GPU Acceleration of B-Trees

Wes Robbins & Rishi Borkar

December 15, 2021

Abstract

Graphics Processing Units (GPUs) have several magnitudes more cores than a CPU. This allows significant acceleration on parallelizable workloads. In this project, we do a deep dive into optimizing GPU performance for data structures and find there are many factors that can degrade performance — warp divergence, memory latency, load imbalance, among others. Special care must be taken to mitigate these factors for any specific use case. We focus on the B-Tree data structure, which is common for database indexing. We survey the related literature to understand current GPU B-Tree methods. Then, we implement a read-capable GPU B-Tree. Experimentally, we find that our GPU B-Tree implementation outperforms our analogous CPU implementation by a factor of 300. While exploring GPU architecture and implementing a GPU B-Tree, we gain meaningful experience with the CUDA programming language. Finally, we explore another use case that may be able to benefit from B-Tree-like data structures on a GPU — Verkle and Merkle trees.

1 Introduction

The focus of our project was understanding and implementing GPU B-trees. There were several introductory steps in this project that were either helpful or necessary for our final B-tree implementation. These introductory topics included understanding GPU history, understanding the GPU architecture, learning challenges of GPU programming, and learning the CUDA programming language. We provide a subsection for each of the topics in the remainder of the introduction. In the second section, we survey GPU B-tree implementations that have been proposed in the literature. In the third section, we discuss our own implementation of a GPU B-tree. The fourth section overview results for our experiments, where we compared execution times between queries on a GPU and queries on a CPU. The fifth section is dedicated to a discussion on Merkle and Verkle trees, and how a GPU implementation could be approached. In the conclusion, we reflect on our multifaceted project with a discussion on the outcome of the project.

History of GPUs

Graphics processing units (GPUs) are designed for handling expensive tasks such as graphics rendering. GPUs can accelerate such tasks by parallelizing computation. GPUs are designed with many more cores than which allows greater parallelization (a standard CPU has 4–8 cores, whereas a mid-range GPU has 1,000s).

The original design of GPUs was motivated by the idea that expensive graphics rendering could be offloaded from the central processing unit to another processor, thus allowing the central

processor to focus on general tasks, such as task scheduling or handling of IO. Although the term 'GPU' was not coined until 1994, the first GPUs were created in the 1970s. Since their inception, GPUs have become ubiquitous for graphics processing. They are found in laptops, desktops, and gaming consoles. The success of GPUs for graphics processing has motivated the use of GPUs in other contexts.

A general-purpose graphics processing unit (GPGPU) was introduced in the early 2000s as a GPU programmed for general-purpose computing, not graphics processing and rendering. The introduction of GPGPUs was motivated by the observation that many general computation tasks can be parallelized. Early adoptions GPGPUs were scientific computing and AI. During the 2000s, several GPU programming APIs were built making GPU accelerated computation more accessible to scientists and developers. These APIs include OpenGL, DirectX, and CUDA. As the use of GPGPUs became more common, Nvidia began developing and marketing its hardware for general-purpose graphics processing, cementing the fate of GPGPUs as an important part of modern computing. In the last decade, GPUs continue to be adopted for different use cases. Notably, modern deep learning methods run exclusively on GPUs. It is even considered that the reason deep learning is successful is for the exact reason that deep learning parallelizes well on GPUs, the processors with the highest throughput today [1]. Another recent use case of a GPGPU is in databases. A GPU database uses GPU computation power to analyze massive amounts of information and return results in milliseconds via GPU co-processing. However, using graphics cards to accelerate data processing is tricky and has both its advantages and disadvantages. He et al. observed that joins are 2-7 times faster on the GPU, whereas selections are 2-4 times slower, due to the required data transfers [2]. The same conclusion was made by Gregg et al., who showed that a GPU algorithm is not necessarily faster than its CPU counterpart, due to the expensive data transfers [3]. One major point for achieving good performance in a GDBMS is therefore to avoid data transfers where possible. Several start-up companies that offer GPU-based databases as a service have launched over the past years such as Brytlyt, Kinetica & MapD (now OmniSciDB). These commercial services mainly focus on data analytics and visualization as their core offering. In 2013, a white paper for OmniDB is released which proposed an optimized GDBMS solution though an advanced memory management and three-tier caching mechanism that minimizes data transfer inefficiency & bottlenecks. [4].

