International Journal of Artificial Intelligence, Data Science, and Machine Learning
Grace Horizon Publication | Volume 3, Issue 1, 10-17, 2022
ISSN: 3050-9262 | https://doi.org/10.63282/30509262/IJAIDSML-V3I1P102

Original Article

High-Throughput Data Structures for GPU-Accelerated Computing

Karthik Reddy
Cloud Solutions Architect, Oracle, UAE

***Abstract** - The increasing demand for high-performance computing in fields like scientific simulations, machine learning, and data analysis has driven the adoption of Graphics Processing Units (GPUs) as accelerators. GPUs offer massive parallelism, but effectively harnessing their power requires careful consideration of data structures. Traditional CPU-centric data structures often become bottlenecks when deployed in GPU environments due to memory access patterns and synchronization overhead. This paper explores the landscape of high-throughput data structures specifically designed for GPU-accelerated computing. We discuss key considerations for GPU data structure design, including memory layout, access patterns, concurrency management, and data transfer strategies. We then delve into specific data structures optimized for GPU execution, such as array of structures vs. structure of arrays, sparse matrix formats, tree-based structures (e.g., B-trees), and hash tables. We analyze their performance characteristics, trade-offs, and suitability for different application domains. Finally, we present case studies demonstrating the effectiveness of these data structures in real-world GPU-accelerated applications and discuss future research directions in this critical area.*

***Keywords** - GPU Computing, High-Throughput Data Structures, Parallel Computing, Memory Access Patterns, Sparse Matrix, Tree Structures, Hash Tables, CUDA, OpenCL.*

1. Introduction

Modern computing systems rely increasingly on heterogeneous architectures, with GPUs playing a significant role in accelerating computationally intensive workloads. GPUs are designed with a massively parallel architecture, featuring thousands of cores that can execute instructions concurrently. However, the performance benefits of GPUs are not automatically realized. Effective utilization requires careful consideration of the algorithm design, memory access patterns, and, most crucially, the choice of appropriate data structures that are optimized for the GPU's architecture.

Traditional CPU-based data structures often fall short when ported to GPUs. This is primarily due to the architectural differences between CPUs and GPUs. CPUs are optimized for low-latency access to relatively small amounts of memory with complex control flow. GPUs, on the other hand, are designed for high-throughput access to large amounts of memory with simpler control flow. Naive porting of CPU data structures can lead to performance bottlenecks due to:

- **Poor Memory Coalescing:** GPUs benefit from coalesced memory accesses, where threads in a warp (a group of 32 threads in NVIDIA GPUs) access contiguous memory locations simultaneously. CPU data structures may not be laid out in memory to facilitate coalesced access.
- **Synchronization Overhead:** Synchronization mechanisms like locks and mutexes, commonly used in CPU multi-threading, can introduce significant overhead on GPUs due to the large number of threads.
- **Data Transfer Bottlenecks:** Moving data between the CPU and GPU memory is a costly operation. Minimizing data transfers is crucial for achieving good performance.
- **Limited Memory Capacity:** GPU memory is typically smaller than CPU memory, requiring careful memory management and potentially limiting the size of the data structures that can be hosted on the GPU.

This paper aims to provide a comprehensive overview of high-throughput data structures specifically designed for GPU-accelerated computing. The goal is to equip researchers and practitioners with the knowledge needed to select and implement appropriate data structures for their GPU-accelerated applications. We will explore different types of data structures, their trade-offs, and their suitability for various application domains.

2. Key Considerations for GPU Data Structure Design

Designing efficient data structures for GPUs requires a deep understanding of memory architecture, concurrency, data transfer strategies, and memory management. Optimizing these aspects is crucial for maximizing performance in GPU computing.

2.1. Memory Layout and Access Patterns

One of the most important aspects of GPU performance is ensuring efficient memory access. Coalesced memory access plays a significant role, as it allows threads within a warp to access contiguous memory locations. This minimizes the number of memory transactions and reduces latency. Poorly organized data can lead to scattered memory accesses, resulting in lower performance due to additional memory fetch operations.

Another important factor is padding and alignment. GPUs have specific memory alignment requirements, and failing to align data properly can lead to inefficient memory accesses. Adding padding can help ensure that data is correctly aligned, reducing access overhead. When organizing data, the choice between Array of Structures (AoS) and Structure of Arrays (SoA) can greatly affect performance. In AoS, each element is a structure containing multiple fields, whereas in SoA, each field is stored in a separate array. SoA is generally preferred in GPU computing as it enables coalesced access to individual fields, improving performance. In some scenarios, strided memory access patterns may be necessary, where threads access memory locations at fixed intervals rather than sequentially. While this can lead to inefficient memory transactions, understanding and optimizing stride patterns can mitigate performance penalties.

