case, the scan descriptors, i.e., (node ID, query ID vector)-pairs must be copied as kernel input.

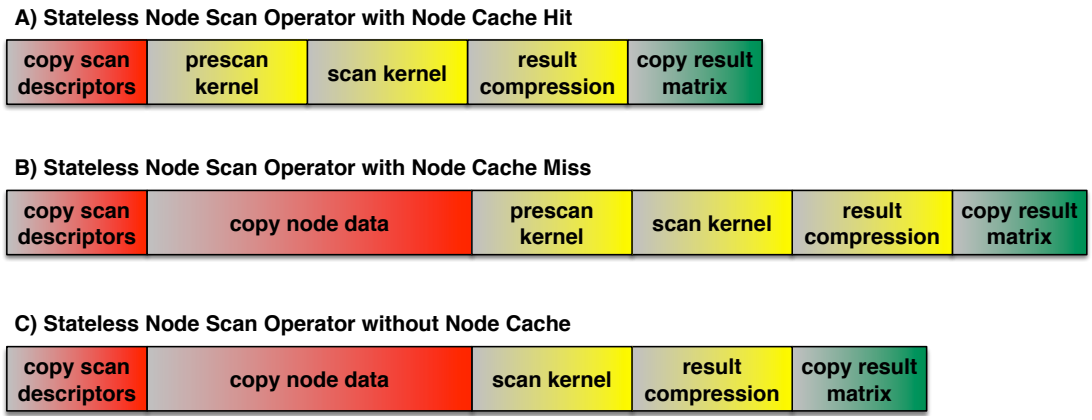


Figure 6.3: Operator Models For Stateless Node Scan Primitive

After the input copy, all device kernels are executed in sequence. The prescan kernel is just required with on-device caching in variants *A* and *B* to translate input IDs into cache buffer offsets, which can be skipped by directly addressing the data in variant *C*. The actual scan kernel is the same for all cases as well as the result compression phase, which is implemented in conjunction with the scan kernel to save the overhead for another kernel invocation. Finally, the (compressed) result matrices need to be copied back to the host, which is the shared result copy phase for all operator versions.

Up to this point, the discussion in this section merely considered the isolated scheduling of single operator instances, each comprising the aforementioned I/O and computation phases. Additional optimizations can be applied, when the scheduling itself is considering sets of operator instances that can be distributed to the available devices, e. g., many query batches that shall be scanned for several nodes. Of course, this adds additional complexity to the decision model, effectively increasing involved scheduling overheads, but it also allows to apply intra-operator optimizations, such as the interleaved and / or pipelined phase execution approach that are sketched in Figure 6.4.

If the processor device supports the concurrent execution of multiple operations and the interconnecting bus between the device and host system supports overlapping I/O requests (e. g., asynchronous I/O requests in one direction and / or full duplex transfers for concurrent transfers in both directions), the different phases of independent operators might be reordered and overlapped. Compared to a strictly sequential execution of the operator set, the overall throughput might be increased, because the latency when an operator phase starts is reduced and available bandwidth capacities might be utilized more efficiently (cf. the data transfer benchmark in Section 5.3.5). In the context of index query processing, each tree traversal iteration will lead to multiple scan primitives of subsequent tree layers that can be scheduled at once. Therefore, applying such an intra-operator optimization makes sense for high-throughput scenarios, where many primitives are queued and the devices operate at their maximum capacity.

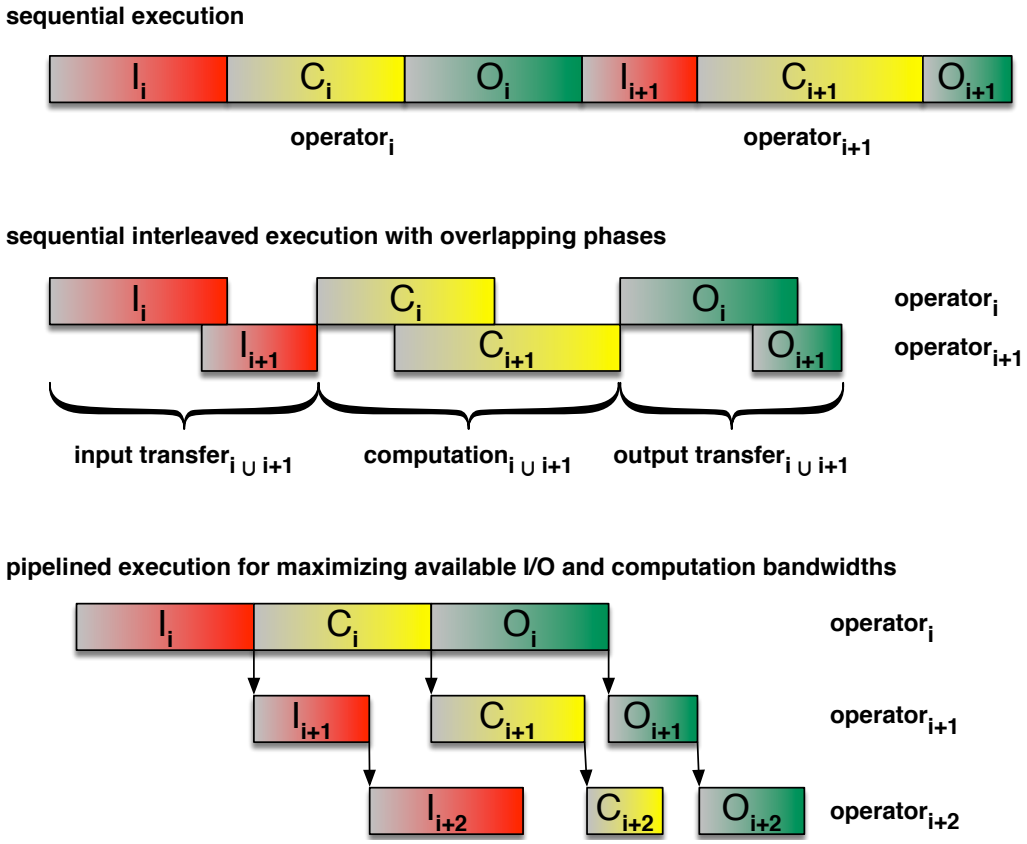


Figure 6.4: Operator Streaming Model for Multiple Processing Primitives

6.1.3 Index Operation Scheduler

The scheduler for generic index operations, which is built upon the platform and operation model, as well as its interactions with other framework components is depicted in Figure 6.5. It is responsible for maintaining cost models for each combination of all available device types and all predicate algorithms that are defined for these platforms. These cost models are used internally to select the best-suiting variant for a processing primitive that shall be executed by the query processor.

The available algorithms are statically provided by the framework library, because the implementation code must exist for each query and device type. In case a specific algorithm is not supported for a certain device class, e. g., because it has not been implemented via the framework's extension points yet, an implementation among the remaining devices will be selected. The available cost models are evaluated at runtime upon a scheduling request for a (set of) processing primitive(s), e. g., (node, query)-batch(es). Therefore, the scheduler extracts important features from the to-be-scheduled primitive(s), i. e., those that are expected to impact the estimated execution costs significantly, such as the size of the input vectors, queries all available cost models for these parameters, and selects the one having the minimum cost estimate.

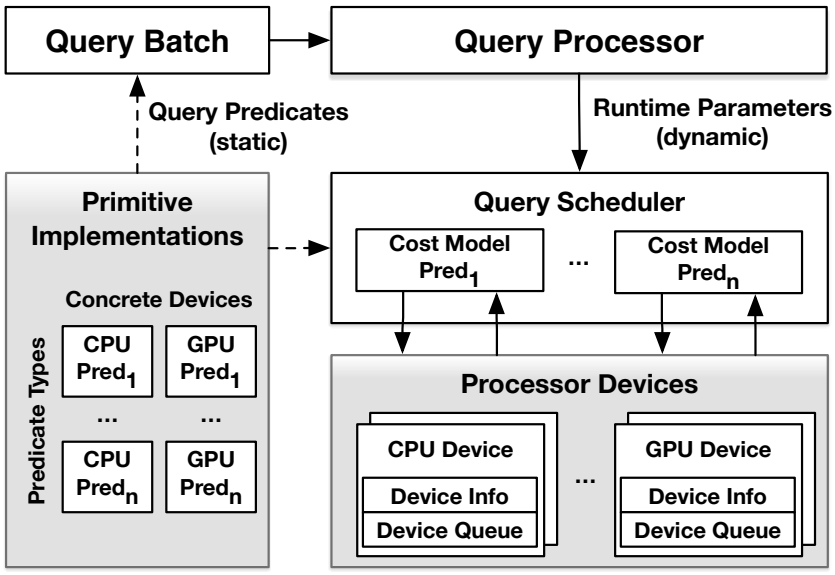


Figure 6.5: Scheduler for Generalized Index Queries

No assumptions are made on how the cost models are implemented. The common interface for each implementation just accepts a processing primitive and returns a cost estimation metric that can be compared between different models of different devices. In the following, the total estimated runtime for executing the primitive is used for this. Internally, any algorithm can be applied to calculate the estimates. In addition to the parameters provided by the processing primitives and the information about the available algorithm implementations, e. g., which I/O and execution phases are involved (cf. Section 6.1.2), the model can query current information about the devices, (cf. Section 6.1.1) and also observe real execution times in order to adapt the models for future decisions. The latter, in particular, allows to implement black-box scheduler models that will be evaluated in the following sections.

6.2 A Self-Tuning Scheduler Implementation

There are various ways for implementing a scheduler for index operations. For example, an analytical approach could be used by modeling the execution hardware via the information that is provided by device drivers, like the bus system bandwidth, and estimating execution times for all operator phases by joining these specifications with the workload characteristics of a specific operation, like a node predicate scan. While this approach offers some abstraction over the available processing hardware by leveraging the abstractions provided by the platform and operator model, the cost models are still hard-coded via rules that are used to calculate runtime estimations based on the available parameters.

To avoid maintenance of such rules and, to some extend, make the models robust regarding hardware changes as well as changes in the available index algorithms provided by the framework, a different approach is followed in this thesis. The main idea is to use a scheduler implementation that automatically derives cost models for the environment where it is deployed by implementing a machine learning algorithm. The models are trained using a one-time calibration phase and may be adapted at runtime in order to refine internal cost functions on-the-fly.

In the following, a first prototypical implementation of this idea is presented, which is based on the database operation scheduler for hybrid CPU/GPU systems that is discussed in Section 6.2.1. Section 6.2.2 presents how this generic scheduler framework can be adapted for the indexing use case, Section 6.2.3 discusses how decision models for index queries are built, and Section 6.2.4 briefly outlines how they are maintained when the framework is used.

6.2.1 Background – A Hardware-Oblivious Operation Scheduling Framework for Hybrid CPU/GPU Systems

Figure 6.6 illustrates a generic scheduling framework that has been developed by Breß [36] and was evaluated as joint work for the indexing use case [37]. The scheduler is mainly comprised of three components:

- an *Algorithm Pool* that maintains a registry of all available algorithm implementations that solve the to-be-scheduled task;
- an *Estimation Component* that maintains cost estimation functions for each of the available algorithm implementations;
- a *Decision Component* that, based on the cost model estimations, selects the best among all available algorithms for a given task instance.

The *Algorithm Pool* stores a (static) list of abstract logical algorithm identifiers A_i, having one entry for each algorithm that is available in the system, e. g., as library implementation, and which can be physically executed on one of the available devices during runtime. That is, when a logical task instance should be executed either on a CPU or a GPU, the pool must have at least two possible implementations, one for each device type, that can be selected for generating the task result. The actual implementations might differ but their logical effect has to be the same.

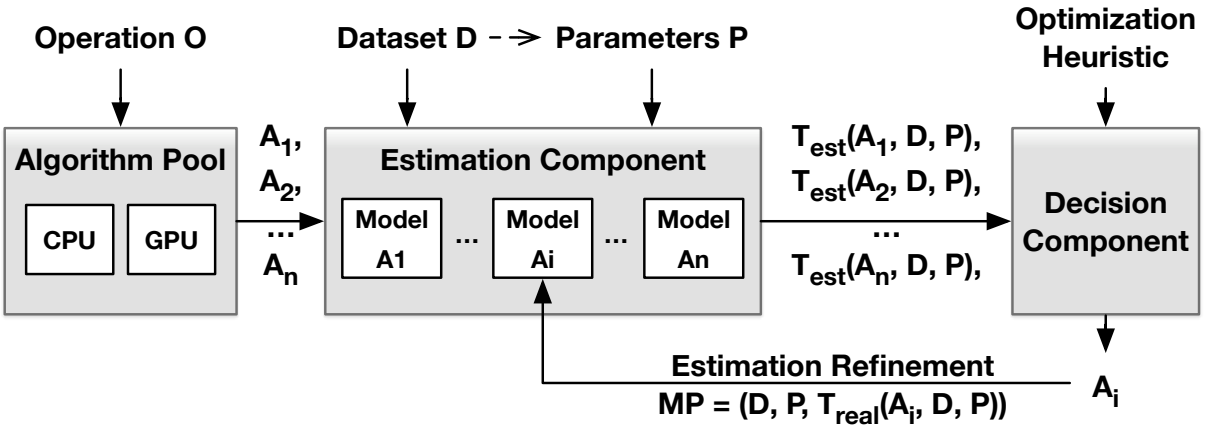


Figure 6.6: Self-Tuning Operation Scheduler for Hybrid Platforms
Source [36, 37]

The *Estimation Component* is responsible to guide the decision of which of the available algorithms should be selected, given a task specific instance. The task instance is described by its input, such as the data set D, onto which the algorithm should be applied. Further, the estimator needs some runtime-specific parameters P, which might be directly derived from the input, e. g., the input size, or which are provided as other task-independent runtime information, e. g., the utilization of each of the available processors, caching infos, etc. A cost model is maintained for each of the available algorithms that can be used. A cost model is simply a function that yields a real number representing the metric that shall be optimized by the decision, e. g., the total execution time, for any, potentially high-dimensional input parameter vector P.

The *Decision Component* calculates all model estimations, and, based on some heuristic, selects the best-fitting algorithm for execution. For example, it may select the algorithm having the minimum estimated execution time T_{est}. The algorithm is executed on its corresponding device and the estimation model can be adapted with the actually observed metric value in order to refine estimations for subsequent scheduling decisions. The model refinement can, for example, be calculated with linear regression.

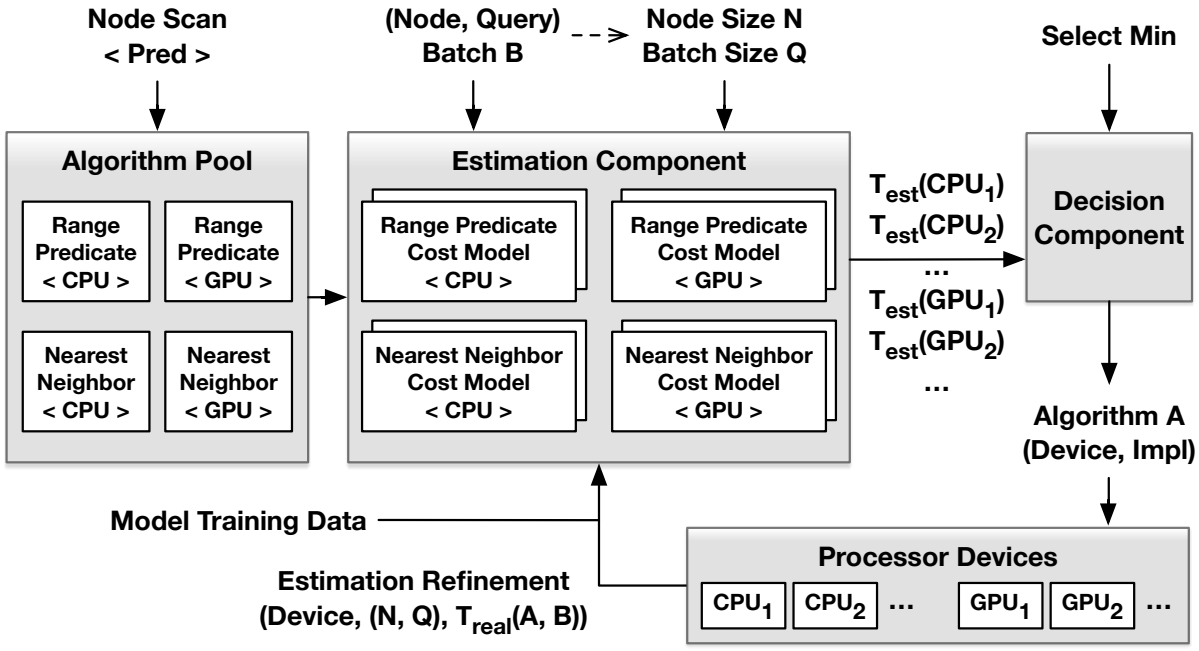


Figure 6.7: Self-Tuning Index Operation Scheduler for Hybrid Platforms

6.2.2 An Adaptive Index Operation Scheduler

This generic scheduling framework has been adapted for the indexing use case [37, 38] by

- mapping the framework's predicate evaluation operations to the generic operator model;
- integrating the operator implementations as algorithms that can be scheduled;
- modeling relevant parameters for guiding the decision;
- configuring the machine learning components to use a regression-based algorithm that builds and maintains internal cost models based on these parameters.

An overview over the adapted components is depicted in Figure 6.7. The available search query algorithms, i. e., the CPU- and GPU-based range predicate scan as well as the nearest neighbor search operation have been registered in the *Algorithm Pool*. The logically differing implementations are distinguished by the predicate type. That is, if a range predicate scan is requested, the scheduler may just choose between the range predicate operator implementations. Integrating other operations later, e. g., for building the index and maintaining its nodes, is straightforward. All that needs to be done is to register an abstract identifier that can be used by the dispatcher in the query processor. In case a specific implementation does not exist yet, e. g., because there is no GPU-based implementation for a specific predicate type, there is no handle for it in the registry and the corresponding processor type does not participate in the algorithm selection process.

The *Estimation Component* maintains a cost model for all combinations of algorithms and processors. Note that the framework might only provide a single algorithm implementation for a specific device class, e. g., a CUDA-based algorithm for all GPUs that can be instantiated for different devices models, or a common OpenCL-based implementation for multi-/many-core CPUs and GPUs. Because each device model might have different characteristics, e. g., a different number of processors and clock frequencies for different GPU generations, it makes sense to separately maintain cost models for each device that is available in the runtime environment. A unique device description that can be used to distinguish them should be available through the corresponding device driver.

The model decision is made for specific primitive instances that shall be scheduled. For the predicate evaluation case, the query batch is provided as well the tree node that shall be scanned. Important parameters are extracted from these primitives and the runtime environment (cf. Section 6.2.3), which are then used to query the available models that return their runtime estimates. The *Decision Component* selects the minimum of all variants and returns the corresponding algorithm identifier. The query processor uses this ID to instantiate the algorithm implementation and dispatch it to the processor who is represented by the model.

The real execution time for this predicate instance on the selected device can be measured and fed back into the model, together with the parameter combination that guided the decision. This *Estimation Refinement* can be used to continuously adapt the cost models when the framework is running. Further, this learning mechanism can be used to initialize the models based on training data sets comprising (parameter combination, runtime) pairs (cf. Section 6.2.4).

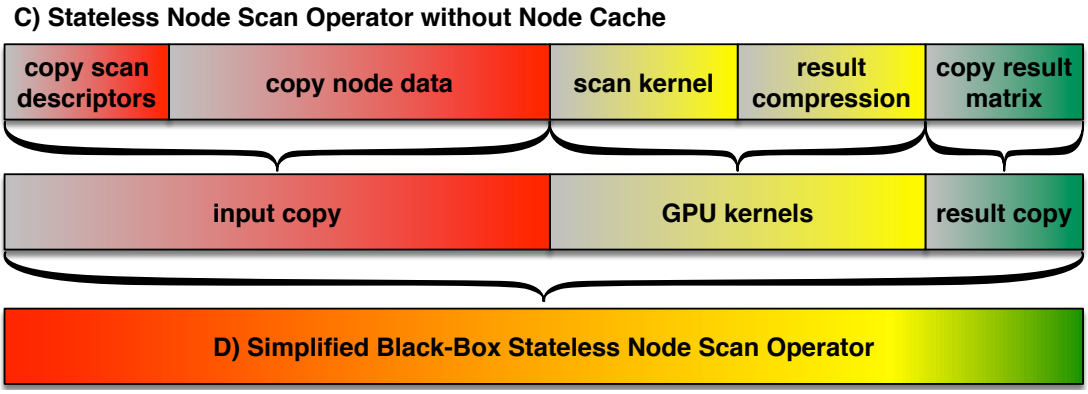


Figure 6.8: Simplified Black-Box Primitive Operator Model

6.2.3 Decision Model

The heart of the scheduler are the decision models that it employs to select the "best" algorithm for the available device types. In the prototype, a regression model is used to learn the total node scan operator runtime over the relevant parameter space based on measurement samples (cf. Figures 5.13 and 5.14). That is, by providing pairs of (parameter, runtime)-measures, the function is trained that can be used later to estimate the runtime for a given, potentially unknown parameter instance that has not been used as training sample before.

Because the prototypical implementation of the node scan operation showed the caching inefficiencies, the base implementation without caching has been selected for the following evaluation. Because it always executes the same phases, irrespective of the state of the device, the operator model from Section 6.1.2 and Figure 6.3 has been simplified as illustrated in Figure 6.8. For scheduling, the simplified operator does not distinguish the algorithm phases anymore. Instead, it is modeled as black-box primitive whose total costs should be estimated. For other implementations, it might make sense to estimate input copy costs analytically and sum it with estimated execution costs that is learned by using the black-box approach. The reason for this is that the input sizes as well as bus system parameters are known in advance and, thus, calculating the input transfer time is straightforward by multiplying the input size with the available bandwidth and adding a transfer latency in case of dGPUs (cf. Section 5.3.5). For CPU implementations, transfer costs can be omitted. However, the runtime for executing the scan depends on the actual predicate implementation. Thus, using a black-box approach simplifies cost modeling significantly and makes the scheduler robust against changes in the underlying implementations.

As the experiments in Section 5.3 have shown, the execution times for the batch-oriented query predicate evaluation mainly depend on the size of the input vectors, because it directly impacts the degree of parallelization that can be leveraged. Therefore, batch size and the number of child slots in a node are used as parameter dimensions for building the regression model. Both parameters can be directly derived from a to-be-scheduled node scan primitive. Other parameters from the environment, e. g., current device queue lengths, processor characteristics, etc. are

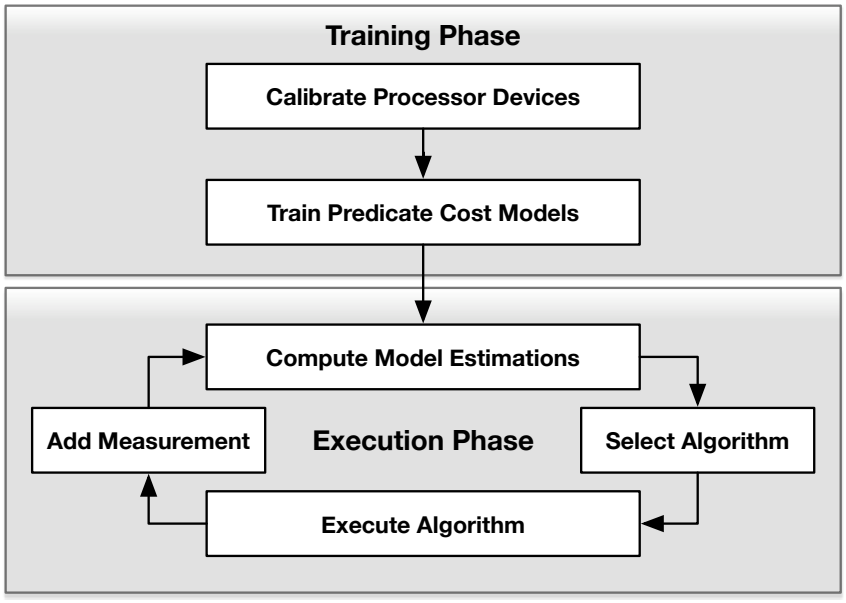


Figure 6.9: Self-Tuning Scheduler Workflow

ignored for now. Integrating such information into the scheduling models for refining estimation results is left for future work. Inter-operation optimizations, such as overlapping operator phases, are also not considered, yet. Independent primitive processing is assumed and just the total runtime for sequentially processing all operator phases is estimated in dependence of the aforementioned primitive parameters.

The initial model for an algorithm instance on a specific device is built through a calibration phase and can be incrementally adapted during runtime, what is discussed in more detail in the next section. After all models are available, they can be probed using the parameters of primitives that shall be scheduled. All estimations are calculated and the (algorithm, device)-combination is selected for execution having the minimum processing time.[1]

6.2.4 Model Maintenance

The cost models used by the scheduler have to be built and maintained in order to derive correct cost estimations. The general workflow for this process is illustrated in Figure 6.9. Since the models do not explicitly encode the cost estimation strategies, but rather learn them for the platform where the framework is deployed, they have to be trained properly.

Therefore, a *Training Phase* is used for building the initial models by probing a set of relevant parameters provided as training set that consists of different (parameter, algorithm instance)-combinations. That is, the parameter space is explored and all algorithms are executed for all available device types, measuring their runtimes that are fed into the regression model. This calibration corresponds to the node scan benchmark that has been described in Section 5.3.1.

[1] The combination of all available models for a single algorithm instance implicitly represents the speedup curves that have been examined in Section 5.3. Although the break-even points are not explicitly encoded, the decision will flip around these points where the minimum changes.

During the *Execution Phase*, the models are used to calculate estimated runtimes for any parameter combination. Depending on how much training effort was spent, the estimations are more or less accurate. Optionally, the initial models might be adapted on-the-fly during runtime by measuring actual execution times, comparing them to estimated ones, and adapting the model with the new sample.

Like any machine learning algorithm, the learned model is susceptible to effects, such as overfitting and oscillating adaptions through the feedback loop in case other parameters, such as concurrent loads on the devices, are also impacting the measured runtimes. But those issues are not followed here any further. Instead, the scheduler component is used as black box, assuming that those questions are addressed there. More details on the scheduler framework can be found in Breß' dissertation [36]. For now, it is simply assumed that the index decision models are trained under laboratory conditions, i. e., without concurrent workloads, etc., for a sufficiently large training sample to derive accurate estimates. No model adaption at runtime is used. The following experiments that have been conducted on the node scan benchmark data will show that using this strategy already leads to good estimation accuracies.

