FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of bachelor's thesis

Title:	Implementation of B-trees on GPU
Student:	Tat Dat Duong
Supervisor:	Ing. Tomáš Oberhuber, Ph.D.
Study program:	Informatics
Branch / specialization:	Computer Science
Department:	Department of Theoretical Computer Science
Validity:	until the end of summer semester 2021/2022

Instructions

1. Familiarise with the basics of programming of GPUs using CUDA.
2. Familiarise with the development of parallel algorithms using TNL library (www.tnl-project.org).
3. Learn and understand available algorithms and data structures (see [1] for example) for B-trees on GPUs.
4. Implement chosen data structures in TNL with the option of run on both GPU and CPU.
5. Implement unit tests for verifying the correct functionality of chosen algorithms.
6. Measure performance speed-ups of implemented data structures compared to suitable containers from STL library or similar data structures on datasets arising from generation of unstructured numerical meshes.

[1] M. A. Awad, S. Ashkiani, R. Johnson, M. Farah-Colton, J. D. Owens, Engineering a high-performance GPU B-Tree, PPoPP '19: Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming, p. 145-157, 2019.

Electronically approved by doc. Ing. Jan Janoušek, Ph.D. on 4 January 2021 in Prague.

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Bachelor's thesis

Implementation of B-trees on GPU

Tat Dat Duong

Department of Computer Science
Supervisor: Ing. Tomáš Oberhuber, Ph.D.

June 27, 2021

Acknowledgements

I want to thank my supervisor Ing. Tomáš Oberhuber, Ph.D., for his guidance, advice, and supervision through the work done on creating this bachelor thesis.

My gratitude extends towards my family and friends, particularly Bc. Nguyen Xuan Thang, for encouraging and helping me during my studies, especially in these trying times.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as a school work under the provisions of Article 60 (1) of the Act.

In Prague on June 27, 2021 . . . . . .

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis

Duong, Tat Dat. *Implementation of B-trees on GPU*. Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2021.

Abstrakt

B-Tree je datová struktura, která provádí vkládání, mazání a vyhledávání klíčů a hodnot se složitostí $\mathcal{O}(\log n)$. Tato práce se zabývá jejich studiem a implementací na kartách GPU. Byly naimplementovány dvě varianty B-Tree: B$^+$Tree a B-Link-Tree, obě patřičně upravené pro paralelní zpracování. Tyto varianty jsou popsané a implementované pro grafické karty NVIDIA v jazyce C++ s pomocí CUDA API a TNL knihovny. V práci je uvedena analýza existujících GPU i CPU řešení a jednotlivé úpravy a optimalizace provedené na výsledných strukturách. Všechny implementace jsou řádně otestované, změřené a porovnané s vybranými implementacemi dostupnými pro GPU a CPU.

Klíčová slova datové struktury, B-strom, Compute Unified Device Architecture, CUDA, grafická karta, GPU, TNL, C++, výpočetní cluster, HPC, řízení paralelního zpracování

Abstract

B-Tree is a data structure that performs inserting, deleting, and searching of key-value pairs in $\mathcal{O}(\log n)$ time. This thesis is about the implementation of a B-Tree capable of execution on GPU cards. Two variants of B-Tree are implemented: B$^+$Tree and B-Link-Tree, both modified to make use of the parallel processing power. These variants are studied and implemented for NVIDIA GPUs using the C++ programming language and CUDA API with the help of the TNL library. This thesis contains an analysis of existing GPU and CPU solutions and explains the changes and optimizations made to the presented solution. All variants are thoroughly tested, measured, and compared against chosen GPU and CPU implementations.

Keywords data structures, B-Tree, Compute Unified Device Architecture, CUDA, graphics processing unit, GPU, TNL, C++, high-performance computing, HPC, concurrency control

Contents

List of Figures

List of Tables

Introduction

Motivation

B-Tree is a well-known data structure often used as an index, a critical primitive used in various applications, such as data mining, decision support systems, and Online analytical processing (OLAP) [13, 12]. Widely found in database systems and file systems [23, 25], B-Tree became the data structure of choice used when dealing with a large amount of arbitrary data of an unknown domain.

Lookup, insertion, and deletion in the tree are in $\mathcal{O}(\log n)$, by keeping the tree height as small as possible regardless of arbitrary updates, thanks to the self-balancing nature of this data structure. Naively, operating on multiple items will thus perform in $\mathcal{O}(n \log n)$. This can be improved upon by using concurrent processing and computation on such data structure when processing multiple items.

One such hardware utilizing many processing cores to process tasks faster is the Graphical Processing Unit (GPU). With the advent of programmable shaders and support for floating-point operations, General-purpose computing on GPU became suitable for high-performance computing. Thanks to their superior computing performance to price ratio, GPUs became an essential staple for enabling high-performance computing for the masses.

Although GPUs have carved their niche and adopted some computational tasks, such as deep learning or cryptocurrency mining, an efficient GPU implementation of a B-Tree is challenging. Much attention needs to be paid when implementing concurrency control to allow high-throughput GPU performance without compromising correctness.

Nevertheless, a GPU-friendly implementation of such data structure can utilize the immense parallel processing power to accelerate querying and updating of multiple items at once. Such optimizations can unlock significant performance gains, benefiting all applications which use B-Trees.

Structure of Work

The main goal of this thesis is to introduce an implementation of a B-Tree built on top of the Template Numerical Library (TNL). This implementation will have the ability to change the execution environment to either run on the GPU or the CPU. The solution will be implemented in C++ and will support CUDA, as these are the prerequisites of the used TNL library.

First, in chapter 1, hardware structure and software architecture will be introduced, with a general primer of both the TNL library and the B-Tree data structure. Previous implementations of such data structure and other state-of-the-art CPU solutions are discussed in chapter 2.

Next, in chapter 3, different variants and modifications of the data structure, such as B^+Tree or B-Link-Tree, are explained and compared against the original B-Tree. The realization section presented in chapter 4 will explain the design decision choices and various implementation and optimization details made for both of the GPU B-Tree variants shown in this thesis.

Last but not least, implementation correctness, testing methodology, and the experimental benchmark results between the developed solution and other chosen CPU and GPU implementations are presented in chapter 5.

Chapter 1

Preliminaries

1.1 GPU architecture

Graphical Processing Unit (GPU) is a type of coprocessor designed to work alongside the CPU, offloading intensive tasks to the coprocessor, accelerating the overall system performance.

Generally speaking, CPUs are low latency, low throughput processors, whereas GPUs are high latency, high throughput processors. A traditional CPU is optimized to execute instructions as fast as possible by reducing *latency* — the duration from the start of the instruction until the availability of results. To achieve this, CPUs have to implement complex strategies such as speculative branch execution, out-of-order execution, and large memory caches. This architecture works well on workflows, which are sequential by nature, where the execution speed of a single thread has a more noticeable impact on overall execution time.

In contrast, a GPU focuses on hiding the latency instead of reducing it by focusing more on *throughput* — the amount of work completed per unit of time. The workflows that GPUs are most suited for are abundant in parallelism, for example, image processing or matrix multiplication. This assumption drives the design of a GPU to sacrifice the execution speed of a single thread in favor of the sheer amount of processing cores to increase throughput. The difference in design philosophies can be seen in fig. 1.1.

Thus, even if the CPU excels at sequential tasks, the GPU can outperform the CPU in highly parallel computations, as the sheer number of cores GPU uses compared to CPU execution amortizes the cost of using a slower one [18].

As the staggering performance of a GPU has not gone unnoticed by the scientific community, work has been done to allow the GPU to process workloads unrelated to graphics, e.g., linear solvers [20], or neural networks [19].

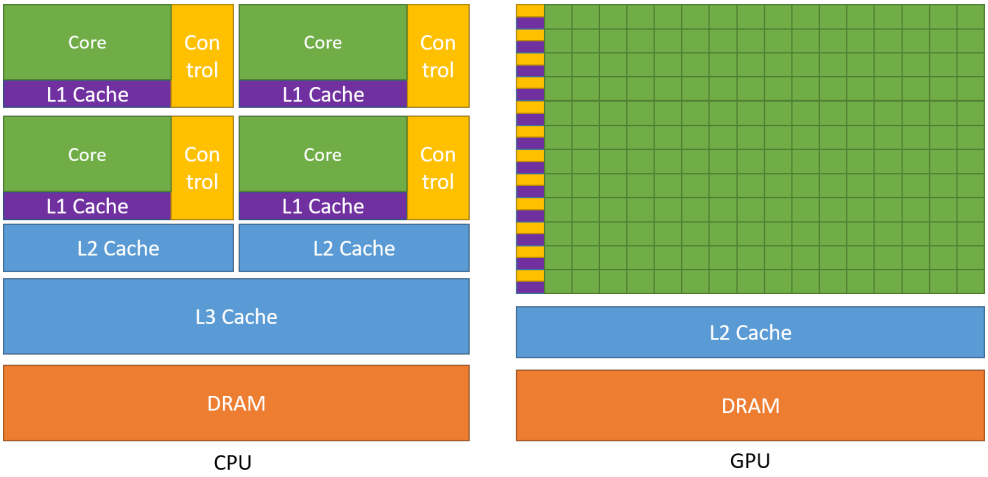

Figure 1.1: Comparison between a typical CPU vs GPU architecture, the main difference being the increased amount of computing cores [18].

1.1.1 Hardware architecture

As the implementation of both B$^+$Tree and B-Link-Tree is based on CUDA (see section 1.2), we're focusing on NVIDIA GPU and its architecture in this work. Since the introduction of unified shared architecture with GeForce 8 Series in 2006 [29], a single NVIDIA GPU is made of an array of *streaming multiprocessors* (SM). The power of a single GPU card depends on the number of these streaming multiprocessors, which may vary between models. Each SM in general have these components:

- *Streaming processors*[1], each containing an arithmetic logic unit (ALU) for integer operations and floating-point unit (FPU) for floating-point operations.
- *Register file*, where threads store their local variables. This register file is shared between all threads running in the SM. The number of available registers per thread is defined by architecture.
- *Instruction cache*, used to accelerate instruction fetching.
- *Shared memory* used to share data between threads in the same thread block (see section 1.1.2).
- *L1 / Texture cache*, which is used to store local memory of the thread or to serve as a texture cache for image data.

Newer architectures include more specialized cores. For example, *tensor cores* in every SP, capable of executing 4x4 matrix multiplication instructions,

[1] Also known as a *CUDA core*

or *RT cores* in every SM, accelerating raytracing, found in recent Ampere architecture (see fig. 1.2).

Figure 1.2: Architecture of an Ampere GA10x Streaming Multiprocessor with RT cores for accelerated raytracing [21].

1.1.2 Memory hiearchy

The GPU has its own memory with its memory hierarchy separate from the rest of the system. Every thread has access to its registry and local memory. Local memory is used to store variables for which we do not have space in registers (registry spilling), or we cannot determine if we do have space (like arrays). The registry and the local memory are not directly accessible to the programmer and are managed by the compiler instead.

Each thread also has access to shared memory, which is shared with all threads executing on the same thread block (see section 1.2). As the memory resides on-chip itself, it has higher throughput and lower latency compared to the global memory. It can be considered a user-managed cache, as the application itself takes care of allocation and access to the cache.

