CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Hardware BVH builder based on the PLOC++ algorithm

Master's thesis in Computer science and engineering

Keivan Saberian

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2023

Master's thesis 2023

Hardware BVH builder based on the PLOC++ algorithm

Keivan Saberian

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2023

Hardware BVH builder based on the PLOC++ algorithm

Keivan Saberian

Supervisor: Ulf Assarsson, Embedded Electronics Systems and Computer Graphics
Examiner: Erik Sintorn, Embedded Electronics Systems and Computer Graphics

Master's Thesis 2023
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2023

Hardware BVH builder based on the PLOC++ algorithm

Keivan Saberian
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

The demand for high-quality visual effects in 3D rendering of real-time applications is on the rise. To meet this demand, researchers have focused on integrating ray tracing support into graphics hardware. However, support for dynamic scenes still poses a significant challenge. This is due to the fact that the underlying spatial data structures, most commonly the bounding volume hierarchy, must be rebuilt every frame in the worst case.

This thesis introduces a hardware accelerator for the construction of bounding volume hierarchies. The proposed hardware is based on the state-of-the-art PLOC++ algorithm, and aims to address the memory-intensive construction through a bandwidth economical approach where most external memory traffic is converted into on-chip streaming traffic similar to PLOCTree [Viitanen et al. 2018]. The proposed unit is on average 2.19 times faster in simulation, with a 3.94 times improvement in memory traffic.

Keywords: computer, science, computer science, engineering, computer graphics, ray tracing, bounding volume hierarchy, graphics hardware, project, thesis.

Acknowledgements

This thesis was made possible thanks to Chalmers for providing the tools necessary. I would also like to extend thanks to my supervisor and examiner for providing both feedback and advice during various stages of the project. Finally, I would like to thank Benthin Carsten for providing me with the PLOC++ repository.

Keivan Saberian, Gothenburg, 2023-06-13

Contents

List of Figures

List of Tables

1

Introduction

Recent developments in GPUs (such as RTX cores in modern NVIDIA GPUs) [Burgess 2020] alongside an increasing demand and interest of high quality visual effects in 3D rendering of real-time applications, has put an emphasis on research for achieving high performance real-time ray tracing [Deng et al. 2017]. A key component in realising such performance is the need to organize scene geometry into spatial acceleration data structures, with the most common data structure being the Bounding Volume Hierarchy (BVH) [Vinkler et al. 2017; Viitanen et al. 2018; Meister et al. 2021; Benthin et al. 2022].

Dynamic scenes, as in real-time applications, requires support for the BVH to be in the worst case reconstructed every frame in order to maintain high visual quality [Viitanen et al. 2018; Benthin et al. 2022]. For this reason, much development has been seen in offloading the BVH construction onto hardware accelerators. However, Viitanen et al. [2018] notes that state-of-the-art accelerators lack foundation in modern algorithms. They therefore instead identified and advocated for the more recent Parallel Locally-Ordered Construction (PLOC) algorithm as more suitable for hardware acceleration.

An improved version of PLOC was published in July 2022, titled PLOC++, which outperforms the original in both computational- and memory requirements [Benthin et al. 2022]. The thesis project aims to design a hardware accelerator for the state-of-the-art BVH construction algorithm PLOC++ in order to improve its build time, memory traffic, and power consumption for the use of real-time ray tracing.

In particular, the following questions will be explored:

1. *How does the chosen hardware implementation of PLOC++ compare to the original software version, as well as the hardware accelerated PLOC algorithm by Viitanen et al. [2018]?*
2. *How well can the proposed design be integrated into desktop-, mobile-, and game console GPUs for the use of real-time ray tracing?*

1.1 Related Work

BVH construction is a widely researched area spawning numerous algorithms, and in turn hardware accelerators [Meister et al. 2021]. This raises the need to compare

builder quality, which is commonly done through the *Surface Area Heuristic* (SAH) metric [Viitanen et al. 2018]. SAH provides an estimation of a given tree node's ray intersection probability through its bounding box's surface area; it was introduced alongside the top-down *SAH sweep* BVH construction algorithm [MacDonald and Booth 1990]. SAH sweep has since then been extended into the *binned SAH sweep* algorithm [Wald 2007], which in turn has become the basis for previous hardware accelerators [Doyle et al. 2013].

As for other accelerators, there exists those based on the *Linear Bounding Volume Hierarchy* (LBVH) algorithm [Viitanen et al. 2017] introduced by Lauterbach et al. [2009]. Furthermore, some accelerators have utilized refitting [Woop et al. 2006; Nah et al. 2015], i.e. algorithms that reuse BVH topology as opposed to constructing new BVHs from scratch, and some are based on k-d trees instead of BVHs [Nah et al. 2014; Liu et al. 2015].

Finally, a hardware accelerator exists for the original PLOC algorithm [Viitanen et al. 2018] that has been shown to produce large performance improvements, while also reducing memory bandwidth requirements, as compared to the original software implementation. This has lead the authors of PLOC++ [Benthin et al. 2022] to note that the implementation of PLOC++ could similarly benefit by having the merged nearest neighbor search and hierarchy construction phase be offloaded to a dedicated hardware unit. Additionally, they suggest that a hardware unit for sorting key-value integer pairs would also be beneficial, since sorting the Morton codes is starting to have similar run-time cost as the merged phase.

The advent of PLOC++, alongside the apparent improvements possible through hardware acceleration, has become motivation for this thesis. This motivation is furthered by the observation that previous builders, as discussed above, lack foundation in modern construction algorithms [Viitanen et al. 2018].