GPU Architecture

GPU architecture can be broken into several levels of hardware hierarchy. A *core* is the atomic level of processing. A GPU core can run a serial stream of instructions. However, A GPU core is more similar to an arithmetic logic unit (ALU) than a CPU and does not support full instruction sets such as X86. Groups of 32 cores are called warps (the small green grids in Figure 1). Each warp has a single instruction cache (also known as the L0 cache). In each clock cycle, all cores run the same instruction. An exception to this is when a core is waiting on operands that have not arrived at the warp. This creates warp divergence, which we discuss in the next subsection. This paradigm of execution is called same instruction multiple data (SIMD). SIMD warps are part of what allows so many cores to be put on a GPU die. However, there are also many limitations of SIMD. Groups of warps (usually 4) each reside under a single streaming multiprocessor (SM). SMs and are responsible for scheduling all subordinate warps. Additionally, groups of two SMs create texture processing clusters, and groups eight texture processing clusters create GPU processing clusters [5].

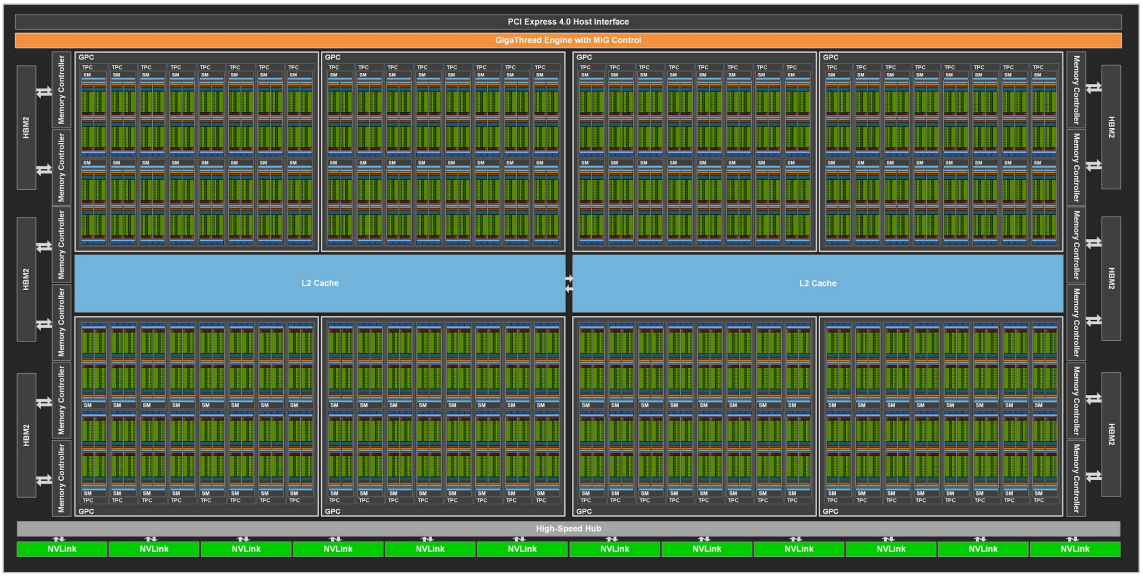

Figure 1: Nvidia Ampere Architecture diagram.[1]

A corresponding memory hierarchy exists for the GPU. In Figure 1, it can be seen that device memory is not on the GPU Die. Instead, it is accessed using memory controllers which can be seen on the sides of the diagram. The relationship between the GPU and GPU ram is the same as that of the CPU and CPU ram. The next level down in the memory hierarchy is the L2 cache. The L2 cache is globally accessible to the device. However, the access time is much faster. It is also smaller than the main memory. The next level is L1 cache. There is an L1 cache local to each SM. Finally, each warp has a set of registers and an L0 cache for storing instructions. The common size of a warp register is 64KB.