Finally, all threads have access to the global memory, available to all threads regardless of the residing multiprocessor. The global memory is the largest memory available on the GPU. But it is also the slowest, as it is stored on a separate chip. Read and writes are done in 32-, 64-, or 128-byte memory transactions [18].

Both local and global memory can be routed through L2 and L1 cache. L1 and L2 cache work in a similar manner as in the CPU to speed up memory access. L1 cache level has lower latency than the L2 cache level when accessing, and both of these caches are faster than accessing the memory directly. Thus, to increase memory throughput, it is desirable to store and read data in faster caches than from memory itself, keeping both cache utilization and hit rate high.

This can be achieved by organizing the memory allocation and access in a way to honor both the *temporal locality*, ensuring the duration between the reuse of specific data is as short as possible, and *spatial locality*, keeping our data access close to each other.

1.2 CUDA programming model

To simplify development on general-purpose GPUs, NVIDIA introduced the Compute Unified Device Architecture (CUDA) programming model in November 2006. As mentioned previously, GPUs are suited for parallel workloads optimizing in total throughput, sacrificing the performance of serial operations. However, not all programs are fully parallel in nature, and some problems are difficult, if not impossible, to formulate in a manner that would benefit from the use of a GPU. Thus, a sane idea would be to utilize both the CPU and the GPU, using GPU in workloads, where parallelism yields significant performance uplift. With CUDA, programmers can write applications that run on the CPU and accelerate parallel workloads with the GPU while using familiar C/C++ programming language for both processors.

In CUDA, the CPU and GPU and their memory are referred to as *host* and *device* respectively. A host manages the memory of both the device and the host itself, and it launches user-defined functions, called *kernels*, which the device executes. A program thus usually executes serial code on the host and parallel code on the device.

1.2.1 Thread hiearchy

Multiple parallel threads execute these kernels on the GPU. These threads are grouped by the programmer or compiler into *thread blocks* and *grids*, which the GPU uses to schedule work. A GPU scheduler maps these abstractions to concrete hardware units: A grid is mapped to a GPU, a thread block is mapped to a *streaming multiprocessor* (SM), and a thread is mapped to a *streaming processor* (SP / CUDA core).

From the implementation side, to launch a kernel function, a programmer must specify the number of threads in a single thread block and the number of thread blocks in a grid. CUDA extends the C++ language with `<<<numBlocks, blockSize>>>` syntax for that purpose.

Every thread in a thread block has an identifier, which is exposed inside the thread as a constant variable: `threadIdx`. Similarly, each thread block can be identified with the block index within a grid, `blockIdx`. With the dimensions of a block known `blockDim`, the global index of a thread can be computed as such:

$$x = blockIdx.x * blockDim.x + threadIdx.x$$

This global index is often used to determine what data is being accessed by a single thread. A programmer may choose to address the threads in a two-, or three-dimensional way instead of a one-dimensional way for the sake of convenience, especially when dealing with matrices or volumes.

1.2.2 SIMT architecture

Only a few threads from a thread block are executed at once by a streaming multiprocessor. Before execution, the SM schedules threads in a group of 32, called a *warp*. A warp scheduler then picks a warp and executes a single instruction with different data on all threads in a warp. This architecture is called Single Instruction, Multiple Threads (SIMT).

All threads in a warp execute the same line of code at once. This rule at face value permits any thread divergence, most notably disallowing the usage of conditionals such as if-else statements. Threads cannot run different branches in parallel, as all threads must run the same instruction simultaneously. As a workaround, divergent branches are executed in series by all threads. Threads not participating in the active execution branch are deactivated while still consuming resources. The cost of execution effectively becomes a sum of the cost of all branches. Generally, warps are at their peak efficiency when all threads run the same code without any branching.

1.2.3 Synchronization

At some point, synchronization needs to happen, either for threads to communicate and share data with each other or to ensure all threads have completed their work at a certain point.

Kernel launches from the host are asynchronous by nature, and the host code will not wait for results unless `cudaDeviceSynchronize()` is invoked. This instruction will block the host until the device has completed all preceding requested tasks [18].

Threads can also be synchronized, as threads do not execute at the same time and thus do not complete their task simultaneously. A synchronization barrier `__syncthreads()` is used to synchronize the threads in a thread block, which all threads must reach to resume execution. This barrier guarantees that all code preceding the instruction is executed before the code after the instruction.

Threads inside a thread block may use shared memory or global memory to cooperate and share data. To ensure all writes to the memory are visible to all other threads, `__threadfence()` can be used to stall the threads until writes to the memory are completed.

Warp-level primitives enable fine-tuned collective operations within warps. Parallel reductions and synchronized data exchange can be achieved with `__shfl_down()` and `__shfl_sync()` respectively. `__syncwarp()` can be utilized to create a synchronization barrier for all threads in a warp, similar to the function `__syncthreads()` for all threads in a thread block.

At last, atomic functions (such as `atomicAdd()` or `atomicCAS()`) can be used to read from or write to shared or global memory while ensuring no interference from other threads until the operation is complete.

1.3 TNL

Template Numerical Library (TNL) [24] is a C++ library offering robust tools for high-performance computing and computational science. The primary goal of this project is to provide a familiar API akin to `STL` while offering significant performance uplift by exploiting the parallel nature of GPUs.

The library makes extensive use of template meta-programming to create a unified interface for both CPU and GPU, which stays the same regardless of the selected execution target device. This interface allows the programmer to enable or disable invocation on GPU without significant rewrites. This programming pattern heavily influenced the design of the implementation.

1.3.1 Views

As mentioned in section 1.1.2, GPU have their own separate memory and address space, which the programmer must keep in mind while developing

GPU-accelerated applications. What is allocated on CPUs is directly not accessible from GPUs and vice-versa. CUDA only offers low-level primitives for managing memory on the GPU, similar to languages like C. TNL helps the programmer with memory management by including helper classes aiming to alleviate the work needed to address memories between the devices.

`TNL::Container::Array` is a template container class for one-dimensional dynamic array. By specifying a type in a template, this class allows the programmer to choose where the data will reside. If the template argument is set to `TNL::Devices::Host`, the container behaves similar to `std::vector`, storing the data in the main memory. However, when the device is set to `TNL::Devices::Cuda`, data is stored in the GPU, and all of the operations need to be invoked with parallelism in mind to utilize that fact.

Data can be directly accessed only from a device where it was previously allocated. To read or write from a different device, copying of data must occur between memories. These cross-device operations are considerably expensive and should thus be used sparingly.

One common problem when developing GPU-accelerated programs with TNL is the ability to supply an instance of Array containers. Object instances cannot be passed to the kernel by reference, and every object must be deep-copied. This implementation detail brings significant performance overhead but also raises the question: How to mirror the changes back to the CPU copy of the object? A companion class is introduced to solve this question: `TNL::Container::ArrayView`. This class implements the proxy design pattern, substituting `TNL::Container::Array`, and allows the user to read and write into the array but permits the user from performing an operation, which may change memory allocation of the array, like resizing.

Chapter 2

State-of-the-art

2.1 Prior Art

Significant work has been done in the past to optimize B-Tree indexing capabilities. This chapter focuses on past work utilizing parallel computing to improve B-Tree performance.

Bw-Tree introduced by Levandoski et al. [17] from Microsoft Research is a B-Tree structure optimized for in-memory data processing and high concurrency access without using locks. To avoid locking, an indirection layer is introduced by maintaining a chain of records for each node. These records allow atomic updates of the memory location of a node using a single compare-and-swap instruction.

Wang et al. [32] expanded the work done in [17] by describing database-specific implementation details regarding iterators and enabling non-unique key support.

Sewall et al. [27] introduced novel modifications to B^+Tree operations in the proposed PALM technique. This technique uses the Bulk Synchronous Parallel model, where queries are grouped, and the work is distributed among threads. This work also optimizes synchronization by avoiding barriers in favor of communicating adjacent threads in a point-to-point manner. Latches are avoided under condition that all search queries have been completed before insertions, and a node may be written by exactly one thread.

Previous projects related to GPU implementation of B-Trees focused on query throughput. Fix et al. [9] measured substantial performance speedup compared to sequential CPU execution by running independent queries in each thread block. Until recently, the general approach for updating is either to perform such updates on the CPU or to rebuild the structure from scratch, which is the case of this implementation.

Kim et al. proposed FAST [15], a configurable high-performance tree optimized for SIMD and multi-core CPU and GPU systems. The structure can be configured towards the target hardware architecture by specifying the

size of a cache line, SIMD register width, and memory page size. Similar to Fix et al., only bulk creation and querying are supported. Updates are done by rebuilding the tree.

Kaczmarski [14] utilized the relationship between a sorted list and demonstrated improved insertion time by presorting keys and proposing a novel bottom-up approach when constructing a tree. The work also explored several methods of key searching within a tree node.

Shahvarani et al. [28] created a B^+Tree variant utilizing the heterogeneous nature of the CPU and the GPU during a search operation. This approach has the benefit of removing the memory capacity restrictions, as the key-value pairs are residing on the host, whereas internal nodes are present on the device.

In work done by Yan et al. [34], instead of having the keys and children of a node in one instance, they are stored into separate arrays. Keys are stored in a one-dimensional array, and children are stored in a prefix-sum array. Storing keys and children into separate array improve memory locality and thus decreasing memory latency.

Awad et al. [2] proposed the most comprehensive GPU implementation of B-Tree to date. Their implementation supports concurrent queries (single, range, successor), insertion, and deletion. To achieve state-of-art performance, B-Link-Tree has been coupled with proactive splitting and Warp Cooperative Work Sharing as proposed by Ashkiani et al. [1].

Chapter 3

Theory

3.1 B-Tree

B-Tree is a balanced search tree used for storing large blocks of data. It captures and maintains the sort order of data and supports searching, sequential retrieval, insertion, and deletion in logarithmic time.

Since their invention 50 years ago [3], B-Trees have been already considered ubiquitous less than ten years later [6]. They can be found in various forms in databases (e.g., PostgreSQL [23]) and file systems (e.g., BTRFS [25]), where a performant self-balancing external index for large blocks of data is required. They can also be found in more applications such as data mining, decision support systems, and Online analytical processing (OLAP) [13, 12].

However, it raises the question of why B-Trees are used for on-disk data and binary search trees are used for in-memory data. The main reason behind this is the high overhead of data access in block-access storage, where byte access is not well supported. A typical example is disk storage, where a disk is divided into blocks. B-Tree exploits this behavior by having its nodes be as large as a whole block.

B-Trees are exceptionally useful for secondary disk-based storage, but it still yields significant improvements even when storing data in memory; as with CPU caches and memory line caches, the memory can be treated as a block-access device.

The main benefit of B-Trees is their shallow height even with many keys inserted; thus, the number of disk accesses to traverse the tree is also low.