2.2. Concurrency Management

GPUs are highly parallel processors, making concurrency management a key aspect of data structure design. Minimizing synchronization is essential, as excessive use of locks and mutexes can create bottlenecks. Instead, atomic operations can sometimes serve as a more efficient alternative for handling shared data. Lock-free data structures are particularly beneficial in GPU computing, as they allow multiple threads to operate concurrently without explicit synchronization mechanisms. These structures rely on atomic operations to ensure data consistency while avoiding the performance overhead associated with locks. Effective workload partitioning further enhances performance by distributing computational tasks in a way that minimizes contention for shared resources. Proper task assignment reduces the need for synchronization and maximizes parallel efficiency.

2.3. Data Transfer Strategies

Since data movement between the CPU and GPU is a slow operation, minimizing data transfers is crucial for performance optimization. Data should be transferred as infrequently as possible to reduce the impact of memory transfer latency. One technique to improve transfer speeds is using pinned memory, which ensures that data resides in page-locked memory and can be transferred more efficiently. Asynchronous data transfers allow the CPU and GPU to work in parallel by overlapping computation with memory transfers. This can hide the latency of data transfers and improve overall execution time. By leveraging APIs that support asynchronous transfers, developers can ensure that the GPU remains busy with computation while data movement is ongoing. In some cases, zero-copy data transfer can eliminate the need for explicit data movement. Shared memory between the CPU and GPU allows both to access the same data without copying it. However, this approach requires careful management to prevent synchronization issues and ensure efficient access patterns.

2.4. Memory Management

GPUs feature different types of memory, each with unique characteristics. Global memory is the largest and most commonly used memory space, but it is relatively slow compared to other types. Shared memory, on the other hand, is a small but fast memory space accessible by all threads within a block. It is particularly useful for caching frequently accessed data, reducing the need to fetch data from slower global memory. Constant memory is a read-only memory space that is cached on the GPU, making it ideal for storing data that remains unchanged throughout kernel execution. Similarly, texture memory is optimized for spatial locality and is commonly used in image processing applications. Efficient memory allocation and deallocation are critical for preventing fragmentation and improving performance. Poor memory management can lead to inefficient resource utilization and increased execution time. Custom memory allocators can be implemented to optimize memory handling for specific data structures, ensuring that memory is efficiently utilized throughout the execution of GPU programs.

3. GPU-Optimized Data Structures: A Deep Dive

This section examines specific data structures optimized for efficient execution on GPUs.

3.1 Array of Structures (AoS) vs. Structure of Arrays (SoA):

As discussed earlier, AoS and SoA represent fundamental data organization choices. Consider a scenario where we need to store data about a set of particles, each having properties like position (x, y, z) and velocity (vx, vy, vz).

AoS: An array of structures would store each particle's data contiguously:

```
struct Particle {
  float x, y, z;
  float vx, vy, vz;
};
```

```
Particle particles[NUM_PARTICLES]; // Array of Particle structures
```

SoA: A structure of arrays would store each property in a separate array:

```
struct ParticleData {
   float x[NUM_PARTICLES];
   float y[NUM_PARTICLES];
   float z[NUM_PARTICLES];
   float vx[NUM_PARTICLES];
   float vy[NUM_PARTICLES];
   float vz[NUM_PARTICLES];
};
ParticleData particle_data;
```

The SoA layout is generally preferred for GPU computing for several reasons:

- **Coalesced Access:** If a kernel primarily accesses only the 'x' coordinates of the particles, the SoA layout allows for coalesced access to particle_data.x, leading to significantly better performance than accessing the 'x' coordinate from the AoS layout.
- **SIMD Execution:** SoA allows the GPU's SIMD units (Single Instruction, Multiple Data) to operate on contiguous data elements, further increasing throughput.
- **Reduced Memory Transactions:** When only a subset of the particle properties is needed, SoA avoids fetching the unnecessary data associated with the other properties, reducing memory traffic.