1.2 Delimitation

As mentioned in Section 1.1, the authors of PLOC++ note that dedicated hardware units for offloading:

1. *The merged nearest neighbor search and hierarchy construction phase*
2. *The sorting of Morton code key-value pairs*

can both lead to large benefits in algorithmic performance. This thesis will focus only on 1), i.e. the merged nearest neighbor search and hierarchy construction phase, as this constitutes the core part of PLOC++, and is also its main separation point from similar algorithms such as PLOC.

1.3 Ethical Considerations

The work of this thesis, its purpose and final product, is done in the hopes of contributing to the vast field of research in real-time ray tracing, such that the

technology can become accessible to a wider audience through consumer grade GPUs, smartphones, and video-game consoles. That said, it can be speculated that such accessibility might be gated by high product pricing, and thus only available to part of the population until said pricing drops as time since commercial launch passes. This kind of speculation can be made for all sorts of technologies, however to bridge the gap between low-end and high-end products, technological advancements are likely to be needed in the first place. It is also important to recognize the possibilities for future technology, and user experiences in various media, that this kind of research can enable.

Furthermore, the hardware can be verified using software simulation as well as synthesis on FPGA (see Section 2.4). The act of simulation, coupled with the reprogrammability of FPGAs, makes it possibly to conduct this research without producing hardware that will go to waste.

2

Background

This section presents necessary background for understanding the thesis work.

2.1 Ray Tracing

Ray casting is a fundamental process in computer graphics that involves firing a ray from a specific location to determine the objects that are present in a particular direction. *Ray tracing*, on the other hand, is a technique built on ray casting in order to determine the light transport between various scene elements. These concepts constitute an inherent component of many modern rendering algorithms. In its most basic form, ray tracing involves emitting rays from the camera through the pixel grid into the scene. The closest object of each ray is then identified, at which point its shadow is computed by firing a ray from the intersection point towards each light source to find if objects are present in between. Additional rays can also be spawned from an intersection point to further refine the lighting and shading of the scene, and the same is true for the number of rays sent per pixel [Akenine-Möller et al. 2018].

2.2 Bounding Volume Hierarchy

A *bounding volume* is a volume enclosing a set of objects. It allows for a simpler geometric representation of its contained objects, and is used as their proxy to speed up computations related to for example rendering or collision detection. Different types of bounding volumes exists. Relevant to this thesis is the *axis aligned bounding box* (AABB). An AABB is a rectangular volume whose faces have normals that align with the axes of the coordinate system [Akenine-Möller et al. 2018].

For real-time rendering, the scene can be organized into bounding volumes forming a tree data structure. The root node of this tree would represent a bounding volume which encompasses the entire scene. An internal node is a bounding volume of other bounding volumes, referred to as children. The root node is an example of an internal node, with the caveat of not having a parent. Finally, leaf nodes contain the actual geometry to render, and thus have no children. This data structure is known as a *bounding volume hierarchy* (BVH) [Akenine-Möller et al. 2018], and is the most commonly used data structure for achieving high performance real-time ray tracing [Vinkler et al. 2017; Viitanen et al. 2018; Meister et al. 2021; Benthin et al. 2022].

In particular, this thesis concerns itself with binary tree BVHs, which means that each internal node has at most two children [Akenine-Möller et al. 2018].

2.3 Morton Code

Morton code [Morton 1966] is a system for assigning integer values to sets of n-dimensional data. This is done by creating a number pattern that orders these sets based on their spatial proximity. This system is sometimes referred to as a Z-curve due to the pattern created in two-dimensional sets.

The Morton code is popular because it is easy to calculate with simple bit-interleaving operations. For example, the n-bit Morton code of vector $\mathbf{v}$ = (x,y,z) is created by interleaving the bits of $\mathbf{v*}$ = (x*,y*,z*) ϵ ((0, $2^{n/3}$), (0, $2^{n/3}$), (0, $2^{n/3}$)). In other words, the coordinates of $\mathbf{v}$ are first mapped into discrete integer values in the range (0, $2^{n/3}$). The bit representation of these coordinates after mapping are then interleaved: $x_0y_0z_0x_1y_1z_1...x_{n/3}y_{n/3}z_{n/3}$ [Vinkler et al. 2017].

2.4 Hardware Description Design

Designing digital circuits and systems at a high level of abstraction is usually done through Hardware Description Languages (HDLs). One such language is SystemVerilog, which can be used in conjunction with design suite tools such as Xilinx Vivado for synthesis and analysis of the hardware description.

Synthesis can be thought of as the process of translating a high-level description of a digital circuit or system into a low-level implementation that can be programmed onto an FPGA (Section 2.5). This is done by converting the HDL design into gate-level representations and logic components. Synthesis, in other words, means that the abstract description of hardware provided in a high-level language, will wire the FPGA circuitry to implement the specified behaviour similar to how a compiler translates software code into CPU readable instructions [Hauck and DeHon 2008].

To note is that SystemVerilog is not only an HDL, but also a Hardware Verification Language (HVL). This means that the language provides additional constructs which can only be used for verification purposes, and not synthesis. The specific subset of SystemVerilog that can be used for synthesis is done at so called *register transfer level* (RTL) [IEEE 2018], and the same is often true for other HDLs [Hauck and DeHon 2008].