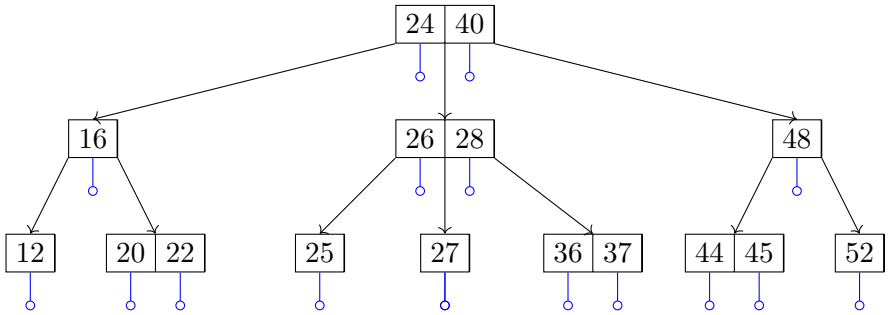

Figure 3.1: B-Tree with $Order = 3$. Blue lines indicate the presence of a pointer to a value.

Definition 1. B-Tree of an order m is a rooted tree with following properties:

1. Every node x has following attributes:
   a) $x.size$, the number of keys in a node,
   b) $x.leaf$, a boolean value indicating whether the node is a leaf node or not,
   c) an list of $x.size$ keys $x.key_1, x.key_2, \ldots, x.key_{x.size}$ sorted in ascending order ($x.key_1 \leq x.key_2 \leq \cdots \leq x.key_{x.size}$).
2. Every internal node x has $x.size + 1$ pointers as its children ($x.child_1, x.child_2, \ldots, x.child_{x.size+1}$).
3. All leaf nodes appear at the same depth, which is the height of tree h.
4. Nodes have upper and lower bounds, which limit the number of keys and children. Assuming m is the order of a B-Tree:
   a) Every node other than the root has at least $\lfloor m/2 \rfloor$ children, thus every node other than root has at least $\lfloor m/2 \rfloor - 1$ keys.
   b) The root node has at least two children, except if it is a leaf node.
   c) Every node may contain at most m children, thus may contain at most $m - 1$ keys.

An example B-Tree with $Order = 3$ can be seen in fig. 3.1. In the case of B-Tree, each key may include a pointer to a specific value, highlighted as blue lines, which is useful when implementing a map-like container.

As a note, B-Trees are a specialization of (a, b)-Trees, where a B-Tree is either an $(a, 2a)$-tree or $(a, 2a + 1)$-tree depending on the oddness / evenness of a. This is also why 2-4 trees (also known as "2-3-4-trees" which in turn are similar to RB-Trees) are B-Trees with an order of 3.

Lemma 1. B-Tree T of order m with $n \geq 1$ keys has height $h = \Theta(\log n)$.

Proof. Assume $t = \lfloor m/2 \rfloor$ as the minimum number of children in a node. The root of T has at least one key; all other nodes contain at least $t - 1$ keys.

Thus, T has at level 1 at least 2 nodes, at level 2 at least $2t$ nodes, and so forth. The last level h will have at least $2t^{h-1}$ nodes.

Define n_{min} as the minimum amount of keys in T of height h as the sum of key count per depth.

$$\begin{aligned} n_{min} &= 1 + (t-1) \cdot (2 + 2t + \cdots + 2t^{h-1}) \\ &= 1 + 2 \cdot (t-1) \cdot \sum_{i=1}^{h} t^{i-1} = 1 + 2 \cdot (t-1) \cdot \sum_{i=0}^{h-1} t^{i} \\ &= 1 + 2 \cdot (t-1) \cdot (\frac{t^h - 1}{t-1}) = 2t^h - 1 \end{aligned} \tag{3.1}$$

The maximum amount of keys at height h is defined similarily:

$$\begin{aligned} n_{max} &= (m-1) \cdot \sum_{i=0}^{h-1} m^i \\ &= (m-1) \cdot (\frac{m^h - 1}{m-1}) = m^h - 1 \end{aligned} \tag{3.2}$$

As $n_{min} \leq n \leq n_{max}$, taking logarithm will prove the theorem. □

As search, insertion, and deletion on a single node takes $\Theta(1)$, time complexity of every operation is $\Theta(\log n)$. The space complexity for B-Tree is $\Theta(n)$.

3.1.1 Search

The search algorithm for B-Trees is trivial but crucial, similar to the search algorithm for binary search trees. Keys are compared against a needle key in every node, starting from the root node. The goal is to find the position of the first key, which is greater than or equal to the needle key, as seen in lines 3–4 in the sample algorithm 3.1.1.

If the key at that position does match the needle key, the key has been found, and the function will return the node containing the needle key and the position of said key. Otherwise, the search will continue from the child node at the said position. The search ends if it ends up in a leaf node, which does not have any additional children.

Line 3–4 in the sample algorithm 3.1.1 can be replaced with a binary search or parallelized search in CUDA, which is more suited towards SIMD systems. More about that topic can be found in chapter 4.

Algorithm 3.1.1: B-Tree Search

```
1 Function Search(node, needle):
2     i ← 0
3     while i < node.size && needle > node.key_i do
4         i ← i + 1
5     if node.key_i = needle then
6         return (node, i)
7     if node.leaf = true then
8         return null
9     return Search(node.child_i, needle)
```

3.1.2 Insertion

First, the tree is traversed for the existence of the needle key. The algorithm will end up in a leaf node if the key is not found. In that case, a key needs to be inserted into the node at the correct position to preserve the key ordering.

If a node after insertion has subsequently become full after insertion, a split operation must occur to preserve the rules of the B-Tree definition 1. The node is considered *full* if the node contains exactly $m - 1$ keys. The median key is chosen as the separator, and the node is split into two smaller nodes based on that separator. The separator is inserted into the parent of the split node, which might trigger the split operation again.

When the root node needs to be split, a new node with the separator as its only key and two subtrees as its children, which the definition 1 permits (only internal nodes must contain at least $\lfloor m/2 \rfloor$ keys).

3.1.3 Deletion

Deletion in B-Trees is analogous to insertion, starting by searching the tree for the desired needle key.

If the searched needle key is found on a leaf node, it can be trivially deleted from the node. If the node has subsequently become *underfull* by not containing at least $\lceil m/2 \rceil$ keys, the tree must be rebalanced.

Deleting a key from an internal node is more complicated, as child nodes are bound to keys, and a key deletion will result in the loss of a child. Thus, instead of deleting, the key is swapped with a successor (or a predecessor) key. The key can then be deleted as it is now located in a leaf node after the swap. Similar rebalancing must occur if the node has become underfull.

Rebalancing of node x is performed when $x.size = \lfloor m/2 \rfloor - 1$. Assume, without loss of generality, a sibling node l on the left of x exists, and i is the key, which binds the node n to its parent. If a left sibling does not exist or

does not have any spare keys to share, a right sibling is used instead, and the rebalancing operation is mirrored.

If the sibling node l has at least $\lfloor m/2 \rfloor + 1$ keys, the borrowing operation can be performed; the key i is prepended to the node x and the last key of l detaches from l and replaces the key i. Last child of l is also detached and prepended to x.

However, if neither the sibling nodes have enough keys, a merge operation occurs. Both the x node and its sibling l are deleted, and a new node is created instead. This new node contains keys and children of both x and l, with key i in the middle. Deletion is, thus, propagated upwards. In the event of a root with no keys and a single child, the single child becomes the new root.

Many implementations forgo rebalancing the tree in favor of replacing the key with a tombstone [12]. The tombstone is useful even when inserting, as a key could be directly inserted in the place of a tombstone if applicable, thus alleviating the cost of inserting a new key into a sorted list. Rebalancing the tree can be postponed and executed later.

3.2 B$^+$Tree

Definition 2. B$^+$Tree is a B-Tree where keys are stored exclusively in leaf nodes.

Separators found in internal nodes can be freely chosen and may not match the actual keys in leaf nodes, as long as these separators split the tree into subtrees and preserve the ordering of the keys. Value pointers are also exclusively stored in leaf nodes, highlighted as blue lines in fig. 3.2.

As the B$^+$Tree does not reuse the keys and may duplicate the keys found in the leaf nodes to use as separators in internal nodes, they do bear increased storage requirements. Compressing techniques on keys can be used to mitigate the increased space complexity in exchange for a slight execution complexity increase due to compression itself.

In most implementations, leaf nodes may include an additional pointer to a right sibling node, as seen in fig. 3.2 highlighted as red arrows, enabling straightforward sequential querying, which is helpful for range querying.

All operations on the B$^+$Tree are simplified thanks to storing the keys exclusively in leaf nodes, as the end state when traversing a tree is always a leaf node.

Tree rebalancing operations in B$^+$Tree, such as splitting full nodes and merging free nodes, are the same as in B-Tree, with the only difference is the removal of the median key in leaf nodes. The median key is only propagated to the parent node after a split; the key itself is preserved in the leaf node, ensuring the rules of B$^+$Tree are not broken.

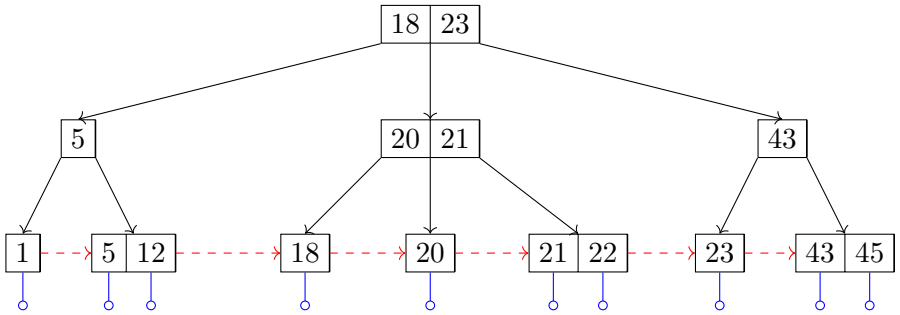

Figure 3.2: B$^+$Tree with $Order = 3$. Blue lines indicate a pointer to a value, red arrows indicate an optional pointer to a sibling node.

3.3 Concurrency Control

The primary goal of concurrency control is to ensure the correctness of results after concurrent operations are performed on the structure. Concurrency control can mean two things, either the correctness and serializability of logical contents or serializability among threads modifying data structure in memory.

In databases, the primary concern is to protect database contents, regardless of the internal representation of said contents. Locks are utilized to separate concurrent transactions. These locks have sophisticated acquiring and releasing mechanisms, usually handled by a lock manager with the support of prioritization and queuing. As these locks ensure the serializability of database contents but not their representation, a B-Tree does not require locks of all non-leaf pages.

In this instance, it must be guaranteed that the operations modifying the data structure in memory are serializable and do not create an invalid or incomplete state of the entire tree, not just its contents. Latches are commonly used for this case, resembling critical sections implemented by mutexes and semaphores. They have the benefit of lower overhead, as these can be implemented with a handful of instructions in comparison to full-fledged lock managers.

More specifically, splitting a full node is a significant challenge for concurrent updates of B-Trees. As one thread is splitting a full B-Tree node, all the other threads must not observe any intermediate or incomplete state. Two latches on different levels must be acquired to atomically serialize changes to the split node, the new sibling of the split node, and the parent of a said split node. Even so, a splitting might propagate towards the root, as a split operation might cause the parent node to subsequently become full, further bottlenecking the concurrency on the entire tree.