Table 1: Comparison of AoS and SoA

Feature	**Array of Structures (AoS)**	**Structure of Arrays (SoA)**
Memory Layout	Contiguous elements with all fields	Separate arrays for each field
Access Pattern	Accessing all fields of one element	Accessing a single field of all elements
Coalesced Access	Difficult to achieve	Easier to achieve
SIMD Efficiency	Lower	Higher
Memory Traffic	Higher if only some fields are needed	Lower if only some fields are needed
Use Cases	CPU-centric workloads, where accessing all fields of a single element is common	GPU-centric workloads, where accessing the same field of many elements is common

3.2 Sparse Matrix Formats

Sparse matrices, where most elements are zero, are common in scientific computing and machine learning. Storing sparse matrices in a dense format is inefficient in terms of memory usage and computation. Several sparse matrix formats have been developed for GPUs, each with its own advantages and disadvantages.

Compressed Sparse Row (CSR): CSR is a widely used format for sparse matrices. It stores the non-zero elements in a single array (values), the column indices of the non-zero elements in another array (col_indices), and the starting indices of each row in a third array (row_pointers).

```
float values[NNZ];       // Non-zero values
int   col_indices[NNZ];   // Column indices of non-zero values
int   row_pointers[N+1];  // Starting index of each row in values and col_indices
```

Here, NNZ is the number of non-zero elements, and N is the number of rows. CSR is efficient for row-wise operations, such as matrix-vector multiplication, and offers good data locality.

Algorithm 1: Sparse Matrix-Vector Multiplication (CSR)

```
Input: CSR matrix (values, col_indices, row_pointers), vector x, vector y
Output: Vector y = A * x

for i = 0 to N-1 do
   y[i] = 0
   for j = row_pointers[i] to row_pointers[i+1]-1 do
      col = col_indices[j]
      y[i] = y[i] + values[j] * x[col]
   end for
end for
```

- **Compressed Sparse Column (CSC):** CSC is similar to CSR, but it stores the matrix column-wise instead of row-wise. It is efficient for column-wise operations.
- **Coordinate (COO):** COO stores each non-zero element as a tuple (row, column, value). It is simple to construct but less efficient for computation than CSR or CSC.
- **ELLPACK/ITPACK:** ELLPACK/ITPACK is a row-oriented format that stores the non-zero elements of each row in a fixed-size array. It is efficient for matrices with a relatively uniform number of non-zero elements per row.

Table 2: Comparison of Sparse Matrix Formats for GPUs

Format	Storage Complexity	Memory Access Pattern	Operations Optimized For	When to Use
CSR	O(NNZ + N)	Row-wise, Coalesced	SpMV (y = A*x)	Row-major matrices, SpMV dominated workloads
CSC	O(NNZ + M)	Column-wise, Coalesced	SpMV (x = A'*y)	Column-major matrices, SpMV dominated workloads
COO	O(3*NNZ)	Random	Matrix Construction	Matrix assembly, not efficient for computation
ELLPACK	O(N * K)	Stride	SpMV (y = A*x)	Matrices with a relatively uniform number of non-zeros per row
Hybrid (e.g., CSR + ELLPACK)	Varies	Varies	Varies	Combine strengths of different formats for irregular sparsity structure

(Where N is the number of rows, M is the number of columns, NNZ is the number of non-zero elements, and K is the maximum number of non-zeros in any row.) The choice of sparse matrix format depends on the specific application and the characteristics of the sparse matrix. For example, CSR is a good choice for matrix-vector multiplication if the matrix is stored in row-major order.

3.3 Tree-Based Structures:

Tree-based structures are used in a wide variety of applications, including search, sorting, and spatial indexing. However, their inherent pointer-based nature poses challenges for GPU implementation due to irregular memory access patterns and synchronization requirements.

- **B-Trees:** B-trees are balanced tree structures that are optimized for disk-based storage. They can also be used on GPUs, but require careful consideration of memory layout and concurrency. The key is to minimize pointer chasing and maximize data locality. Some techniques include:
    - **Wide Nodes:** Using wider nodes (containing more keys) can reduce the height of the tree and minimize the number of memory accesses.
    - **Node Packing:** Packing multiple nodes into a single memory block can improve data locality.

  - **Bulk Updates:** Performing updates in batches can amortize the cost of synchronization.
- **Binary Trees:** Binary trees can be represented using arrays, eliminating the need for explicit pointers. This can improve memory access patterns. For instance, a complete binary tree of height *h* can be stored in an array of size $2^{h+1} - 1$. The root is stored at index 1, and the left and right children of a node at index *i* are stored at indices 2*i* and 2*i*+1, respectively. This implicit representation is often used in heap-based algorithms on GPUs.
- **Quadtrees/Octrees:** Quadtrees (in 2D) and Octrees (in 3D) are hierarchical data structures used for spatial indexing. They recursively divide space into quadrants (2D) or octants (3D). GPU implementations often use a linear representation of the tree, such as a Morton order, to improve memory access patterns.