6.3 Evaluation of Generalized Query Traversals on a Hybrid CPU/GPU Platform

In order to evaluate the framework's benefits for end-to-end index query processing workloads, several experiments have been conducted that are discussed in this section. The experiments are based on a tree traversal workload simulation that is described in Section 6.3.1. By varying various parameters that impact vectorized tree traversals, different workloads can be simulated. The results of each run can be joined with the results obtained from the node scan benchmark in order to estimate how the traversal would behave on different hardware platforms.

The first goal of the evaluation is to asses whether the framework's scheduler represents a suitable approach to leverage the strengths of different processor types in hybrid systems for executing index queries. Section 6.3.2 answers the question whether automatically learned decision models can be used to derive reasonable scheduling decisions, i. e., how accurate calibrated cost estimation functions represent real operator execution on the available (co-)processors. The actual benefit of hybrid scheduling over processing index queries on a single processor type is evaluated in Section 6.3.3 for various simulation parameters. It answers the question whether the runtime improvement that can be achieved by using a GPU in conjunction with a CPU outweighs scheduling overheads, and, in total, improves the algorithms' performance for end-to-end tree traversals.

The second question that is evaluated in this section is whether batch processing, as one of the framework's core ideas, results in higher query throughputs compared to the original GiST execution model as baseline. Therefore, the hybrid batch processing model is compared to the GiST-like CPU-based query-at-a-time model in Section 6.3.4. Further, a CPU-optimized implementation that utilizes caching effects is also included in the comparisons, because the experiments in Section 5.3 have shown that the evaluation of simple query predicates is more efficient on a tuned CPU implementation than on a GPU.

6.3.1 Index Traversal Workload Simulation

As discussed in Section 5.3.1, the execution of a query batch in an index tree is impacted by many parameters that influence the performance of the CPU and GPU node scan operations. The impact of the *number of slots* as well as the *query batch size* has already been analyzed in the node scan benchmark for the scope of a single node. In order to evaluate how different scheduling strategies influence end-to-end query performance, i. e., when queries flow through an entire tree, the whole traversal process needs to be performed, making scheduling decisions for each node that is visited along the way from the tree's root towards its leaf layer.

As illustrated in Figure 6.10, this traversal can be simulated for different workload characteristics, utilizing the fact that a logical tree traversal is independent from the physical implementation of the underlying algorithms. That is, each node scan is processed using the same (simulated) primitive algorithm and, thus, produces the same results for the given inputs parameters, no matter which predicate scan implementation is actually applied on which hardware platform. In case of inner nodes, the result of a scan is a set of input parameters for the next traversal iteration that is fed into the same simulation again. Otherwise, i. e., in case of leaf nodes, the traversal terminates. The *node scan simulation* can be configured with different *workload parameters* that influence which output shall be produced. For each workload class, the primitive parameters that occured during the entire simulation can be recorded and used afterwards to simulate different scheduling strategies that select one of the available execution units for a given parameter instance. Because the operator runtimes have already been profiled for the relevant parameter space, the measurements from the previous chapter can be reused as *cost model* to calculate runtimes for the whole traversal.

The algorithm that is used to simulate a tree traversal is depicted in Algorithm 6.1. For simplicity, it assumes that the *number of slots* does not change after the index was created and the tree is fully populated, i. e., all slots are filled in each node, which implies that all possible nodes exist for a given *tree height*. This assumption is satisfied to some extend, because the underlying GiST framework implements height-balanced trees, where node maintenance algorithms guarantee minimum fill factors. For modeling other scenarios, the simulation might be adapted in this respect. Because the tree parameters are considered static, the only thing that the simulation has to produce is a histogram of batch sizes that occur when an initial set of *root queries* flows through the tree (line 31).

When the simulation is initialized, it simply creates an empty histogram and inserts a set of initial batches into the queue of operations that still need to be processed (lines 1–2). Two variants exist here (cf. Algorithm 6.2) – either all batches are processed independently or all batches that would belong to the same node are merged after each iteration in order to maximize batch sizes during the traversal. In the independent processing case, a large initial batch exceeding the maximum size is split into smaller ones that are treated as if they originate from different tree nodes above the root (Algorithm 6.2, lines 1–12) and, thus, would be simulated independently in the following. If batch merging is simulated, merely one initial batch comprising all root queries is created (Algorithm 6.2, lines 13–16).

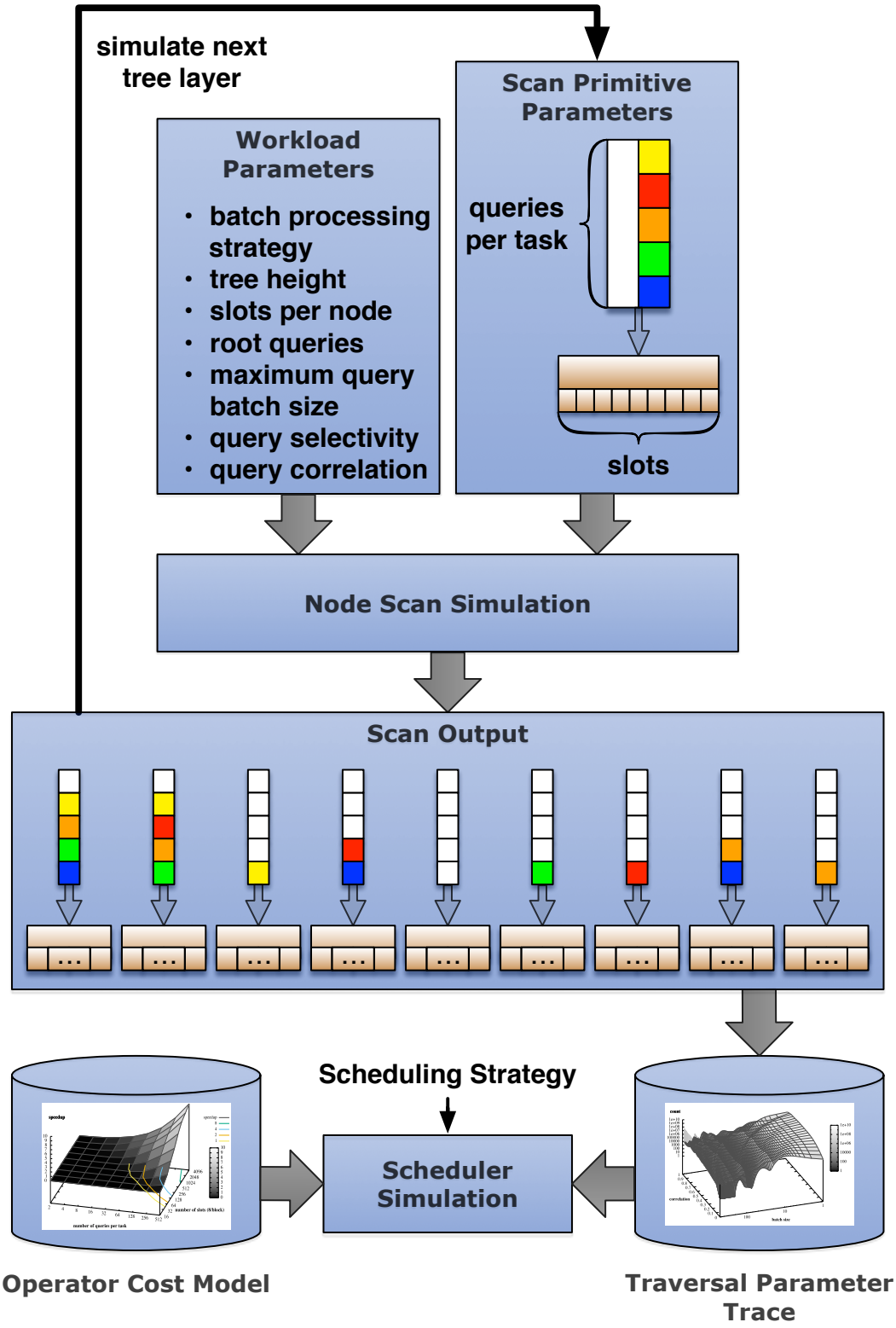


Figure 6.10: Index Tree Traversal Simulation

Algorithm 6.1 Index Query Simulation

Input: slotsPerNode, treeHeight, numRootQueries, maxBatchSize, batchProcessingStrategy, queryCorrelation (in %), querySelectivity (in %)
Output: batchSizeHistogram

```
1:  batchSizeHistogram = Histogram( maxBatchSize )
2:  rootBatches = createRootBatches( primitiveQueue, numRootQueries,
                                     maxBatchSize, batchProcessingStrategy )
3:  totalEntries = querySelectivity / 100 * slotsPerNode ^ treeHeight
4:  nodeSelectivity = treeHeight-root(totalEntries) / slotsPerNode
5:  while not primitiveQueue.empty() do
6:     ( treeLayer, numQueries ) = primitiveQueue.pop()
7:     fullBatches = numQueries / maxBatchSize
8:     partialBatchSize = numQueries % maxBatchSize
9:     batchSizeHistogram[ maxBatchSize ] += fullBatches
10:    batchSizeHistogram[ partialBatchSize ] += 1
11:    if treeLayer < treeHeight then
12:       childQueries = Histogram( slotsPerNode )
13:       numMatchingSlots = nodeSelectivity * slotsPerNode
14:       for query ∈ [0 .. numQueries) do
15:          correlatedQueries = 0
16:          for slot ∈ [0 .. numMatchingSlots) do
17:             if rand( 0, 100 ) < queryCorrelation then
18:                childQueries[ correlatedQueries ] += 1
19:                correlatedQueries++
20:             end if
21:          end for
22:          remainingSlots = randomShuffle( [correlatedQueries .. slotsPerNode) )
23:          for slot ∈ [correlatedQueries .. numMatchingSlots) do
24:             matchingChild = remainingSlots[ slot ]
25:             childQueries[ matchingChild ] += 1
26:          end for
27:       end for
28:       createChildBatches( primitiveQueue, childQueries, treeLayer + 1 )
29:    end if
30: end while
31: return batchSizeHistogram
```

Algorithm 6.2 *createRootBatches*()

Input: primitiveQueue, numRootQueries, maxBatchSize, batchProcessingStrategy
Output: insert an initial set of batch simulation tasks into the queue that flow through the tree

```
1:  rootLayer = 0
2:  if batchProcessingStrategy = INDEPENDENT_BATCHES then
3:      fullBatches = numQueries / maxBatchSize
4:      partialBatchSize = numQueries % maxBatchSize
5:      for fullBatch ∈ [0 .. fullBatches) do
6:          fullRootBatch = ( rootLayer, maxBatchSize )
7:          primitiveQueue.push( fullRootBatch )
8:      end for
9:      if partialBatchSize > 0 then
10:         partialRootBatch = ( rootLayer, partialBatchSize )
11:         primitiveQueue.push( partialRootBatch )
12:     end if
13: else // simulate batch merging for each node
14:     rootBatch = ( rootLayer, numRootQueries )
15:     primitiveQueue.push( rootBatch )
16: end if
```

Algorithm 6.3 *createChildBatches*()

Input: primitiveQueue, childQueries, treeLayer
Output: pushes a new simulation task for all matching children

```
1:  for batchSize ∈ childQueries do
2:      if batchSize > 0 then
3:          childBatch = ( treeLayer, batchSize )
4:          primitiveQueue.push( childBatch )
5:      end if
6:  end for
```

The main simulation considers hardware-specific bounds for the maximum batch size that can be used for a node scan by splitting a batch into batches of maximum size plus a smaller-sized remainder (lines 7–10). This maximum batch size parameter defines the number of buckets in the output and can be used to optimize batch sizes for one specific hardware platform, e. g., smaller ones for CPUs and larger ones for GPUs. If the maximum tree layer has not been reached, yet, i. e., an inner node is simulated in the current iteration, the actual workload simulation is performed, which, based on the workload class parameters, generates a list of batches for all child nodes (lines 11–29) and inserts them into the queue (line 28, Algorithm 6.3). Because the maximum batch size is ignored here, it is simulated that all matching queries for one child node would be merged into a single batch after the iteration, which will be virtually split into the minimum number of batches once this child is processed. Independent batch processing is achieved by splitting the initial batch beforehand, which guarantees that no child batch exceeding the maximum size exists.

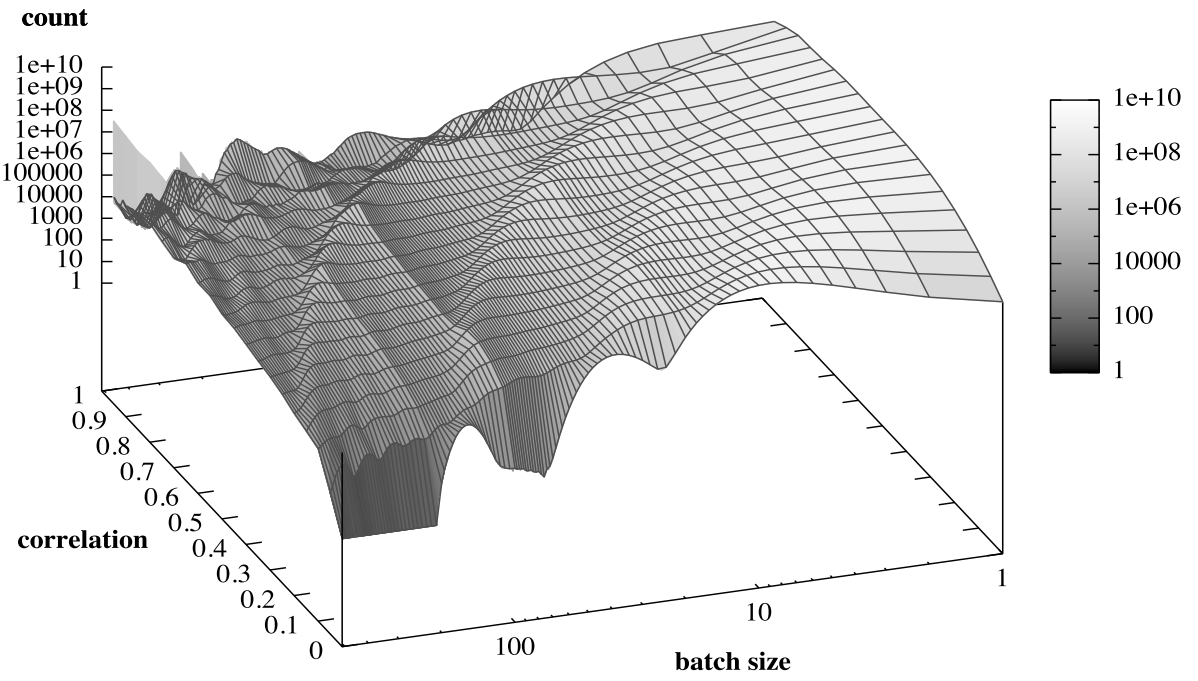


Figure 6.11: Distribution of Query Batch Sizes during a simulated Tree Traversal with varying Query Correlation, 0.1% Selectivity, and an Initial Batch of 10,000 Queries, where result batches a merged for each selected tree node

The child node batches are also simply constructed as histogram (line 12), because all queries are treated equally during the simulation and it does not matter which query follows which path in the tree. Workload parameters influencing the generation of child batches are the *selectivity* and *correlation* of the simulated query predicate. The *nodeSelectivity* denotes how many child nodes are selected by the predicate (line 13). It is derived from the desired *querySelectivity* that denotes how many result entries shall be selected as ratio of the total number of entries in the tree (lines 3–4). The latter can be derived from the number of nodes and the tree height, assuming a balanced tree with full nodes. Currently the simulation merely assumes a uniform data distribution by keeping the selectivity constant for all nodes during entire traversal. If skew should be simulated, this needs to be adapted, e. g., by increasing the selectivity on the path from the root towards the leaf nodes, which would reduce the number of nodes that need to be visited.

The distribution of queries over a node's slots and, therefore, over the different search space regions the tree is covering, is configured via the *correlation* parameter. It influences which of the slots are selected by a single query, relative to the others in the same batch (lines 15–21). The correlation is modeled as probability to select the next slot with the lowest unused ID. The remaining, uncorrelated slots are chosen randomly with equal probability (lines 22–26). Metaphorically speaking, correlated queries follow the leftmost paths in the tree while all others are evenly distributed over all possible paths.

Each simulation run produces a histogram of query batch sizes for each tree layer, which can be aggregated and joined with the execution times of a single batch on different processor devices. The aggregated histograms for varying query correlations are illustrated in Figure 6.11 for a full tree with 5 layers and 96 slots per node, i. e., a total number of ≈8 billion indexed entries, which is a realistic data volume, e. g., for CAD applications. Each histogram corresponds to one (batch size, count)-slice in the surface plot. One can see that small batch sizes dominate the overall figure and just a small number of larger batches exist. This is expected, because queries spread more inside the tree with each simulated layer. The amplitudes of the corresponding "waves" differ in at least one order of magnitude, because, depending on the correlation, input batch sizes are reduced for the next layer when queries are distributed over the approx. 100 slots. For the zero-correlation case, queries are uniformly distributed over the entire search space, leading to a large number of small batches that mostly occur at the leaf level. An increasing correlation flattens the surface, because queries select the same regions and, therefore, form less, but larger batches for subsequent layers. There is a minor peak for the maximum batch size, which results from the previously described cutoff mechanism. Varying the selectivity mostly impacts the height of the curve as it represents a simple factor, which increases the number of result items that are distributed using the correlation setting.

The impact of query selectivity and correlation is illustrated in more detail in Figure 6.12. An increasing selectivity significantly impacts the total number of batches, because the number of queries grows exponentially with the number of tree layers. Increasing the correlation reduces the number batches, but increases their sizes, because queries follow similar paths. In general, batch sizes are reduced with each additional tree layer, which can be seen best for the 0 %-Correlation case, where the mean value shifts from right to left. There is an exception, however, for tree layers 2 and 3, where layer 3 should have a lower mean than layer 2, because it is near the leaf nodes. This effect is caused by the simulated batch merging after each iteration. The maximum-sized ones can be seen at the peak on the right in each diagram. All the other batch sizes illustrate the distribution of remainders after the artificial split. After layer 3, there are no max-sized batches anymore and the size of all remainders can only decrease after this point.

The aggregated batch sizes over all tree layers are illustrated at the bottom of Figure 6.12. On the secondary axis, the speedup values from the node scan benchmark are plotted. In order to calculate the impact of a scheduling decision, the corresponding runtimes are joined with the batch count. Currently, the simulation merely multiplies them, which would correspond to a sequential execution of all node evaluations during the traversal – like a single-threaded execution of each node-batch operation on a single core. Inter-operator optimizations, such as executing multiple scans at once, what would be useful for fully utilizing all available processing resources, of GPUs in particular, are not considered in the following. Such an optimization would shift the benchmark lines, but does not impact the general shape of the curves. Thus, the runtime and throughput metrics in the following have to be interpreted as rather theoretical, but comparable values that denote an abstract definition of "work", which is executed for a single batch on a device. This suffices to analyze and discuss the impact of different processing strategies, but does not yield absolute values that are comparable to other real-world scenarios. An evaluation of the latter is not in the scope of this thesis and left for future work. The vertical bars belonging to the speedup values illustrate the break even points. All batches to the left of them would perform better on the CPU, while the batches on the right excel on a GPU.

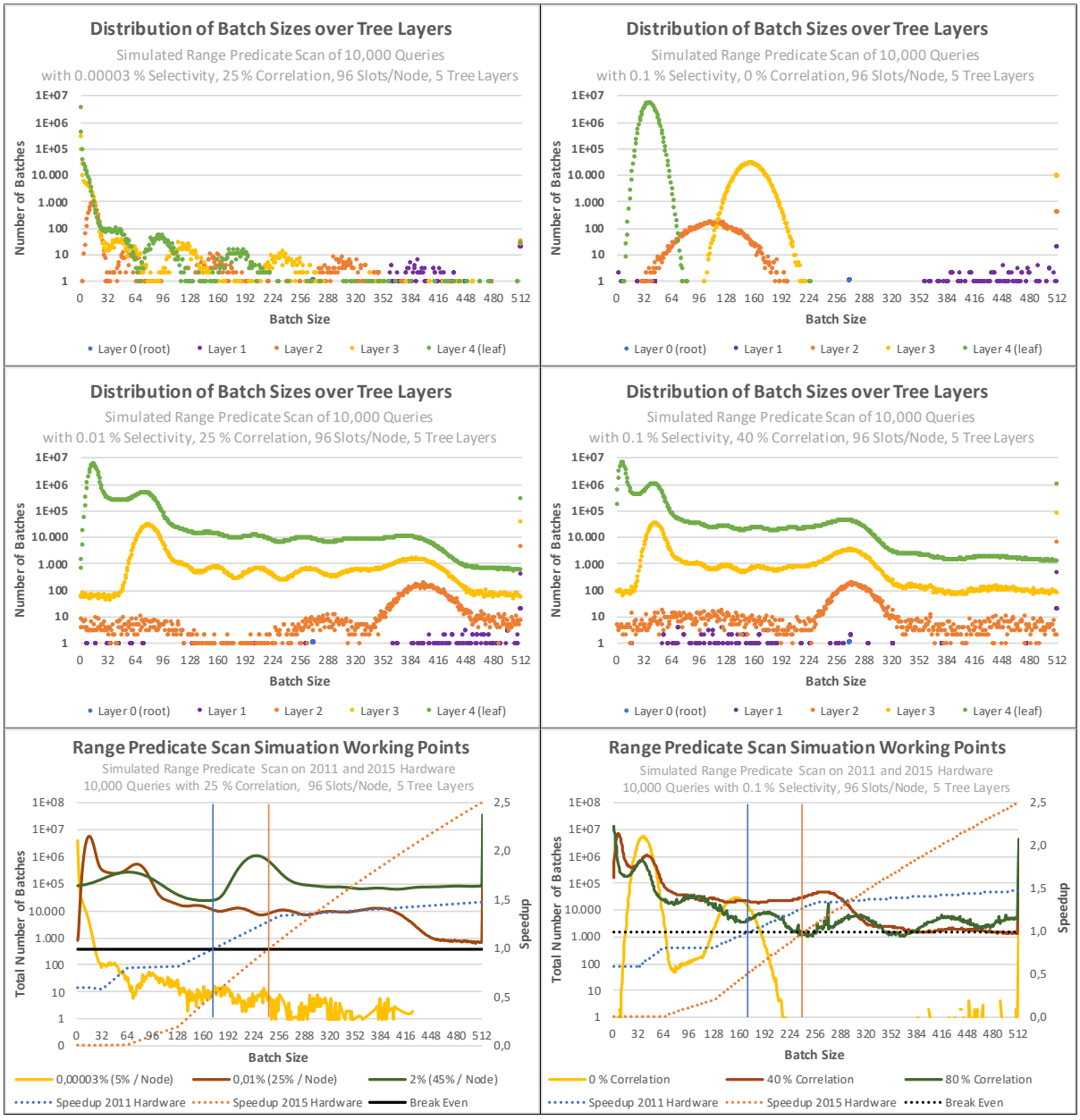
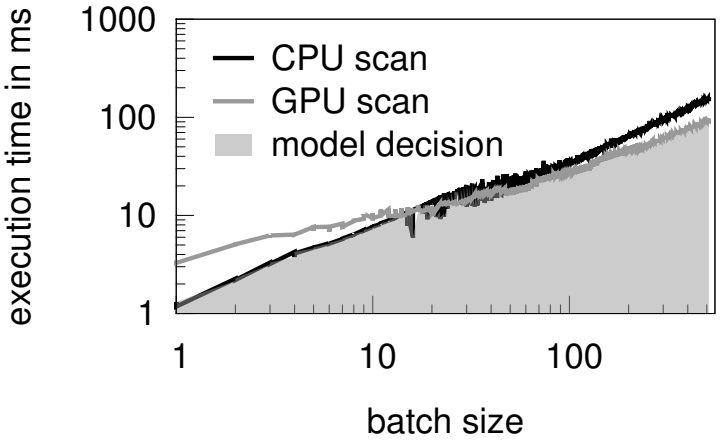


Figure 6.12: Impact of Query Selectivity and Correlation on a Simulated Tree Traversal
Low Selectivity (left, top), Medium Selectivity (left, middle)
No Correlation (right, top), Medium Correlation (right, middle)
Aggregated Batch Sizes and Corresponding CPU/GPU Speedup Values (bottom)



Algorithm	Relative Error
CPU Scan	8.99 %
GPU Scan	3.99 %

Figure 6.13: Range Predicate Regression Model Decisions after Training Phase

6.3.2 Decision Model Accuracy

A decision model that is employed by the scheduler during the final step of the workload simulation selects an execution unit for a given primitive instance. Ideally, this would be the one having the lowest execution costs for the corresponding parameter values, which will be referred to as the *ideal strategy* in the following. But in reality, the proposed strategy that learns and predicts execution costs via regression on training data is expected to have some inaccuracies caused by estimation errors. These errors shall be analyzed here in order to assess whether a regression-based training of such a model is a valid approach for the indexing use case.

In order to train the model, the operator runtime measurements obtained in Section 5.3 were used to feed the regression algorithm that has to select between the CPU and GPU classes. For choosing appropriate statistical methods, several experiments have been performed with the ALGLIB [31] package. Using the *least squares method* and *spline interpolation* led to good estimation errors for one-dimensional parameters and *multi-parameter fitting* for multi-dimensional ones. Because the simulation currently only uses the batch size as parameter, the optimization problem is one-dimensional. If a node's fill factor is also modified, i. e., the number of slots is varied, it becomes a two-dimensional problem that corresponds to learning the speedup plots over the (batch size, slots)-parameter space.

The results for the range predicate evaluation on the 2011 hardware are illustrated in Figure 6.13. The diagram on the left hand side illustrates the runtimes of each algorithm as solid lines. For smaller input sizes, the overhead for invoking GPU kernels and data transfers dominate the execution and the CPU algorithm is faster. On larger inputs, the GPU can fully utilize its advantage through parallel processing when computation becomes dominating. The break-even points, where both curves cross, can be observed for medium-sized batches, which should be found by the trained decision model. This confirms the results from Section 5.3.2, where speedups as the quotient of both curves has been discussed.