3.4 B-Link-Tree

Previous approaches include locking a subtree of highest affected node [26], which, albeit straightforward, reduced concurrency. To alleviate the bottleneck without risking inconsistency, *B-Link-Tree* relaxes the definition of B^+Trees. As explained by Graefe [12]:

Definition 3. B-Link-Tree is a B^+Tree with following properties:

1. Each node x has additional attributes:
   - $x.sibling$, a pointer to a right sibling node at the same level,
   - $x.highkey$, the upper bound of all keys found in the subtree rooted by x (every key found in x is less then $x.highkey$).
2. Does not require locks nor latches for reading.

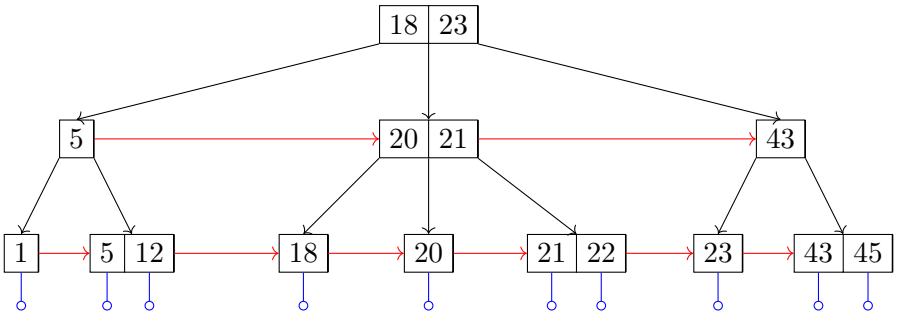

Figure 3.3: B-Link-Tree with $Order = 3$. Blue lines indicate a pointer to a value, red arrows indicate an optional pointer to a sibling node.

An example tree can be seen in fig. 3.3, where it is shown that the idea of a sibling pointer from B^+Tree is extended towards all levels of the tree, not just the leaf nodes.

3.4.1 Insertion and Search

Splitting during node insertion is divided into two independent steps: splitting a node and inserting the split node with its new separator key to the parent node.

Assuming node x is a full node, which needs to be split, shown in step (a) of fig. 3.4. When splitting the node x, a new right sibling node x'' is created, as seen in step (b). The node x'' inherits the high key and the sibling pointer from the split node x, whereas the x node is updated (marked as x') with a new $x.highkey = x''.key_0$ and sibling pointer. Thus, an internal node does exist without a parent in between the operations.

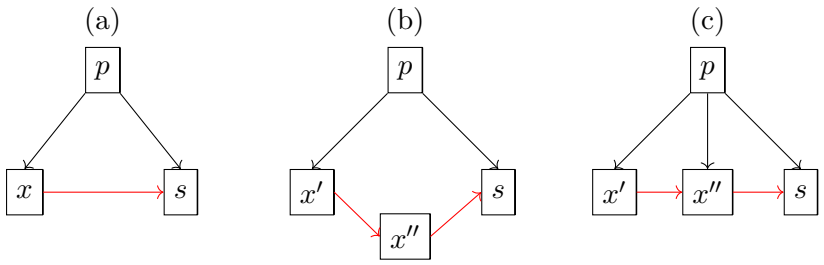

Figure 3.4: Possible states of a B-Link-Tree split operation.

As the final step of node splitting, both the separator key and the pointer to the newly split node y are inserted in the parent node p, seen in step (c). Similar to the insertion in B-Tree, a split operation might trigger additional splitting in higher levels.

Tree traversal is modified to honor $x.highkey$ by returning the node at $x.sibling$ when the target key k is larger or equal to $x.highkey$. With these additional attributes, it is possible to traverse the entire tree, even though some nodes (such as x'' in step (b) of fig. 3.4) are yet to be inserted into the parent node.

3.4.2 Proof of correctness

The following theorems need to be proven to prove the correctness of each operation performed on the B-Link-Tree [16]:

- *Deadlock freedom* — threads performing operations on the B-Link-Tree cannot produce a deadlock.
- *Correct tree modifications* — the tree must appear as a valid tree for all nodes at any time.
- *Correct interactions* — concurrent operations do not interfere with one another.

Theorem 1. *Deadlock freedom*: threads performing operations on the B-Link-Tree cannot produce a deadlock.

Proof. By imposing a total ordering of nodes, cycles are eliminated from the system. Thus the system is deadlock-free.

To prove a total ordering of nodes, the following ordering $a < b$, where a and b are nodes of the tree, is considered:

1. If a and b are not on the same distance from the root node, then $a < b$ if and only if the path from the root node to node a is shorter than the path from the root node to node b (bottom-up condition),

2. if a and b are on the same level, then $a < b$ if and only if node b is reachable from node a by following a chain of one or more *sibling* pointers (left-to-right condition).

It can be shown that during insertion operation the total ordering of nodes is preserved. If $a < b$ at time t_0 at the start of insertion, then $a < b$ is preserved for $\forall t, t > t_0$, as a node x during split operation will create a node x' and x'', where $x' < x''$ and:

$$\forall y, y < x \Leftrightarrow y < x'$$

$$\forall y, x < y \Leftrightarrow x'' < y$$

Therefore, insertion will not break the total ordering of nodes.

Latches for the nodes are acquired by following the ordering; thus, once a latch is acquired for a node, no other latch will be acquired on any node below it, nor on any node on the same level. □

Theorem 2. *Correct tree modifications*: the tree must appear as a valid tree for all nodes at any time.

Proof. The tree must appear as a valid tree for all threads at any time except for the modifying thread. Assuming writing is done by invoking *writeNode* function, which will write to the storage atomically.

The insertion operation, assuming performed on a node x of the tree, will therefore write only in these circumstances:

1. *writeNode*(x) — if node x is not considered full and is safe for rewriting,
2. *writeNode*(x'') — writing a newly allocated node x', which was created in the process of splitting node x,
3. *writeNode*(x') — rewriting the node x as part of the splitting process, $x.sibling$ is set to point at node x'.

Even though *writeNode*(x'') and subsequent *writeNode*(x') operation are done in two writes, it can be shown, that these operations are equivalent to a single change in tree structure:

- When *writeNode*(x'') is executed, no other node has a pointer at x''. Therefore this operation will not modify the tree structure,
- when *writeNode*(x') is subsequently executed, the $x'.sibling$ is set to point to x''. This operation therefore does both modify the node x and introduce newly allocated node x' to the tree in a single operation.

□

Theorem 3. *Correct interactions*: concurrent operations do not interfere with one another.

Proof. Assume t_i is the time when the insertion process I writes node x to storage, and t_r is the time when another operation P reads the same node x. All of the operations are considered atomic, thus $t_r \neq t_i$.

First, the case of $t_r > t_i$ is considered: an operation P is reading a node x after the insertion is done. Any changes that the insertion process I does will preserve the correct tree structure, as proven in theorem 2.

In the case of $t_r < t_i$, where insertion I happens after operation P does a read, the proof is broken into three possible scenarios, which the insertion might perform:

1. Simple insertion of a key-pointer pair into node x without splitting,
2. splitting of node x, where the inserted key is placed in the left node x', same as the node, which has been split ($x' = x$),
3. splitting of node x, where the inserted key is placed in the right node x'', the newly allocated node.

In the first scenario, I performs the insertion in node x without additional splitting if it is inserting into a leaf node ($x.leaf = true$), no other pointers are inserted by the process I. The operation P behaves as if the read happened before insertion.

If the insertion happens on an internal node ($x.leaf = false$), a key-pointer pair created by splitting a lower-level node z'' is inserted into the node x. This scenario is the only one where a key-pointer pair could propagate upwards to node x. The operation P will be able to utilize the link pointers $z.sibling$ to reach both the original node and the newly split node.

In the second and third scenarios, the process I has split the node x into two nodes x' and x''. If the process happens on a leaf node, P will continue as if no insertion has occurred. Similar to the first scenario, the only possible case where the process I needs to split a non-leaf node is when a child node z went through a split and a new separator key and a pointer to z'' is being inserted into the node x.

Both the insertion and search in the node z'' below node x will be correct thanks to theorem 2. It only remains to prove the correctness of split operation on node x.

Assume z'' is the node, which has been created by splitting a node below x and z' the node on the left of node z'' ($z'.sibling = z$). If the search would not follow the pointer of z, the operation P will proceed as usual because the nodes x' and x'' contain the same set of pointers as node x with the addition of the pointer to z''. Otherwise, if the search would follow the pointer to z'', had the process P read the node x after the split, the operation P will proceed

instead to z' instead. The operation P is then able to reach node z'' by using the sibling pointer of node z'.

If the process P is another insertion process, it is either searching for the correct node, backtracking upwards, or attempting to insert into node x. In the case of searching, the proof is the same as if the process P was a search operation.

In the case of backtracking, the node n might have been split multiple times since the last read when traversing downwards in the search operation. In this scenario, the insertion process will find the proper target node for insertion using link pointers, as the order between nodes is preserved, and the newly split nodes of node n will be found on the right side of node n.

Finally, the process P can attempt to insert a key (or a key-pointer pair) into node n. In this scenario, only one process can hold a latch for the node n. After the latch has been acquired by one of the insertion processes, the other will attempt to read the node n afterward. As the read occurs after write, theorem 2 proves this interaction to be correct.

□

Chapter 4

Realisation

In this chapter, implementation details and design choices made to construct a B-Tree implementation on GPU are presented. The entire implementation is written in CUDA C++ with the usage of TNL.

4.1 Warp Cooperative Work Sharing strategy

Warp-cooperative work-sharing strategy (WCWS) is used to improve the performance of each operation by exploiting the behavior of NVIDIA GPUs and how threads are executed in each SM. As SMs have a limited number of cores, threads are organized into groups of 32 threads called *warps* (see section 1.2).

As threads can communicate with each other in a warp using warp-wide communication primitives in CUDA, these threads can synchronize with each other and work on a single task.

An example is shown in listing 1 with entry-point being the `KernelTask` function. Each thread will read a single key based on the thread ID calculated on line 22. A reference to `toFind` boolean flag is passed to `WarpTask`. This flag is used in a `ballot` call on line 9, which returns a bitmap indicating which thread lane in a warp has `toFind` variable set to `true`. A single lane is selected as the main lane whose task will be processed by the entire warp. This lane is selected by calculating the position of the least significant bit set to 1 via the `__ffs` function. An elected key is shared between all warp threads from the selected lane via a synchronization primitive shown in line 11, and the `executeOperation` continues with the key. If the operation succeeds, `toFind` is set to `false`, which will be reflected in the next `ballot` call. The loop will end after all tasks assigned to the warp threads were completed.