2.5 Field-programmable Gate Array

A *Field-programmable gate array* (FPGA) is a hardware device that can be configured, and at any point re-programmed, to implement a wide variety of tasks. The term *re-programmed* in this sense means that configurations are not permanent, but rather programmed into the chip in such a way that new specifications can be provided to override the configuration with possibly a whole new set of functionality.

To clarify: reprogramming can be done for something as simple as small bug fixes, or to extend behaviour with new features, or even to implement something entirely different. This is done through so called programming points in a specific portion of the FPGA memory. All logic and routing elements are controlled by these memory bits, and can be configured by supplying a file or bitstream into the device.

As opposed to microprocessors, computations in an FPGA are implemented spatially, which allows for the processing of a large number of operations in parallel across the silicon chip. This gives potential for large performance savings in terms of execution time, power, and silicon area [Hauck and DeHon 2008].

3

Algorithm

This section starts with an overview of the PLOC and PLOC++ algorithms, before proceeding to explaining the process of translating these to a hardware suitable implementation.

3.1 Parallel Locally-Ordered Clustering

Parallel Locally-Ordered Clustering (PLOC) [Meister and Bittner 2018] is a bottom to top BVH construction algorithm, where clusters of scene triangles are iteratively merged until only one cluster (the top-level BVH node) remains. Each scene triangle initially corresponds to one unique cluster. These initial clusters are in a pre-processing step sorted in 1D along the Morton codes of their centroid. The main part of the algorithm then consists of repeated applications of the following three phases:

- Nearest-neighbor search: Based on the property that nearby indices in the Morton code sorted array are likely to be close in 3D-space, the algorithm finds and stores the closest neighboring cluster in a search radius R of neighboring indices in the sorted array, for each cluster. In other words, for each index i in the array, find the corresponding index in the range [i - R, i + R] that minimizes the distance function. In particular, the distance between two clusters is defined as the area of their merged bounding volume, with ties broken by lower index. This function is said to follow a non-decreasing property, as explored by [Walter et al. 2008], which claims that no better neighbor will emerge in the future for currently mutually agreeing nearest-neighbors.
- Merging of clusters: After finding the nearest-neighbor of each cluster, a second CUDA kernel is launched where any mutually agreeing nearest-neighbors are merged into one AABB. This merged cluster overrides the lower index of the two merged nodes, while the upper index is marked as invalid. Nodes without any mutual nearest-neighbor are left unchanged. Valid/Invalid metadata is stored in a separate array.
- Compaction: The final phase removes any invalid entries of the array. This is done through three kernel launches, and employs a prefix sum to determine the position of valid node indices. A prefix sum of a sequence A, is a second sequence B, such that each element at index x in B is the sum of all elements

up to and including index x in A.

The array does not get re-sorted in Morton code order after each iteration due to its required cost, and through the observation that Morton code orderings are sufficiently kept between iterations for the approximate nature of the nearest-neighbor search.

3.2 PLOC++

The authors of PLOC identify three main bottlenecks of the algorithm. The first is the complexity of the nearest-neighbor search. The second is the overhead associated with five kernel launches required for each iteration of the algorithm, and the third is iterating through-, and performing memory operations on auxiliary arrays (for example for keeping track of valid/invalid states and performing the prefix sum during the compaction phase).

PLOC++ [Benthin et al. 2022] is an extension of PLOC that improves performance by addressing these bottlenecks. This thesis work concerns itself only with *PLOC++ Two-Level*, an alternate version of PLOC++, which is the one to be implemented in hardware. For brevity, PLOC++ Two-Level will henceforth be referred to simply as PLOC++.

The algorithm splits the Morton code sorted array into independent sequences such that each sequence independently constructs a BVH within an OpenCL work group. The algorithm allows each such sequence to build up to their respective root node in only one kernel launch. This comes from the assumption that sequences are independent. Viewing each sequence as independent means that the output of the nearest neighbor search in one such sequence, is the only data required for merging its clusters. This data can therefore be stored in local memory, as opposed to global, because it is irrelevant to other sequences. The nearest neighbor search and merging phase can thus be done in a single kernel launch. Furthermore, the compaction phase can happen in the same kernel launch as well, because the assumption of independence means that changes in one sequence does not affect other sequences.

However, to maintain BVH quality, each independent sequence does not fully construct its BVH. Instead, the iterative process stops once the total number of clusters within a sequence reaches a given threshold. When all sequences have reached this threshold, they get merged to one range, on which a second kernel is launched to build the top-level bounding volumes. The total number of kernel launches is thus reduced from five times the number of PLOC iterations, to only two.

Efficiently partitioning the Morton code sorted array into independent ranges is done by viewing the Morton codes as forming a binary tree, such that the root node corresponds to the full range over all clusters. The two children of each node can then be viewed as splitting the range into two independent sequences. This tree is traversed top-down, breadth-first, to split the largest ranges until the desired number of ranges have been made.

Furthermore, changes have been made to the nearest neighbor search algorithm itself

in order to reduce its impact on the overall performance. Recall from Section 3.1 that the search is done in a range [i - R, i + R] for each position i of the input array, and a search radius R. The search complexity thus becomes $N*R*2$, where N is the number of elements in the array. This is reduced by half to $N*R$ in PLOC++ by utilizing the commutative property of the distance function *distance(x, y) = distance(y, x)*, which means that computing distance(x, y) contributes to the nearest neighbor search for both of clusters x and y. Exact implementation details will however be omitted as they are not necessary to understand the hardware implementation.