Model decisions after calibrating the model are illustrated as shaded area. By looking at this result qualitatively, the model made the right decisions, because the shaded area stays underneath the minimum curve. Decision errors are not visually observable, because they occur near the break-even point(s) where both curves overlap. Quantitative prediction error results that were aggregated over samples from the entire parameter space are listed in the table on the right hand side. The error rates are below 10%, which is considered sufficiently good for guiding indexing scheduling decisions, because the absolute errors are not relevant as long as the final decision is made right for those cases, where execution times differ significantly. Deviations around break-even points do not have major impacts on the overall runtime, because operator execution times are almost equal on both hardware platforms for these cases.

In summary, the qualitative results prove that the regression-based approach for training a hybrid scheduler is suitable for the analyzed scenario. The evaluation has not been repeated for the other node scan benchmarks in Section 5.3, i. e., neither for a newer hardware configuration nor the nearest neighbor predicate, because the qualitative results are not expected to change. The other speedup curves have comparable shapes, just with different slopes for the runtime measures, and show clear break-even points that have correctly been found for the already analyzed sample.

6.3.3 Hybrid Scheduling Strategies

The sole decision accuracy is not sufficient to evaluate scheduling models as a whole. Although wrong scheduling decision may lead to severe performance degradations for a single operator execution, it may have negligible consequences too. The latter happens when a wrong decision is made for parameters near break-even-points, where runtimes of the available algorithms are almost equal. Therefore, a different metric has to be defined that evaluates how much the application would benefit in total by using a hybrid scheduling strategy instead of another one for a specific workload.

To quantify the performance gain for end-to-end tree traversal, the *model speedup* metric is used, which is defined as:

$$model\ speedup(DM_i \rightarrow DM_j, W) = \frac{T_{DM_i}(W)}{T_{DM_j}(W)} \tag{6.1}$$

This ratio indicates how the measures used as optimization goal, T for total execution runtime in this case, will change, when a decision model DM_j would be used instead of another decision model DM_i for specific workload W. A hybrid approach j is beneficial when the speedup, over trivial models i that always choose the same algorithm for the same processing device, is greater than 1. Otherwise, longer runtimes caused by making the decision, potential decision errors, and other overheads, e. g., for learning and adapting model parameters during runtime, cancel any performance gain.

In general, a workload represents a set of tasks that shall be scheduled based on their parameters. In the following, the simulated workloads described in Section 6.3.1 have been used and the following scheduling strategies were evaluated:

- DM_{hybrid}, which represents the hybrid scheduling model that was learned during the calibration phase;
- $DM_{optimal}$, which represents the hypothetical optimal hybrid model that always selects the fastest algorithm for the best processing device;
- DM_{CPU} and DM_{GPU}, which represent trivial decision models that always select the same algorithm for the same processing device, independent from any task parameters.

6.3.3.1 Runtime Impact of Hybrid Decision Models

The objective of the first experiment is to assess whether the speedups for evaluating a query batch on a single tree node do also result in speedups, when hybrid scheduling is applied for end-to-end tree traversals. Therefore, the simulation has been executed for the following parameters and the results for different (static) scheduling models have been compared for different query selectivities and correlations, which would be dynamic at runtime:

Tree Parameters:

- Slots per Node: 96
- Tree Height: 5
- Simulated Number of Entries: $96^5 = 8,153,726,976$

Hardware Platforms:

- 2011 CPU (Intel Xeon X5690)
- 2011 GPU (Nvidia Tesla C2050)

Query Processing Parameters:

- Predicate: Range Query for 3D R-Tree
- Root Queries: 10,000
- Max Batch Size: 512 (GPU-optimized)
- Batch Processing Strategy: Merging (maximizing batch size)

Scheduling Strategies:

- *CPU-only*
- *GPU-only*
- *Hybrid*
- *Optimal Hybrid*

A 5-layer tree has been simulated having 96 slots per node, because this configuration has shown clear regions of CPU- and GPU-optimal batch sizes in the stateless range predicate node scan benchmark. Those differences are expected to impact the total query evaluation runtimes on the examined hybrid system. Only the 2011 hardware has been examined here, because a single platform suffices to compare the impact of scheduling decisions if there are cases where one processor types outperforms the other. The different hardware generations are compared with each other in the next section.

The simulated tree stores a total of ≈ 8 billion entries (≈ 99% in the leaf layer), which corresponds to a 3-dimensional R-Tree of size 364.5 GB, assuming that each MBR key is stored uncompressed as two 3-dimensional points using 8 bytes per coordinate (double-precision). This tree may be stored in-memory on a reasonably-sized host, but does not fit into the GPU memory, satisfying the assumption of the benchmark. As query workload, 10,000 range predicate queries have been selected arbitrarily, assuming that the application can generate a sufficiently large batch at once, e. g., for an index-nested-loop join. The simulation has been configured to use a maximum batch size of 512, as the maximum for caching in GPU shared memory, and to merge the result batches of each node after each iteration. This setting optimizes for the GPU, because batch sizes are maximized during the traversal. The reason for this is that in a hybrid processing model, the GPU should take as much load as possible, leaving the majority of remaining lower-sized batches for CPU execution, where offloading does not pay off.[2]

Each combination of two scheduling strategies can be compared with each other using the speedup metric in order to assess potential performance gains (or losses) when a base model is substituted by another one. The *ideal decision model* represents the upper bound for improvements that can never be achieved by any hybrid scheduling approach in case decision errors occur or additional runtime overheads are considered. But it can be used as replacement for another model to evaluate how close an implementable strategy gets to the theoretical optimum when it is employed instead. The *trivial strategies* do not make any real scheduling decisions, because just a single device type is used. Depending on the workload parameters, i. e., the corresponding batches that need to be processed, it is expected that these strategies suffer from low performance for tasks where the device is not optimal. Because such differences can be significant, this might negatively impact the overall runtime, depending on how often such task configurations occur in the whole end-to-end query run. Comparing them to the ideal strategy gives a clue how much performance is lost by not implementing a hybrid model. The *hybrid decision model* should not show such inefficiencies, because it switches to the best-fitting device type after analyzing the operation's parameters. As the evaluation in the previous section has shown, involved error rates are quite low and, thus, it is expected that the hybrid scheduling algorithm gets close to the ideal optimum. Comparing the hybrid decision model to the trivial ones should indicate attainable performance gains for real-world implementations, provided that the workload matches the simulated conditions.

The results of the different scheduling strategies for the 2011 hardware are illustrated in Figure 6.14. The normalized total runtimes for the simulated end-to-end tree traversal of 10,000 queries are illustrated as bars on the primary y-axis, model speedups are illustrated as interpolated lines on the secondary y-axis. The upper figure shows how these metrics are impacted when the query correlation is increased, while the selectivity is varied in the lower figure.

[2] This assumes linear behavior of the execution runtimes corresponding to a speedup value. However, as the evaluation in Section 5.3.2.1 has shown, this is not necessarily true due to caching effects on the CPU. The impact of the latter are examined later in Section 6.3.4.

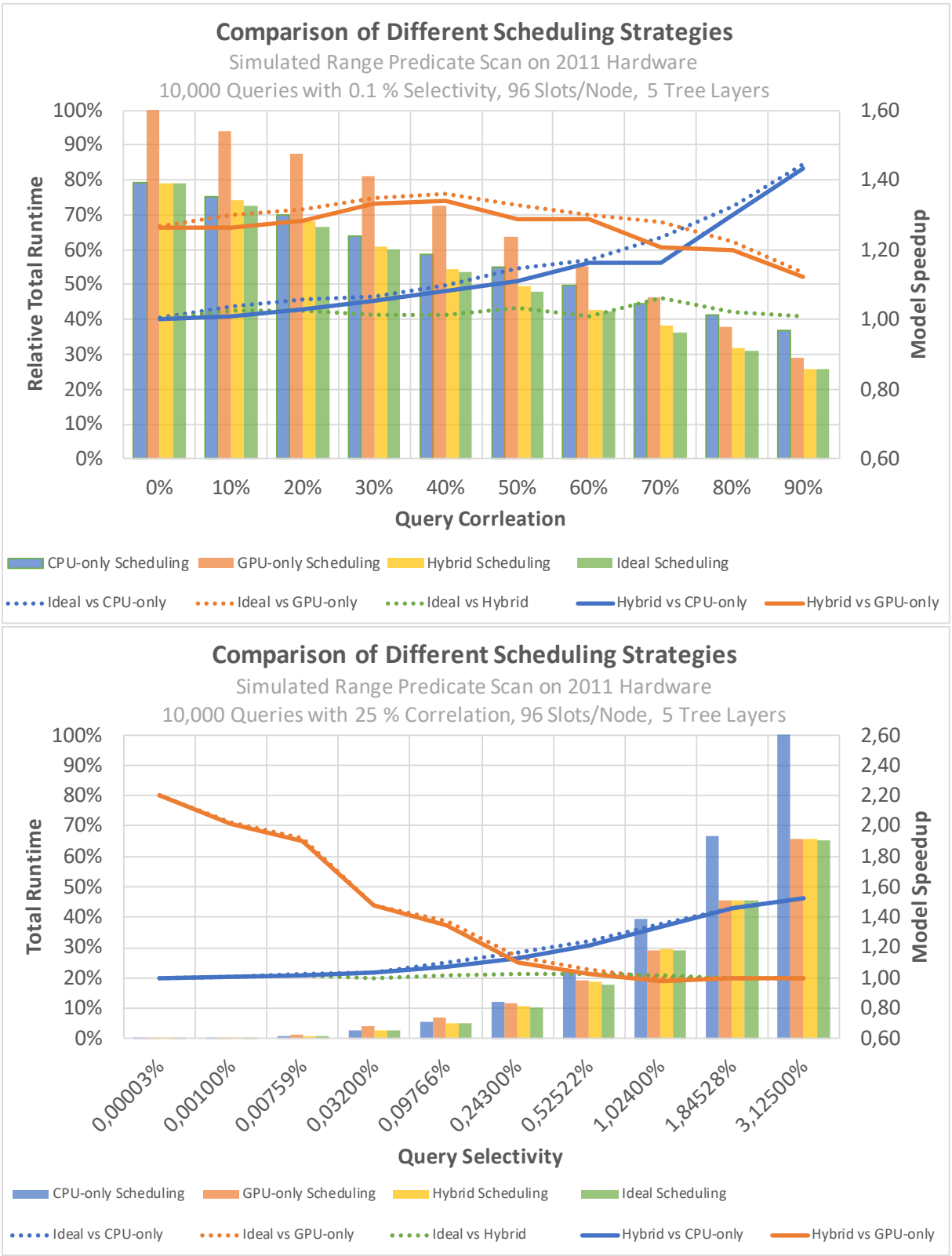


Figure 6.14: Comparison of Different Scheduling Strategies on 2011 Hardware

By looking at the total runtimes, one can see that an increasing correlation leads to decreasing execution times, because queries select similar paths in the tree and, therefore, a lower number of nodes has to be visited during the traversal. When selectivity is increasing, the runtimes increase significantly, because more nodes need to be visited in each iteration. By comparing total execution times of the trivial strategies with each other, it can be seen that the best device changes for end-to-end traversals for certain parameter combinations, even without employing hybrid scheduling (for workloads with 70% – 80% correlation and 0.1% selectivity / 25% correlation and 0.1% – 0.2% selectivity). High query correlations lead to large batch sizes, which favors the GPU, while high selectivities lead to large fan-outs in each iteration and lower batch sizes, which is good for the CPU. In all cases, the total execution times for the hybrid strategy stay below or are almost equal to the minimum of the trivial strategy runtimes, which indicates that it is beneficial to switch between devices on a per-operator basis while the tree is traversed. The hybrid scheduling runtimes are almost equal to the ideal one, which means that decision errors did not impact the overall query runtimes noticeably, because these errors occur around break-even points.

The model speedup that is also illustrated in the diagrams allows even better comparisons of the different scheduling strategies. Comparing the ideal strategy to the trivial versions reveals significant room for improvement of up to a factor of 2.2 that could be leveraged by applying a hybrid strategy. This is caused by the large batches at the topmost tree layers and the many small-sized batches near its leaves. The model improvements comparing the hybrid scheduling models with the trivial ones are close to the ideal case. That is, the theoretical improvement factors can almost be reached using the implemented hybrid strategy. Directly comparing the latter to the ideal model shows that it only differs slightly from the optimum, having a speedup ≈ 1 over the entire parameter space. That is, as expected, decision errors that occurred during the simulated traversal have a low impact on the overall runtime. The experiment with varying selectivities shows that for very high selectivities, choosing the trivial GPU-only approach would not cause any notable performance degradations, because each query batch produces a lot of child batches of size near or above the respective break-even point. When this point at $\approx 0.5\%$ selectivity is reached, the learned decision model sometimes, incorrectly, suggests the CPU due to approximation errors, leading to a speedup that is slightly below 1, i. e., a slowdown. But it stays below 3%, which is acceptable for the performance gains that can be achieved via hybrid platform usage for all the other cases.

The experiment has shown that, under the assumptions that have been made, using a hybrid processing model is beneficial for hybrid systems and can lead to significant performance improvements, compared to the execution on just a single processor type. Further, it has been shown that the decision errors made by the trained regression-based scheduler are negligible in the end-to-end view, because overall processing times just slightly differ from the optimal case. Thus, employing such a self-calibrating model in the framework's scheduler represents a reasonable way to design this component so that it can adapt itself to any system configuration. Because the deviation of the self-calibrated model from the optimal model is negligible, only the optimal hybrid case will be used as baseline in the following. This significantly reduces the remaining parameter space that needs to be explored.

6.3.3.2 Runtime Impact of Different Hardware Platforms

The objective of the second experiment is to evaluate how the evolution of different hardware generations impacts execution times for end-to-end tree traversals. Therefore, the simulation has been executed for the following parameters and different scheduling models have been compared for varying query selectivities and correlations, like the in the previous experiment:

Tree Parameters:

- Slots per Node: 96
- Tree Height: 5
- Simulated Number of Entries: $96^5 = 8,153,726,976$

Hardware Platforms:

- *2011 CPU (Intel Xeon X5690)*
- *2011 GPU (Nvidia Tesla C2050)*
- *2015 CPU (Intel Core i7-6700)*
- *2015 GPU (Nvidia GeForce GTX TITAN)*

Query Processing Parameters:

- Predicate: Range Query for 3D R-Tree
- Root Queries: 10,000
- Max Batch Size: 512 (GPU-optimized)
- Batch Processing Strategy: Merging (maximizing batch size)

Scheduling Strategies:

- CPU-only
- GPU-only
- Optimal Hybrid

The setup is similar to the previous evaluation, but in addition to the 2011 hardware system also evaluates the 2015 setup. Because the results of the trained scheduling model did not differ significantly from the optimal case in the previous setup, only the optimal strategy is compared as hybrid model to the trivial ones, which enhances readability of the diagrams and also saves model training efforts. All the other parameters have been kept unchanged in order to gain comparable results and assess the framework's ability to tune itself to changed hardware configurations, without manual interventions. In particular, the maximum batch size has not been adapted for the newer GPU model. Increasing it would extend the range of batches that qualify for GPU execution and has slight impacts on the distribution of batch sizes during the simulation.

Because the overall speedup curve does not change qualitatively, similar results are expected for both hardware generations, i. e., significant speedups of hybrid over trivial scheduling. Quantitatively, the 2015 speedup curve has shifted slightly (cf. Figure 5.13 and Figure 6.12), i. e., larger batches are required for reaching break even points and also its slope increased significantly, which is expected to emphasize the performance differences between both processor types. Therefore, these changes should lead to much larger speedup factors when comparing hybrid and trivial decision models. Absolute throughput values are not compared here, because they are covered in the following sections.

Figure 6.15 illustrates the model speedups for the 2011 and 2015 hardware setups. As expected, the speedup factors increased significantly, in particular for the GPU-only execution for low selectivities and correlations. This can be explained by the significant runtime differences between the 2015 CPU and GPU for small-sized batches, where the CPU execution is up to 200x faster than the GPU counterpart. Because most of the batches are below the break even size (cf. Figure 6.12), this performance gap receives a very large weight when the overall runtime is accumulated, leading to overall speedups of 100x of the optimal hybrid over GPU-only scheduling. The performance benefits for large batch sizes leads to moderate speedups of hybrid over CPU-only execution. Up to 2x can be achieved here for extreme cases of very large correlations and selectivities. An interesting observation is a slight drop in the speedup value for the 90% correlation parameter, which is unexpected. This effect is analyzed in more detail in the next sections, when the entire query parameter space is explored.

The results of this experiment have shown that the evolved hardware had major impacts on the execution runtimes in the otherwise unchanged framework. That is, the hybrid approach became even more important for the new hardware setup, because it clearly favors different execution units for different workload classes. Each class is characterized by the query batch sizes in this case, which is not evenly distributed in the end-to-end evaluation. The framework is able to utilize these differences automatically, because the speedups increased without changing any configuration settings. Although some additional adjustments might be necessary to optimize algorithms for a specific device, the overall picture did not change. That is, the proposed framework is able to generalize the underlying hardware and adapt itself to an altered environment via cost model calibration.

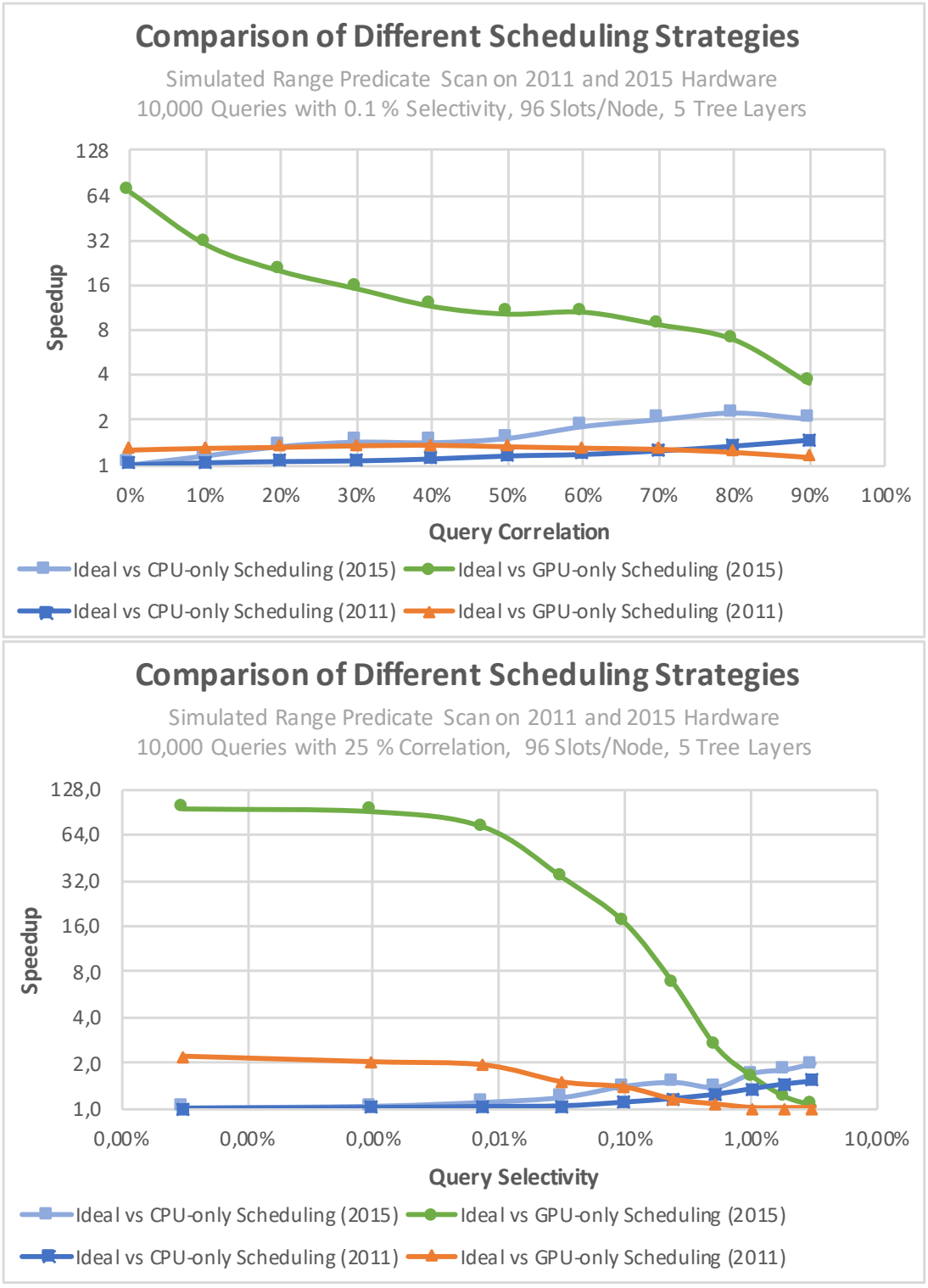


Figure 6.15: Comparison of CPU, GPU, and hybrid Scheduling Strategies on 2011 and 2015 Hardware

6.3.3.3 Runtime Impact of Batch Processing Strategies

The objective of the third experiments is to evaluate the benefits of the result batch merging that is applied by the framework after each iteration. Because all batches exceeding the maximum size that is supported by the hardware need to be partitioned internally before they can be scheduled to the device, the node scan operations are returning smaller batches for their child node entries. This partitioning reduces the amount of parallelism that can be applied in subsequent iterations. Therefore, the framework can merge the results of each child node after all batch partitions have been processed. Compared to the independent case, where all partitions are treated as independent work units, this batch merging will cause additional synchronization overheads, but maximizes batch sizes during the traversal. To analyze the impact of this setting on different batch size distribution metrics and the node scan execution on different device types, the simulation has been executed for the following parameters:

Tree Parameters:

- Slots per Node: 96
- Tree Height: 5
- Simulated Number of Entries: $96^5 = 8,153,726,976$

Hardware Platforms:

- 2015 CPU (Intel Core i7-6700)
- 2015 GPU (Nvidia GeForce GTX TITAN)

Query Processing Parameters:

- Predicate: Range Query for 3D R-Tree
- Root Queries: 10,000
- Max Batch Size: 512 (GPU-optimized)
- *Batch Processing Strategies: Merging (maximizing batch size) / Independent (no batch optimization)*

Scheduling Strategies:

- CPU-only
- GPU-only
- Optimal Hybrid

This setup uses the same workload as the previous experiments and merely modifies the framework's internal behavior when batches are evaluated for a single node. The evaluation has just been performed for the 2015 hardware, but for the full selectivity-correlation parameter space to assess the impact of the batch merge for different dynamic query workload classes.

Because the batch merging is expected to increase the average batch size during the traversal and the overall number of query predicates that need to be evaluated is independent from this, the total number of batches that need to be processed should decrease significantly, compared to the independent processing case. This should result in reduced execution times and higher query throughputs, because less node batches need to be evaluated. Further, their evaluation should be more efficient, because these batches are larger than the independently processed ones that may start with the maximum size, but quickly become smaller when queries spread over the search space with each additional tree layer.

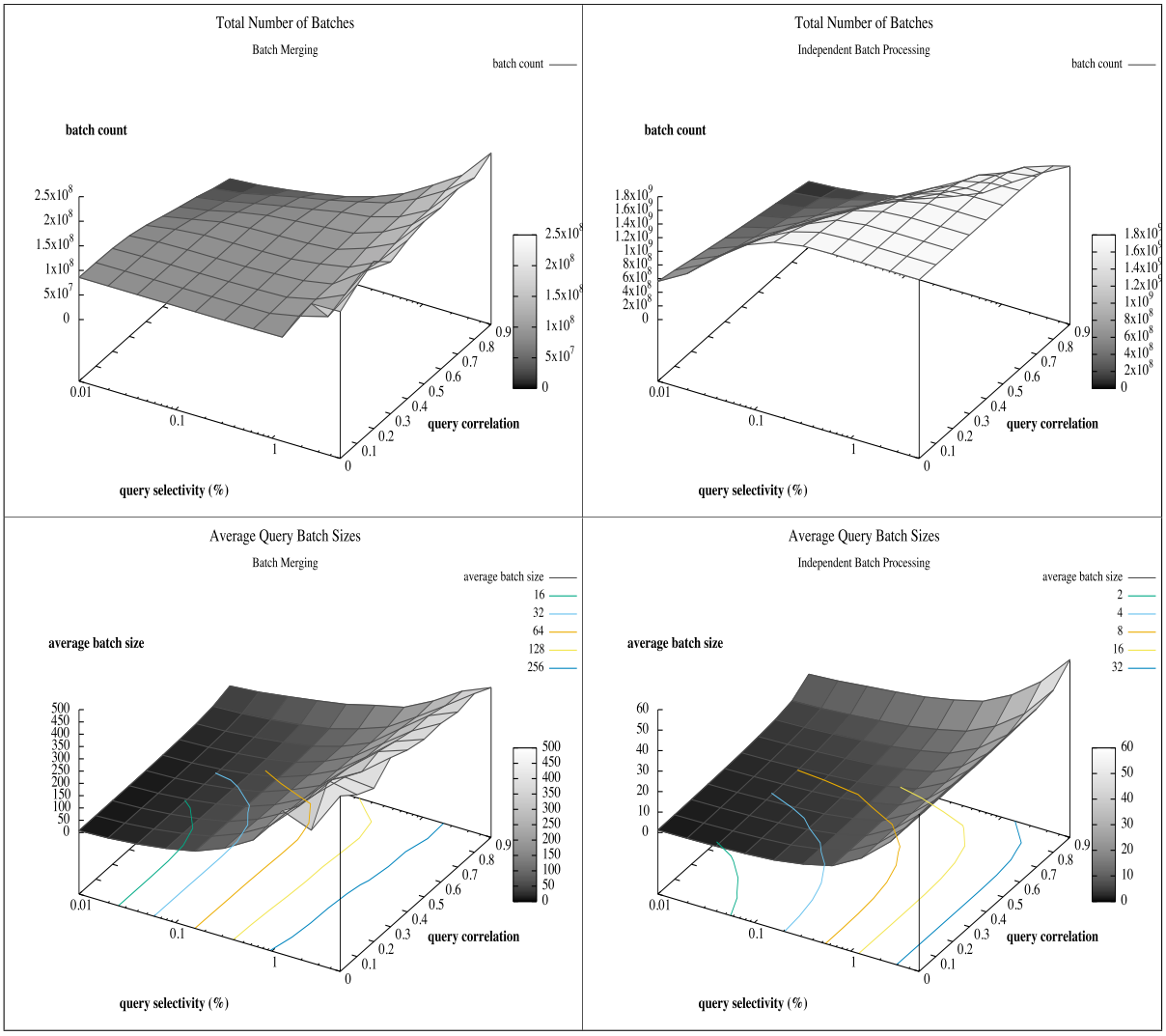


Figure 6.16: Batch Processing Strategies: Total Batch Distribution Metrics
Batch Merging (left), Independent Batch Processing (right)
Total Number of Batches (top), Average Batch Size (bottom)

Different batch size metrics are depicted in Figures 6.16, showing how the total number of batches (top) and their average size (bottom) changes when batches are merged after each iteration (left) or processed independently (right). As expected, the merge after each iteration reduces the total number of batches significantly, by an order of magnitude in this scenario. For large selectivities and without merging, the number of batches remains flat, which indicates that all nodes are covered by some of the initial queries. The average batch size also behaves as expected, i. e., it increases with increasing correlation and selectivity and this increase is much larger when merging is applied.