This strategy is used in all of the B^+Tree and B-Link-Tree operations, especially during updates. Better memory coalescing and vectorized load and store is achieved by utilizing every thread in a warp for a single task. As a side effect, the *Order* of a B-Tree node must not exceed the number of thread lanes in a warp: 32 in contemporary GPUs.

```
1  template <typename KeyType>
2  __device__ static void WarpTask(
3    Node *root, KeyType key, bool &toFind
4  ) {
5    using cg = cooperative_groups;
6    auto threadBlock = cg::this_thread_block();
7    auto warp = cg::tiled_partition<32>(threadBlock);
8
9    uint32_t queue = 0;
10   while ((queue = warp.ballot(toFind))) {
11     uint32_t currentLane = __ffs(queue) - 1;
12     KeyType electedKey = warp.shfl(key, currentLane);
13
14     if (executeOperation(electedKey)) {
15       toFind = false;
16     }
17   }
18 };
19
20 template <typename KeyType>
21 __global__ static void KernelTask(
22   Node *root, ArrayView<KeyType, Devices::Cuda> keys
23 ) {
24   auto threadId = threadIdx.x + blockIdx.x * blockDim.x;
25   bool toFind = threadId < keys.getSize();
26
27   KeyType key;
28   if (toFind) {
29     key = keys.getElement(threadId);
30   }
31
32   WarpTask<KeyType>(root, key, toFind);
33 };
```

Listing 1: Example code used to implement the Warp Cooperative Work Sharing strategy. `KernelTask` is a kernel function invoked from the CPU, `WarpTask` is a function called from GPU. The `executeOperation` function found in line 13 shall accept a single value for processing.

4.2 Warp-based operations

As discussed in section 4.1, both GPU implementations use the WCWS strategy, and all threads of a warp are cooperating to process a single task. Whether it is an insertion, removal, or query operation, thus, it does make sense to use this implementation detail to speed up commonly used, performance-critical computations.

One such computation is finding the correct key index in a node, whether to find a specific key in a leaf node or to find the right child node to traverse into the correct sub-tree. This computation essentially boils down to finding the index of a lower and upper bound key, the former being the first key in a node that is greater or equal to a searched key, the latter being the first node key greater than a searched key.

Algorithm 4.2.1: Lower bound in sorted list

Input: $keys[0 \dots n], needle$
Result: Position of a lower bound key

```
1  index ← 0, size ← n
2  while size > 0 do
3      step ← size/2
4      it ← index + step
5      if keys[it] < needle then
6          index ← it + 1
7          size ← size − step + 1
8      else
9          size ← step
10 return index
```

On the host implementation, the lower-bound key is found by performing a binary search, as the keys are stored in a sorted manner in each node, thus allowing a search with time complexity $\mathcal{O}(\log_2 n)$. Implementation of a lower bound function can be seen in algorithm 4.2.1. However, as all 32 threads perform a single task in a warp, a binary search is sub-optimal on the GPU, as it will inherently execute in series for the reasons of thread divergence, explained in section 1.2.2.

Thus, it is more efficient to break down the work for every thread to be able to participate, seen in a lower-bound example in listing 2. A lower bound (and upper bound) search is done as follows: each thread in a warp will read a key from a node and perform a comparison. The results are aggregated as a bitmap using `warp.ballot()`, where N-th bit indicates whether a thread at index N fulfills the comparison (see line 9). The position is obtained by invoking `__ffs` on line 10, which returns the index of a first thread with a fulfilled comparison.

```
1  __device__ static uint32_t lowerBound(
2    KeyType *keys, uint32_t size, KeyType needle
3  ) {
4    using cg = cooperative_groups;
5    auto threadBlock = cg::this_thread_block();
6    auto warp = cg::tiled_partition<32>(threadBlock);
7
8    uint64_t rank = warp.thread_rank();
9    uint32_t ballot = warp.ballot(keys[rank] >= needle);
10   uint32_t idx = __ffs(ballot) - 1;
11
12   asm("min.u32 %0, %1, %2;" : "=r"(idx) : "r"(idx), "r"(size));
13   return idx;
14 }
```

Listing 2: A warp-friendly implementation of finding a lower-bound key.

The resulting position must not be greater or equal to the size of a processed node, as reading keys outside the expected node size is undefined behavior. As the data type of a variable storing the position is independent of the used data type for keys or values, a single `asm` statement on line 12 is used to clamp the position. This optimization was done specifically to enforce the compiler to use a single `min.u32` statement rather than generating branch and jump statements, thus reducing the number of cycles spent on these critical sections.

This optimization does bring an additional constraint to the structure of a B-Tree node: the *Order* of a tree must not exceed the number of threads available in a single warp, as if it does, some keys will become inaccessible.

4.3 Proactive Splitting

As specified in operations of section 3.1, to perform an update operation, such as insertion or deletion, on a tree, the target node must be first found by traversing top-down. If the node becomes either full or does not have enough keys, a split or merge operation must occur, which requires propagation upwards and might cascade the changes to the root node. A pointer to a parent node stored in a node or a stack of visited notes must be implemented to have the ability to traverse back to the root, which brings additional unnecessary complexity to the GPU.

Thus, in the implementation found in this thesis, any splitting operations are done proactively, where splits occur before descending further down the tree. This method benefits from not visiting any node twice, guaranteeing every operation to update at most two levels, and preventing any cascading

back to the root. Furthermore, the implementation needs to track only two nodes simultaneously, avoiding implementing an explicit stack on the device or keeping track of the parent node inside the node structure itself.

However, this method does come with the disadvantage of premature splitting, as the node does not know whether either of its descendants will become full, require a split, and cause a cascade back to the node. B-Trees implementing bottom-up insertion could do a smaller amount of splitting in general.

4.4 Latching and Concurrency Control

For our implementation of B-Trees, an additional attribute $x.writelock$ is used to prevent multiple threads from writing to the same node at the same time.

In the B$^+$Tree implementation of insertion, traversal is prevented if encountering a node with a write lock to ensure data serializability, as the node with a latch is most likely not in a correct state and may lead the traversal operation to an incorrect subtree. The search is thus restarted from the start if reaching a node with a latch, sharing the properties of a back-off lock.

B-Link-Tree has this limitation lifted, as the tree is always traversable, even if the node is latched for writing. As described in section 3.4, each node has a $x.sibling$ pointer referencing next at the same depth. At every depth, the nodes are essentially chained in a linked list. With this addition, a correct node is always found with the help of the additional path to reach every node.

In both of the implementations, `warp.sync()` and `__threadfence()` is often needed to ensure correct order of instructions used for memory access. Latch coupling is used to ensure safe upgrading and downgrading of write latches when traversing the tree vertically: a latch of a parent node is preserved until the target child node is successfully latched. If the attempt to latch a child node fails, the parent latch is released.

Latching itself is done by a `BNodeLatch` class. This class does include two separate template specializations, with the CUDA specialization utilizing atomic instructions such as `atomicAdd()` and `atomicCAS()` to avoid serializability issues when acquiring or releasing a latch. Host specialization does not require a critical section, as the implementation runs on a single thread. The class is not explicitly bound to any of the B-Tree implementations, and a template class of a node must be provided when instantiating the latch.

4.5 Bulk Insert

As described by Graefe [12], there is a strong relationship between B-Trees and sorting, which we can utilize. Optimal construction of B-Tree shall forgo incremental insertion in favor of building the tree from a presorted list.

As both B-Link-Tree and B$^+$Tree internally store their key-value pairs in their nodes, incremental insertion can be avoided. Instead, the tree can be

constructed in a bottom-up approach, creating all nodes of each level one by one. Sample pseudocode can be seen in algorithm 4.5.1.

Algorithm 4.5.1: Bulk Insert

Input: *inputKeys*: list of keys to be stored
inputValues: list of values, each one is bound to a key

```
1 (keys, values) ← sortByKey(inputKeys, inputValues)
2 nodeCount ← calculate nodeCount
3 (keys, children) ← createLeafKernel(keys, values)
4 while children.len() > 1 do
5 |  (keys, children) ← createInternalKernel(keys, children)
6 root ← children[0]
```

First, the key-value pairs are sorted by key using the built-in `thrust::sort` function, as TNL does not have a sort method yet. Sorted pairs are passed to the `createLeafKernel`, which will divide the pairs and fit them into nodes. A fixed offset is chosen to make sure each node is not immediately full. The expected node count is then calculated by dividing the number of pairs by the desired node size.

$$nodeCount = \left\lceil \frac{|inputKeys|}{Order - Offset} \right\rceil, \quad Order > 0$$

Each kernel invocation of `createInternalKernel` will return separator keys and pointers of the created nodes. These keys and children nodes are passed back to the kernel to create an additional level until the kernel creates only one node. This node then becomes the root of the tree, as seen at line 6 in algorithm 4.5.1.

A significant additional benefit of constructing the tree in such a manner is the improved memory locality of the tree. The reason is that B-Tree nodes at the same level are created and inserted into memory simultaneously. Thus the nodes reside in the memory block near each other.

4.6 Allocation

As the allocator for the data structure, a bump allocator is used (also known as stack allocator) in both of the implementations. The bump allocator allocates a continuous linear section of memory and works by increasing a pointer at the next unused memory.

To handle the allocation of nodes, a section of memory is allocated beforehand on the CPU. This continuous section of memory can be resized by invoking the `setSize` method, which will allocate a new memory segment and copy the remaining data.

Atomicity is key for successful allocation, as the allocator can be invoked concurrently from different threads. The `allocate` method will move the

```
1 template <typename Object>
2 struct BumpAllocator {
3   int *mOffset;
4   ArrayView<Object, Devices::Cuda> mData;
5
6   __device__ Object *allocate(int size) {
7     return mData.getData() + atomicAdd(mOffset, size);
8   }
9 };
```

Listing 3: Implementation snippet of a bump allocator with an `allocate` method.

memory offset atomically via the `atomicAdd` function. This function will return the previous offset, which is used to obtain the address on the newly available global memory block, as seen on line 7 in listing 3. Atomic instructions are used to ensure the serializability of the operations, as the allocator is invoked concurrently from different threads.

Proper care needs to be taken of the root node and its address when moving the pointers to a different memory location, for example, while resizing the underlying memory block of the allocator. The address of the root node can be trivially updated in this instance by using pointer arithmetic.

The main benefit of the bump allocator is its simplicity and performance. Dynamic global memory allocation inside a device function is considerably slower [30] than the bump allocator. It is also allocated on a size-limited heap. By default, the heap is only 8 MB large and may be resized only once in a during application runtime [18], which is not suitable for storing large amounts of data in global memory.

However, the presented allocator is not as versatile when compared to traditional allocators, as the acquired memory cannot be deallocated because the returned `mOffset` only increases.

4.7 Node structure

This section describes the proposed B$^+$Tree and B-Link-Tree node structure, found in the listing 4. Comments and helpers methods were removed for the sake of brevity.

`uint16_t` have been chosen in favor of smaller data types, as most instructions in the ISA do not support operand types smaller than 16-bits and instead convert them to larger data types via a `cvt` conversion statement [22].

```
1  template <typename KeyType, typename ValueType, size_t Order>
2  struct BPlusNode {
3    uint16_t mLeaf;
4    uint16_t mSize;
5    uint16_t mWriteLock;
6
7    KeyType mKeys[Order];
8    ValueType mValues[Order];
9
10   BNode * mChildren[Order];
11   BNode * mSibling;
12 }
13
14 template <typename KeyType, typename ValueType, size_t Order>
15 struct BLinkNode {
16   uint16_t mLeaf;
17   uint16_t mSize;
18   uint16_t mWriteLock;
19
20   KeyType mHighKey;
21   uint16_t mHighKeyFlag;
22
23   KeyType mKeys[Order];
24   ValueType mValues[Order];
25   volatile BNode * mChildren[Order];
26   volatile BNode * mSibling;
27 }
```

Listing 4: The node structures used in B$^+$Tree and B-Link-Tree.