For example, the Nvidia RTX 3090 released last September has a retail price of $1500 and has 10496 cores, 6,144kB of L2 cache, a streaming multiprocessor count of 82 with 128 KB of L1 cache per SM. The RTX 3090 belongs to Nvidia's Ampere microarchitecture class. Ampere released in 2020, aims to further enable high-performance computing (HPC) and AI use cases. Enhancements in Ampere including 3rd generation NVLink and Tensor cores, structural sparsity (the conversion of unneeded parameters to zeros to enable AI model training), 2nd generation ray-tracing cores, multi-instance GPU (MIG) to enable partitioning of A100 GPUs into individual logically isolated and secure GPU instances.

Programming Frameworks

Two major frameworks are used for programming GPUs, namely the Compute Unified Device Architecture (CUDA), the Open Compute Language (OpenCL) and OpenACC. All of these frameworks implement the kernel programming model and provide APIs that allow the host CPU to manage computations on the GPU and data transfers between CPU and GPU. In contrast to CUDA, which

[1]Image from https://www.nextplatform.com/2020/05/28/diving-deep-into-the-nvidia-ampere-gpu-architecture/.

supports NVIDIA GPUs only, OpenCL can run on a wide variety of devices from multiple vendors. However, CUDA offers advanced features such as allocation of device memory inside a running kernel or Uniform Virtual Addressing (UVA), a technique where CPUs and GPUs share the same virtual address space and the CUDA driver transfers data between CPU and GPU transparently to the application. OpenACC is the youngest programming standard for parallel computing and was initially released in 2015 by a group of companies comprising Cray, CAPS, Nvidia, and PGI to simplify parallel programming of heterogeneous CPU/GPU systems. The choice between these three parallel computing platforms depends on your goals and the environment you work in. CUDA is widely used in academia, and it's also considered to be the easiest one to learn. OpenCL is by far the most portable parallel computing platform, although programs written in OpenCL still need to be individually optimized for each target platform.

Challenges

IO Bottlenecks. A standard GPU usually has only around 24GB of memory. Due to this limitation, data has to be constantly transferred over from the CPU or disk to the GPU for processing. A GPU faces two major IO bottlenecks. The first is the classical disk IO, and the second bottleneck is the PCIe bus. GPU-accelerated operators are of little use for disk-based systems, where most time is spent on disk I/O. Since the GPU improves performance only once the data is in main memory, time savings will be small due to a large amount of time spent on data transfer from disk/CPU to GPU memory.

Performance. Deciding the optimal processing device for a given operation is a difficult task. GPUs are well suited for easily parallelizable operations (e.g., predicate evaluation, arithmetic operations), while the CPU is the vastly better fit when it comes to operations that require complex control structures or significant inter-thread communications (e.g., hash table creation or complex user-defined functions). Selecting the optimal device for a given operation is a non-trivial operation, and - due to the large parameter space, applying simple heuristics is typically insufficient. Breβ et al. argue that there are four major factors that need to be considered for such a decision (1) the operation to execute, (2) the features of the input data (e.g., data size, data type, operation selectivity, data skew), (3) the computational power and capabilities of the processing devices (e.g., number of cores, memory bandwidth, clock rate), and (4) the load on the processing device (e.g., even if an operation is typically faster on the GPU, one should use the CPU when the GPU is overloaded) [6]. Therefore, we argue that a complex decision model, that incorporates these four factors is needed to decide on an optimal operator placement.

2 GPU B-Tree Background

As early as the mid-2000s, GPU B-tree implementations were discussed in the literature. Many of the aforementioned B-Trees were implemented as cache-sensitive search trees(CSS-trees), which have the principle property of being static [7]. In 2011, Fix et al. first propose using *braided parallelism*, running independent tasks in parallel, to permit multiple read queries to execute at the same time [8]. Kaczmarski was an early work to focus on improving insertion times and found that bulk inserts improved upon previously unsatisfactory results [9].