Table 3: Comparison of Tree-Based Structures for GPUs

Structure	**Memory Access Pattern**	**Concurrency Challenges**	**Use Cases**	**Optimization Considerations**
B-Tree	Irregular, Pointer-based	High synchronization needs	Database indexing, Search	Wide nodes, Node packing, Bulk updates
Binary Tree	More Regular (Array Representation)	Moderate	Heap-based algorithms, Sorting	Array representation, Implicit indexing
Quadtree/Octree	Regular (Linear representation)	Moderate	Spatial indexing, Collision detection	Morton order, Linear representation

3.4 Hash Tables

Hash tables are a fundamental data structure for storing and retrieving data based on a key. They present challenges for GPU implementation due to collisions and concurrency.

- **Collision Resolution:** Common collision resolution techniques include:
  - **Separate Chaining:** Each bucket in the hash table points to a linked list of elements that hash to the same bucket. This is generally not well-suited for GPUs due to the pointer-based nature of linked lists.
  - **Linear Probing:** When a collision occurs, the algorithm probes the next available bucket in the table. This can lead to clustering, but is more amenable to GPU implementation.
  - **Cuckoo Hashing:** Uses multiple hash functions and moves elements around to resolve collisions. Can provide good performance on GPUs with careful implementation.
- **Concurrency:** Concurrent access to hash tables can be managed using:
  - **Atomic Operations:** Atomic operations can be used to update the hash table in a thread-safe manner.
  - **Lock-Free Techniques:** Lock-free hash tables allow multiple threads to access and modify the table concurrently without the need for explicit locks.
  - **Fine-Grained Locking:** Dividing the hash table into smaller buckets and using locks at the bucket level can reduce contention.

Algorithm 2: Linear Probing Hash Table Insert (GPU)

```
Input: Hash table (table), key, value
Output: Inserted (key, value) into hash table
index = hash(key) % TABLE_SIZE

while True do
  current_key = atomicCAS(&table[index].key, EMPTY, key) //Atomic Compare-and-Swap
  if current_key == EMPTY then
    table[index].value = value;
    return;
  else if current_key == key then
```

```
        table[index].value = value; //Update Existing
        return;
    else
        index = (index + 1) % TABLE_SIZE; //Linear Probe
    end if
end while
```
(Note: EMPTY represents an empty slot in the table. atomicCAS is an atomic compare-and-swap operation.)

Table 4: Comparison of Hash Table Techniques for GPUs

Technique	Collision Resolution	Concurrency Management	Memory Access Pattern	Advantages	Disadvantages
Separate Chaining	Linked Lists	Difficult	Irregular	Simple to implement	Poor memory locality, unsuitable for GPUs
Linear Probing	Linear Probing	Atomic Operations	Stride	Better memory locality than chaining	Clustering, performance degrades with load
Cuckoo Hashing	Multiple Hash Functions	Atomic Operations	Potentially Irregular	Potentially high performance	Complex implementation

4. Case Studies

Several case studies highlight how GPU-optimized data structures enhance performance in real-world applications. These optimizations are crucial for achieving high efficiency in scientific computing, machine learning, data analysis, image processing, and computer graphics.

- **Scientific Computing:** In fields such as molecular dynamics, fluid dynamics simulations, and finite element analysis, sparse matrices are widely used to represent complex mathematical models. Since many of the elements in these matrices are zero, storing them in a dense format would waste memory and computation power. Instead, compressed storage formats like Compressed Sparse Row (CSR) and Compressed Sparse Column (CSC) are used to efficiently represent sparse matrices. Optimizing matrix-vector multiplication on GPUs, often through warp-based parallelization and shared memory usage, significantly accelerates these simulations. By leveraging GPU acceleration, researchers can perform large-scale simulations much faster than with traditional CPU-based methods.
- **Machine Learning:** Deep learning and machine learning workloads require processing vast amounts of data, making efficient memory access and computation essential. Structuring data in a Structure of Arrays (SoA) format rather than an Array of Structures (AoS) improves memory coalescence, leading to faster data retrieval and processing. Additionally, GPU-optimized matrix operations, such as those provided by cuBLAS and cuDNN libraries, significantly boost training and inference performance. In tasks like natural language processing (NLP) and recommendation systems, embedding layers are often implemented using hash tables. Using GPU-accelerated hash tables allows efficient handling of categorical data and large vocabulary sizes, ensuring faster lookup operations during training and inference.
- **Data Analysis:** Many data analysis tasks, such as data mining and graph processing, involve handling massive datasets that require efficient storage and retrieval mechanisms. Traditional CPU-based approaches may struggle with performance bottlenecks when processing large-scale graphs, such as social networks or web graphs. GPU-optimized data structures, such as hash tables and adjacency lists, can significantly speed up operations like searching, filtering, and relational joins. By leveraging parallelism, GPUs can process millions of records or graph edges simultaneously, enabling real-time or near-real-time analysis. This is particularly useful in applications like fraud detection, social network analysis, and large-scale database querying.
- **Image Processing:** Image processing algorithms often involve structured access patterns, where pixels are read and modified in a predefined order. Efficient memory access is crucial to maintaining high performance, as random access to global memory can introduce significant latency. Utilizing texture memory, which is optimized for spatial locality, allows faster access to pixel data. Additionally, hierarchical data structures such as quadtrees (for 2D images) and octrees (for 3D