3.3 From Software to Hardware

Listing 1 shows an abstract overview of the PLOC++ algorithm. Each of the components inside the PLOC SWEEP loop, i.e. NN-search, merge, and compaction phases, constitute their own independent loops across each cluster in the sequence. In other words, a phase does not start until the previous is completed for each element.

```
Preprocessing:
    Compute Morton codes for each cluster
    Sort clusters according to Morton codes
    Split the Morton code sorted array into independent sequences

Main Algorithm:
    For each independent sequence:
        while (number of clusters > threshold) : PLOC SWEEP
            Find nearest neighbor of each cluster in radius R
            Merge mutually agreeing nearest neighbors
            Remove empty indices after merging

    Concatenate independent sequences into one range
    Build top-level BVH by repeatedly applying PLOC SWEEPs
```

Listing 1: Overview of PLOC++

While PLOC++ presents fewer operations on auxiliary data buffers as compared to the original PLOC, the overview in Listing 1 still requires a lot of such operations in each iteration of the PLOC SWEEP loop. Performing these operations on data buffers residing within the GPU memory becomes problematic, because the data sizes involved in these processes exceed the permissible cache limits and result in inefficiencies that compromise the overall system's performance. When designing a dedicated hardware unit, the latency might even increase unless a large enough storage is present locally within the unit. To note however is that the data size is in the worst case equal to the scene's primitive count.

Designing a hardware unit with enough local memory to store such an amount becomes unfeasible because the primitive count differs between scenes, and therefore difficult to predict. It might also not be desirable to accommodate such storage

due to its large area needs, as well as energy requirements. Finally, providing the storage does not remedy the inherent problem of latency associated with performing memory operations. The hardware should therefore present an approach that minimizes memory buffer traffic and elevates local data transfers, thereby enhancing throughput.

This necessitates a two-part solution, with the initial step involving the transformation of single PLOC SWEEP units into autonomous, pipelined, streaming processes. This step minimizes latency between the NN-search, merging, and compaction phases by starting the next phase for a given cluster as soon as its independent requirements are met. The pipelined streaming approach also makes it possible for clusters to move between phases without additional memory operations. The second step involves performing multiple on-chip sweeps, further negating traffic between sweep units and augmenting the system's reliance on on-chip data streams.

Listing 2 shows an overview of the algorithm encapsulating the desired behaviour of PLOC SWEEP units. The algorithm is designed for a throughput of one AABB per clock cycle. It takes input from FIFO memory and performs the entire PLOC SWEEP loop from Listing 1 on its independent AABB sequence using only local circular buffers of size B, where B must be a power of 2 in order to take advantage of the quick modulo operation performed by the mask() function.

The phases have been merged into one loop through the observation by Viitanen et al. [2018], that once nearest[i] elements have been processed by the NN-search, there is enough information to perform the merging/compaction phase up until nearest[i - search radius]. This is because individual clusters only depend on clusters within the search radius. As such, this algorithm is similar to the one presented in PLOCTree [Viitanen et al. 2018], however with some slight modifications, as well as the addition of the improved NN-search algorithm of PLOC++.

As mentioned in Section 3.2, the NN-search takes advantage of the commutative property of the distance function. After initializing the nearest[] array to a large default value, the search is performed in the range [i+1, i + search radius] for each cluster *i*. The 32-bit distance of each pair (i, i + r), is then left-shifted to allow the relative offset to the neighbor be stored in the lower 32-bits. In actuality, this can be implemented in SystemVerilog without explicit bit manipulation through the use of packed structs.

Finally, once each sweep pipeline has finished building their BVH, the top-level BVH is built by concatenating the sequences into one range that is given as input to only one sweep pipeline. This mirrors the implementation of PLOC++ where independent BVHs are first built, and then collected into one range for constructing the top-level BVH in a total of two kernel launches.

```
input[B];
nearest[B] = MAX_VALUE;
is_merged[B] = false;
search_radius = 16;
encode_mask = NOT((1 << 32) - 1);

c_idx = 0;
while(threshold not reached)
  ----Nearest Neighbor Search-------
    min_dist_and_offset = MAX_VALUE
    input[mask(c_idx + search_radius)] = fifo_in.pop();
    nearest[mask(c_idx + search_radius)] = MAX_VALUE;

    for (r = 1; r <= search_radius; r++)
        dist = distance(input[c_idx], input[mask(c_idx+r)]);
        my_offset = r;
        other_offset = (i-r);
        encoded_dist = (dist << 32) AND encode_mask;
        my_dist_and_offset = encoded_dist OR my_offset;
        other_dist_and_offset = encoded_dist OR other_offset;

        min_dist_and_offset = min(min_dist_and_offset,
            my_dist_and_offset);
        min(nearest[mask(c_idx+r)], other_dist_and_offset);
    end for
    min(nearest[c_idx], min_dist_and_offset);

    --------Merging & Compaction-----------
    l_idx = mask(c_idx - search_radius);
    l_nearest = mask(l_idx + lower 32 bits of nearest[l_idx]);

    if(l_idx = mask(l_nearest + lower 32 bits of nearest[l_nearest])
        && l_idx < l_nearest)
            output <= merge(input[l_idx], input[l_nearest]);
            is_merged[l_nearest] = true;
    else if (!is_merged[l_idx])
        output <= input[l_idx];
    else
        //merged in previous iteration
---------------
mask(i32 value){
    return value AND (B - 1); //B = size of input[]
}
```

Listing 2: Algorithm describing the behaviour of PLOC SWEEP units.