The first fully dynamic indexing data structure on a GPU was a log-structured merge (LSM) tree which demonstrated satisfactory performance on read and write queries. Recently, Awad et al.

propose a dynamic GPU B-Tree [10]. This method stands out from previous approaches in that all modifications are completed on GPU. Additionally, they demonstrate improved performance on almost all operations compared to the previously mentioned LSM tree. To the best of our understanding, the results in this paper demonstrate the best performance and trade-offs of proposed GPU B-Tree implementations. Awad et al. discuss several design decisions that are critical to their approach. First, they leverage the SIMD architecture by using a method called *Warp Cooperative Work Sharing* (WCWS). WCWS prevents warp divergence (discussed in the Introduction) by having a single warp handle one query at a time (compared to each core handling a single query). Next, they use a B-link-tree, which has a pointer between all sibling nodes – not just leafs, to reduce which portions of a tree need to be locked. See Figure 2 for an example of how B-link-trees improve upon vanilla B-trees for parallel read and insert queries. Additionally, Awad et al. use restarts more often than spinlocks (previous work used spinlocks). Restarts involve restarting the search from either a parent or root node when a lock is encountered, whereas a spinlock waits for the Awad et al. demonstrate improved experimental results by using restarts in most cases[2]. The combination of these factors allows for parallelization even with high rates of insertions or deletions.

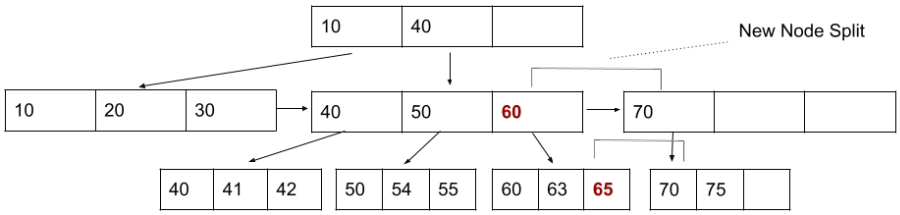

Figure 2: A B-link-tree allows us to lock only the child node sub-tree during a split, not the parent node sub-tree. Consider the scenario where an insertion into the root of the tree has caused a higher level split at a non-leaf. During a node split, there exists a point in execution such that the node has split, but the parent node has not been updated. If a read query is happening simultaneously, the stale reference in the parent node can cause the query to be incorrect. Traditionally, to prevent this invalid read, the entire sub-tree is locked. However, with a link between siblings, this is unnecessary. In the above diagram, consider a query for the key 75 during the insertion process of new item 65. Normally, the link from 60 would be followed. However, we can instead follow the side-link and drop to the next level via the link from 70. It can be noticed that the leaf-level link between 65 and 70 could have been followed to get to 75. However, if there is a split high up the tree, following leaf-links becomes intractable to traverse significant portions of the list of the linked-list of leaves. Additionally, other intermediate sub-tree locks may block traversal of the leaf node linked-list.

3 Implementation

We implement a read only GPU B-tree in CUDA. We make use of three primary CUDA API calls in our implementation. First, we use `cudaMalloc` to allocate memory on the device. The second API call is `cudaMemCopy` to copy the data to the device. Third, we use a kernel call with the chevron notation to denote the number of threads and blocks – `<<<blocks,threads>>>`. Last,

[2]The conditions for restarting vs. spinlock is nuanced and outside the scope of this description. More information can be found in Section 3.5 and Algorithm 2 in [10].

we use the `cudaDeviceSynchronize` call to ensure all queries have completed before moving on to validating the queries. When launching the kernel, we use the maximum number of threads per block (1024) and then as many blocks as is necessary to accommodate the number of queries.

4 Results

# of Queries	GPU Wall Time (ms)	CPU Wall Time (ms)	Relative Acceleration
10^0	0.042	.005	0.12
10^1	0.065	.031	0.48
10^2	0.076	.249	3.27
10^3	0.124	1.889	15.23
10^4	0.132	14.701	111.36
10^5	0.422	122.149	289.45
10^6	3.761	1179.370	313.66
10^7	36.048	11494.600	319.27
10^8	356.72	111026.132	311.87

Table 1: Execution time comparison between CPU and GPU.