As seen in the `BLinkNode` variant at line 25–26, `mChildren` and `mSibling` are both using the `volatile` qualifier to avoid incorrect memory access optimization by the compiler. This qualifier tells the compiler to assume that the contents of the variable may be changed or used at any time by another thread. Therefore all references to this variable will compile into an actual memory read or write [18].

Chapter 5

Testing

5.1 Environment

The measurements are captured on the `gp1` system provided by Faculty of Nuclear Sciences and Physical Engineering, Czech Technical University with the following specifications seen in table 5.1:

CPU	Intel Xeon CPU E5-2630 v3 (2.40 GHz)
GPU	NVIDIA Quadro P6000 (24 GB, CUDA 6.1)
RAM	128 GB
Driver	465.31
OS	Arch Linux (kernel 4.12.9, KPTI enabled)
Host compiler	GCC g++ 11.1.0
Device compiler	nvcc v11.3.58

Table 5.1: Hardware and software specification of the `gp1` system.

Furthermore, the optimization flag is set to `-O3`. C++ dialect is set to C++17 when benchmarking against different implementations, although C++14 is supported as well.

5.2 Testing methodology

It is hard to prove the complete absence of errors and bugs of the solution, as the execution of code on the device is not deterministic. Nevertheless, for each of the B-Tree implementations, a set of unit tests is prepared. This unit test suite rigorously examines if each operation preserves the valid tree state, which has proved to be helpful when optimizing the performance of the tree without sacrificing correctness. The unit tests also test the proper usage of templates, verifying the validity of operations when handling non-integer

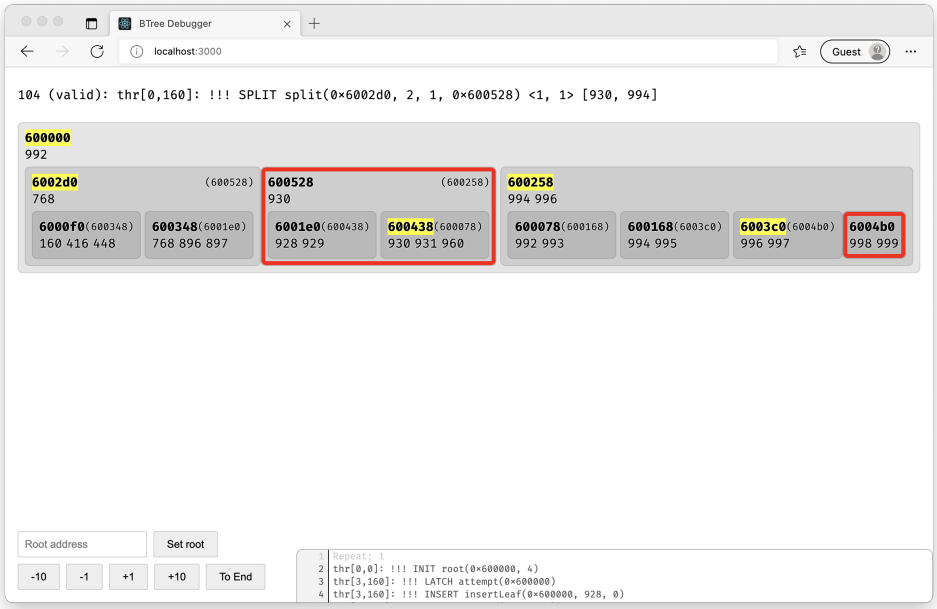

Figure 5.1: Web based visual debugger displaying the internal state of a tree. Red boxes denote split nodes not yet inserted to the parent node. A highlighted node address indicate that a thread has latched that node.

keys or values. GoogleTest framework [11] is the framework of choice used for writing these unit test suites.

For the host implementation, unit tests do include assertions of the internal states of a node to aid the development and catch bugs early on. In comparison, the device implementations rely on the assertions provided by the TNL instead. These assertions were used to detect the same issues covered by GoogleTest unit tests on the host side.

Still, debugging of a kernel is a pain-staking process, even though `cuda-gdb` does work on device code. Pinpointing race conditions can be difficult, especially without any prior context, and any potential issues might arise only when dealing with a large number of items in B-Tree. To better understand the behavior of GPU B-Tree operations and help find concurrency issues easier, a complimentary web-based visual debugger has been developed in React [8], seen in fig. 5.1.

Macro functions have been inserted through the implementations, which are enabled by defining a `DEBUGGER` flag in the source code. With this flag, the implementation will emit debugging commands into the standard output, which can be inserted into the web app.

In the case of multiple B-Tree variants, to avoid duplicating tests and ensure the correctness of both implementations, typed tests are used to repeat the same logic over a list of types [10], as seen in listing 5. The `typename` of

```
template <typename C> class TestSuite : public testing::Test {
public:
  using Implementation = C;
};

using TypeList = ::testing::Types<A, B>;
TYPED_TEST_SUITE(TestSuite, TypeList);

TYPED_TEST(TestSuite, Test) {
  typename TestFixture::Implementation impl;
  // ... rest of test implementation
}
```

Listing 5: Snippet of a GoogleTest typed test

```
using DeviceType = Benchmark::Device::Host;
using NumericType = uint32_t;
Benchmark::execute<DeviceType, NumericType>(
  "test", [](Benchmark::BenchTimer<DeviceType> &timer,
             const std::vector<NumericType> &input) {
    timer.start();
    // insert operation
    timer.stop("insert");

    timer.start();
    // query operation
    timer.stop("query");
  });
```

Listing 6: Sample usage of the benchmark methods found in `benchmark.hpp`.

each class will be available as a static field within `TestFixture` itself.

5.3 Benchmarking methodology

Companion benchmarking methods have been created to provide a standard interface across different implementations of our choice. A tested solution is wrapped in a lambda function, accepting a timer instance and an input sequence as its arguments. This lambda function is invoked ten times, and an arithmetic mean of captured results is computed and stored on a disk. An example code can be seen in listing 6 and the source code of the benchmarking methods can be found in `/benchmark/_common/benchmark.hpp`.

The `Benchmark::BenchTimer` class will capture the time spent between

the invocations of `start()` and `stop()`. This class does support multiple timers within a single instance by specifying a key string in the stop function, as seen in lines 8 and 12 of listing 6. Internally, CPU-based implementations are measured using `std::chrono::high_resolution_clock`, whereas GPU-based solutions will use the built-in CUDA runtime API, which is more precise, as it does not include the time spent on potential synchronization.

In most scenarios, the time spent on copying the data between the CPU and GPU were excluded in the actual measurement.

5.3.1 Datasets used in benchmarking

Different data sequences were generated as the input data for benchmarking created implementations and existing solutions. Assuming n is the expected size of generated data sequence, these datasets were used:

- *Ascending* – sorted sequence $0..(n-1)$,
- *Descending* – inverse of ascending dataset, a sorted sequence $(n-1)..0$,
- *Almost sorted* – sorted sequence $0..(n-1)$ with 5 random swaps,
- *Shuffle* – generated sequence shuffled with `std::shuffle`,
- *Gaussian* – a sequence generated by producing n random values around mean with standard deviation.

For all of the measurements done in section 5.4, 32-bit values are used both as keys and values, as some implementations chosen in section 5.3.2 do not function properly when different data types are utilized. For this reason, no benchmark tests utilizing non-integer keys or values required for comparison on datasets from unstructured numerical meshes are presented here. An example of such a test implemented for TNL based implementations can be found in the source code (`/benchmark/tnl/tnl_mesh_cuda.cu`).

5.3.2 Chosen implementations for comparison

As for the implementations themselves, the following projects were chosen for the benchmarks:

- *OWG* (owensgroup/GpuBTree) – implementation of a GPU based B-Link-Tree by Awad et al. [2],
- *PALM* (runshenzhu/palmtree) – an implementation of PALM Tree [27] by Xian et al. [33]; a concurrent lock-free B^+Tree scaling up to 16 cores,
- *TLX* – a collection of C++ data structures, algorithms and helpers by Bingman et al. [5].

`std::map` is chosen as the baseline when calculating the performance speedup. Internally, `std::map` tends to be implemented as a self-balanced red-black tree [7].

Additional implementations, such as the Bw-Tree implementation by Wang et al. [31], or STX B$^+$Tree by Timo Bingman [4], were considered as well. But either due to memory leak issues or deprecation by the author, these solutions were not included in the comparison. Nevertheless, the obtained results can be found in the attached medium.

As the OWG implementation is the only readily available GPU implementation of the B-Tree, additional work has been done to resolve issues encountered while benchmarking. Patches were created for the OWG implementation to mitigate memory leaks. These patches can be found in the benchmarking source code and will be automatically applied when CMake is invoked.

5.4 Results

Chosen implementations described in section 5.3.2 were measured against different sequences mentioned in section 5.3.1.

For the speedup comparison, `std::map` is chosen as a baseline. 32-bit key-value pairs sequences are generated as the input data. First column indicate the size of the sequence used. Second column contains the raw execution time spent by `std::map` on a benchmark test. Remaining speedup values are presented as the ratio of the execution time spent by `std::map` to the execution time spent by a tested implementation:

$$Speedup = S = \frac{\tau_{CPU}}{\tau_{GPU}}$$

5.4.1 Query performance

In table 5.2 the query speedup on *shuffled* sequence is shown. For smaller sequences, `std::map` tends to be faster than other compared implementations. However, as the input size increases, the solutions utilizing parallelization overtake the baseline implementation.

As soon as the input size is larger than 2^{10}, both the TNL$^+$ and TNL^{link} becomes faster than the baseline `std::map`. A notable observation can be made when comparing TNL$^+$ (implementation of B$^+$Tree) with TNL^{link} (implementation of B-Link-Tree). Even though the concurrency control is better in TNL^{link} than in TNL$^+$, the reduced instruction count and reduced pointer chasing through sibling links overshadow the performance benefits of such concurrency control, resulting in similar speedup on *shuffled* and *gaussian* input sequence.

Different behavior can be observed in table 5.3, where the query speedup is shown on *ascending* input sequence. The performance of the CPU implemen-

2^x	STL $\tau[ms]$	TNL$^+$ S	TNL^{link} S	OWG S	PALM S	TLX S	TNL^{host} S
10	0.055	0.730	0.814	1.383	1.767	0.804	0.588
11	0.133	1.416	1.583	2.722	1.512	0.861	0.615
12	0.288	3.024	3.383	5.825	2.035	0.798	0.615
13	0.742	7.703	8.363	14.415	3.904	0.964	0.737
14	1.901	11.457	16.556	30.823	4.980	1.111	0.785
15	4.632	21.204	21.189	36.124	5.800	1.162	0.891
16	10.918	29.534	25.694	44.915	7.080	1.200	0.955
17	26.074	32.771	33.447	61.745	8.694	1.260	0.999
18	62.571	34.897	38.197	78.075	10.333	1.337	0.978
19	169.116	49.068	51.854	105.834	13.923	1.505	1.113
20	480.989	121.280	74.452	153.818	19.921	1.562	1.083
21	1217.690	108.249	91.548	192.396	25.343	1.617	1.269
22	2989.450	97.953	112.298	230.330	31.187	1.709	1.475
23	6975.730	187.227	138.958	×	36.500	1.702	1.476
24	16584.300	155.728	160.673	×	40.687	1.789	1.476
25	43311.200	226.363	201.184	×	53.206	2.089	2.129

Table 5.2: Search speed-up of chosen implementations compared to `std::map` for various input sizes. *Shuffled* sequence is used as input.