4

Hardware Architecture

An overview of the architecture can be seen in Figure 4.1. Its main components are: memory, PLOC SWEEP units, as well as a control unit for handling input and output to each sweep unit.

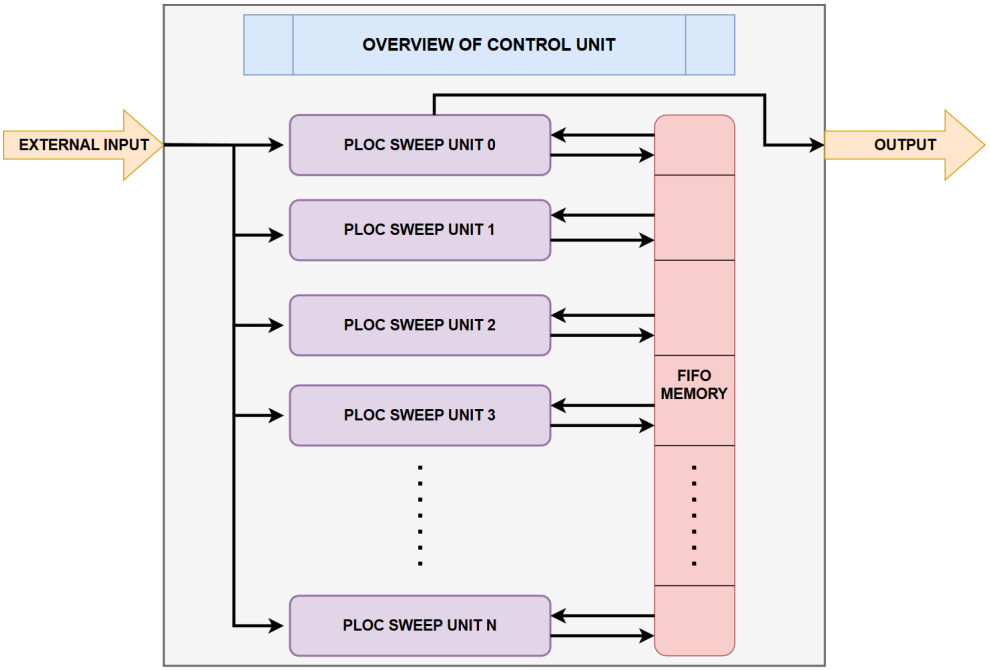

Figure 4.1: System Architecture of a hardware unit with N sweep units. As indicated by the figure, both sweep units and memory reside inside the control module similar to how various PC components reside on a motherboard. Note that the figure is meant as a simplified overview for visualization purposes, as opposed to a detailed schematic of the logic inside the control unit.

The control unit receives external input in the form of scene primitives and supplies them to the PLOC SWEEP units in parallel. The input primitives are sorted externally before being received such that the control unit can correctly supply them to their corresponding sweeps through simple indexing. This continues for each sweep until its entire external input has been provided, after which the control unit proceeds to send input to the sweep through its associated FIFO memory.

The PLOC SWEEP units are designed as a translation of the algorithm in Listing 2, and can be seen in Figure 4.2. A sweep unit consists of distance metric evaluators

(DMEs), comparator units, a merge unit, as well as local circular buffers for storing auxiliary data of AABBs currently in its pipeline. Each unit is assigned an independent sequence of the Morton code sorted array, which is stored inside its own FIFO section of the memory, and is populated in accordance with the Morton code ordering.

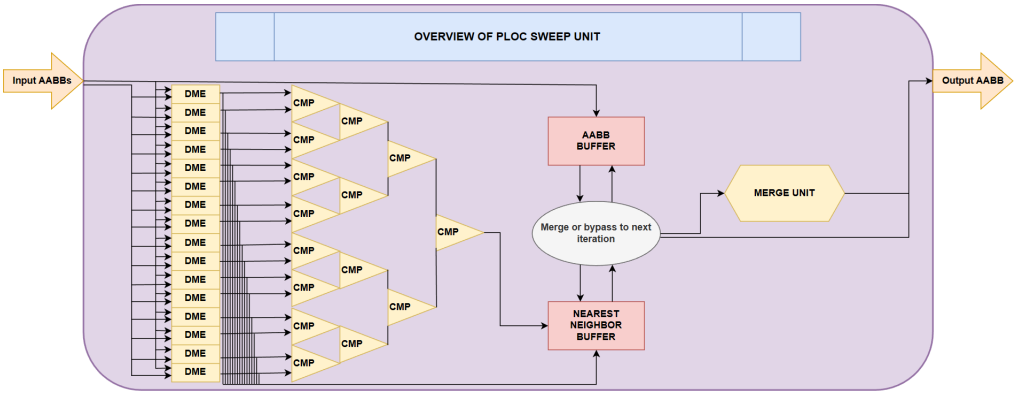

Figure 4.2: Architecture of a PLOC SWEEP unit with search radius of 16. Here, DME stands for Distance Metric Evaluator, and CMP refers to a comparator unit.

At the start of every clock cycle, the control unit supplies the sweep unit with batches of 1 + *search radius* AABBs, and the following events take place in order:

1. The input AABBs are sent to *search radius* DME units where $distance(C_i, C_{i+r})$ is computed for C_i = *lowest index AABB in batch*, and $r \epsilon \{1..$ *search radius*$\}$. Recall from Section 3.1 that the distance between two AABBs is defined as the area of their merged bounding volume.