images) are used for spatial indexing and efficient region-based operations. These structures enable optimized operations like adaptive image compression, object detection, and efficient nearest-neighbor searches in high-resolution images.

- **Ray Tracing:** Ray tracing is a fundamental technique in computer graphics for rendering realistic images by simulating light interactions with objects. The computational cost of testing rays against a large number of objects can be prohibitive without efficient spatial indexing. Bounding Volume Hierarchies (BVHs) are widely used for accelerating intersection tests by hierarchically organizing objects in a scene. Optimized BVH traversal on GPUs is critical for achieving real-time performance in applications such as video game rendering and movie special effects. By leveraging warp-wide processing and memory-efficient data structures, GPU-accelerated ray tracing engines can achieve photorealistic rendering in real time, making them indispensable for modern graphics applications.

5. Future Research Directions

The field of high-throughput data structures for GPU-accelerated computing is continuously evolving, with numerous opportunities for innovation and optimization. Future research directions focus on enhancing the adaptability, efficiency, and usability of GPU-optimized data structures to meet the growing demands of emerging applications. One promising direction is automatic data structure selection, where tools and frameworks can intelligently choose the optimal data structure for a given application based on its characteristics. Machine learning models could be trained to predict performance based on workload patterns, hardware configurations, and memory access behaviors, enabling automated optimization without requiring manual tuning. Another key area is the development of adaptive data structures that dynamically adjust to changing data characteristics at runtime. For example, sparse matrix formats could switch between Compressed Sparse Row (CSR) and Compressed Sparse Column (CSC) depending on the access pattern, ensuring optimal performance under varying workloads. Similarly, GPU-friendly dynamic data structures could restructure themselves on the fly to maintain efficient memory access and parallel execution.

With the rapid growth of new computing paradigms, there is a need for specialized data structures for emerging applications. In particular, graph neural networks (GNNs) require efficient representations for large-scale graphs, while quantum computing simulations may benefit from novel GPU-optimized representations of quantum states and operations. Developing customized data structures tailored for these domains could significantly accelerate computations and improve scalability. Integration with high-level programming languages is another crucial area of future research. Many developers rely on high-level languages like Python for data science, machine learning, and scientific computing. Creating seamless GPU-optimized data structures that can be directly utilized in frameworks like TensorFlow, PyTorch, and NumPy would lower the barrier to entry and enhance productivity without requiring extensive knowledge of CUDA or low-level GPU programming. Hardware-aware data structure design explores co-optimization between data structures and GPU architectures. By leveraging specific GPU features—such as memory hierarchies, warp scheduling, and tensor cores data structures can be fine-tuned for maximum performance. This could involve developing custom memory controllers or specialized processing units for operations like tree traversal, hash table lookups, and sparse matrix manipulations. Advanced memory management techniques for GPUs, including efficient garbage collection, memory pooling, and defragmentation strategies, could help improve overall performance and scalability. As GPUs are increasingly used for general-purpose workloads beyond graphics, robust memory management solutions will be essential for handling complex data structures efficiently.