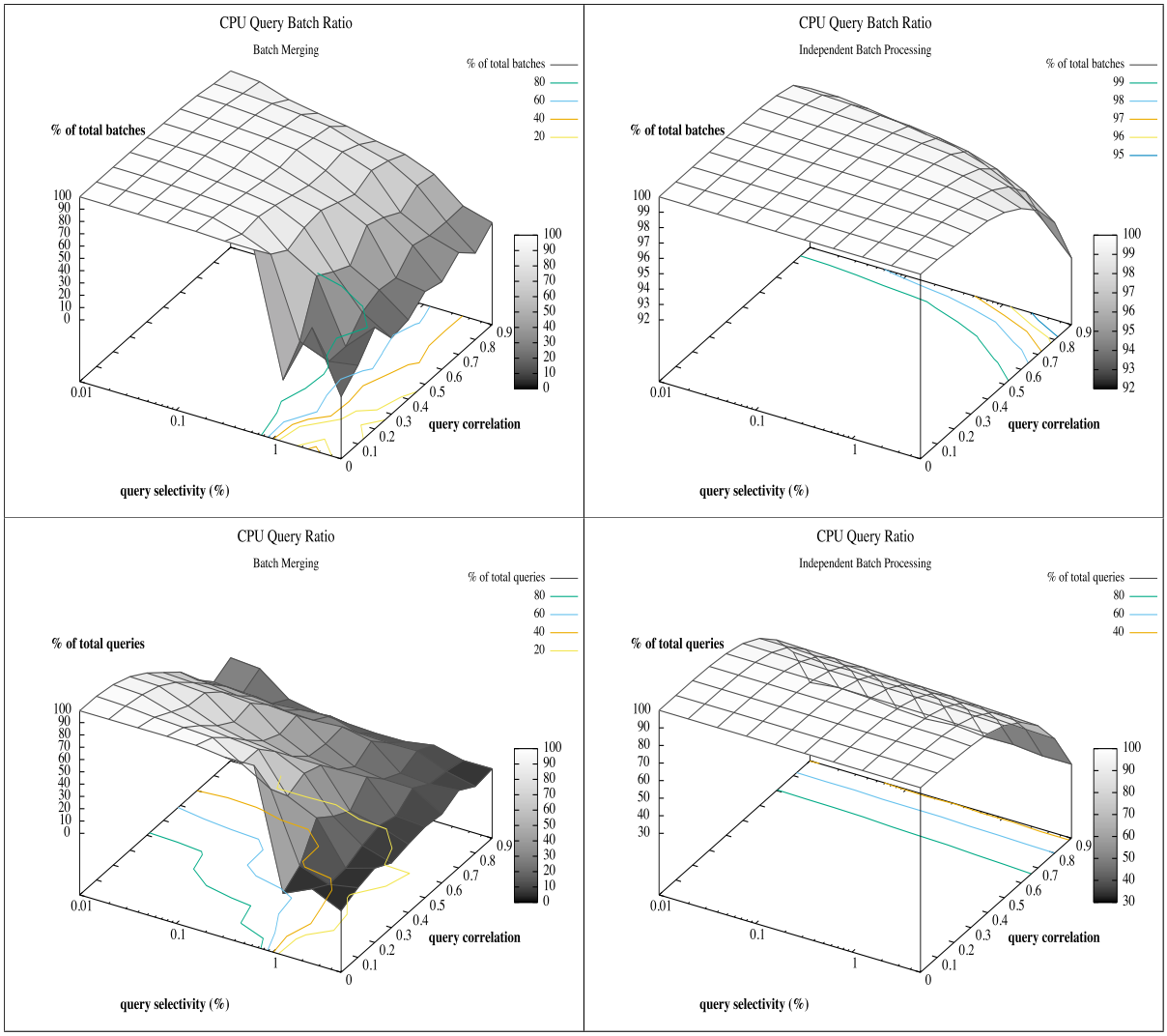


Figure 6.17: Batch Processing Strategies: CPU Batch Distribution Metrics
Batch Merging (left), Independent Batch Processing (right)
Batch Ratio (top), Query Ratio (bottom)

The larger batches also shift the entire workload towards GPU processing, which can clearly be seen in Figure 6.17 and Figure 6.18 that illustrate the ratio of all batches and their corresponding queries, where the CPU processing is faster than the GPU (Figure 6.17) and vice versa (Figure 6.18). While almost all batches qualify for CPU processing for small selectivities and correlations, this changes quickly for the batch merging case once these parameters increase. As expected, this does not happen for independent processing. It is remarkable that, although only a minor fraction of all batches exceed the break even batch size here (less than 8%), they cover the majority of all queries that need to be processed during the entire traversal if they are correlated. That is, even if no batch merging is applied, the GPU may accelerate a large fraction of the entire query processing load when the initial batch size is sufficiently large.

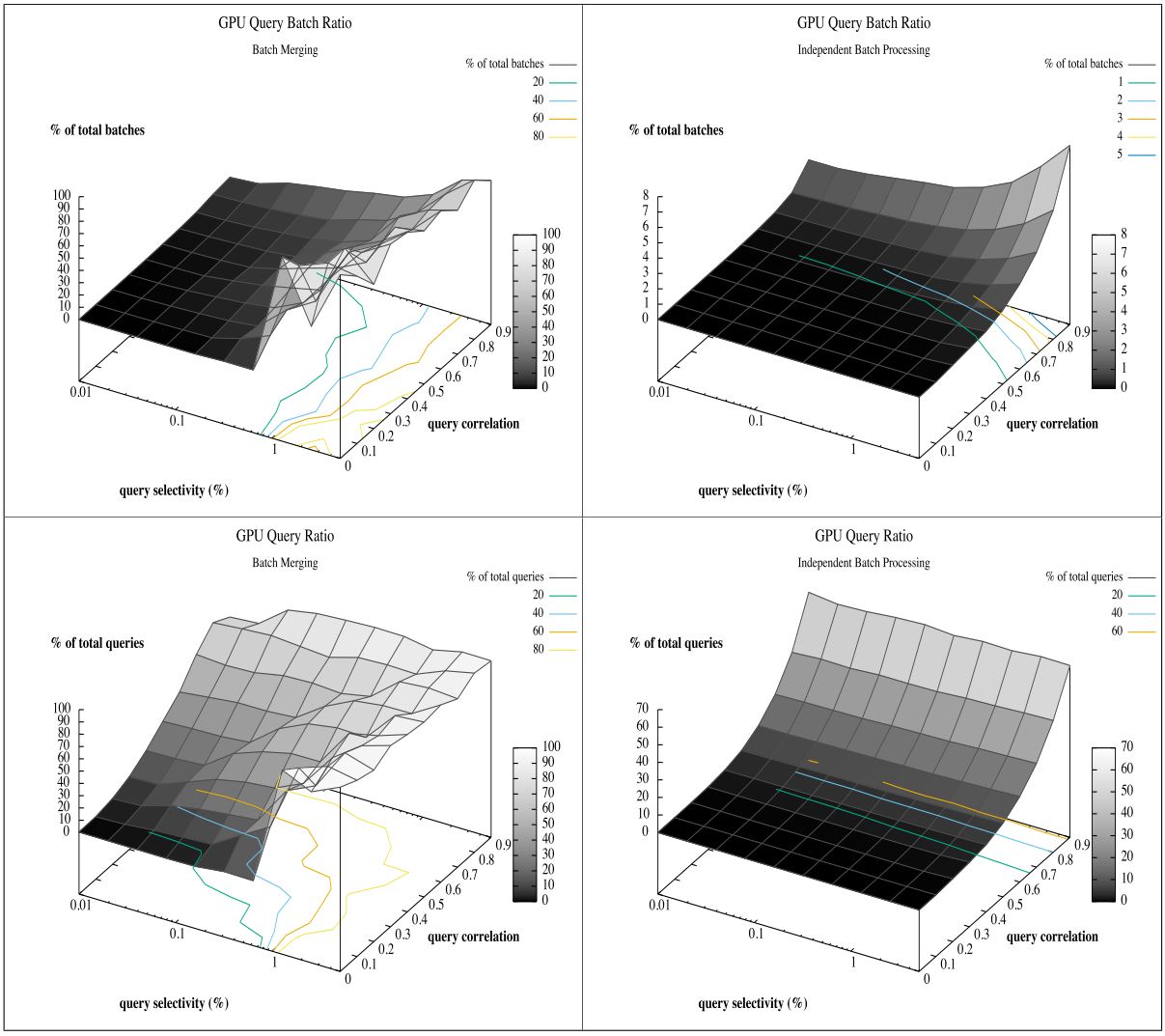


Figure 6.18: Batch Processing Strategies: GPU Batch Distribution Metrics
Batch Merging (left), Independent Batch Processing (right)
Batch Ratio (top), Query Ratio (bottom)

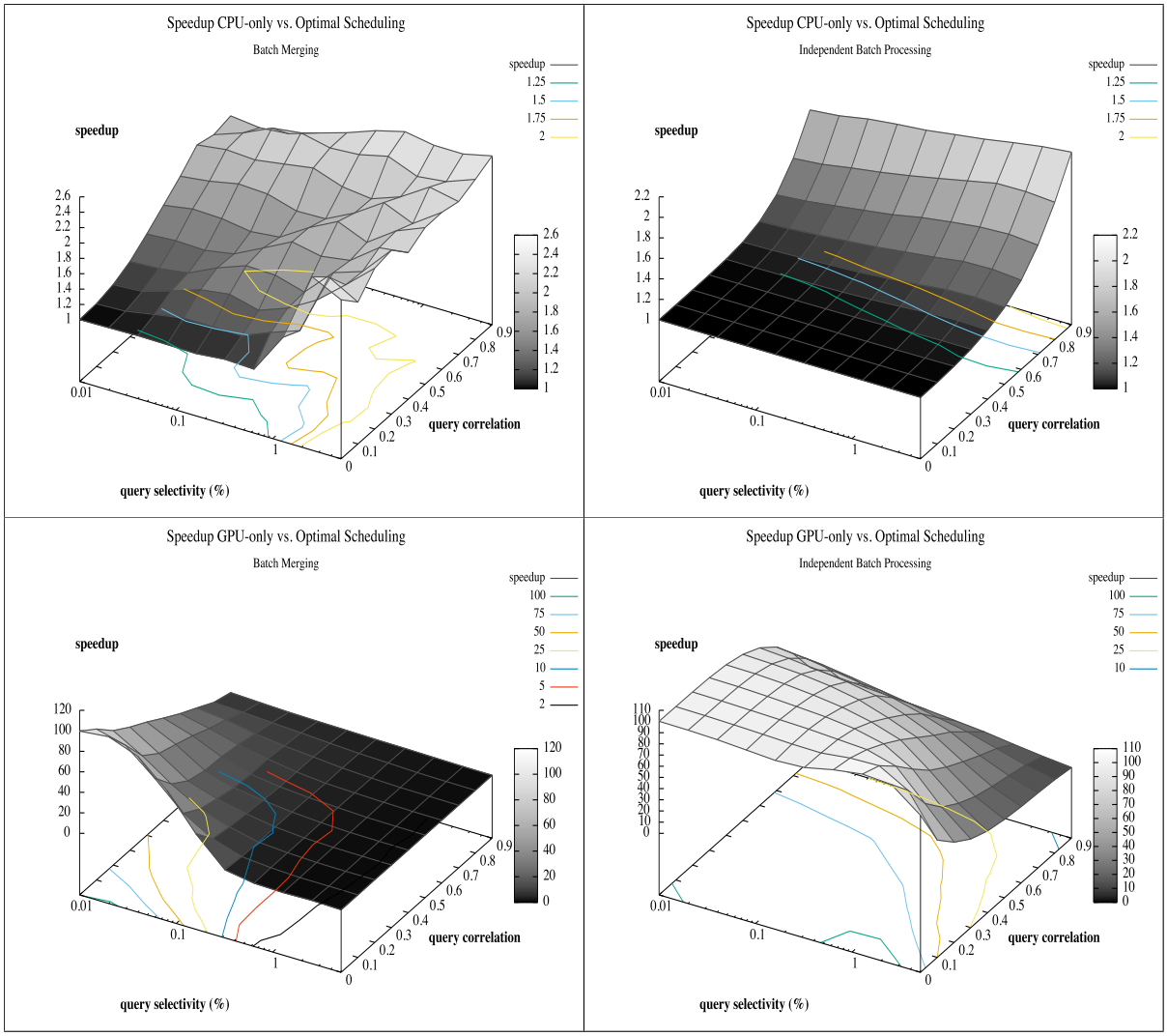


Figure 6.19: Batch Processing Strategies: Hybrid Scheduling Speedups
Batch Merging (left), Independent Batch Processing (right)
CPU-only vs. Optimal Scheduling (top),
GPU-only vs. Optimal Scheduling (bottom)

The benefit of using hybrid processing over the trivial strategies is illustrated in Figure 6.19, where speedup values are plotted. Like in the previous experiment, speedups are almost equal to one for the extremum cases, i. e., CPU-only processing is nearly optimal for low selectivities and correlations and GPU-only processing is near-optimal for larger parameter values. It can be seen that batch merging shifts the thresholds where notable speedups can be achieved through hybrid processing towards lower parameter values. Further, it also increases absolute speedup values slightly, which matches the initial expectations.

The optimal query throughputs that can be achieved via hybrid processing are plotted in Figure 6.20 for both batch handling strategies. An increasing selectivity negatively impacts the throughputs, because much more tree nodes need to be processed during the traversal. An increasing correlation increases query throughputs for the independent case, where batch sizes decrease less if queries follow the same paths. If batches are merged, this effect is reduced,

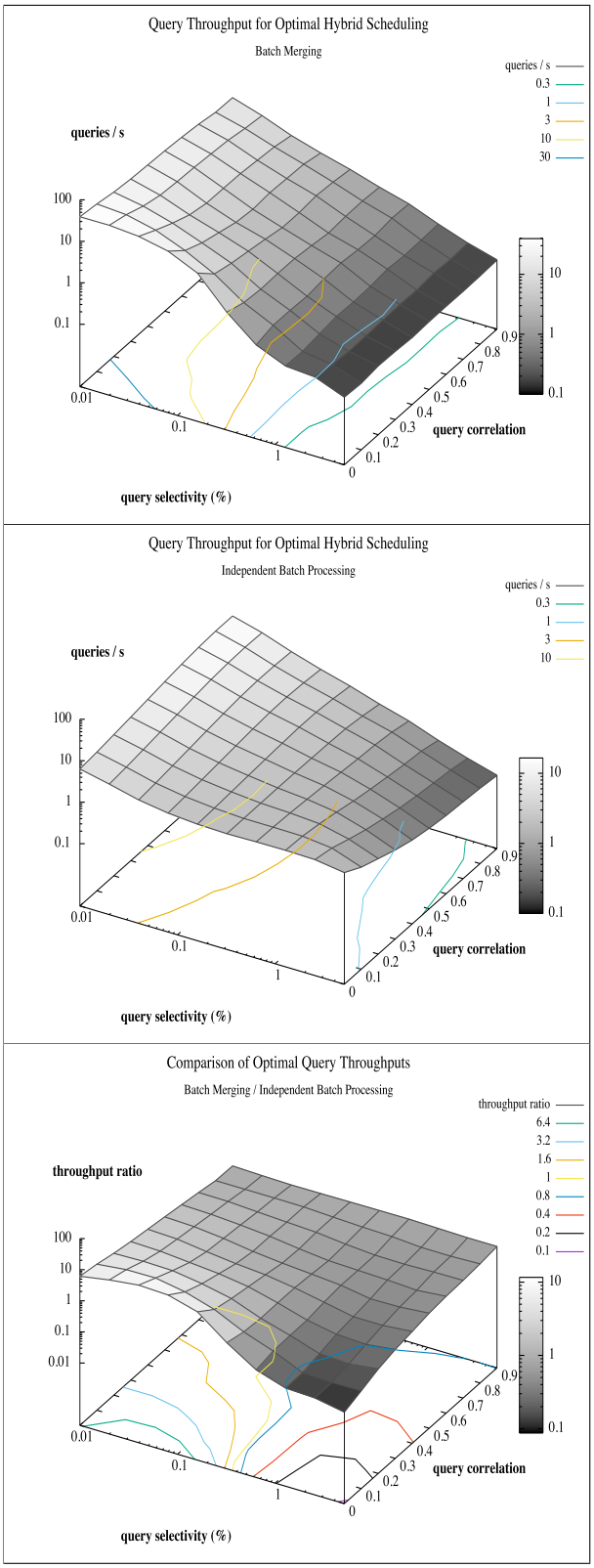


Figure 6.20: Batch Processing Strategies: Query Throughputs
Batch Merging (top), Independent Batch Processing (middle)
Throughput Ratio (bottom)

which means that merging intermediate results before the next iteration can compensate lower correlations and also yields sufficiently large batches that can be processed efficiently. Directly comparing both framework configurations by their throughput ratio (bottom of Figure 6.20) shows that applying batch merging can lead to speedups of up to 10x over independent processing, which matches the expectations. However, contrary to the expectations, this is not true for all parameter configurations. For larger selectivities, this ratio flips and independent processing becomes up to 10x faster. The reason for this is that the average batch size approaches 64 in this case, which is optimal for CPU processing, where, due to caching effects, absolute query throughputs exceed those that can be achieved by the GPU for the maximum batch size.

The experiment has shown that maximizing batches during the traversal via synchronization points for merging intermediate results can lead to significant performance improvements for hybrid scheduling approaches. But the experiment also revealed some workload-specific conditions, where the exact opposite was the case. These effects where caused by non-linear behavior of absolute query runtimes, which are analyzed in more detail in Section 6.3.4.

6.3.3.4 Runtime Impact of Tree Parameters

The objective of the fourth experiments is to evaluate how tree parameters impact the batch size distribution and, thereby, the hybrid scheduling speedups. Therefore, the simulations have been configured with the following settings:

Tree Parameters:

- *Slots per Node: 96 / 2048*
- *Tree Height: 5 / 3*
- *Simulated Number of Entries:*
  96^5 = 8,153,726,976 /
  2048^3 = 8,589,934,592

Hardware Platforms:

- 2015 CPU (Intel Core i7-6700)
- 2015 GPU (Nvidia GeForce GTX TITAN)

Query Processing Parameters:

- Predicate: Range Query for 3D R-Tree
- Root Queries: 10,000
- Max Batch Size: 512 (GPU-optimized)
- Batch Processing Strategies:
  Merging (maximizing batch size)

Scheduling Strategies:

- CPU-only
- GPU-only
- Optimal Hybrid

In this setup, the number of slots per node is varied. In addition to the small node having 96 slots, a large node with 2096 slots is analyzed. In order to gain comparable results, in particular for the selectivity parameter, the number of entries in the tree should not be changed significantly. Therefore, the tree height had to be reduced from 5 to 3 layers.

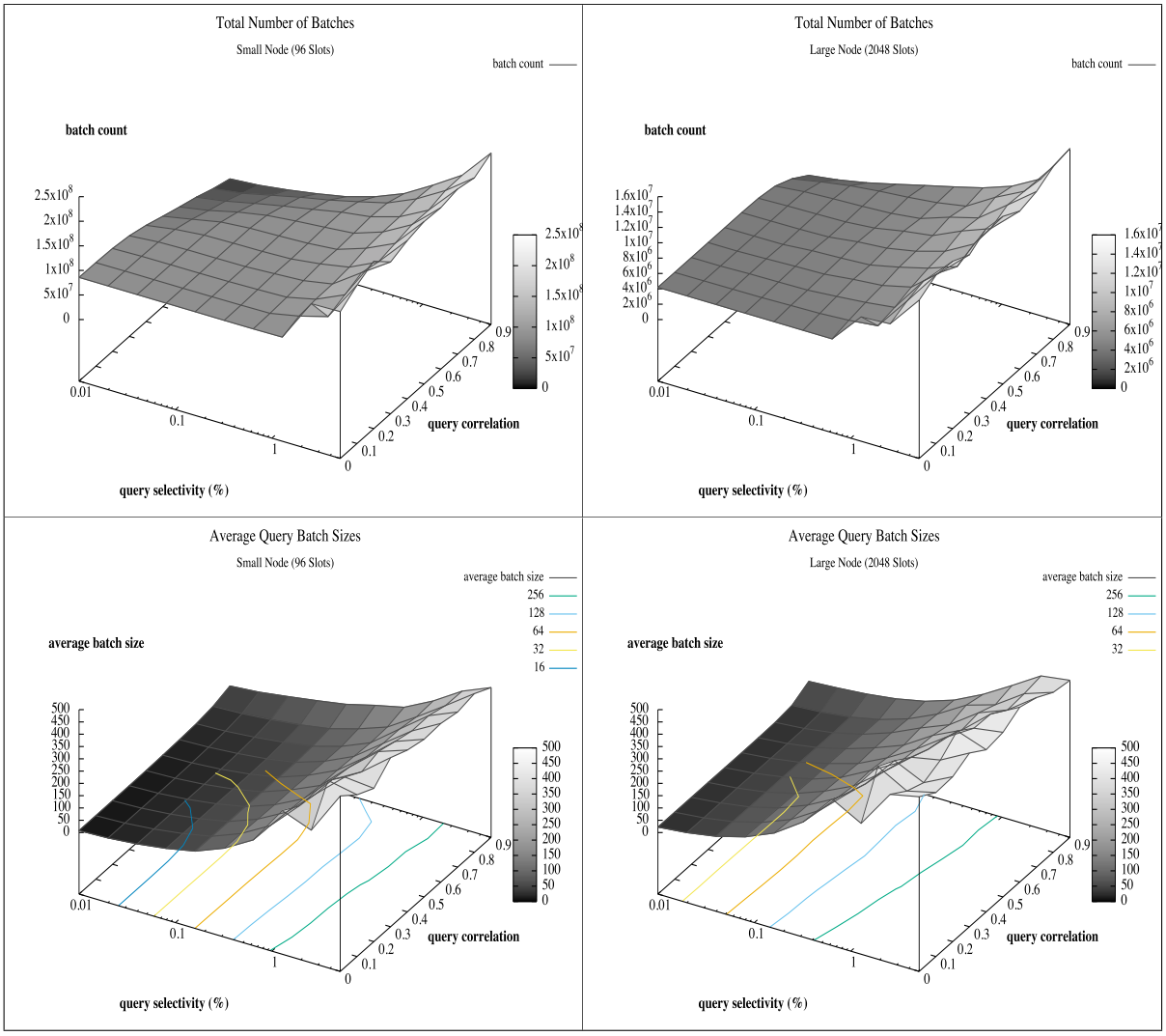


Figure 6.21: Tree Parameters: Total Batch Distribution Metrics
Small Node (left), Large Node (right)
Total Number of Batches (top), Average Batch Size (bottom)

Because the tree with larger nodes has fewer layers that need to be traversed, it is expected that this reduces the number of batches during the traversal. But their distribution over the simulated search space should not change significantly, because the workload parameters did not change. If the same selectivity is used for both configurations, approx. the same number of entries will be returned. Because less batches are expected for larger nodes, their average sizes should increase, shifting workload from CPU-processing towards the GPU.

Since the GPU speedup is much larger for large node sizes, it is expected that this also results in a significant increase of end-to-end speedups when the GPU is used in these cases. But this does not necessarily need to result in higher query throughputs. On the one hand, much more processing can be done on a single node, which should increase the parallelization benefits and, due to the reduced number of batches, less nodes need to be scanned. On the other hand,

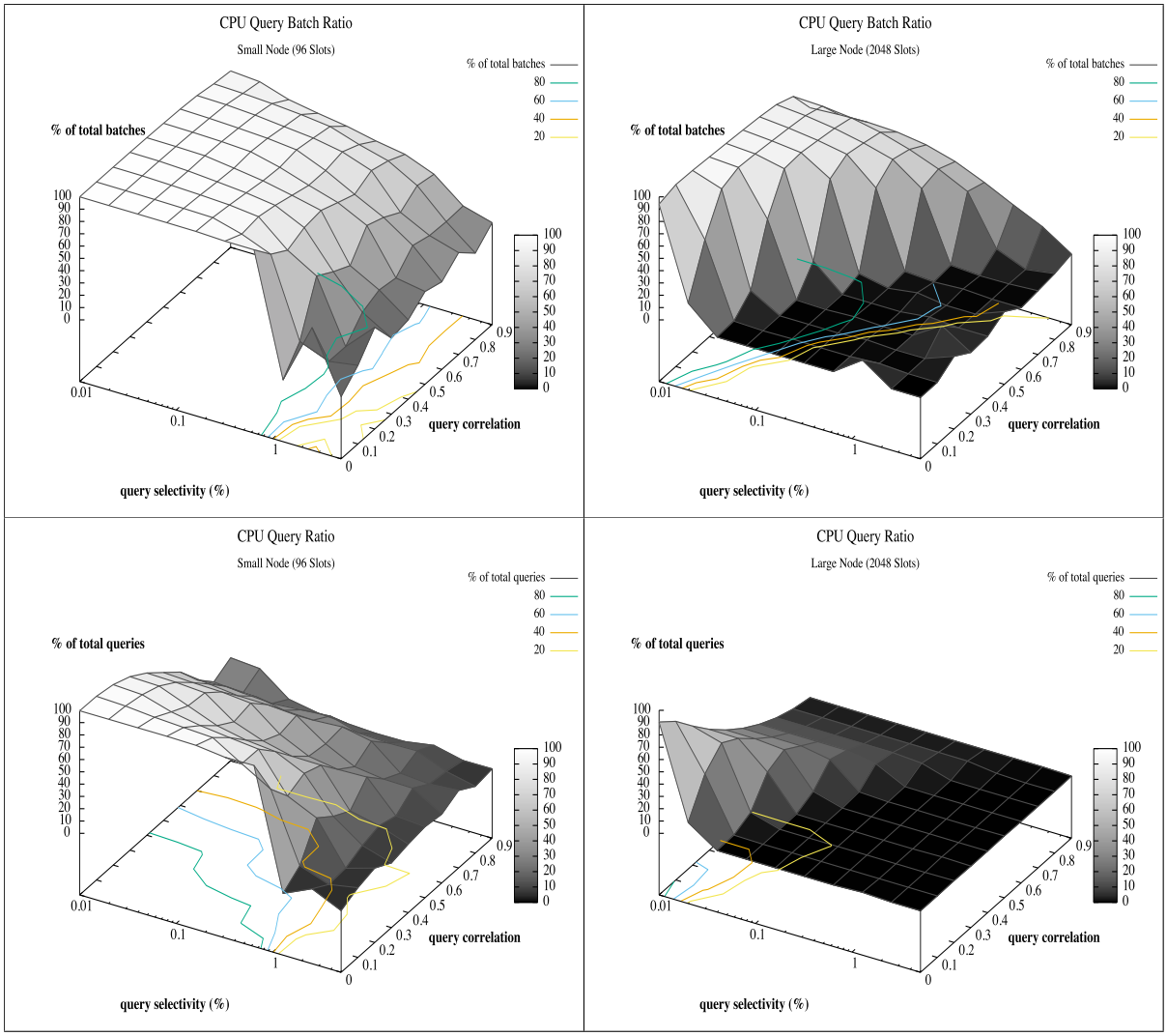


Figure 6.22: Tree Parameters: CPU Batch Distribution Metrics
Small Node (left), Large Node (right)
Batch Ratio (top), Query Ratio (bottom)

however, as the observations in the previous sections have shown, the absolute throughputs for CPU processing on small and medium-sized batches outperforms the GPU in many cases. Therefore, the GPU-optimized configuration should not be optimal for this scenario.