In our experiments, we compare our B-Tree GPU implementation to an analogous CPU implementation. The CPU implementation uses the same underlying structure as the GPU implementation. All experiments are run on a Tesla P100 (2017 model) with 16 gigabytes of memory and 3,564 cores.

We create a synthetic index of data of by creating an ordered array of keys and a corresponding array values. Elements in the value array are generated randomly. The values simulate a pointer to an address on disk, which may be entirely independent of the key.

All results in Table 1 are reported on an index over 100 millions elements. For our experiment, we run queries over the range 1–10^8. We generate queries by randomly selecting a value with in the range of total number of values. For n queries, an additional n-sized array is allocated for the query responses. We are able to check the correctness of our implementation by comparing the query array to the expected values (which are stored in the `values` array). We collect wall times with a the `cudaEventStream` timer[3] for both the CPU and the GPU. Each row in Table 1 represents a different number of queries. The column on the right of Table 1 presents the relative acceleration from the GPU, calculated by dividing the CPU time by GPU time.

From Table 1, we can see that for few queries, the CPU outperforms the GPU, and for many queries (≥ 100) the GPU outperforms the CPU. Our main result is that we can achieve up to an approximately 300 factor speed up by using a GPU for B-Tree reads. We also find that the near-maximal acceleration can is reached around 10^5 queries on the P100. For greater than 10^5 queries, the factor of acceleration begins to plateau. Additionally, we find that the time to transfer the data from host to device is 188.34 milliseconds (averaged over 10 runs). By comparing to CPU times in the table, it can be seen that this transfer time non-negligible for $< 10^6$ queries. As discussed earlier, data transfer can become a bottleneck if the frequent data transfer is required.

[3]CUDA Library timer that gives wall time independent of which device is running. Docs found at https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__EVENT.html.

We also point that the CPU is timed on a completely serial implementation and could be optimized with a multi-threaded and multi-core implementation. However, our GPU code is not perfectly optimized. Considering these factors, we believe 300 factor acceleration is a reasonable assessment of a single GPU performance for the described task.

5 Verkle & Merkle Trees

A stretch goal for this project was to adopt an implementation of a GPU B-Tree to a Verkle or Merkle tree. A GPU implementation of these has the potential to accelerate computation of the corresponding Verkle or Merkle digests. While we did not complete these implementations during the semester, we researched this topic alongside other aspects of our project.

What are Verkle and Merkle Trees?

A Merkle tree, also known as a hash tree, is a method for crytographically hashing a dataset. Each leaf node in a Merkle tree stores the hash of an individual data item and a hash of a concatenation of hash values of the child nodes is saved in the successive parent nodes. This is repeated until we receive a root hash (also called a digest) of the tree. The digest is the only hash that is needed to be stored by any verifier in order to verify the authenticity of the data received by it which it does using the Merkle Proofs provided by the provider [11].

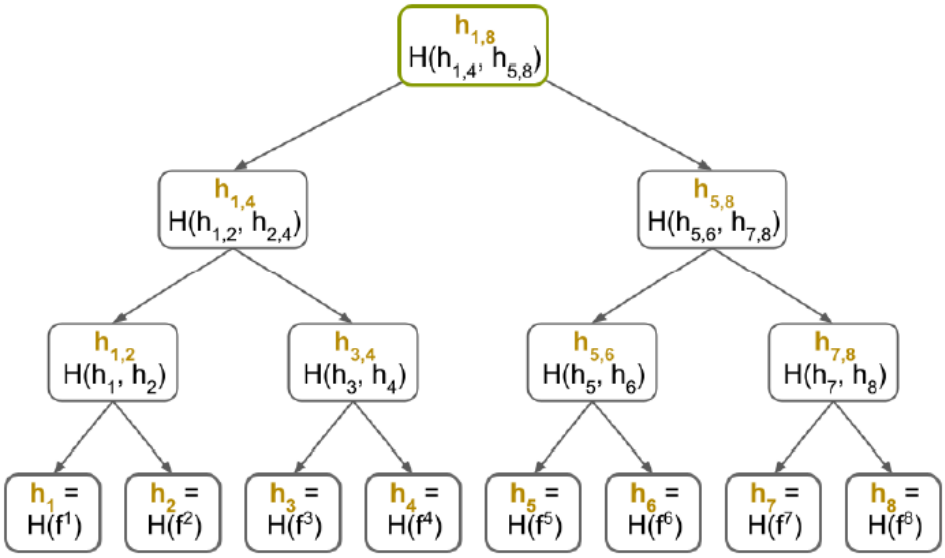

Figure 3: Merkle Tree [4]