6. Conclusion

High-throughput data structures are fundamental to achieving optimal performance in GPU-accelerated computing. Efficiently managing memory layout, access patterns, concurrency, and data transfer strategies is crucial for designing GPU-friendly data structures. While adapting traditional data structures for parallel execution on GPUs presents challenges, the potential performance benefits are substantial. This paper has provided an overview of key considerations for GPU data structure design, emphasizing memory coalescence, data organization strategies like Structure of Arrays (SoA), and optimized formats for sparse data. We explored specialized structures such as hash tables and tree-based indexing methods that enhance performance across various applications. Case studies demonstrated the effectiveness of these optimizations in domains including scientific computing, machine learning, data analysis, image processing, and ray tracing. Future research in this field is essential to fully harness the power of GPU computing. Advances in automatic data structure selection, adaptive data structures, hardware-aware optimizations, and seamless high-level language integration will drive further improvements. As GPUs continue to evolve, so too must the data structures that enable their efficient use, unlocking new possibilities for high-performance computing and innovative applications.

References

[1] Ament, M., Ziegler, G., & Dachsbacher, C. (2012). Implementing efficient parallel data structures on GPUs. In M. Pharr & R. Fernando (Eds.), *GPU Gems 2* (pp. 521–545). Addison-Wesley Professional.

[2] Chen, Z., Xu, J., Tang, J., Kwiat, K., & Kamhoua, C. (2015). G-Storm: GPU-enabled high-throughput online data processing in Storm. *2015 IEEE International Conference on Big Data (Big Data)*, 307–312. https://doi.org/10.1109/BigData.2015.7363771

[3] Fan, B., Andersen, D. G., & Kaminsky, M. (2013). MemC3: Compact and concurrent memcache with dumber caching and smarter hashing. *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, 371–384.

[4] Garland, M., & Kirk, D. B. (2010). *Programming massively parallel processors: A hands-on approach*. Morgan Kaufmann.

[5] Herlihy, M., & Shavit, N. (2012). *The art of multiprocessor programming*. Morgan Kaufmann.

[6] Jünger, D., Kobus, R., Müller, A., Hundt, C., Xu, K., Liu, W., & Schmidt, B. (2020). WarpCore: A library for fast hash tables on GPUs. *arXiv preprint* arXiv:2009.07914. https://doi.org/10.48550/arXiv.2009.07914

[7] Kipfer, P., & Westermann, R. (2005). Improved GPU sorting. *GPU Gems 2*, 733–746.

[8] Maier, D., & Torp, K. (2019). Concurrent growing of hash tables: A GPU perspective. *Proceedings of the VLDB Endowment*, 12(11), 1593–1605. https://doi.org/10.14778/3342263.3342631

[9] NVIDIA Corporation. (2020). CUDA C++ programming guide. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

[10] Qureshi, Z., Mailthody, V. S., Gelado, I., Min, S. W., Masood, A., Park, J., Xiong, J., Newburn, C. J., Vainbrand, D., Chung, I.-H., Garland, M., Dally, W., & Hwu, W.-m. (2022). GPU-initiated on-demand high-throughput storage access in the BaM system architecture. *arXiv preprint* arXiv:2203.04910. https://doi.org/10.48550/arXiv.2203.04910

[11] Ragan-Kelley, J., Barnes, C., Adams, A., Paris, S., Durand, F., & Amarasinghe, S. (2013). Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *ACM SIGPLAN Notices*, 48(6), 519–530. https://doi.org/10.1145/2491956.2462176

[12] Reinders, J. (2007). *Intel threading building blocks: Outfitting C++ for multi-core processor parallelism*. O'Reilly Media.

[13] Sengupta, S., Harris, M., Zhang, Y., & Owens, J. D. (2007). Scan primitives for GPU computing. *Graphics Hardware*, 97–106.

[14] Shavit, N. (2011). *Data structures in the multicore age*. Communications of the ACM, 54(3), 76–84. https://doi.org/10.1145/1897852.1897873

[15] Sutter, H. (2005). The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's Journal*, 30(3), 202–210.

[16] Tzeng, S., & Wei, L.-Y. (2008). Parallel white noise generation on a GPU via cryptographic hash. *Proceedings of the 2008 Symposium on Interactive 3D Graphics and Games*, 79–87. https://doi.org/10.1145/1342250.1342264

[17] Zhang, K., Wang, K., Yuan, Y., Guo, L., Lee, R., & Zhang, X. (2015). Mega-KV: A case for GPUs to maximize the throughput of in-memory key-value stores. *Proceedings of the VLDB Endowment*, 8(11), 1226–1237. https://doi.org/10.14778/2809974.2809981

[18] Ziegler, G., Theußl, T., & Purgathofer, W. (2002). Fast rendering of complex scenes by hierarchical rasterization. *Proceedings of the Vision Modeling and Visualization Conference 2002*, 251–258.