The batch distribution metrics, which are plotted in Figure 6.21 fully match all expectations, i. e., the batch size distribution is not impacted by the node size for comparable workload settings, but the overall number of batches is reduced when nodes with a larger fan-out are used.

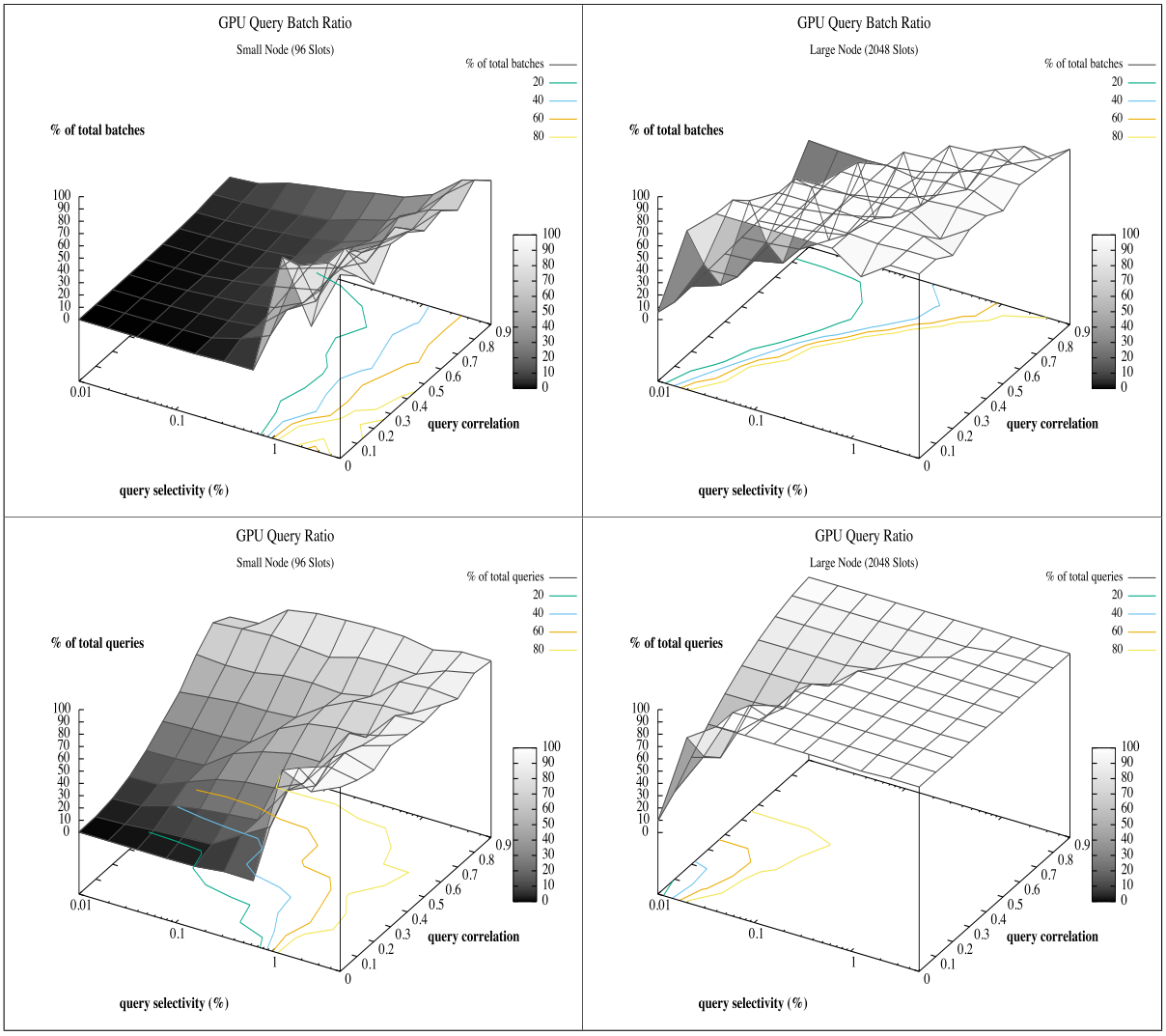


Figure 6.23: Tree Parameters: GPU Batch Distribution Metrics
Small Node (left), Large Node (right)
Batch Ratio (top), Query Ratio (bottom)

The distribution of batches to each processor type, which is illustrated in Figure 6.22 and Figure 6.23, also matches the expectations, i. e., the large-node workload is shifted towards the GPU. Because the break-even batch size reduced significantly from ≈ 256 to ≈ 32 queries for the large node setting (cf. Figure 5.15), the border between the CPU-optimal and the GPU-optimal area is much sharper, compared to the small node case.

The batch size distribution also results in much larger speedups that can be achieved with hybrid processing, compared to the CPU-only execution (cf. Figure 6.24). But because much of the parameter space is optimal for GPU processing, the GPU-only penalty is significantly lower for large tree nodes. The unexpected inverted behavior of the correlation impact must have been caused by the batch size cutoff mechanism that leaves a larger fraction of small-sized remainders for low selectivities and large correlations, which fall into the CPU-optimal range.

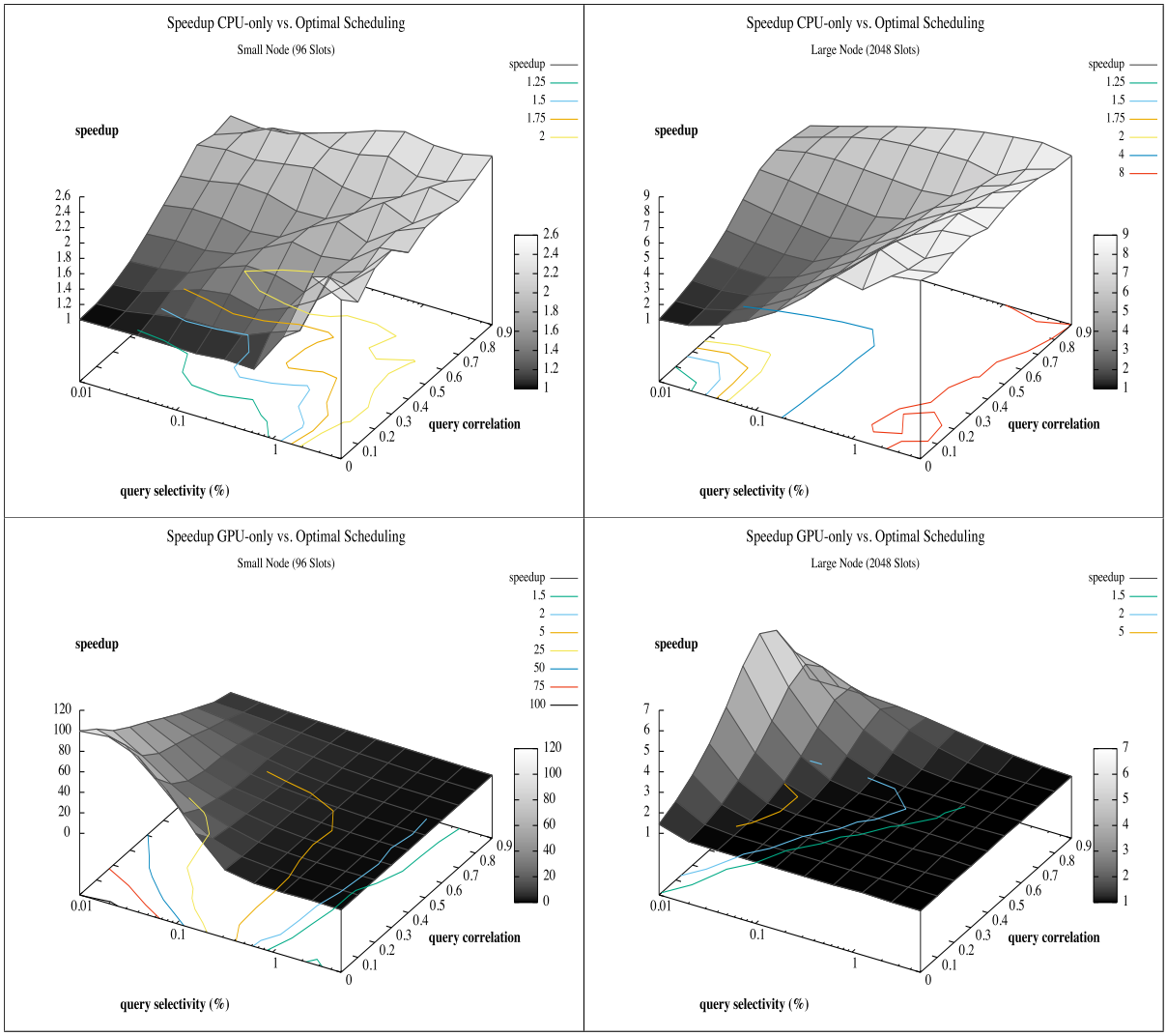


Figure 6.24: Tree Parameters: Hybrid Scheduling Speedups
Small Node (left), Large Node (right)
CPU-only vs. Optimal Scheduling (top),
GPU-only vs. Optimal Scheduling (bottom)

By comparing query throughputs with each other (cf. Figure 6.25), it can be seen that larger throughputs can be achieved when smaller nodes are used. Only for large selectivities, where the entire tree is scanned anyway, this behavior changes. This can be explained by the increased pre-filtering in upper tree layers for smaller nodes, i. e., there are fewer predicate evaluations for those child nodes that would not have been visited if additional tree layers were added on top, instead of distributing the same number of entries over fewer layers.

In summary, the node size strongly impacts the overall query performance on a tree. If merely the predicate evaluation phase needs to be considered and I/O can be ignored, because the tree can be stored in-memory, this rather static parameter should be tuned for the execution runtime. However, additional measurements are needed to evaluate this in more detail, which is not in the scope of this thesis. In particular, memory access patterns should be analyzed for both device types to tune the algorithms further and derive recommendations for the optimal node size.

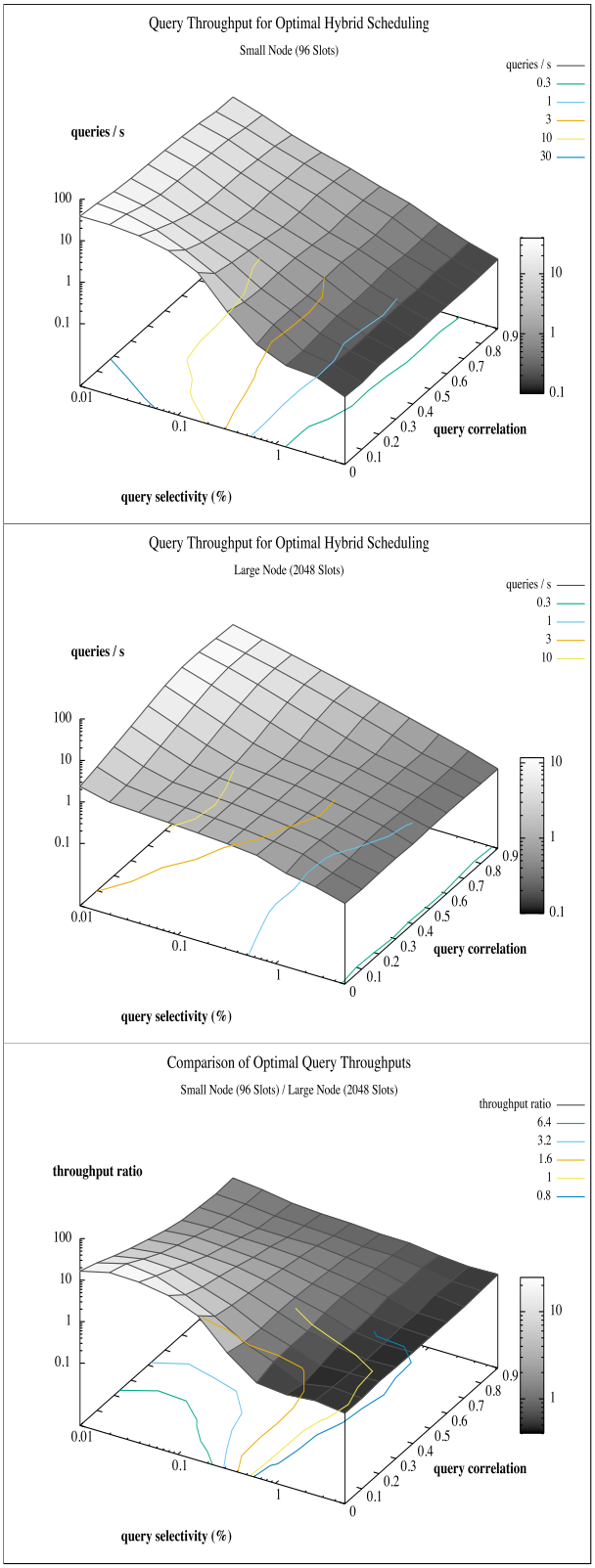


Figure 6.25: Tree Parameters: Query Throughputs
Small Node (top), Large Node (middle)
Throughput Ratio (bottom)

6.3.3.5 Runtime Impact of Query Parameters

The objective of the fifth experiments is to evaluate how the size of the initial query batch, as the parameter describing the amount of independent work units that can be processed in parallel by the framework, impacts batch size distributions and hybrid scheduling speedups. Therefore, the simulations have been configured with the following settings:

Tree Parameters:

- Slots per Node: 96
- Tree Height: 5
- Simulated Number of Entries: $96^5 = 8,153,726,976$

Hardware Platforms:

- 2015 CPU (Intel Core i7-6700)
- 2015 GPU (Nvidia GeForce GTX TITAN)

Query Processing Parameters:

- Predicate: Range Query for 3D R-Tree
- *Root Queries: 1,000 / 10,000 / 100,000*
- Max Batch Size: 512 (GPU-optimized)
- Batch Processing Strategies: Merging (maximizing batch size)

Scheduling Strategies:

- CPU-only
- GPU-only
- Optimal Hybrid

In this setup, merely the initial batch size is varied. The other parameters remain unchanged. In particular, the batch merging strategy is applied. Otherwise, scaling the initial batch size does not lead to any major changes of batch size distributions during the traversal. As discussed in Section 6.3.3.3, independent processing would partition the initial batch in the root layer, capping it at the maximum size that is supported by the hardware, and intermediate batch sizes would be strictly decreasing in subsequent layers.

Since batch merging is applied, it is expected that the average batch size increases during the traversal if the initial batch size is increased, because the framework can group more intermediate results for further processing. Of course, the total number of batches should also increase significantly as more queries are streamed through the tree. Increasing the amount of root queries is expected to favor GPU over CPU execution. Further, the overall throughputs shall also increase, because the framework can leverage more parallelism. However, caching effects might impact absolute throughputs again, leading to unexpected behavior for this metric.

Analyzing the batch metrics in Figure 6.26 for all initial batch sizes shows that the assumptions are fully met. The total batch counts increase almost linearly with the increasing initial batch size and increasing the initial workload also increases the size of intermediate batches that are generated after each predicate evaluation step. For the largest root batch of 100,000 queries, one can see that the average batch size reaches the maximum possible value for medium selectivities. That is, due to batch merging, almost all intermediate batches are of maximum size, which favors GPU execution.

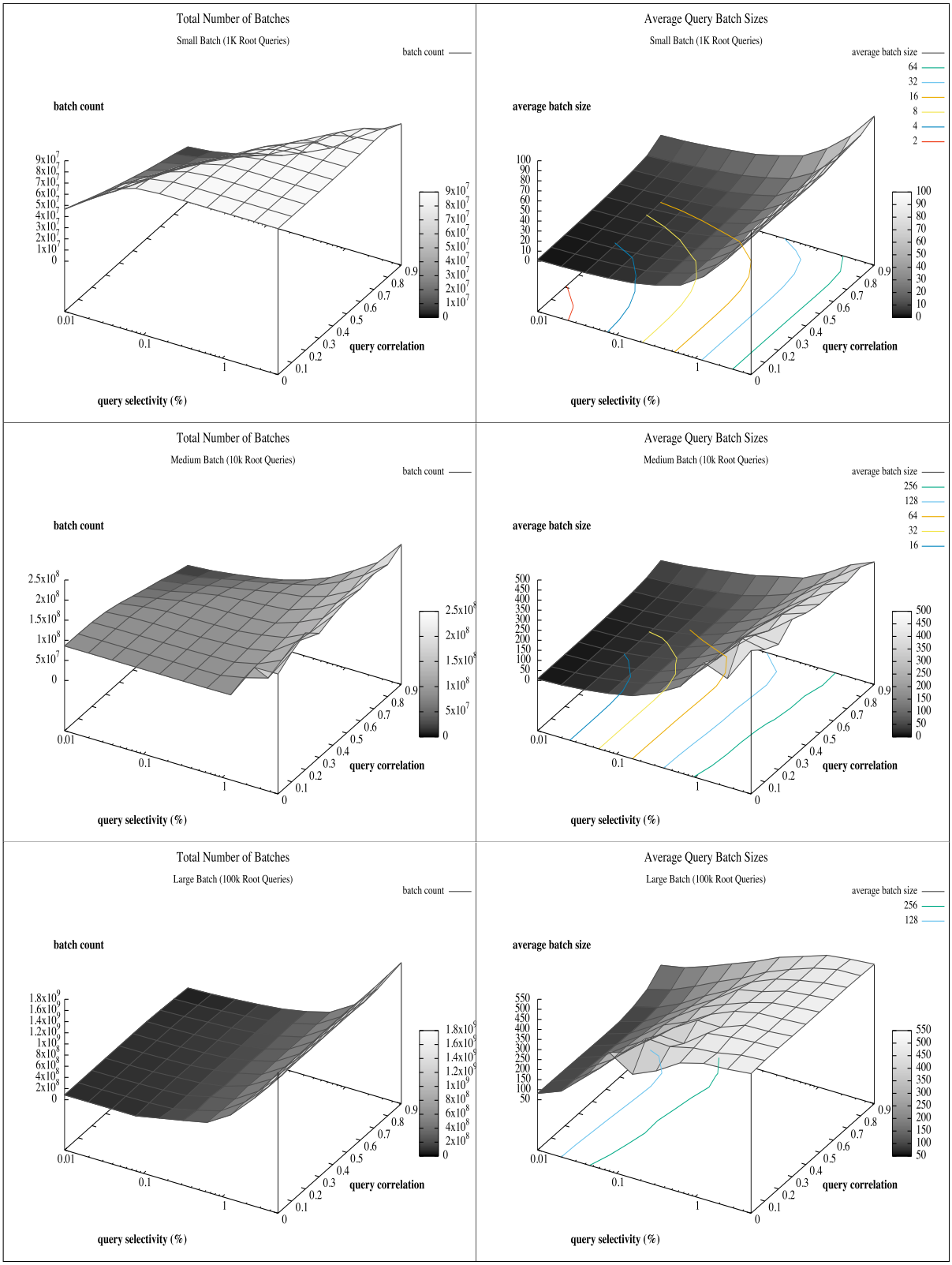


Figure 6.26: Query Parameters: Total Batch Distribution Metrics
Small Batch (top), Medium Batch (middle), Large Batch (bottom)
Total Number of Batches (left), Average Batch Size (right)

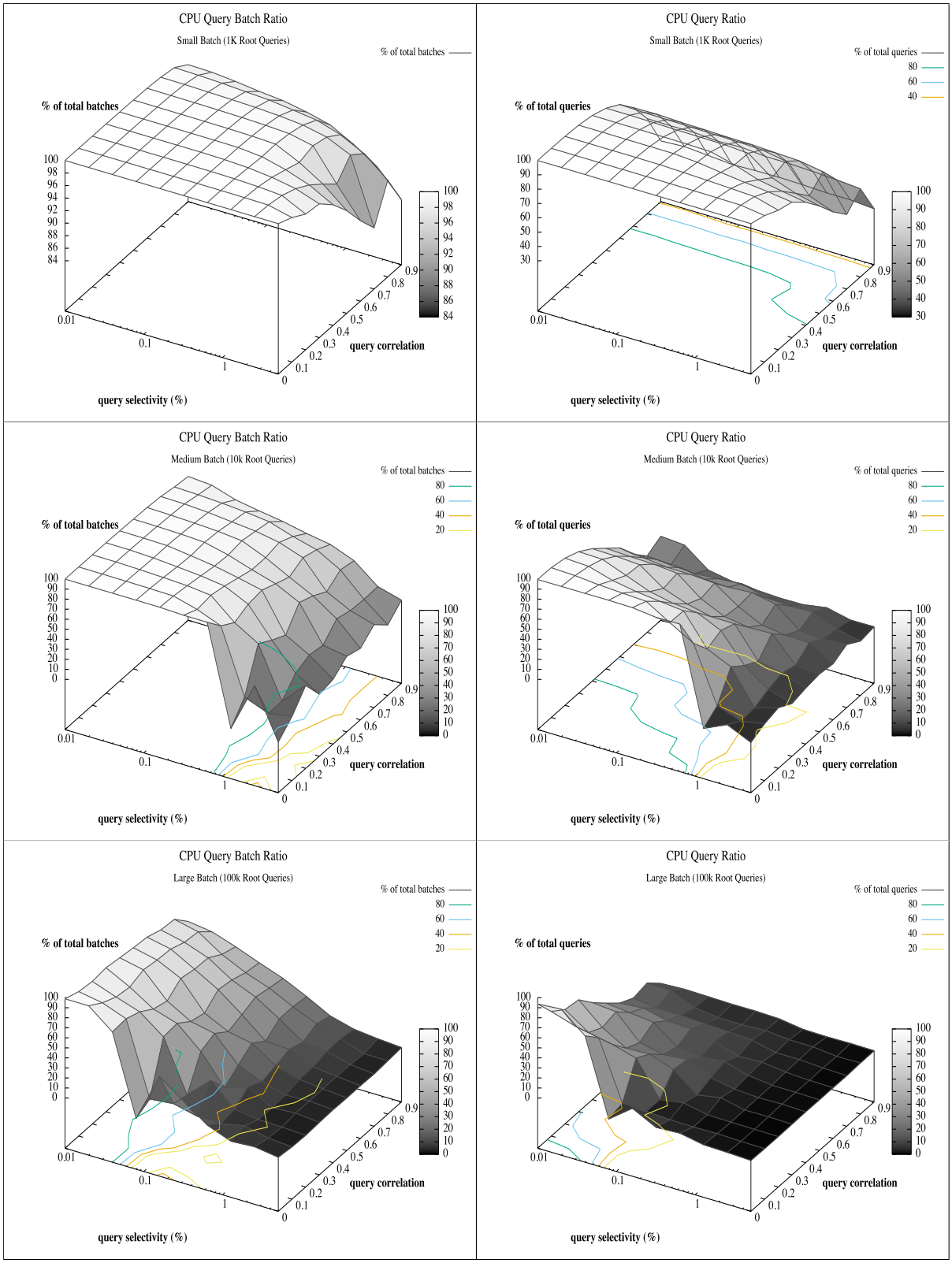


Figure 6.27: Query Parameters: CPU Batch Distribution Metrics
Small Batch (top), Medium Batch (middle), Large Batch (bottom)
Batch Ratio (left), Query Ratio (right)

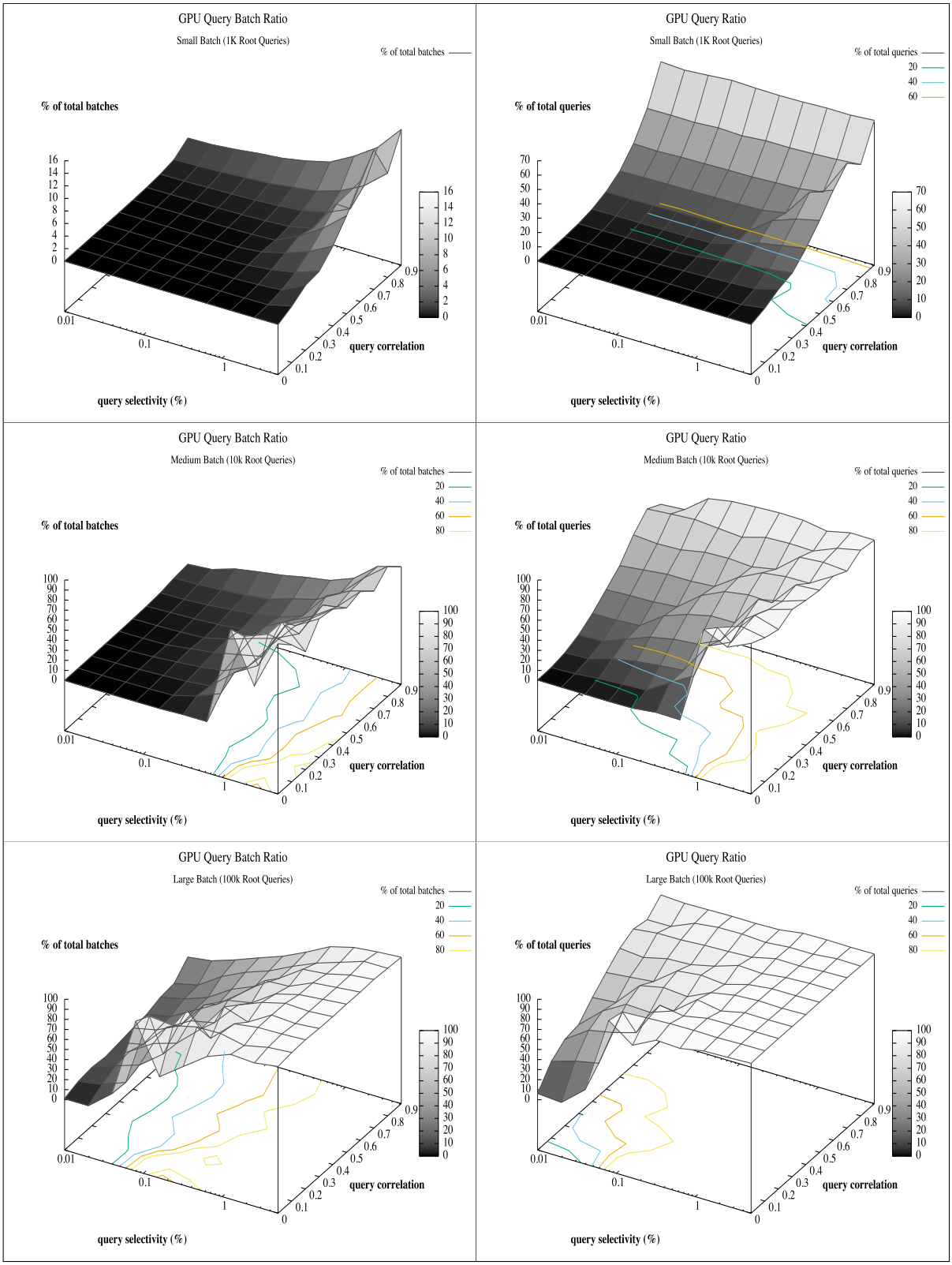


Figure 6.28: Query Parameters: GPU Batch Distribution Metrics
Small Batch (top), Medium Batch (middle), Large Batch (bottom)
Batch Ratio (left), Query Ratio (right)

The workload shift towards the GPU can be clearly observed in Figure 6.27 and Figure 6.28, which also matches the expected behavior. The overall shape of the curves is not impacted by the initial batch size. Increasing the latter merely shifts each plane to the left, i. e., GPU execution becomes beneficial for lower selectivity and correlation values. Therefore, the speedup discussion for hybrid execution vs. processing on a single processor type has been omitted.