2^x	STL $\tau[ms]$	TNL$^+$ S	TNL^{link} S	OWG S	PALM S	TLX S	TNL^{host} S
10	0.045	0.582	0.668	1.075	1.827	0.839	0.678
11	0.097	1.029	1.134	1.906	1.178	0.885	0.705
12	0.199	2.068	2.324	4.048	1.574	0.793	0.728
13	0.406	3.787	4.588	7.920	2.049	0.784	0.712
14	0.842	4.201	6.787	13.267	2.158	0.803	0.751
15	1.879	7.654	10.023	14.823	2.531	0.883	0.836
16	4.321	9.529	10.164	17.443	2.856	0.957	0.890
17	10.829	10.986	11.581	23.388	3.556	1.130	1.080
18	26.453	15.397	15.930	30.847	4.378	1.324	1.269
19	60.116	17.887	18.116	35.975	4.915	1.415	1.448
20	128.294	15.199	17.847	35.448	5.349	1.504	1.509
21	271.900	18.845	19.115	38.846	5.673	1.529	1.511
22	602.152	23.065	20.274	43.738	6.318	1.669	1.522
23	1308.570	21.677	20.167	×	6.905	1.739	1.797
24	2921.260	23.215	23.499	×	7.346	1.874	2.144
25	6591.670	25.359	26.415	×	8.272	2.102	2.336

Table 5.3: Search speed-up of chosen implementations compared to `std::map` for various input sizes. *Ascending* sequence is used as input.

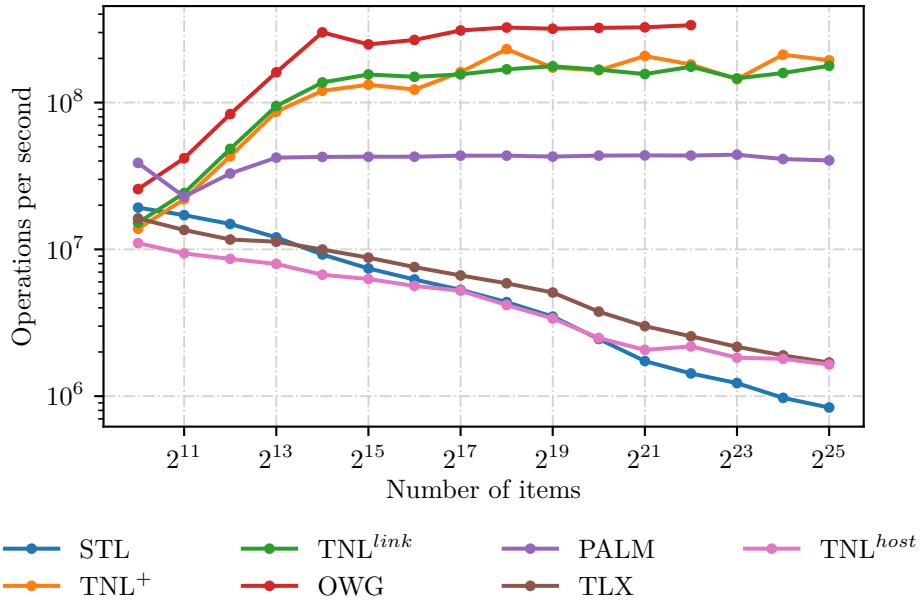

Figure 5.2: Graph comparing number of search queries per second between chosen implementations for various input sizes. *Gaussian* distribution is used as input.

tation has significantly improved, thus reducing the measured query speedup on the GPU implementations.

In all input distributions, the OWG implementation of GPU B-Tree performs 2× to 3× faster than both the TNL^{link} and TNL^{+}, regardless of input size, which can be seen in fig. 5.2. Internally, OWG utilizes explicit memory read and write operations with hand-crafted `asm` statements, where each thread in a warp reads a 32-bit word from global memory instead of relying on implicit read instructions generated by the compiler. This approach yields better memory utilization in exchange for the loss of generality. On a related note, even with applied patches, the OWG implementation suffers from frequent crashes caused by invalid memory accesses and deadlocks for input sequences larger than 2^{22}. Thus the OWG results are missing for these input sizes.

5.4.2 Insertion performance

In table 5.4 it can be shown that `std::map` is generally not effective even on a single thread, as both TLX and TNL^{host} overtake the baseline when inserting more than 2^{15} items.

One notable observation can be gathered from the results of the PALM tree, as the insertion time is comparably the same as the querying execution

2^x	STL $\tau[ms]$	TNL$^+$ S	TNL^{link} S	OWG S	PALM S	TLX S	TNL^{host} S
10	0.083	0.139	0.207	0.398	3.193	0.914	0.693
11	0.185	0.279	0.397	0.772	2.099	0.950	0.704
12	0.395	0.492	0.752	1.433	3.505	0.943	0.706
13	0.903	1.028	1.485	2.487	3.541	1.005	0.768
14	2.183	1.989	2.679	3.763	4.968	1.111	0.838
15	5.312	3.041	3.894	4.000	6.237	1.229	0.962
16	12.429	4.596	6.102	4.818	7.921	1.294	1.012
17	29.280	8.119	10.864	9.469	9.349	1.367	1.071
18	67.215	12.981	15.803	19.933	10.853	1.384	1.099
19	168.193	19.835	22.629	38.064	13.497	1.511	1.185
20	461.217	31.699	33.518	65.485	18.864	1.751	1.171
21	1222.530	42.197	44.516	99.201	25.044	1.897	1.381
22	3054.340	51.295	54.066	134.295	31.361	1.956	1.564
23	7299.760	61.146	63.606	×	37.566	1.956	1.589
24	17217.300	69.850	72.824	×	41.691	2.009	1.631
25	44176.200	84.574	89.070	×	53.398	2.286	2.243

Table 5.4: Insertion speed-up of chosen implementations compared to `std::map` for various input sizes. *Shuffled* sequence is used as input.

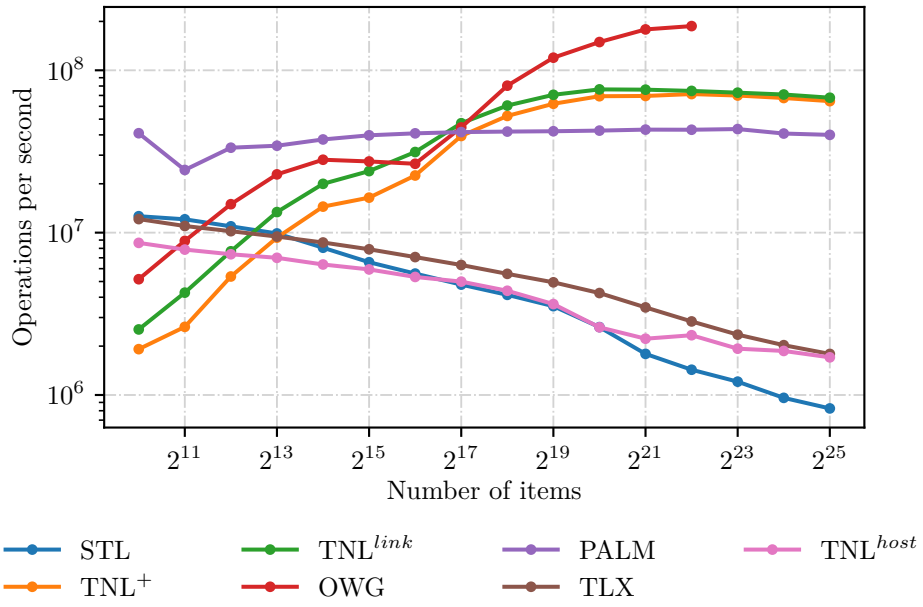

Figure 5.3: Graph comparing the number of inserted keys-value pairs per second between chosen implementations for various input sizes. *Gaussian* distribution is used as input.

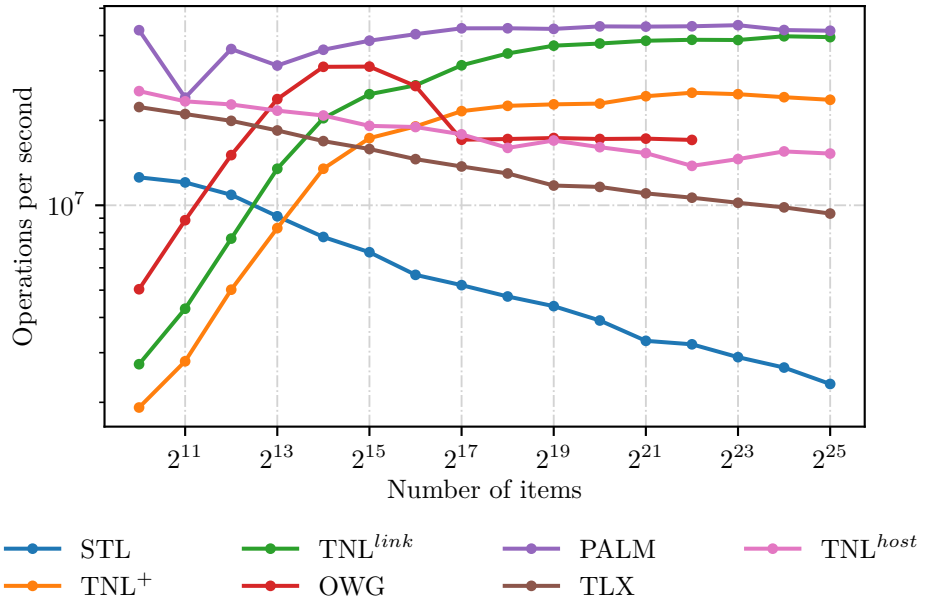

Figure 5.4: Graph comparing the number of inserted keys-value pairs per second between chosen implementations for various input sizes. *Increasing* sequence is used as input.

time, a result more in line with other CPU implementations. All GPU-based implementations are slower than PALM when inserting less than 2^{17} items. This performance is consistent regardless of the distribution of incoming sequence, as seen in figs. 5.3 and 5.4, where PALM is the most performant implementation in the scenario of *increasing* input sequence.

The performance difference between OWG and TNL implementations is similar compared to the previous query benchmarks with *shuffled* input sequence, where TNL structures are generally $\approx 3\times$ slower than OWG. When inserting key-value pairs in a sorted manner, as seen in fig. 5.4, OWG becomes slower than both of the TNL implementations when inserting 2^{17} items and more.

Conclusion

Goals and results

The goal of this body of work was to study and understand the B-Tree data structure with all of its variants and introduce an implementation of a B-Tree data structure capable of execution on the GPU.