Verkle Trees combine techniques used in Merkle Tree as well as Vector Commitment Schemes to provide a tradeoff between bandwidth and computational power. A Verkle Tree is basically a

[4]Image from https://decentralizedthoughts.github.io/2020-12-22-what-is-a-merkle-tree/.

Merkle Tree in which cryptographic hash functions are replaced with Vector Commitments.The use of Vector Commitments helps it to reduce the proof size from $O(klog_k n)$ for a k-ary Merkle Tree to $O(log_k n)$ for a k-ary Verkle Tree. [12]

Vector Commitments

Along with handling the structure of the Verkle tree, an implementation also assumes the responsibility of computing the vector commitments at each step. Vector commitment computations are non-trivial and generally require numbers larger than 64 bits — one Go language implementation used 256 bit ints[5]. We provide in-depth description of an RSA-based realization of vector commitments to allow for a discussion on GPU implementation[6].

There are two steps to calculate a vector commitment. First, a key is generated that satisfies the RSA assumption for each element in the commitment. Second, a digest is calculated by multiplying over the exponentiation of each element's key, to the power of the element [13].

More formally, let q be the length of the vector of elements and choose a hash function H that maps the index to a prime number. Calculate a prime-index e for each element as follows: $e_i = H(i) \mid i \in \{1..q\}$. Let $= p_1 \cdot p_2$ where p_1, p_2 are large primes. Chose a generator g from a hidden-group that is the RSA problem is hard under. Calculate keys as follows

$$S_i = g^{\prod_{j \in \{1..q\}, j \neq i} {e_j}} \mid i \in \{1..q\}. \tag{1}$$

Let $m_i..m_q$ be the elements of the original vector. The vector commitment C is the set from the following exponentiations:

$$C \leftarrow S_i^{m_i} \mid i \in \{1..q\}. \tag{2}$$

There are also two steps to verify if an element existed in the original vector. The vector commitment is *opened* as follows:

$$\Lambda_i = \left(\sqrt[e_i]{\prod_{j=1, j \neq i}^{q} S_j^{(1/e_i)^{m_j}}} \right) mod\ N. \tag{3}$$

Element m can then be verified to be the i_{th} element in the original vector if the following equality holds:

$$C == S_i^m \cdot \Lambda_i^{e_i}. \tag{4}$$

On a GPU? The greatest GPU acceleration would be realized by paralleling the computation of a single vector commitment and parallelizing several vector commitments at once as the Verkle tree digest is calculated. However, as discussed earlier, even if we are just able to do the latter the acceleration still might be worthwhile. Given that we are satisfied running a single vector commitment in serial, it still leaves with the question of how to approach implementing on a GPU.

To do an implementation, we need to know the size of the numbers we are working with. The calculation for the number of bits needed is dominated by the size of the numbers in the exponents

[5] https://github.com/lunfardo314/verkle

[6] Our vector commitment description is adopted from the original vector commitment paper by Dario Catalano and Dario Fiore [13] and a blog post by Alin Tomescu at https://alinush.github.io/2020/11/24/Catalano-Fiore-Vector-Commitments.html

— i.e the generator g, prime-index e, and element m. We can approximate the number of bits needed as

$$b_g \cdot b_e \cdot (b_m / b_e) = b_g \cdot b_m \tag{5}$$

where b is the number of bits needed to represent a certain value. If both m and g are standard 4-byte integers, we need to store approximately 1,024 bit numbers or 4,096 bits if doubles. Our gut feeling is that this can't even be done as a proof of concept with only 64-bit values. g and m would both need to be 8 bits and the root factor of e and the mod N would fall apart for values exclusively in the range 0 to 64.