The query throughputs are compared in Figure 6.29. The medium batch size of 10,000 initial queries has been used as common baseline and is illustrated at the top. Comparing it to the small batch size on the left hand side shows that processing the medium batch outperforms the smaller one for small selectivities. For larger selectivities, this, again, changes due to caching effects, because the smaller intermediate batches are better-suited for CPU execution, which outperforms the GPU in terms of absolute query throughputs in this scenario (cf. Figure 5.15). Comparing the medium to the large initial batch shows that the throughput values are almost equal. That is, the medium-sized one already saturates the processors by generating sufficiently large batches for most selectivity and correlation settings. Unexpected peaks in the throughput ratio are, most likely, caused by CPU caching effects, too.

In summary, a sufficiently large number of initial queries is required by the framework to leverage its batch processing capabilities. This can only be achieved if the application is generating enough queries that can be processed independently without synchronizing on their results. But there is an upper bound for this, where adding additional independent load does not lead to further improvements. This information can, for example, be used to optimize the framework internally, e. g., for implementing a buffering strategy that gathers a sufficiently large number of queries before scheduling the whole batch to the available processing units.

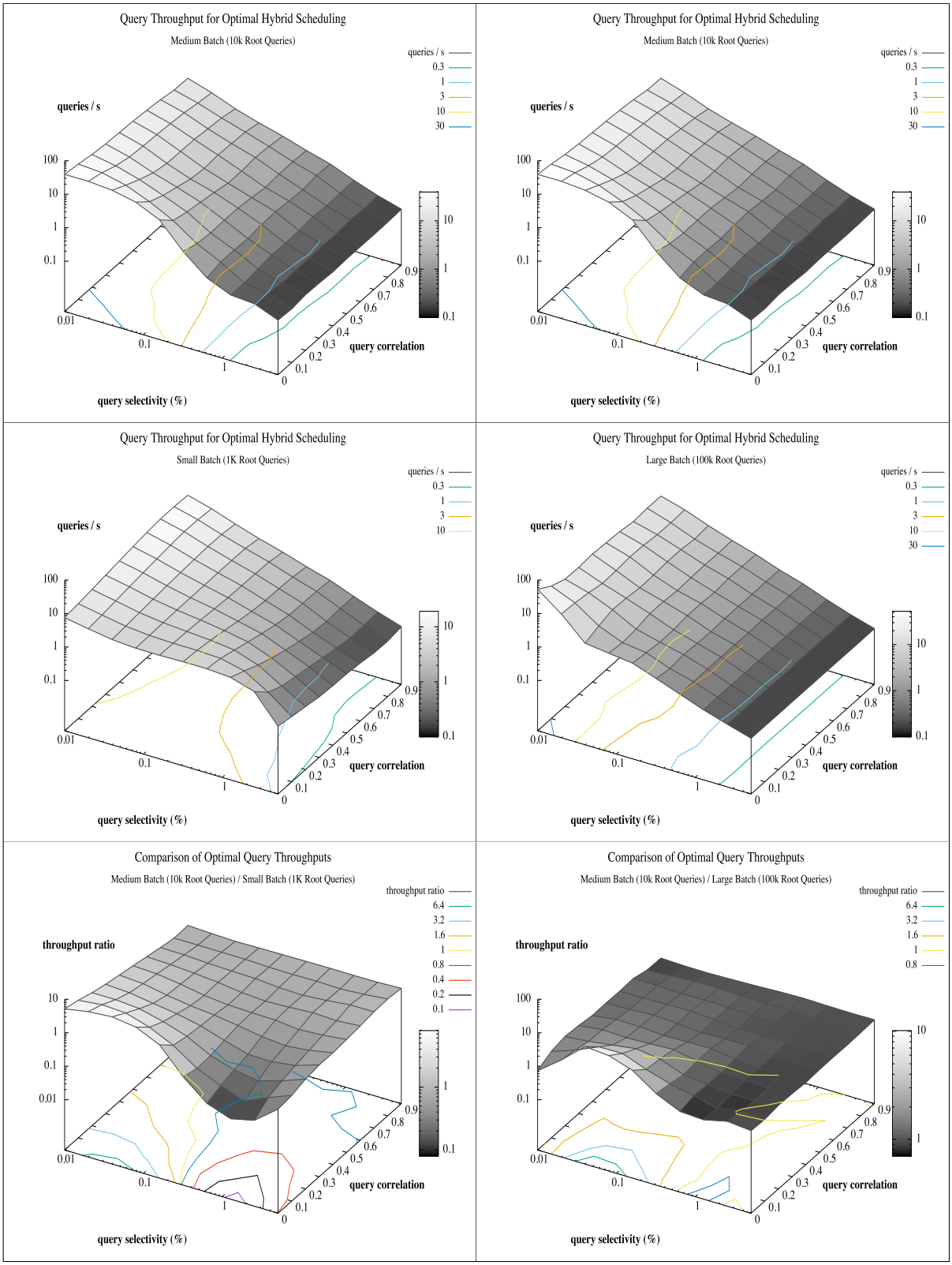


Figure 6.29: Query Parameters: Query Throughputs
Medium Batch (top), Small Batch (middle, left), Large Batch (middle, right)
Throughput Ratio (bottom)

6.3.4 CPU-Optimization & Comparison with GiST

The objective of the final experiment is to evaluate peak end-to-end throughputs that can be achieved when queries flow through the tree and the predicate evaluation for each node has been tuned for each processor types. The single-node throughput analysis of Section 5.3.3 has shown that there are significant performance gaps between CPUs and GPUs for different framework parameters. A CPU-optimized configuration is able to outperform the GPU-optimized version for the analyzed range predicate. Therefore, both versions should be compared here to assess how the single-node evaluation performance impacts entire tree traversals. Further, the strategies that can be implemented via the proposed index framework are compared to GiST to quantify performance benefits that can be achieved over this common base line.

Tree Parameters:

- Slots per Node: 96
- Tree Height: 5
- Simulated Number of Entries: $96^5 = 8,153,726,976$

Hardware Platforms:

- 2015 CPU (Intel Core i7-6700)
- 2015 GPU (Nvidia GeForce GTX TITAN)

Query Processing Parameters:

- Predicate: Range Query for 3D R-Tree
- Root Queries: 10,000
- *Max Batch Sizes:*
  1 (GiST simulation) /
  64 (CPU-optimized) /
  512 (GPU-optimized)
- Batch Processing Strategies:
  Merging (maximizing batch size)

Scheduling Strategies:

- CPU-only
- Optimal Hybrid

In order to tune the framework for a specific execution unit, the maximum batch size needs to be varied. Maximizing it optimizes the framework for GPU and hybrid processing, because more load is generated that is better-suited for GPU processing, whose performance should, theoretically, be higher than the CPU's. The medium batch size that is optimal for cache-local CPU processing has been determined via the peak throughput that has been measured in Section 5.3.3. It also depends on other parameters, like the node size and the CPU model, which have been fixed in this scenario. In order to simulate how GiST would process the queries in the initial batch, a batch size of one has been used, which simulates iterative query-at-a-time processing for each node. Compared to the iterative batch-wise execution on a CPU, the difference is that GiST traverses the full tree for a single query before continuing with the next one. Node data might be cached in a buffer pool to save I/O time, but will be evicted from lower-level CPU caches once a node has been scanned, because the next node(s) on the path

towards the leaf layer will overwrite it. In the CPU-optimized version, the query loop is pushed down one in order to evaluate all queries on cached node data before the next one is fetched. The node scan benchmark metrics that will be joined with batch counts have been collected, assuming in-memory node storage but cold CPU caches. Therefore, the GiST simulation via minimal batch size is a valid and comparable approach.

For optimized CPU and GPU processing, the framework maximizes batch sizes during the traversal (up to the platform-specific maximum size) by applying the merging strategy after each iteration. The GPU-only scheduling strategy has been omitted in the comparisons, because none of the batches exceeds the break even point in the CPU-optimized setting and, therefore, it will always be slower. Instead, the optimal hybrid strategy is compared, which always picks the best execution unit for a particular batch, but is not aware of other batch sizes that may lead to even higher throughputs on another processor.

Reducing the maximum batch size for CPU-optimized execution increases the total number of batches that have to be processed, compared to the GPU-optimized setting. But since CPU query throughputs are higher for most of the smaller-sized ones, it is expected that the overall query throughput is also higher for the end-to-end traversal. Although the optimal hybrid scheduler correctly selects the CPU for smaller-sized batches, their total number and, therefore, their contribution to the overall runtime is expected to be much lower, because the cutoff prefers larger batch sizes, which impacts the batch size distribution. Compared to simulated GiST processing, both vectorized framework setups, the CPU-optimized as well as the optimal hybrid version, should lead to significantly increased throughputs.

The batch size distributions for small and large cutoff thresholds are plotted in Figure 6.30. As expected, the total number of batches increases by almost an order of magnitude for the lower threshold. The average batch size also decreases and approaches the reduced maximum already for medium selectivities, because the initial batch size is large enough so that most of the nodes can be processed with maximum performance. The total number of queries that have to be processed, which is plotted at the bottom of Figure 6.30, is not affected by the internal batch partitioning at all. They represent logical evaluations that are independent from the fact whether vectorized processing is applied or not.

Absolute end-to-end query throughputs for each scenario are illustrated at the left hand side of Figure 6.31. As expected, the best throughputs are achieved for the CPU-optimized version. For this case, an increasing correlation also leads to better results, because more batches reach the optimal maximum size. But the increasing number of max-sized batches caused by an increasing selectivity cannot be compensated via higher throughputs and the overall performance drops, like in the other scenarios. Directly comparing all configurations with each other by their throughput ratios, which are illustrated on the right hand side of Figure 6.31, shows that both vectorized processing approaches outperform the simulated GiST execution for all query workloads. The hybrid strategy is up to 12 times faster than GiST and using the optimized CPU version may even lead to speedups of 40x and more. Comparing hybrid and CPU-optimized processing with each other shows that the significant performance gap caused by CPU caching effects for a single node also leads to better overall throughputs for CPU-optimized processing. Speedups of 3–12x can be reached for most of the simulated workload settings.

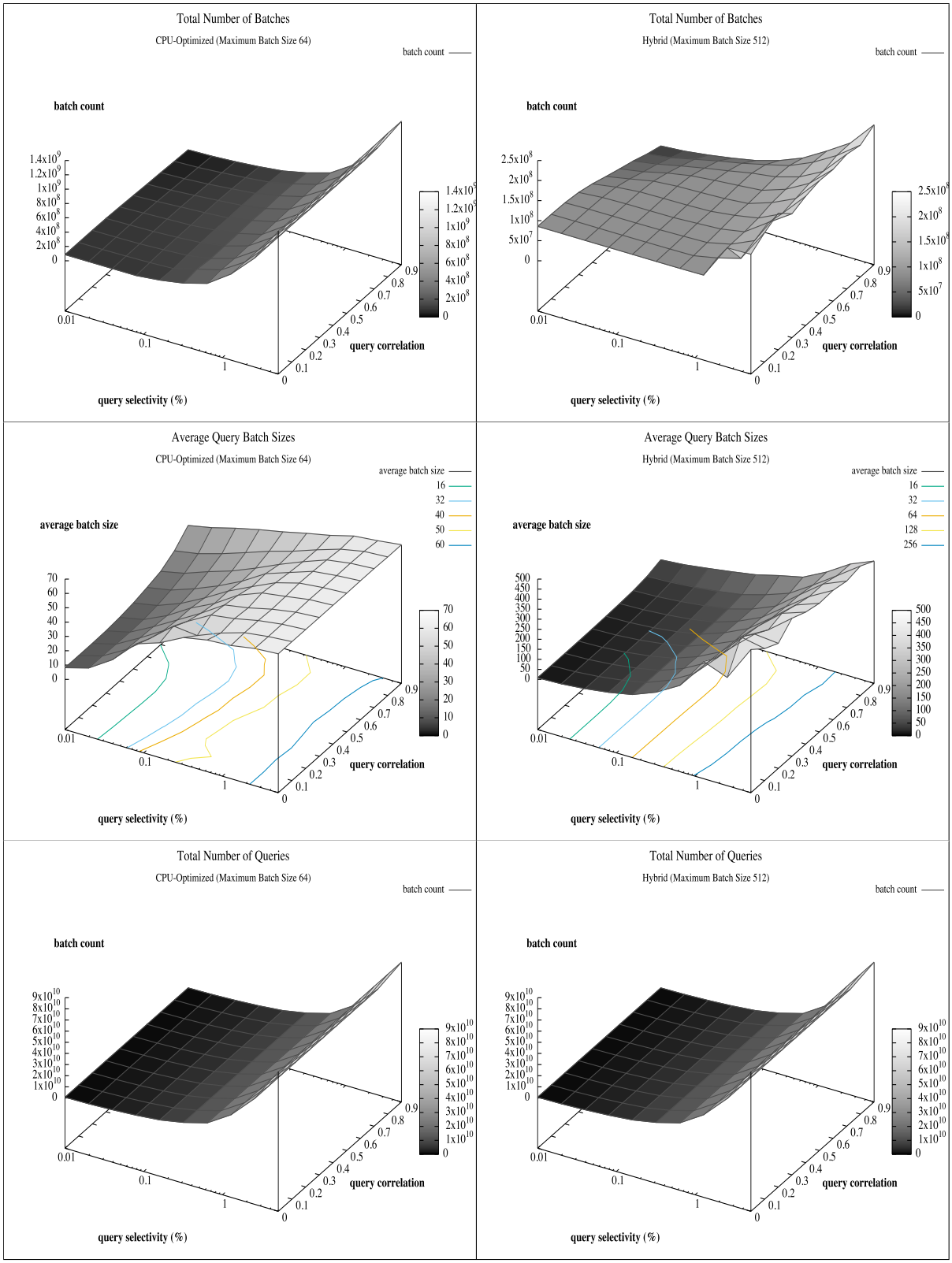


Figure 6.30: Maximum Batch Size: Query Batch Distribution
CPU-Optimized (left), Hybrid-Optimized (right) Total Number of Batches (top), Average Batch Size (middle), Total Number of Queries (bottom)

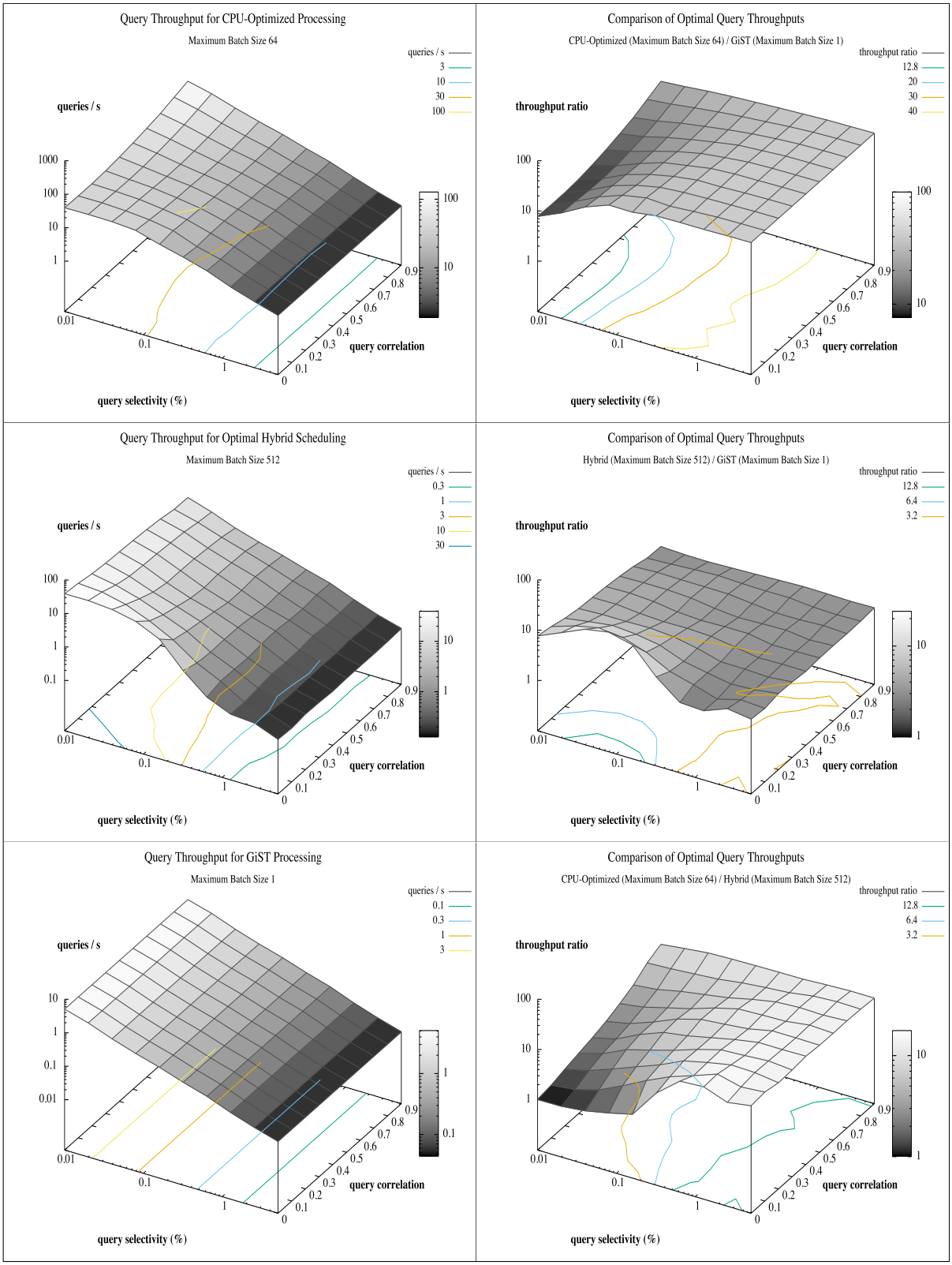


Figure 6.31: Maximum Batch Size: Query Throughputs
Left: CPU-Optimized (top), Hybrid (middle), GiST (bottom)
Right: CPU-Optimized/GiST Ratio (top), Hybrid/GiST Ratio (middle), CPU-Optimized/Hybrid Ratio (bottom)

The evaluation has shown that vectorized processing used by the extended GiST framework can lead to significant performance gains, compared to the query-at-a-time approach followed by the original GiST implementation. The framework can be optimized for CPU-execution as well as for hybrid processing on CPUs and GPUs by adapting a single parameter. For the analyzed scenario, the CPU-optimized version yields the best results as caching effects for smaller-sized query batches led to an overall better performance than a configuration that tries to offload as much load as possible to a GPU coprocessor. Nevertheless, this behavior might be different for other scenarios. For example, when other predicate implementations are used that are computationally intensive, GPU and hybrid processing will likely excel in such cases.

6.3.5 Discussion of Hybrid Scheduling Evaluation Results

The evaluation in this chapter demonstrated that the learned regression-based model for estimating operator execution times is accurate. The query traversal simulation has shown that estimation errors just marginally impacted the overall execution time, which differed less than 5% from the optimal case. Further, the scheduler could be successfully adapted to an unknown hardware environment after an initial calibration. In summary, it could be shown that using the adaptive scheduler in the proposed index framework extension is a reasonable approach.

The performance of hybrid query processing was impacted by many parameters, static configurations, like node and maximum batch size, as well as dynamic workload-specific ones, like batch sizes. It could be shown that for static configurations, where workload-specific conditions exist where one processor outperforms the other, hybrid scheduling outperformed processing on a single device type. Therefore, in general, adding hybrid system support is a valuable extension for the index framework.

However, for the analyzed use case, there was a static framework configuration, where the CPU always outperformed the GPU. Thus, hybrid processing on both devices was not beneficial under the assumptions that were made, i. e., sequential vectorized processing of (node, query batch)-pairs that use a simple predicate evaluation logic. Nevertheless, the framework could be easily adjusted to this situation by changing a single configuration parameter in order to optimize it for a cache-conscious CPU-based execution. Compared to the original GiST implementation, this led to a throughput improvement of 40x and more in the simulated workload. Even the suboptimal hybrid framework configuration outperformed GiST in this scenario by up to an order of magnitude. In summary, it could be shown that vectorizing query execution in GiST significantly boosts query throughputs, compared to the original query-at-a-time model.

In future work, the framework should be adapted, so that it also considers the maximum batch size during the calibration phase and schedules multiple batches at once for achieving optimal performance on a given platform. In addition, the tree traversal simulation may be improved, e. g., by adding support for simulating different data distributions. In order to evaluate the framework for real-world scenarios, a valuable extension would be to integrate the framework with QuEval [126, 178].

6.4 Summary

In this chapter, it has been shown how support for multiple processor types can be added to the generalized indexing framework in order to leverage benefits of a hybrid system. Therefore, a scheduler has been integrated into the framework's generalized execution layer, which automatically learns operator execution cost models via regression during an initial one-time calibration phase. Based on these models, the scheduler is able to estimate total execution costs for a tree traversal operation and select the best-suiting device for processing it. Details about the scheduler framework can be found in corresponding publications [36, 37].

For evaluating the scheduler, an index query simulator has been provided and used to evaluate scheduling strategies for different workload classes and processor hardware generations. In the experimental evaluation, the regression-based scheduler proved be to good for the analyzed workload scenario. Only minor decision errors were made and, because they occurred near break-even points, these errors resulted in less than 5 % performance loss, compared to the theoretical optimum. That is, by using the scheduler, the framework is able to adapt itself to the available runtime environment.

Based on these results, additional experiments have proven the benefits of hybrid scheduling over single-processor execution. Significant speedups could be observed for end-to-end tree traversals, because the filtering step that is applied for each query in each tree node, leads to mixed workload distributions that are optimal for different device types. Compared to the GiST-like baseline, hybrid processing was always faster and led to 3–12x speedups for the analyzed workload configurations.

By changing the scheduler configuration, the framework could also be optimized for CPU-based execution that implicitly utilizes lower-level CPU caches. As experiments in the previous chapter have shown, such a configuration was even faster than any of the GPU-based executions for the analyzed workload. Experiments with the CPU-optimized version have shown that it is able to outperform hybrid processing on the same workload by additional factors of 3–12x, i. e., up to 40x in total, compared to the GiST baseline. This demonstrated that the framework is flexible and can be easily adjusted to maximize throughputs of generalized index queries on different platforms.

Chapter 7

Conclusion and Outlook

7.1 Conclusion

In this thesis, it has been demonstrated how algorithms operating on generic tree-based index structures can be adapted to the architectural characteristics of modern (co-)processor devices. Based on GiST, a state-of-the-art framework for modeling arbitrary height-balanced search trees, a new framework extension has been developed that allows to schedule query operations to arbitrary processors for (parallel) execution.