Core concepts behind the CUDA programming model were studied and explained; how it allows the programmer to execute GPU code from the CPU, how threads are organized and assigned to each CUDA execution core, and how a programmer can synchronize threads on the GPU. An introduction to the Template Numerical Library has been made to understand how template metaprogramming can provide a unified interface for different execution environments.

Two variants of B-Tree were introduced and described: B$^+$Tree and B-Link-Tree. Both of the variants were appropriately modified to support concurrent updates and reads. All of the implementations were written in C++ with the help of the TNL library.

The principles behind the Warp Cooperative Work Sharing strategy were outlined, and different warp-friendly optimizations were introduced to utilize the entire warp to accelerate a single operation, avoiding unnecessary warp divergence and improving performance. Proactive splitting has been utilized to avoid implementing a stack for keeping track of parent nodes, and various concurrency control methods used in both of the variants were discussed.

Multiple GoogleTest based unit test suites were written to ensure proper functionality. A complementary web tool has been developed to visualize operations done in a multithreaded environment.

Finally, both of the implementations were measured and compared against STL and other state-of-the-art GPU and CPU B-Tree implementations. Measurements of both variants against `std::map` show a substantial speedup: $\approx 200\times$ when querying and $\approx 85\times$ when inserting random sequences of length 2^{24}–2^{25}. When compared to the GPU B-Tree implementation by Awad et al.

[2], both of the variants implemented in this thesis are slower. It should be noted that both the B$^+$Tree and B-Link-Tree perform similarly, suggesting the reduced instruction count and reduced unnecessary pointer chasing in B$^+$Tree overshadow the concurrency improvements of B-Link-Tree.

Future work

The implementation presented in this thesis, as shown in section 5.4, is $\approx 3\times$ slower than the solution from Awad et al. [2], even though it is not certain their implementation perform correctly on large input sequences. Further work needs to be done to improve memory access patterns. Another point of improvement can be the implementation of range queries.

Bibliography

1. ASHKIANI, Saman; FARACH-COLTON, Martin; OWENS, John D. A dynamic hash table for the GPU. In: *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 2018, pp. 419–429.
2. AWAD, Muhammad A. et al. Engineering a High-Performance GPU B-Tree. In: *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*. Washington, District of Columbia: Association for Computing Machinery, 2019, pp. 145–157. PPoPP '19. ISBN 9781450362252. Available from DOI: 10.1145/3293883.3295706.
3. BAYER, R.; MCCREIGHT, E. Organization and Maintenance of Large Ordered Indices. In: *Proceedings of the 1970 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control*. Houston, Texas: Association for Computing Machinery, 1970, pp. 107–141. SIGFIDET '70. ISBN 9781450379410. Available from DOI: 10.1145/1734663.1734671.
4. BINGMANN, Timo. *STX B+ Tree C++ Template Classes v0.9* [comp. software]. [N.d.] [visited on 2021-06-24]. Available from: https://github.com/bingmann/stx-btree.
5. BINGMANN, Timo. *TLX: Collection of Sophisticated C++ Data Structures, Algorithms, and Miscellaneous Helpers* [comp. software]. 2018 [visited on 2021-06-24]. Available from: https://panthema.net/tlx.
6. COMER, Douglas. Ubiquitous B-Tree. *ACM Comput. Surv.* 1979, vol. 11, no. 2, pp. 121–137. ISSN 0360-0300. Available from DOI: 10.1145/356770.356776.
7. CPPREFERENCE.COM. *std::map* [online]. [N.d.] [visited on 2021-06-24]. Available from: https://en.cppreference.com/w/cpp/container/map.

8. FACEBOOK INC. *React: A JavaScript library for building user interfaces* [comp. software]. [N.d.] [visited on 2021-04-27]. Available from: `https://reactjs.org/`.

9. FIX, Jordan; WILKES, Andrew; SKADRON, Kevin. Accelerating Brai ded B+ Tree Searches on a GPU with CUDA. In: *2nd Workshop on Applications for Multi and Many Core Processors: Analysis, Implementation, and Performance (A4MMC), in conjunction with ISCA*. 2011.

10. GOOGLE LLC. *Advanced googletest Topics* [comp. software]. [N.d.] [visited on 2021-04-27]. Available from: `https://github.com/google/googletest/blob/master/docs/advanced.md`.

11. GOOGLE LLC. *GoogleTest* [comp. software] [visited on 2021-04-27]. Available from: `https://github.com/google/googletest`.

12. GRAEFE, Goetz. Modern B-Tree Techniques. *Found. Trends Databases.* 2011, vol. 3, no. 4, pp. 203–402. ISSN 1931-7883. Available from DOI: `10.1561/1900000028`.

13. GUPTA, H.; HARINARAYAN, V.; RAJARAMAN, A.; ULLMAN, J.D. Index selection for OLAP. In: *Proceedings 13th International Conference on Data Engineering.* 1997, pp. 208–219. Available from DOI: `10.1109/ICDE.1997.581755`.

14. KACZMARSKI, Krzysztof. B^+-tree optimized for GPGPU. In: *OTM Confederated International Conferences On the Move to Meaningful Internet Systems.* 2012, pp. 843–854.

15. KIM, Changkyu et al. Designing Fast Architecture-Sensitive Tree Search on Modern Multicore/Many-Core Processors. *ACM Trans. Database Syst.* 2011, vol. 36, no. 4. ISSN 0362-5915. Available from DOI: `10.1145/2043652.2043655`.

16. LEHMAN, Philip L.; YAO, s. Bing. Efficient Locking for Concurrent Operations on B-Trees. *ACM Trans. Database Syst.* 1981, vol. 6, no. 4, pp. 650–670. ISSN 0362-5915. Available from DOI: `10.1145/319628.319663`.

17. LEVANDOSKI, Justin J.; LOMET, David B.; SENGUPTA, Sudipta. The Bw-Tree: A B-Tree for New Hardware Platforms. In: *Proceedings of the 2013 IEEE International Conference on Data Engineering (ICDE 2013).* USA: IEEE Computer Society, 2013, pp. 302–313. ICDE '13. ISBN 9781467349093. Available from DOI: `10.1109/ICDE.2013.6544834`.

18. NVIDIA CORPORATION. *CUDA C++ Programming Guide* [online]. [N.d.] [visited on 2021-04-05]. Available from: `https://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf`.

19. NVIDIA CORPORATION. *cuDNN Developer Guide* [online] [visited on 2021-04-30]. Available from: `https://docs.nvidia.com/deeplearning/cudnn/developer-guide/index.html`.

20. NVIDIA CORPORATION. *cuSOLVER* [online] [visited on 2021-04-30]. Available from: `https://docs.nvidia.com/cuda/cusolver/index.html`.

21. NVIDIA CORPORATION. *NVIDIA Ampere GA102 GPU Architecture* [online]. [N.d.] [visited on 2021-05-02]. Available from: `https://www.nvidia.com/content/PDF/nvidia-ampere-ga-102-gpu-architecture-whitepaper-v2.pdf`.

22. NVIDIA CORPORATION. *Parallel Thread Execution ISA* [online]. [N.d.] [visited on 2021-06-27]. Available from: `https://docs.nvidia.com/cuda/pdf/ptx_isa_7.3.pdf`.

23. OBERHUBER, Tomáš; KLINKOVSKÝ, Jakub; WODECKI, Aleš. *PostgreSQL: Documentation: 13: Implementation* [comp. software]. [N.d.] [visited on 2021-04-27]. Available from: `https://www.postgresql.org/docs/13/btree-implementation.html`.

24. OBERHUBER, Tomáš; KLINKOVSKÝ, Jakub; WODECKI, Aleš. *Template Numerical Library* [comp. software]. [N.d.] [visited on 2021-04-27]. Available from: `https://tnl-project.org/`.

25. RODEH, Ohad; BACIK, Josef; MASON, Chris. BTRFS: The Linux B-Tree Filesystem. *ACM Trans. Storage*. 2013, vol. 9, no. 3. ISSN 1553-3077. Available from DOI: `10.1145/2501620.2501623`.

26. SAMADI, Behrokh. B-trees in a system with multiple users. *Information Processing Letters*. 1976, vol. 5, no. 4, pp. 107–112.

27. SEWALL, Jason et al. PALM: Parallel Architecture-Friendly Latch-Free Modifications to B+ Trees on Many-Core Processors. *Proc. VLDB Endow.* 2011, vol. 4, no. 11, pp. 795–806. ISSN 2150-8097. Available from DOI: `10.14778/3402707.3402719`.

28. SHAHVARANI, Amirhesam; JACOBSEN, Hans-Arno. 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*. San Francisco, California, USA: Association for Computing Machinery, 2016, pp. 1523–1538. SIGMOD '16. ISBN 9781450335317. Available from DOI: `10.1145/2882903.2882918`.

29. SHIMPI, Anand Lal; WILSON, Derek. NVIDIA's GeForce 8800 (G80): GPUs Re-architected for DirectX 10. *Anandtech* [online]. 2006 [visited on 2021-04-27]. Available from: `https://www.anandtech.com/show/2116/8`.

30. VINKLER, Marek; HAVRAN, Vlastimil. Register efficient dynamic memory allocator for GPUs. In: *Computer Graphics Forum*. 2015, vol. 34, pp. 143–154. No. 8.

31. WANG, Ziqi; ZHU, Runshen. *Open BwTree: An open sourced implementation of Bw-Tree in SQL Server Hekaton* [online]. [N.d.] [visited on 2021-06-24]. Available from: `https://github.com/wangziqi2013/BwTree`.

32. WANG, Ziqi et al. Building a Bw-Tree Takes More Than Just Buzz Words. In: *Proceedings of the 2018 International Conference on Management of Data*. Houston, TX, USA: Association for Computing Machinery, 2018, pp. 473–488. SIGMOD '18. ISBN 9781450347037. Available from DOI: `10.1145/3183713.3196895`.

33. XIAN, Ran; ZHU, Runshen. *An implementation of Intel's concurrent B+Tree (Palm Tree)* [comp. software]. [N.d.] [visited on 2021-06-24]. Available from: `https://github.com/runshenzhu/palmtree`.

34. YAN, Zhaofeng; LIN, Yuzhe; PENG, Lu; ZHANG, Weihua. Harmonia: A High Throughput B+tree for GPUs. In: *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*. Washington, District of Columbia: Association for Computing Machinery, 2019, pp. 133–144. PPoPP '19. ISBN 9781450362252. Available from DOI: `10.1145/3293883.3295704`.

Appendix A

Acronyms

CPU Central Processing Unit.

CUDA Compute Unified Device Architecture.

GPU Graphical Processing Unit.

ISA Instruction Set Architecture.

SIMT Single Instruction, Multiple Threads.

SM Streaming Multiprocessor.

TNL Template Numerical Library.

Appendix B

Contents of enclosed SD card

```
README.md ..... description of content
benchmark ..... directory of the benchmarking suite
benchmark-data ..... captured benchmark data
debugger ..... sources of the complementary B-Tree debugger
implementation ..... implementation sources
    scripts ..... hotfix scripts for editor support
    src ..... source code of the implementation
    test ..... integration and unit tests
```