2. The DME output of clusters (C_i, C_{i+r}) is checked against the currently saved distance of the nearest neighbor of C_{i+r}, while also being sent to a tree of comparator units. The comparator units compare the outputs of each DME to find the lowest distance. This distance is checked against the distance of the currently saved nearest neighbor of C_i. At this point, the nearest neighbor of C_i in range $[C_{i-r}, C_{i+r}]$ has been found.

3. Cluster C_{i-r} is checked for any mutually agreeing nearest neighbor. If one is found, and if not merged in a previous pass, then the two are sent to the merge unit before being output from the pipeline to be stored in the FIFO memory. If there is no mutually agreeing nearest neighbor, then the cluster is output from the pipeline as is. If merged in a previous pass, then the cluster will not be output, and will instead eventually be overwritten in a future iteration of the pipeline.

Once the sweep units have built their BVH up until the threshold as in Listing 2, the top-level BVH will be constructed by one sweep pipeline. This process is handled by the control unit, and must respect the Morton code ordering of the input. Note that this process can start as soon as the sweep unit whose input sequence contains the lowest index AABBs has finished building, and thus said sweep unit can be appointed to construct the top-level pipeline.

5

Evaluation

The hardware, as described in Section 4, was implemented at the register transfer level in SystemVerilog. The number of PLOC SWEEP units was set to 32, and similarly to PLOC++ [Benthin et al. 2022], the search radius was set to 16 and the iterative build process for each independent sweep terminates once *search radius* or fewer clusters remain.

The design ran build simulations of various scenes, from which performance metrics in terms of execution time, power consumption, and memory traffic were measured and compared to the GPU implementation (`https://github.com/embree/embree/tree/ploc`) running on Intel Arc A750. The simulation was conducted on hardware similar to the Intel Arc A750 in order to achieve a fairer comparison.

PLOCTree [Viitanen et al. 2018], on the other hand, does not have a public repository available. As such, the proposed hardware will only have its area and power consumption compared to the reported values of PLOCTree in its original paper.

In pre-processing, the scene primitives were extracted from OBJ files, then sorted in Morton code order, and finally split into independent sequences in Python. These sequences were then input to the hardware unit in parallel. Thus, all performance measurements for both the proposed hardware design and the GPU implementation exclude any pre-processing and regard only the construction from primitives to root node.

As a side note: while the implementation and its resulting BVH have been thoroughly tested for correctness in simulation as well as compared to the GPU implementation, the actual output BVH of a synthesized implementation was not loaded into a software ray tracer for validating correctness of the resulting trees, nor for measuring their ray tracing performance. This is due to a lack of available FPGAs at Chalmers this current semester. That said, the algorithm in Listing 2 has been implemented in software and have had its results rendered for verification purposes prior to being realized as hardware.

6

Results

The main results are summarized in Table 6.1. The detailed performance results can be found in Figure 6.1, while area and power is shown in Table 6.2. Here, TDP stands for *thermal design power*. TDP is the amount of power a processor is designed to dissipate without exceeding the highest possible operating temperature of its transistors. It is often used to estimate chip power consumption [Esmaeilzadeh et al. 2011].

As mentioned in Section 5, the proposed design was evaluated using a search radius of 16 and consists of 32 sweep units in total. While the results heavily depends on these parameters, it should be made clear that the hardware has been designed to work independently of their values. Both parameters can be changed before synthesis without requiring changes in the code. In this case, the search radius was chosen to be the same as in the original PLOC++ for the sake of comparison, and the number of sweep units was chosen to provide an appropriate trade-off between performance, power consumption, and area.

Table 6.1: Summary of results.

	GPU PLOC++, Intel Arc A750	PLOCTree [Viitanen et al. 2018]	HW PLOC++ (**simulated**)
Area (mm^2)	10962	2.43	15.1
Clock (GHz)	2.05	1.0	2.05
TDP (W)	225	0.362	5.16
Build time	219%	-	100%
Mem. traffic	394%	-	100%

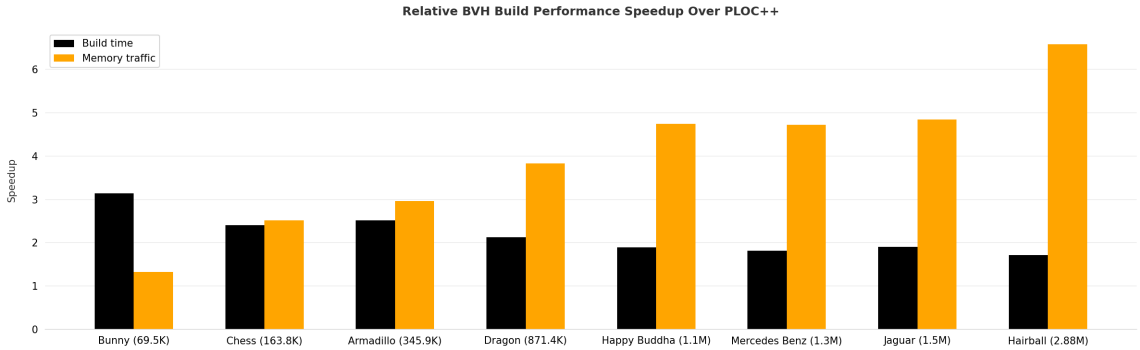

Figure 6.1: Relative performance speedup of the proposed hardware unit over the GPU implementation.