So, we know that if we want to do vector commitments we need big numbers. What's the support like on GPUs? In general, not great. Several works have done POC GPU cryptography implementations, however many of them are for older architectures. We were not able to find a well-maintained Big Num library.

Several works have looked into accelerating and RSA encryption on GPUs. [14] focus on the large exponentiation-mod[7] and successfully implement the operation on a GPU. The takeaways for their experimental results are similar to ours – GPU slower on single large exponent-mod calculation, faster on many. However, most cryptography-on-GPU papers note that the GPU implementations our more susceptible to side-channel attacks (hardware exploitation) than CPUs.

Our conclusion for a potential Verkle tree GPU implementation is as follows. It is certainly possible given the big number operations are properly handled. Some acceleration would certainly by observed for in-memory Verkle trees. Still, several primary barriers could prevent this from being practical in a real-world system. Namely, 1) acceleration needs to compensate for memory transfer bottleneck, 2) someone needs to develop a maintained GPU BigNum library, and 3) GPUs are currently more susceptible to side-channel attacks.

6 Conclusion

Overall, this has been interesting project. We have researched GPU architecture and learned the basics of the CUDA programming language. During the project, we became familiar with state-of-the-art approaches and implemented a GPU B-tree from scratch. In our experiments, we find our GPU B-tree runs 300 times faster than a similar CPU implementation for read queries. Additionally, we learned of Merkle tree, Verkle tree, and Vector commitments.

References

[1] Sara Hooker. The hardware lottery. *Communications of the ACM*, 64(12):58–65, 2021.

[2] Bingsheng He, Ke Yang, Rui Fang, Mian Lu, Naga Govindaraju, Qiong Luo, and Pedro Sander. Relational joins on graphics processors. *Proceedings of the 2008 ACM SIGMOD international conference on Management of data - SIGMOD 08*, 2008.

[3] Chris Gregg and Kim Hazelwood. Where is the data? why you cannot debate cpu vs. gpu performance without the answer. *(Ieee Ispass) Ieee International Symposium On Performance Analysis Of Systems And Software*, 2011.

[7] Exponent operation immediately follows by a modular operation.

[4] Shuhao Zhang, Jiong He, Bingsheng He, and Mian Lu. Omnidb: Towards portable and efficient query processing on parallel cpu/gpu architectures. *Proc. VLDB Endow.*, 6(12):1374–1377, aug 2013.

[5] Timothy Morgan. Diving deep into the nvidia amphere architecture, 2020.

[6] Sebastian Breß, Felix Beier, Hannes Rauhe, Kai-Uwe Sattler, Eike Schallehn, and Gunter Saake. Efficient co-processor utilization in database query processing. *Information Systems*, 38(8):1084–1096, 2013.

[7] Rui Fang, Bingsheng He, Mian Lu, Ke Yang, Naga K. Govindaraju, Qiong Luo, and Pedro V. Sander. Gpuqp: Query co-processing using graphics processors. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*, SIGMOD '07, page 1061–1063, New York, NY, USA, 2007. Association for Computing Machinery.

[8] Jordan Fix, Andrew Wilkes, and Kevin Skadron. Accelerating braided 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.

[9] Krzysztof Kaczmarski. B + -tree optimized for gpgpu. In *On the Move to Meaningful Internet Systems: OTM 2012*, pages 843–854, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[10] Muhammad A Awad, Saman Ashkiani, Rob Johnson, Martín Farach-Colton, and John D Owens. Engineering a high-performance gpu b-tree. In *Proceedings of the 24th symposium on principles and practice of parallel programming*, pages 145–157, 2019.

[11] Alin Tomescu. What is a merkle tree?

[12] John Kuszmaul. Verkle trees.

[13] Dario Catalano and Dario Fiore. Vector commitments and their applications. In *International Workshop on Public Key Cryptography*, pages 55–72. Springer, 2013.

[14] Eduardo Ochoa-Jiménez, Luis Rivera-Zamarripa, Nareli Cruz-Cortés, and Francisco Rodríguez-Henríquez. Implementation of rsa signatures on gpu and cpu architectures. *IEEE Access*, 8:9928–9941, 2020.