Therefore, an additional execution layer has been integrated into GiST that is located below its existing abstraction layers for modeling arbitrary indexes structures on a logical level. Within the new execution layer, tree traversal algorithms were broken down to generic vectorized processing primitives that can be specialized for arbitrary device types. A sample implementation for CPUs has been provided as well as a new out-of-core GPU algorithm for generalized stateless and stateful search queries, which, unlike prior art, does not require that an index is fully stored within a GPU's main memory. Further, the processing layer has been extended by a new adaptive scheduler that allows to leverage the strengths of different device types in a hybrid system. For mixed workload classes, i. e., for runtime-specific parameters where one device clearly outperforms the others, the scheduler is able to select the best execution unit where total execution times, including overheads for intermediate data transfers, are expected to be minimal. By using a regression-based training phase to build models for estimating execution times of each primitive on each available device type, the scheduler is able to adapt itself to arbitrary hardware configurations, without modifying the framework code.

The experimental evaluation of the framework extension has shown that by using the vectorized execution layer, query throughputs can be increased by an order-of-magnitude and more, compared to the original query-at-a-time processing model implemented by GiST. It could be shown that for large workloads, using the GPU clearly outperforms the CPU, while the latter was much faster for smaller loads. It could also be shown that the hybrid scheduler is able to adapt itself to these performance characteristics. After an initial calibration on different platforms, the scheduler made near-optimal decisions for runtime-specific workload parameters.

However, the analysis has also shown that a CPU-optimized implementation is able to outperform the best hybrid variant by an order of magnitude if the framework is statically configured for cache-local CPU processing. Thus, using the GPU as coprocessor was not beneficial for the analyzed setup, because examined queries could not fully utilize all GPU resources to compensate involved overheads. Nevertheless, this picture might change for other query workloads and it could be demonstrated that the extended framework design itself is flexible enough to tune GiST for a specific hardware environment by simply adapting some configuration settings.

7.2 Outlook

The GiST framework extension that has been developed in the course of this thesis represents a solid foundation for future enhancements to broaden its applicability to a wider range of indexing use cases and execution platforms. In addition to internal improvements, like tuning the algorithms and refining scheduling decision models, the following tasks should be addressed:

- **Update support**: So far, just read-only query workloads have been considered for the parallelization. A valuable extension, therefore, is to parallelize index updates and integrate these algorithms under the same execution model in order to leverage benefits of hybrid scheduling. GiST's internal clustering algorithms are a good place to start in this direction, because they need to solve an optimization problem of picking the best search space split, when node entries shall be distributed over multiple tree nodes. The main challenge here will be to properly synchronize shadow copies of node data that are created internally when query execution primitives are offloaded to a coprocessor device. Further, the integration with GiST's transaction manager needs to be examined in more detail in order to avoid any side effects that may cause transactional inconsistencies.
- **Support for additional processor types**: Currently, the prototypical framework implementation just supports CPUs and dGPUs. Extending this to other processors that are available on the market, such as APUs or MIC devices, would be a valuable contribution, because they will likely show completely different behaviors for intermediate data transfers, for example. Porting the framework implementation from CUDA to OpenCL will be helpful to accomplish this task as OpenCL is supported by a wider range of devices.
- **Integration with an index evaluation framework**: Although, many aspects of the framework extension have already been evaluated, these experiments just represent a first step towards fully understanding all facets of the whole index modeling area. In order to gain a holistic view and provide guidance to the fundamental question which index should be used under which circumstances, it would be beneficial to integrate GiST with an evaluation framework, e. g., QuEval [126, 178], that allows to plug in different index structures in order to assess their performance for various workloads and algorithmic configurations.
- **Evaluation of real-world applications**: Evaluating the enhanced framework with real-world applications will provide valuable insights whether the performance improvements that have been evaluated under lab conditions can also be observed for real workloads. First experiments have already been conducted with image rendering engines. However, the mandatory rewrite for porting the applications' data- and control flows to a batch processing model could not be achieved within the scope of this thesis.

Appendix A

Glossary of CUDA and OpenCL Terminology

CUDA and OpenCL share a similar programming model but specify slightly different terminology for related concepts. This section summarizes the most important terms which are used throughout this thesis. Their mappings to corresponding CUDA and OpenCL terminology is included as well as a brief explanation of the concept behind each term. The original specification of these terms can be found among others in the "NVIDIA CUDA C Programming Guide" [160] and "The OpenCL Specification" [6].

A.1 Platform & Hardware-Related Terms

Host

CUDA Term: **Host**
OpenCL Term: **Host**

Main system executing sequential program parts and orchestrating the offloading of parallel program parts to coprocessor *Devices*.

Device

CUDA Term: **Device**
OpenCL Term: **Device**

Coprocessor devices that can be used to offload parallel program portions, comprised of multiple, parallel, and independent vector processors.

Multiprocessor

CUDA Term: **Streaming Multiprocessor (SM)**
OpenCL Term: **Compute Unit**

Vector processor comprising of multiple scalar processors (*Thread Processors*).

Thread Processor

CUDA Term: **Thread Processor / CUDA Core**
OpenCL Term: **Processing Element**

Scalar processor for executing instructions of a parallel program instance.

Device Driver

CUDA Term: **Driver**
OpenCL Term: **Driver**

Software offering APIs to interact with the device hardware within the *Host* program.

Installable Client Driver

CUDA Term: **–**
OpenCL Term: **Installable Client Driver (ICD)**

Light-weight OpenCL driver software thats wraps *Device Drivers* of multiple vendors for loading them during runtime and transparently dispatching driver API calls.

A.2 Interaction between Host and Devices

Heterogeneous Platform

CUDA Term: –
OpenCL Term: **Platform**

Object representing the combination of a *Host* with potentially multiple connected coprocessor *Devices* within a (OpenCL) program. Can be used to query *Device* properties and creating *Device Contexts* for interacting with them. Implicitly defined in CUDA because only a single CUDA driver is allowed while OpenCL allows multiple *Devices* for each available *Device Driver* in the system.

Device Context

CUDA Term: **Context**
OpenCL Term: **Context**

Object representing the environment for interacting and managing resources associated with specific *Devices*.

Device Command Queue

CUDA Term: **Stream**
OpenCL Term: **Command Queue**

Object controlled by the *Host* for scheduling operations to a single *Device*. Operations can be synchronized via *Events*.

Event

CUDA Term: **Event**
OpenCL Term: **Event**

Object encapsulating the status of an (asynchronous) operation scheduled to a *Device* via a *Device Command Queue*. Can be used to query the execution status, profile its execution, and synchronize operations.

Device Program

CUDA Term: **Program**
OpenCL Term: **Program**

Set of *Kernels*, corresponding helper functions, and defined constants that have been compiled for execution on a *Device*.

Kernel

CUDA Term: **Kernel**
OpenCL Term: **Kernel**

Function representing a parallel portion of a program that can be invoked from the *Host* for executing it on a coprocessor *Device*. Will be instantiated once for each unique *Thread ID* in the overall index space.

Kernel Instance

CUDA Term: –
OpenCL Term: **Kernel Instance**

Instantiation of a parallel *Kernel* function, including its arguments and corresponding index space representing the problem size.

A.3 Execution Model Terminology

Thread ID

CUDA Term: **Thread ID**
OpenCL Term: **Global ID**

Unique ID identifying a single unit of work within the entire index space of a data-parallel *Device Program* that is executed by a *Kernel Instance*.

Thread

CUDA Term: **Thread**
OpenCL Term: **Work Item**

Finest granular unit of work in a *Kernel* program that is executed by a *Thread Processor*.

Thread Block

CUDA Term: **Thread Block**
OpenCL Term: **Work Group**

A collection of *Threads*, with possible communication between them. Executed by independent *Multiprocessors*.

Grid

CUDA Term: **Grid**
OpenCL Term: **ND Range Kernel**

Independent pieces of a *Kernel Instance*, i. e., a collection of multiple *Thread Blocks*, without means of direct communication between them. A scheduling unit for whole *Devices* where they are executed in parallel by the available *Multiprocessors*. Scheduling is done via *Device Command Queues* and synchronization can be achieved via *Events*.

Warp

CUDA Term: **Warp**
OpenCL Term: **Wave Front**

A set of *Threads* that is used as scheduling unit by *Multiprocessors* of the underlying *Device* hardware. Usually executed in lockstep mode as SIMD/SIMT instructions.

A.4 Memory Model Terminology

Host Memory

CUDA Term: **Host Memory**
OpenCL Term: **Host Memory**

Random access memory that is available in the address space of the *Host*. Without *Unified Memory* it is read/writable only from the host part of a heterogenous program.

Pinned Host Memory

CUDA Term: **Pinned / Page-Locked Host Memory**
OpenCL Term: **–**

Host Memory which is managed by the *Device Driver* instead of the operating system. Allows the driver to schedule asynchronous memory transfer operations via DMA controllers.

Device Memory

CUDA Term: **Device Memory**
OpenCL Term: **Global Memory**

Random access memory in the address space of a *Device* that is read/writable by the *Host* as well as by *Threads* that are executed on the *Device*.

Unified Memory

CUDA Term: **Unified Memory**
OpenCL Term: **Shared Virtual Memory**

Virtual common address space for both, *Host Memory* and *Device Memory*. Memory accesses are transparently translated to copy operations by the *Device Driver* if needed, i. e., in case the *Device* does not share a common memory bus with the *Host*.

Shared Memory

CUDA Term: **Shared Memory**
OpenCL Term: **Local Memory**

Random access memory on a *Device* that is read/writable by all *Threads* within in a single *Thread Block*. Can be used as cache and for implementing communication between *Threads*. Shared access can be synchronized via *Barriers*.

Barrier

CUDA Term: **Barrier**
OpenCL Term: **Work Group Barrier**

Synchronization primitive for *Threads* within a *Thread Block*. Enforces that all concurrently executed *Threads* have reached this instruction before any other *Thread* in the *Thread Block* can continue operation.

Private Memory

CUDA Term: **Register**
OpenCL Term: **Private Memory**

Memory that is solely read/writable by a single *Thread*.

Constant Memory

CUDA Term: **Constant Memory**
OpenCL Term: **Constant Memory**

Special *Device Memory* region with read-only access for all *Threads*.

Texture Memory

CUDA Term: **Texture Memory**
OpenCL Term: **–**

Special *Device Memory* region that is optimized for image texture operations.

Bibliography

[1] libgist v.1.0. `http://gist.cs.berkeley.edu/libgistv1`.

[2] libgist v.2.0/amdb v.1.0. `http://gist.cs.berkeley.edu/libgist-2.0`.

[3] OGRE 3D - Object-Oriented Graphics Rendering Engine. `http://www.ogre3d.org`.

[4] Persistency of Vision Ray Tracer (pov-ray). `http://www.povray.org`.

[5] The OpenCL Specification Version 1.2, 2012.

[6] The OpenCL Specification Version 2.0, 2015.

[7] The OpenCL C++ Specification version 1.0, Apr. 2016.

[8] Advanced Micro Devices, Inc. OpenCL Optimization Case Study Fast Fourier Transform – Part II. Technical report, Advanced Micro Devices, Inc, December 2011.

[9] Advanced Micro Devices, Inc. AMD Accelerated Parallel Processing. *OpenCL Programming Guide*, November 2013.

[10] Advanced Micro Devices, Inc. Sea Islands Series Instruction Set Architecture. Technical report, Advanced Micro Devices, Inc, February 2013.

[11] A. Ailamaki. Managing Scientific Data: Lessons, Challenges, and Opportunities. In *Proceedings of the 2011 international conference on Management of data*, SIGMOD '11, pages 1045–1046, New York, NY, USA, 2011. ACM.

[12] A. Ailamaki, D. J. DeWitt, M. D. Hill, and D. A. Wood. DBMSs on a Modern Processor: Where Does Time Go? In *Proceedings of the 25th International Conference on Very Large Data Bases*, VLDB '99, pages 266–277, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.

[13] V. H. Allan, R. B. Jones, R. M. Lee, and S. J. Allan. Software Pipelining. *ACM Comput. Surv.*, 27(3):367–432, Sept. 1995.

[14] Altera. OpenCL on FPGAs for GPU Programmers. *Altera Whitepaper*, 2014.

[15] G. M. Amdahl. Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), pages 483–485, New York, NY, USA, 1967. ACM.

[16] P. M. Aoki. Generalizing "Search" in Generalized Search Trees. In *Data Engineering, 1998. Proceedings., 14th International Conference on*, pages 380–389, Feb 1998.

[17] W. G. Aref and I. F. Ilyas. SP-GiST: An Extensible Database Index for Supporting Space Partitioning Trees. *Journal of Intelligent Information Systems*, 17:215–240, 2001. 10.1023/A:1012809914301.

[18] L. Arge. The Buffer Tree: A Technique for Designing Batched External Data Structures. *Algorithmica*, 37(1):1–24, Sep 2003.

[19] B. Barney et al. Introduction to Parallel Computing. *Lawrence Livermore National Laboratory*, 6(13):10, 2010.

[20] R. Bayer and E. McCreight. Organization and Maintenance of Large Ordered Indices. In *Proceedings of the 1970 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control*, SIGFIDET '70, pages 107–141, New York, NY, USA, 1970. ACM.

[21] R. Bayer and K. Unterauer. Prefix B-Trees. *ACM Trans. Database Syst.*, 2(1):11–26, Mar. 1977.

[22] F. Beier, T. Kilias, and K.-U. Sattler. GiST Scan Acceleration using Coprocessors. In *Proc. of 8th Int. Workshop on Data Management on New Hardware*, DaMoN '12, pages 63–69. ACM, 2012.

[23] F. Beier and K.-U. Sattler. GPU-GIST – A Case of Generalized Database Indexing on Modern Hardware. *it-Information Technology*, 59(3):141–149, 2017.

[24] F. Beier, K.-U. Sattler, C. Dinh, and D. Baumgarten. Dataflow Programming for Big Engineering Data. In *Datenbanksysteme für Business, Technologie und Web (BTW), 16. Fachtagung des GI-Fachbereichs Datenbanken und Informationssysteme (DBIS) Workshopband, 2.-3.3.2015 in Hamburg, Germany. Proceedings*, 2015.

[25] J. L. Bentley. Multidimensional Binary Search Trees Used for Associative Searching. *Commun. ACM*, 18(9):509–517, Sept. 1975.

[26] S. Berchtold, C. Böhm, and H.-P. Kriegal. The Pyramid-Technique: Towards Breaking the Curse of Dimensionality. In *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data*, SIGMOD '98, pages 142–153, New York, NY, USA, 1998. ACM.

[27] J. V. d. Bercken, B. Blohsfeld, J.-P. Dittrich, J. Krämer, T. Schäfer, M. Schneider, and B. Seeger. XXL - A Library Approach to Supporting Efficient Implementations of Advanced Database Queries. In *Proceedings of the 27th International Conference on Very Large Data Bases*, VLDB '01, pages 39–48, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc.

[28] R. Binna, E. Zangerle, M. Pichl, G. Specht, and V. Leis. HOT: A Height Optimized Trie Index for Main-Memory Database Systems. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, pages 521–534, New York, NY, USA, 2018. ACM.

[29] G. E. Blelloch. Prefix Sums and their Applications. *Report CMU-CS-90-190, School of Computer Science, Carnegie Mellon University*, 1990.

[30] R. D. Blumofe and C. E. Leiserson. Scheduling Multithreaded Computations by Work Stealing. *J. ACM*, 46(5):720–748, Sept. 1999.

[31] S. Bochkanov and V. Bystritsky. ALGLIB. `http://www.alglib.net`, 2013. [Online; accessed 26-April-2013].

[32] P. Bohannon, P. Mcllroy, and R. Rastogi. Main-Memory Index Structures with Fixed-size Partial Keys. *SIGMOD Rec.*, 30(2):163–174, May 2001.

[33] C. Böhm, S. Berchtold, and D. A. Keim. Searching in High-dimensional Spaces: Index Structures for Improving the Performance of Multimedia Databases. *ACM Comput. Surv.*, 33(3):322–373, Sept. 2001.

[34] P. A. Boncz, M. Zukowski, and N. J. Nes. MonetDB/X100: Hyper-Pipelining Query Execution. In *Proceedings of International conference on verly large data bases (VLDB) 2005*. Very Large Data Base Endowment, 2005.

[35] E. L. Bosworth. *Design and Architecture of Digital Computers: An Introduction*. Columbus State University, 2011.

[36] S. Breß. *Efficient Query Processing in Co-Processor-Accelerated Databases*. PhD thesis, Otto von Guericke University Magdeburg, 2015.

[37] S. Breß, F. Beier, H. Rauhe, K.-U. Sattler, E. Schallehn, and G. Saake. Efficient Co-Processor Utilization in Database Query Processing. *Information Systems*, 38(8):1084–1096, November 2013.

[38] S. Breß, F. Beier, H. Rauhe, E. Schallehn, K.-U. Sattler, and G. Saake. Automatic Selection of Processing Units for Coprocessing in Databases. In *16th East-European Conference on Advances in Databases and Information Systems*, Lecture Notes in Computer Science. Springer, September 2012.

[39] B. Brüderlin, M. Heyer, and S. Pfützner. Interviews3D: A Platform for Interactive Handling of Massive Data Sets. *IEEE Computer Graphics and Applications*, 27(6):48–59, Nov 2007.

[40] R. Budruk, D. Anderson, and T. Shanley. *PCI express system architecture*. Addison-Wesley Professional, 2004.

[41] K. Chakrabarti and S. Mehrotra. Efficient Concurrency Control in Multidimensional Access Methods. In *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data*, SIGMOD '99, pages 25–36, New York, NY, USA, 1999. ACM.

[42] R. Chappell, B. Toll, and R. Singhal. Intel Next Generation Micro Architecture Codename Haswell: New Processor Innovations. In *Intel Developer Forum, San Francisco*, 2012.

[43] H. Chavan, R. Alghamdi, and M. F. Mokbel. Towards a GPU Accelerated Spatial Computing Framework. In *2016 IEEE 32nd International Conference on Data Engineering Workshops (ICDEW)*, pages 135–142, May 2016.

[44] E. Chávez, G. Navarro, R. Baeza-Yates, and J. L. Marroquín. Searching in Metric Spaces. *ACM Comput. Surv.*, 33(3):273–321, Sept. 2001.

[45] S. Chen, P. B. Gibbons, and T. C. Mowry. Improving Index Performance Through Prefetching. In *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data*, SIGMOD '01, pages 235–246, New York, NY, USA, 2001. ACM.

[46] G. Chrysos. Intel Xeon Phi X100 Family Coprocessor - the Architecture. Technical report, Intel Corporation, November 2012.

[47] P. Ciaccia, M. Patella, and P. Zezula. M-tree: An Efficient Access Method for Similarity Search in Metric Spaces. In *Proceedings of the 23rd International Conference on Very Large Data Bases*, VLDB '97, pages 426–435, San Francisco, CA, USA, 1997. Morgan Kaufmann Publishers Inc.

[48] D. Comer. Ubiquitous B-Tree. *ACM Comput. Surv.*, 11(2):121–137, June 1979.

[49] H. Consortium et al. HyperTransport I/O Technology Overview: An Optimized, Low-latency Board-level Architecture. *The HyperTransport Consortium, www. hypertransport. com*, 2004.

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

[51] M. Daga and M. Nutter. Exploiting Coarse-Grained Parallelism in B+ Tree Searches on an APU. In *Proceedings of the 2012 SC Companion: High Performance Computing, Networking Storage and Analysis*, SCC '12, pages 240–247, Washington, DC, USA, 2012. IEEE Computer Society.

[52] H. Doraiswamy, H. T. Vo, C. T. Silva, and J. Freire. A GPU-based Index to Support Interactive Spatio-Temporal Queries over Historical Data. In *2016 IEEE 32nd International Conference on Data Engineering (ICDE)*, pages 1086–1097, May 2016.

[53] P. Du, R. Weber, P. Luszczek, S. Tomov, G. Peterson, and J. Dongarra. From CUDA to OpenCL: Towards a performance-portable solution for multi-platform GPU programming. *Parallel Computing*, 38(8):391 – 407, 2012. {APPLICATION} {ACCELERATORS} {IN} {HPC}.

[54] P. Estérie, M. Gaunard, J. Falcou, and J.-T. Lapresté. Practical SIMD acceleration with Boost.SIMD. *Meeting C++*, 2012.

[55] J. Fang, H. Sips, L. Zhang, C. Xu, Y. Che, and A. L. Varbanescu. Test-Driving Intel Xeon Phi. In *Proceedings of the 5th ACM/SPEC International Conference on Performance Engineering*, ICPE '14, pages 137–148, New York, NY, USA, 2014. ACM.

[56] J. Fang, A. L. Varbanescu, and H. Sips. A Comprehensive Performance Comparison of CUDA and OpenCL. In *2011 International Conference on Parallel Processing*, pages 216–225, Sept 2011.

[57] R. Fang, B. He, M. Lu, K. Yang, N. K. Govindaraju, Q. Luo, and P. V. Sander. GPUQP: Query Co-processing Using Graphics Processors. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*, SIGMOD '07, pages 1061–1063, New York, NY, USA, 2007. ACM.

[58] A. Fava, E. Fava, and M. Bertozzi. Mpipov: a parallel implementation of pov-ray based on mpi. In *PVM/MPI*, pages 426–433. Springer, 1999.

[59] F. J. Fernandez, B. Gehrels, B. Lalande, M. Łoskot, and L. J. Simonson. Boost.Geometry.Index. `http://www.boost.org/doc/libs/1_61_0/libs/geometry/doc/html/geometry/spatial_indexes.html`.

[60] R. A. Finkel and J. L. Bentley. Quad Trees a Data Structure for Retrieval on Composite Keys. *Acta Informatica*, 4(1):1–9, 1974.

[61] J. Fix, A. Wilkes, and K. Skadron. Accelerating Braided B+-Tree Searches on a GPU with CUDA. In *Proceedings of the 2nd Workshop on Applications for Multi and Many Core Processors: Analysis, Implementation, and Performance (A4MMC)*, 2011.

[62] T. Foley and J. Sugerman. KD-tree Acceleration Structures for a GPU Raytracer. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, HWWS '05, pages 15–22, New York, NY, USA, 2005. ACM.

[63] P. Francisco. *The Netezza Data Appliance Architecture: A Platform for High Performance Data Warehousing and Analytics*. IBM Redbooks, 2011. http://www.redbooks.ibm.com/abstracts/redp4725.html.

[64] V. Gaede and O. Günther. Multidimensional Access Methods. *ACM Comput. Surv.*, 30(2):170–231, June 1998.

[65] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Pearson Education, 1994.

[66] B. Gaster, L. Howes, D. R. Kaeli, P. Mistry, and D. Schaa. *Heterogeneous Computing with OpenCL: Revised OpenCL 1.2*. Elsevier, 2012.

[67] B. R. Gaster and L. Howes. The OpenCL C++ Wrapper API 1.2.6, 2012.

[68] G. Giannikis, G. Alonso, and D. Kossmann. SharedDB: Killing One Thousand Queries with One Stone. *Proc. VLDB Endow.*, 5(6):526–537, Feb. 2012.

[69] F. Gieseke, J. Heinermann, C. Oancea, and C. Igel. Buffer kd-Trees: Processing Massive Nearest Neighbor Queries on GPUs. In *Proceedings of The 31st International Conference on Machine Learning*, pages 172–180, 2014.

[70] A. Glassner. *An Introduction to Ray Tracing*. The Morgan Kaufmann Series in Computer Graphics. Elsevier Science, 1989.

[71] M. Goldfarb, Y. Jo, and M. Kulkarni. General Transformations for GPU Execution of Tree Traversals. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 10:1–10:12, New York, NY, USA, 2013. ACM.

[72] A. Goldhammer and J. Ayer Jr. Understanding performance of PCI express systems. *Xilinx WP350, Sept*, 4, 2008.

[73] J. Goldstein, R. Ramakrishnan, and U. Shaft. Compressing Relations and Indexes. In *Proceedings of the Fourteenth International Conference on Data Engineering*, ICDE '98, pages 370–379, Washington, DC, USA, 1998. IEEE Computer Society.

[74] G. Graefe. The Five-Minute Rule 20 Years Later: And How Flash Memory Changes the Rules. *Queue*, 6(4):40–52, July 2008.

[75] G. Graefe. Modern B-Tree Techniques. *Found. Trends databases*, 3(4):203–402, Apr. 2011.

[76] A. Grama. *Introduction to Parallel Computing*. Pearson Education, 2003.

[77] J. Gray and G. Graefe. The Five-Minute Rule Ten Years Later, and other Computer Storage Rules of Thumb. *SIGMOD Rec.*, 26(4):63–68, Dec. 1997.

[78] J. Gray and F. Putzolu. The 5 Minute Rule for Trading Memory for Disc Accesses and the 10 Byte Rule for Trading Memory for CPU Time. *SIGMOD Rec.*, 16(3):395–398, Dec. 1987.

[79] C. Gregg and K. Hazelwood. Where is the Data? Why You Cannot Debate CPU vs. GPU Performance Without the Answer. In *Performance Analysis of Systems and Software (ISPASS), 2011 IEEE International Symposium on*, pages 134–144, April 2011.

[80] O. Guenther and A. Buchmann. Research Issues in Spatial Databases. *SIGMOD Rec.*, 19(4):61–68, Dec. 1990.

[81] A. Guttman. R-trees: A Dynamic Index Structure for Spatial Searching. In *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*, SIGMOD '84, pages 47–57, New York, NY, USA, 1984. ACM.

[82] M. Hadjieleftheriou, E. Hoel, and V. J. Tsotras. SaIL: A Spatial Index Library for Efficient Application Integration. *GeoInformatica*, 9(4):367–389, 2005.

[83] M. Harris, S. Sengupta, and J. D. Owens. Parallel Prefix Sum (Scan) with CUDA. *GPU gems*, 3(39):851–876, 2007.

[84] M. Harvey and G. D. Fabritiis. Swan: A tool for porting CUDA programs to OpenCL. *Computer Physics Communications*, 182(4):1093 – 1099, 2011.

[85] A. Heinecke, M. Klemm, and H.-J. Bungartz. From GPGPU to Many-Core: Nvidia Fermi and Intel Many Integrated Core Architecture. *Computing in Science & Engineering*, 14(2):78–83, 2012.

[86] J. M. Hellerstein, J. F. Naughton, and A. Pfeffer. Generalized Search Trees for Database Systems. In *Proceedings of the 21th International Conference on Very Large Data Bases*, VLDB '95, pages 562–573, San Francisco, CA, USA, 1995. Morgan Kaufmann Publishers Inc.

[87] T. H. Hetherington, M. O'Connor, and T. M. Aamodt. MemcachedGPU: Scaling-up Scale-out Key-value Stores. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, SoCC '15, pages 43–57, New York, NY, USA, 2015. ACM.