The hardware boasts an average improvement of 2.19x and 3.94x for build time and memory traffic respectively over the GPU implementation during simulation. Initially, results after synthesis were planned to be measured. However, as of yet, no FPGA has been available for use due to a high demand of FPGA resources at Chalmers this semester. Once synthesized, the observed performance numbers are expected to be lower. This is because:

1. The clock frequency of modern FPGAs tend to be much lower than the selected value of 2.05GHz. In particular, the PYNQ-Z2, which was originally meant to be used, has a reported maximum frequency of 650Mhz [Xilinx 2018].

2. The run-time behaviour of actual hardware is unlikely to be as idealistic as compared to simulation. Furthermore, memory traffic will likely differ between architectures due to their differences in available memory resources.

That said, it is difficult to predict by how much the results will differ after synthesis, because it depends on the actual hardware used. In this regard, simulation might actually prove to give a fairer comparison as it can be tailored to match the GPU's specifications.

It should also be kept in mind that, as previously mentioned, the number of sweep units, as well as the search radius, can be changed independently of the design specification. Different values will have different performance results. For example, increasing the number of sweep units will lead to faster build times and better memory traffic, but will require more area while also increasing TDP.

Additionally, while the simulated design shows a decent improvement in memory traffic, there is still much more room for further optimization. The following lists a smorgasbord of techniques for further improvements in memory efficiency, which becomes increasingly important as the number of scene primitives increases:

- Increased on-chip memory.
- More sophisticated memory management as opposed to the simplistic FIFO.
- Compression techniques.
- Increased number of sweep units.

- Pipelined sweep units such that the output of one becomes input to the other, similar to [Viitanen et al. 2018].
- Increased number of independent sequences per sweep unit.

Note also that the last item on the list is done during pre-processing, as opposed to requiring changes in the design. Furthermore, any combination of these techniques can be used independently of each other. That said, as before, these changes come with their own trade-offs in terms of build performance, power consumption, area usage, and even tree quality.

Table 6.2: Area and power breakdown of a single PLOC SWEEP unit.

Component	Area (mm^2)	Power (mW)
FIFO memory	0.2	63
Distance metrics evaluators	0.16	39
Comparator tree	0.03	40
Merge unit	0.02	6
Data buffers	0.04	12
Total	0.45	160

As shown in Table 6.2, a single sweep unit has a rough power of estimate of 160mW. Given the large number of sweep units used, this results in quite a high TDP for the hardware unit (5.16W). For reference, game benchmarks have shown a power consumption of 1-3W for smartphones in 2014 [Pathania et al. 2014]. While recent smartphones have become more advanced since, and thus likely consume a larger amount of power, the current configuration of hardware PLOC++ would still constitute a large percentage of the power budget. If lower performance is of less concern, then it becomes simple to reduce the TDP by reducing the number of sweep units. However, depending on the number of sweep units having to be reduced, PLOC-Tree [Viitanen et al. 2018] might become more suitable due to its smaller TDP requirements. The same is also true for any area concerns.

On desktop or console GPUs however, the proposed hardware would only account for a very small percentage of the overall TDP and silicon area. In this case, there might even be enough room to possibly increase the number of sweep units for further performance gains without affecting the overall TDP by much.

7

Conclusion

The proposed hardware accelerator for BVH construction based on the PLOC++ algorithm has shown an average improvement of 2.19x and 3.94x during simulation in terms of build time and memory traffic respectively, as compared to the GPU implementation. This result depends however on the search radius size, the number of sweep units, and the number of independent sequences assigned to each sweep unit. These parameters can be changed without impacting the design specification.

Similarly, both power consumption and silicon area are also dependant on the number of sweep units. A reduced number could allow the hardware to fit into mobile GPUs, much like PLOCTree [Viitanen et al. 2018], at the cost of build performance. On the other hand, the current design is small enough to fit into desktop and console GPUs. In this case, the number of sweep units could even be increased for further performance gains while still remaining a small percentage of the overall power consumption.

This thesis has shown an implementation of a hardware BVH builder based on the state-of-the-art construction algorithm PLOC++. While the current configuration has yet to be synthesized, the overall design has potential for performance improvements over the original for ray tracing in both desktops, consoles, and possibly smartphones.

Bibliography

Tomas Akenine-Möller, Eric Haines, Naty Hoffman, Angelo Pesce, Michat Iwanicki, and Sébastien Hillaire. *Real-Time Rendering, Fourth Edition*. A. K. Peters, Ltd., USA, 4th edition, 2018. ISBN 0134997832.

Carsten Benthin, Radoslaw Drabinski, Lorenzo Tessari, and Addis Dittebrandt. Ploc++ parallel locally-ordered clustering for bounding volume hierarchy construction revisited. *Proceedings of the ACM on Computer Graphics and Interactive Techniques*, 5(3):1–13, 2022.

John Burgess. Rtx onthe nvidia turing gpu. *IEEE Micro*, 40(2):36–44, 2020. doi: 10.1109/MM.2020.2971677.

Yangdong Deng, Yufei Ni, Zonghui Li, Shuai Mu, and Wenjun Zhang. Toward real-time ray tracing: A survey on hardware acceleration and microarchitecture techniques. *ACM Comput. Surv.*, 50(4), aug 2017. ISSN 0360-0300. doi: 10.1145/3104067. URL `https://doi.org/10.1145/3104067`.

Michael J. Doyle, Colin Fowler, and Michael Manzke. A hardware unit for fast sah-optimised bvh construction. *ACM Trans. Graph.*, 32(4), jul 2013. ISSN 0730-0301. doi: 10.1145/2461912.2462025. URL `https://doi.org/10.1145/2461912.2462025`.