[88] M. D. Hill and M. R. Marty. Amdahl's Law in the Multicore Era. *Computer*, 41(7):33–38, July 2008.

[89] G. R. Hjaltason and H. Samet. Index-Driven Similarity Search in Metric Spaces (Survey Article). *ACM Trans. Database Syst.*, 28(4):517–580, Dec. 2003.

[90] D. R. Horn, J. Sugerman, M. Houston, and P. Hanrahan. Interactive k-D Tree GPU Raytracing. In *Proceedings of the 2007 Symposium on Interactive 3D Graphics and Games*, I3D '07, pages 167–174, New York, NY, USA, 2007. ACM.

[91] IBM Corporation. Simultaneous Multi-Threading on POWER7 Processors. *IBM Whitepaper*, 2010.

[92] Intel Corporation. Beignet OpenCL Driver. `https://www.freedesktop.org/wiki/Software/Beignet/`.

[93] Intel Corporation. Intel Hyper-Threading Technology.

[94] Intel Corporation. Intel Turbo Boost Technology in Intel Core Microarchitecture (nehalem) Based Processors. *White Paper*, 2008.

[95] Intel Corporation. Introduction to the Intel QuickPath Interconnect. *White Paper*, 2009.

[96] Intel Corporation. The Intel Xeon Phi Product Family: Highly-Parallel Processing for Unparalleled Discovery. Technical report, Intel Corporation, 2013.

[97] Intel Corporation. *Intel Xeon Phi Coprocessor System Software Developers Guide*, Mar. 2014.

[98] Intel Corporation. Intel Xeon Processor E7-8800/4800/2800 v2 Product Family. *Intel Datasheet*, 2014.

[99] J. Jeffers and J. Reinders. *Intel Xeon Phi Coprocessor High-Performance Programming*. Newnes, 2013.

[100] P. Jiménez, F. Thomas, and C. Torras. 3D Collision Detection: A Survey. *Computers & Graphics*, 25(2):269 – 285, 2001.

[101] G. Junker. *Pro OGRE 3D Programming*. Apress, 2007.

[102] H. Kaiser. Plain Threads are the GOTO of Today's Computing. In *Meeting C++*, 2014.

[103] T. Kajtazi. Development and Evaluation of a Storage Manager for a Rendering Engine. Master's thesis, Technische Universität Ilmenau, 2014.

[104] T. Kaldewey, J. Hagen, A. Di Blas, and E. Sedlar. Parallel Search on Video Cards. In *Proceedings of the First USENIX Conference on Hot Topics in Parallelism*, HotPar'09, pages 9–9, Berkeley, CA, USA, 2009. USENIX Association.

[105] K. Karimi, N. G. Dickson, and F. Hamze. A Performance Comparison of CUDA and OpenCL. *CoRR*, abs/1005.2581, 2010.

[106] H. Kasim, V. March, R. Zhang, and S. See. *Network and Parallel Computing: IFIP International Conference, NPC 2008, Shanghai, China, October 18-20, 2008. Proceedings*, chapter Survey on Parallel Programming Model, pages 266–275. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[107] Khronos Group. Khronos Launches Heterogeneous Computing Initiative. Press Release, June 2008.

[108] Khronos Group. List of OpenCL-conformant Products, May 2016.

[109] T. Kiefer, T. Kissinger, B. Schlegel, D. Habich, D. Molka, and W. Lehner. ERIS Live: A NUMA-aware In-memory Storage Engine for Tera-scale Multiprocessor Systems. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 689–692, New York, NY, USA, 2014. ACM.

[110] T. Kilias. A GPU-based Index Framework. Master's thesis, Technische Universität Ilmenau, 2012.

[111] 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.

[112] C. Kim, J. Chhugani, N. Satish, E. Sedlar, A. D. Nguyen, T. Kaldewey, V. W. Lee, S. A. Brandt, and P. Dubey. Designing Fast Architecture-sensitive Tree Search on Modern Multicore/Many-core Processors. *ACM Trans. Database Syst.*, 36(4):22:1–22:34, Dec. 2011.

[113] J. Kim, S. Hong, and B. Nam. A Performance Study of Traversing Spatial Indexing Structures in Parallel on GPU. In *2012 IEEE 14th International Conference on High Performance Computing and Communication 2012 IEEE 9th International Conference on Embedded Software and Systems*, pages 855–860, June 2012.

[114] J. Kim, W. Jeong, and B. Nam. Exploiting Massive Parallelism for Indexing Multi-Dimensional Datasets on the GPU. *IEEE Transactions on Parallel and Distributed Systems*, 26(8):2258–2271, Aug 2015.

[115] K. Kim, S. K. Cha, and K. Kwon. Optimizing Multidimensional Index Trees for Main Memory Access. *SIGMOD Rec.*, 30(2):139–150, May 2001.

[116] M. Kim, L. Liu, and W. Choi. A GPU-aware Parallel Index for Processing High-dimensional Big Data. *IEEE Transactions on Computers*, pages 1–1, 2018.

[117] W.-S. Kim, W.-K. Loh, and W.-S. Han. CC-GiST: Cache Conscious-Generalized Search Tree for Supporting Various Fast Intelligent Applications. In S. Mehrotra, D. Zeng, H. Chen, B. Thuraisingham, and F.-Y. Wang, editors, *Intelligence and Security Informatics*, volume 3975 of *Lecture Notes in Computer Science*, pages 657–658. Springer Berlin / Heidelberg, 2006.

[118] W.-S. Kim, W.-K. Loh, and W.-S. Han. *Computational Science – ICCS 2006: 6th International Conference, Reading, UK, May 28-31, 2006. Proceedings, Part III*, chapter Performance Analysis of the Cache Conscious-Generalized Search Tree, pages 648–655. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006.

[119] L. B. Kish. End of Moore's law: thermal (noise) death of integration in micro and nano electronics. *Physics Letters A*, 305(3):144–149, 2002.

[120] T. Kissinger, B. Schlegel, D. Habich, and W. Lehner. KISS-Tree: Smart Latch-free In-memory Indexing on Modern Architectures. In *Proceedings of the Eighth International Workshop on Data Management on New Hardware*, DaMoN '12, pages 16–23, New York, NY, USA, 2012. ACM.

[121] S. Kockara, T. Halic, K. Iqbal, C. Bayrak, and R. Rowe. Collision Detection: A Survey. In *2007 IEEE International Conference on Systems, Man and Cybernetics*, pages 4046–4051, Oct 2007.

[122] D. Komatitsch, S. Tsuboi, C. Ji, and J. Tromp. A 14.6 Billion Degrees of Freedom, 5 Teraflops, 2.5 Terabyte Earthquake Simulation on the Earth Simulator. In *Proceedings of the 2003 ACM/IEEE Conference on Supercomputing*, SC '03, pages 4–, New York, NY, USA, 2003. ACM.

[123] M. Kornacker, C. Mohan, and J. M. Hellerstein. Concurrency and Recovery in Generalized Search Trees. *SIGMOD Rec.*, 26(2):62–72, June 1997.

[124] D. Kossmann and P. M. Fischer. Batched Processing for Information Filters. In *21st International Conference on Data Engineering (ICDE'05)(ICDE)*, volume 00, pages 902–913, 04 2005.

[125] M. Kunjir and A. Manthramurthy. Using Graphics Processing in Spatial Indexing Algorithms. Technical report, Indian Institute of Science, 2009.

[126] V. Köppen, M. Schäler, and R. Schröter. Toward Variability Management to Tailor High Dimensional Index Implementations. In *2014 IEEE Eighth International Conference on Research Challenges in Information Science (RCIS)*, pages 1–6, May 2014.

[127] C. Lameter. NUMA (Non-Uniform Memory Access): An Overview. *Queue*, 11(7):40:40–40:51, July 2013.

[128] V. W. Lee, C. Kim, J. Chhugani, M. Deisher, D. Kim, A. D. Nguyen, N. Satish, M. Smelyanskiy, S. Chennupaty, P. Hammarlund, R. Singhal, and P. Dubey. Debunking the 100X GPU vs. CPU Myth: An Evaluation of Throughput Computing on CPU and GPU. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, pages 451–460, New York, NY, USA, 2010. ACM.

[129] W. Lehner, A. Ungethüm, and D. Habich. Diversity of processing units. *Datenbank-Spektrum*, 18(1):57–62, Mar 2018.

[130] V. Leis, P. Boncz, A. Kemper, and T. Neumann. Morsel-driven Parallelism: A NUMA-aware Query Evaluation Framework for the Many-core Age. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 743–754, New York, NY, USA, 2014. ACM.

[131] V. Leis, A. Kemper, and T. Neumann. The Adaptive Radix Tree: ARTful Indexing for Main-Memory Databases. In *Data Engineering (ICDE), 2013 IEEE 29th International Conference on*, pages 38–49, April 2013.

[132] M. Levoy. Efficient Ray Tracing of Volume Data. *ACM Trans. Graph.*, 9(3):245–261, July 1990.

[133] K. I. Lin, H. V. Jagadish, and C. Faloutsos. The TV-tree: An Index Structure for High-Dimensional Data. *The VLDB Journal*, 3(4):517–542, Oct. 1994.

[134] P. Lindstrom and G. Turk. Fast and Memory Efficient Polygonal Simplification. In *Visualization '98. Proceedings*, pages 279–286, Oct 1998.

[135] G. Liu, H. An, W. Han, X. Li, T. Sun, W. Zhou, X. Wei, and X. Tang. FlexBFS: A Parallelism-aware Implementation of Breadth-first Search on GPU. *SIGPLAN Not.*, 47(8):279–280, Feb. 2012.

[136] J. Liu, N. Hegde, and M. Kulkarni. Hybrid CPU-GPU Scheduling and Execution of Tree Traversals. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '16, pages 41:1–41:2, New York, NY, USA, 2016. ACM.

[137] D. Löber. GPU-unterstützte Nachbarschaftssuche auf R-Bäumen. Master's thesis, Technische Universität Ilmenau, 2013.

[138] D. B. Lomet. *Grow and Post Index Trees: Role, Techniques and Future Potential*, pages 181–206. Springer Berlin Heidelberg, Berlin, Heidelberg, 1991.

[139] C. T. Lu, J. Dai, Y. Jin, and J. Mathuria. GLIP: A Concurrency Control Protocol for Clipping Indexing. *IEEE Transactions on Knowledge and Data Engineering*, 21(5):714–728, May 2009.

[140] L. Luo, M. Wong, and W.-m. Hwu. An Effective GPU Implementation of Breadth-First Search. In *Proceedings of the 47th Design Automation Conference*, DAC '10, pages 52–55, New York, NY, USA, 2010. ACM.

[141] L. Luo, M. D. F. Wong, and L. Leong. Parallel Implementation of R-Trees on the GPU. In *17th Asia and South Pacific Design Automation Conference*, pages 353–358, Jan 2012.

[142] L. M. Maas, T. Kissinger, D. Habich, and W. Lehner. BUZZARD: A NUMA-aware In-memory Indexing System. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 1285–1286, New York, NY, USA, 2013. ACM.

[143] D. Madeira, A. Montenegro, E. Clua, and T. Lewiner. GPU Octrees and Optimized Search. In *Proceedings of VIII Brazilian Symposium on Games and Digital Entertainment*, pages 73–76, 2009.

[144] S. Manegold, P. A. Boncz, and M. L. Kersten. Optimizing Database Architecture for the new Bottleneck: Memory Access. *The VLDB Journal*, 9(3):231–246, Dec 2000.

[145] Y. Mao, E. Kohler, and R. T. Morris. Cache Craftiness for Fast Multicore Key-value Storage. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, pages 183–196, New York, NY, USA, 2012. ACM.

[146] H. Markram. The Blue Brain Project. *Nature Reviews Neuroscience*, 7(2):153–160, 2006.

[147] G. Martinez, M. Gardner, and W. c. Feng. CU2CL: A CUDA-to-OpenCL Translator for Multi- and Many-Core Architectures. In *Parallel and Distributed Systems (ICPADS), 2011 IEEE 17th International Conference on*, pages 300–307, Dec 2011.

[148] T. G. Mattson, B. A. Sanders, and B. L. Massingill. *Patterns for Parallel Programming*. Pearson Education, 2004.

[149] M. McCool, A. Robison, and J. Reinders. *Structured Parallel Programming: Patterns for Efficient Computation*. Morgan Kaufmann. Elsevier/Morgan Kaufmann, 2012.

[150] D. Meagher. Geometric Modeling Using Octree Encoding. *Computer graphics and image processing*, 19(2):129–147, 1982.

[151] U. Milic, O. Villa, E. Bolotin, A. Arunkumar, E. Ebrahimi, A. Jaleel, A. Ramirez, and D. Nellans. Beyond the Socket: NUMA-aware GPUs. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-50 '17, pages 123–135, New York, NY, USA, 2017. ACM.

[152] S. Mittal and J. S. Vetter. A Survey of CPU-GPU Heterogeneous Computing Techniques. *ACM Computing Surveys (CSUR)*, 47(4):69, 2015.

[153] S. Mittal and J. S. Vetter. A Survey of CPU-GPU Heterogeneous Computing Techniques. *ACM Comput. Surv.*, 47(4):69:1–69:35, July 2015.

[154] R. Mueller, J. Teubner, and G. Alonso. Data Processing on FPGAs. *Proc. VLDB Endow.*, 2(1):910–921, Aug. 2009.

[155] J. Nickolls, I. Buck, M. Garland, and K. Skadron. Scalable Parallel Programming with CUDA. *ACM Queue*, 6(2):40–53, Mar. 2008.

[156] NVIDIA Corporation. CUDA Toolkit Archive.

[157] NVIDIA Corporation. NVIDIA's Next Generation CUDA Compute Architecture: Fermi. *Nvidia Whitepaper*, 2009.

[158] NVIDIA Corporation. OpenCL Best Practices Guide. *Nvidia Corporation*, May 2010.

[159] NVIDIA Corporation. Tesla M2050/M2070 GPU Computing Module. Technical report, NVIDIA Corporation, 2010.

[160] NVIDIA Corporation. Nvidia CUDA C Programming Guide Version 4. 2. *Nvidia Corporation*, 2012.

[161] NVIDIA Corporation. NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110. *Nvidia Whitepaper*, 2012.

[162] NVIDIA Corporation. Nvidia Tesla GPU Accelerators. Technical report, NVIDIA Corporation, 2013.

[163] NVIDIA Corporation. NVIDIA NVLink High-Speed Interconnect: Application Performance. *Nvidia Whitepaper*, 2014.

[164] NVIDIA Corporation. Tesla K80 GPU Accelerator Board Secification. Technical report, NVIDIA Corporation, 2015.

[165] A. Papakonstantinou, K. Gururaj, J. A. Stratton, D. Chen, J. Cong, and W. M. W. Hwu. FCUDA: Enabling Efficient CPU-Compilation of CUDA Kernels onto FPGAs. In *Application Specific Processors, 2009. SASP '09. IEEE 7th Symposium on*, pages 35–42, July 2009.

[166] T. Plachetka. POV || Ray: Persistence of Vision Parallel Raytracer. In *Proc. of Spring Conf. on Computer Graphics, Budmerice, Slovakia*, pages 123–129, 1998.

[167] S. K. Prasad, M. McDermott, X. He, and S. Puri. GPU-based Parallel R-Tree Construction and Querying. In *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*, pages 618–627, May 2015.

[168] D. Qiu, S. May, and A. Nüchter. GPU-Accelerated Nearest Neighbor Search for 3D Registration. In M. Fritz, B. Schiele, and J. Piater, editors, *Computer Vision Systems*, volume 5815 of *Lecture Notes in Computer Science*, pages 194–203. Springer Berlin / Heidelberg, 2009.

[169] B. Raducanu, P. A. Boncz, and M. Zukowski. Micro Adaptivity In Vectorwise. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD, 2013)*, June 2013.

[170] J. Rao and K. A. Ross. Cache Conscious Indexing for Decision-Support in Main Memory. In *Proceedings of the 25th International Conference on Very Large Data Bases*, VLDB '99, pages 78–89, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.

[171] J. Rao and K. A. Ross. Making B+- Trees Cache Conscious in Main Memory. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, SIGMOD '00, pages 475–486, New York, NY, USA, 2000. ACM.

[172] S. Rosendahl. CUDA and OpenCL API Comparison. *Presentation for T-106.5800 Seminar on GPGPU Programming*, 106, 2010.

[173] N. Roussopoulos, S. Kelley, and F. Vincent. Nearest Neighbor Queries. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, SIGMOD '95, pages 71–79, New York, NY, USA, 1995. ACM.

[174] M. Sadoghi, K. A. Ross, M. Canim, and B. Bhattacharjee. Making Updates Disk-I/O Friendly Using SSDs. *Proc. VLDB Endow.*, 6(11):997–1008, Aug. 2013.

[175] S. Saini, J. Chang, and H. Jin. *High Performance Computing Systems. Performance Modeling, Benchmarking and Simulation: 4th International Workshop, PMBS 2013, Denver, CO, USA, November 18, 2013. Revised Selected Papers*, chapter Performance Evaluation of the Intel Sandy Bridge Based NASA Pleiades Using Scientific and Engineering Applications, pages 25–51. Springer International Publishing, Cham, 2014.

[176] M. Sarwat, M. F. Mokbel, X. Zhou, and S. Nath. Generic and Efficient Framework for Search Trees on Flash Memory Storage Systems. *GeoInformatica*, 17(3):417–448, 2013.

[177] K.-U. Sattler, J. Teubner, F. Beier, and S. Breß. Many-Core-Architekturen zur Datenbankbeschleunigung - Tutorial. In *Datenbanksysteme für Business, Technologie und Web (BTW), 16. Fachtagung des GI-Fachbereichs Datenbanken und Informationssysteme (DBIS) Workshopband, 2.-3.3.2015 in Hamburg, Germany. Proceedings*, 2015.

[178] M. Schäler, A. Grebhahn, R. Schröter, S. Schulze, V. Köppen, and G. Saake. QuEval: Beyond High-dimensional Indexing à La Carte. *Proc. VLDB Endow.*, 6(14):1654–1665, Sept. 2013.

[179] B. Schlegel, R. Gemulla, and W. Lehner. k-Ary Search on Modern Processors. In *Proceedings of the Fifth International Workshop on Data Management on New Hardware*, DaMoN '09, pages 52–60, New York, NY, USA, 2009. ACM.

[180] D. Schmidt, M. Stal, H. Rohnert, and F. Buschmann. *Pattern-Oriented Software Architecture, Patterns for Concurrent and Networked Objects*. Pattern-Oriented Software Architecture. Wiley, 2013.

[181] L.-C. Schulz, D. Broneske, and G. Saake. An eight-dimensional systematic evaluation of optimized search algorithms on modern processors. *Proc. VLDB Endow.*, 11(11):1550–1562, July 2018.

[182] T. K. Sellis, N. Roussopoulos, and C. Faloutsos. The R^+-Tree: A Dynamic Index for Multi-Dimensional Objects. In *Proceedings of the 13th International Conference on Very Large Data Bases*, VLDB '87, pages 507–518, San Francisco, CA, USA, 1987. Morgan Kaufmann Publishers Inc.

[183] J. Sewall, J. Chhugani, C. Kim, N. Satish, and P. Dubey. PALM: Parallel Architecture-Friendly Latch-Free Modifications to B+ Trees on Many-Core Processors. *Proc. VLDB Endowment*, 4(11):795–806, 2011.

[184] A. Shahvarani and H.-A. Jacobsen. A Hybrid B+-tree As Solution for In-Memory Indexing on CPU-GPU Heterogeneous Computing Platforms. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 1523–1538, New York, NY, USA, 2016. ACM.

[185] D. Singh. Implementing FPGA Design with the OpenCL Standard. *Altera whitepaper*, 2011.

[186] D. Singh. Implementing FPGA Design with the OpenCL Standard. *Altera Whitepaper*, 2011.

[187] N. Sitchinava and N. Zeh. A Parallel Buffer Tree. In *Proceedings of the Twenty-fourth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '12, pages 214–223, New York, NY, USA, 2012. ACM.

[188] K. Skadron. Implications of the Power Wall in a Manycore Era. Lecture Slides LAVA/HotSpot Lab, Dept. of Computer Science University of Virginia, 2007.

[189] T. Stich. Fermi Hardware & Performance Tips. Technical report, NVIDIA Corporation, 2010.

[190] A. Stougiannis, M. Pavlovic, F. Tauheed, T. Heinis, and A. Ailamaki. Data-driven Neuroscience: Enabling Breakthroughs via Innovative Data Management. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 953–956, New York, NY, USA, 2013. ACM.

[191] J. Stuecheli. Power Technology For a Smarter Future. *IBM White Paper*, 2014.

[192] H. Sutter. The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software. *Dr. Dobb's journal*, 30(3):202–210, 2005.

[193] F. Tauheed, L. Biveinis, T. Heinis, F. Schurmann, H. Markram, and A. Ailamaki. Accelerating Range Queries for Brain Simulations. In *2012 IEEE 28th International Conference on Data Engineering*, pages 941–952, April 2012.

[194] P. B. Volk, D. Habich, and W. Lehner. GPU-based Speculative Query Processing for Database Operations. In *Proceedings of the 1st International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures*, 2010.

[195] G. Wang and W. Song. Communication-Aware Task Partition and Voltage Scaling for Energy Minimization on Heterogeneous Parallel Systems. In *Parallel and Distributed Computing, Applications and Technologies (PDCAT), 2011 12th International Conference on*, pages 327–333, Oct 2011.

[196] B. Wile. Coherent Accelerator Processor Interface (CAPI) for POWER8 Systems. *IBM White Paper*, 2014.

[197] N. Wirth. A Plea for Lean Software. *Computer*, 28(2):64–68, Feb 1995.

[198] W. A. Wulf and S. A. McKee. Hitting the Memory Wall: Implications of the Obvious. *SIGARCH Comput. Archit. News*, 23(1):20–24, Mar. 1995.

[199] X. Xiao, T. Shi, P. Vaidya, and J. J. Lee. R-tree: A Hardware Implementation. In *CDES'08*, pages 3–9, 2008.

[200] S. Yalamanchili. Multicore Computing - Evolution. Lecture Slides Georgia Institute of Technology.

[201] T. Yamamuro, M. Onizuka, T. Hitaka, and M. Yamamuro. VAST-Tree: A Vector-advanced and Compressed Structure for Massive Data Tree Traversal. In *Proceedings of the 15th International Conference on Extending Database Technology*, EDBT '12, pages 396–407, New York, NY, USA, 2012. ACM.

[202] S. You, J. Zhang, and L. Gruenwald. Parallel Spatial Query Processing on GPUs Using R-Trees. In *Proceedings of the 2Nd ACM SIGSPATIAL International Workshop on Analytics for Big Geospatial Data*, BigSpatial '13, pages 23–31, New York, NY, USA, 2013. ACM.

[203] B. Yu, H. Kim, W. Choi, and D. Kwon. Parallel Range Query Processing on R-Tree with Graphics Processing Unit. In *2011 IEEE Ninth International Conference on Dependable, Autonomic and Secure Computing*, pages 1235–1242, Dec 2011.

[204] B. Yu, H. Kim, W. Choi, and D. Kwon. Accelerating Range Query Processing on R-tree Using Graphics Processing Units. *IEICE TRANSACTIONS on Information and Systems*, 96(12):2776–2785, 2013.

[205] S. Zeuch, J.-C. Freytag, and F. Huber. Adapting Tree Structures for Processing with SIMD Instructions. In *EDBT*, 2014.

[206] F. Zhang, P. Di, H. Zhou, X. Liao, and J. Xue. RegTT: Accelerating Tree Traversals on GPUs by Exploiting Regularities. In *2016 45th International Conference on Parallel Processing (ICPP)*, pages 562–571, Aug 2016.

[207] F. Zhang and J. Xue. Poker: Permutation-based SIMD Execution of Intensive Tree Search by Path Encoding. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, CGO 2018, pages 87–99, New York, NY, USA, 2018. ACM.

[208] K. Zhang, J. Hu, B. He, and B. Hua. DIDO: Dynamic Pipelines for In-Memory Key-Value Stores on Coupled CPU-GPU Architectures. In *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*, pages 671–682, April 2017.

[209] K. Zhang, K. Wang, Y. Yuan, L. Guo, R. Lee, and X. Zhang. Mega-KV: A Case for GPUs to Maximize the Throughput of In-memory Key-value Stores. *Proc. VLDB Endow.*, 8(11):1226–1237, July 2015.

[210] K. Zhang, K. Wang, Y. Yuan, L. Guo, R. Li, X. Zhang, B. He, J. Hu, and B. Hua. A Distributed In-memory Key-value Store System on Heterogeneous CPU-GPU Cluster. *The VLDB Journal*, 26(5):729–750, Oct. 2017.

[211] J. Zhou and K. A. Ross. Buffering Accesses to Memory-Resident Index Structures. In *Proceedings of the 29th International Conference on Very Large Data Bases - Volume 29*, VLDB '03, pages 405–416. VLDB Endowment, 2003.

[212] M. Zukowski, P. A. Boncz, N. J. Nes, and S. Héman. MonetDB/X100 - A DBMS In The CPU Cache. *IEEE Data Engineering Bulletin*, 28(2):17 – 22, June 2005.

Eidesstattliche Erklärung / Affirmation

Ich versichere, dass ich die vorliegende Arbeit ohne unzulässige Hilfe Dritter und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Die aus anderen Quellen direkt oder indirekt übernommenen Daten und Konzepte sind unter Angabe der Quelle gekennzeichnet.

Weitere Personen waren an der inhaltlich-materiellen Erstellung der vorliegenden Arbeit nicht beteiligt. Insbesondere habe ich hierfür nicht die entgeltliche Hilfe von Vermittlungs- bzw. Beratungsdiensten (Promotionsberater oder anderer Personen) in Anspruch genommen. Niemand hat von mir unmittelbar oder mittelbar geldwerte Leistungen für Arbeiten erhalten, die im Zusammenhang mit dem Inhalte der vorgelegten Dissertation stehen.

Die Arbeit wurde bisher weder im In- noch im Ausland in gleicher oder ähnlicher Form einer Prüfungsbehörde vorgelegt.

Ich bin darauf hingewiesen worden, dass die Unrichtigkeit der vorstehenden Erklärung als Täuschungsversuch angesehen wird und den erfolglosen Abbruch des Promotionsverfahrens zu Folge hat.

Ilmenau, April 16, 2019