Hadi Esmaeilzadeh, Ting Cao, Yang Xi, Stephen M. Blackburn, and Kathryn S. McKinley. Looking back on the language and hardware revolutions: Measured power, performance, and scaling. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, page 319332, New York, NY, USA, 2011. Association for Computing Machinery. ISBN 9781450302661. doi: 10.1145/1950365.1950402. URL `https://doi.org/10.1145/1950365.1950402`.

Scott Hauck and André DeHon. *Reconfigurable Computing: The Theory and Practice of FPGA-Based Computation*. Morgan Kaufmann, Amsterdam, 1st edition, 2008. ISBN 9780123705228.

IEEE. Ieee standard for systemverilog–unified hardware design, specification, and verification language. *IEEE Std 1800-2017 (Revision of IEEE Std 1800-2012)*, pages 1–1315, 2018. doi: 10.1109/IEEESTD.2018.8299595.

C. Lauterbach, M. Garland, S. Sengupta, D. Luebke, and D. Manocha. Fast bvh

construction on gpus. *Computer Graphics Forum*, 2009. ISSN 1467-8659. doi: 10.1111/j.1467-8659.2009.01377.x.

Xingyu Liu, Yangdong Deng, Yufei Ni, and Zonghui Li. Fasttree: A hardware kd-tree construction acceleration engine for real-time ray tracing. In *2015 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1595–1598, 2015.

David J. MacDonald and Kellogg S. Booth. Heuristics for ray tracing using space subdivision. *The Visual Computer*, 6:153–166, 1990.

Daniel Meister and Jií Bittner. Parallel locally-ordered clustering for bounding volume hierarchy construction. *IEEE Transactions on Visualization and Computer Graphics*, 24(3):1345–1353, 2018. doi: 10.1109/TVCG.2017.2669983.

Daniel Meister, Shinji Ogaki, Carsten Benthin, Michael J. Doyle, Michael Guthe, and Jií Bittner. A survey on bounding volume hierarchies for ray tracing. *Computer Graphics Forum*, 40(2):683–712, 2021. doi: https://doi.org/10.1111/cgf.142662. URL `https://onlinelibrary.wiley.com/doi/abs/10.1111/cgf.142662`.

Guy M Morton. A computer oriented geodetic data base and a new technique in file sequencing. 1966.

Jae-Ho Nah, Hyuck-Joo Kwon, Dong-Seok Kim, Cheol-Ho Jeong, Jinhong Park, Tack-Don Han, Dinesh Manocha, and Woo-Chan Park. Raycore: A ray-tracing hardware architecture for mobile devices. 33(5), sep 2014. ISSN 0730-0301. doi: 10.1145/2629634. URL `https://doi.org/10.1145/2629634`.

Jae-Ho Nah, Jin-Woo Kim, Junho Park, Won-Jong Lee, Jeong-Soo Park, Seok-Yoon Jung, Woo-Chan Park, Dinesh Manocha, and Tack-Don Han. Hart: A hybrid architecture for ray tracing animated scenes. *IEEE Transactions on Visualization and Computer Graphics*, 21(3):389–401, 2015. doi: 10.1109/TVCG.2014.2371855.

Anuj Pathania, Qing Jiao, Alok Prakash, and Tulika Mitra. Integrated cpu-gpu power management for 3d mobile games. In *Proceedings of the 51st Annual Design Automation Conference*, DAC '14, page 16, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450327305. doi: 10.1145/2593069.2593151. URL `https://doi.org/10.1145/2593069.2593151`.

Timo Viitanen, Matias Koskela, Pekka Jääskeläinen, Heikki Kultala, and Jarmo Takala. Mergetree: A fast hardware hlbvh constructor for animated ray tracing. *ACM Trans. Graph.*, 36(5), oct 2017. ISSN 0730-0301. doi: 10.1145/3132702. URL `https://doi.org/10.1145/3132702`.

Timo Viitanen, Matias Koskela, Pekka Jääskeläinen, Aleksi Tervo, and Jarmo Takala. Ploctree: A fast, high-quality hardware bvh builder. *Proceedings of the ACM on Computer Graphics and Interactive Techniques*, 1(2):1–19, 2018.

Marek Vinkler, Jiri Bittner, and Vlastimil Havran. Extended morton codes for high performance bounding volume hierarchy construction. In *Proceedings of High Performance Graphics*, HPG '17, New York, NY, USA, 2017. Association

for Computing Machinery. ISBN 9781450351010. doi: 10.1145/3105762.3105782. URL https://doi.org/10.1145/3105762.3105782.

Ingo Wald. On fast construction of sah-based bounding volume hierarchies. In *2007 IEEE Symposium on Interactive Ray Tracing*, pages 33–40, 2007. doi: 10.1109/RT.2007.4342588.

Bruce Walter, Kavita Bala, Milind Kulkarni, and Keshav Pingali. Fast agglomerative clustering for rendering. In *2008 IEEE Symposium on Interactive Ray Tracing*, pages 81–86, 2008. doi: 10.1109/RT.2008.4634626.

Sven Woop, Erik Brunvand, and Philipp Slusallek. Estimating performance of a ray-tracing asic design. In *2006 IEEE Symposium on Interactive Ray Tracing*, pages 7–14, 2006. doi: 10.1109/RT.2006.280209.

Xilinx. *PYNQ-Z2 User Manual*, 2018. URL https://www.mouser.com/datasheet/2/744/pynqz2_user_manual_v1_0-1525725.pdf.