FastTree: A Hardware KD-Tree Construction Acceleration Engine for Real-Time Ray Tracing

Xingyu Liu, Yangdong Deng, Yufei Ni, Zonghui Li
Institute of Microelectronics
Tsinghua University
{liuxingyu11, niyf13, lizonghui11}@mails.tsinghua.edu.cn, dengyd@tsinghua.edu.cn

***Abstract*—The ray tracing algorithm is well-known for its ability to generate photo-realistic rendering effects. Recent years have witnessed a renewed momentum in pushing it to real-time for better user experience. Today the construction of acceleration structures, e.g., kd-tree, has become the bottleneck of ray tracing. A dedicated hardware architecture, FastTree, was proposed for kd-tree construction by adopting a fully parallel construction algorithm. FastTree was validated by an FPGA prototype and evaluated as an ASIC implementation. Experiment result shows FastTree outperforms existing hardware construction engines by a factor of nearly 4X at a similar area and power budget.**

Keywords*—*ray tracing, kd-tree, kd-tree construction, hardware acceleration, real-time ray tracing

I. Introduction

Ray tracing has long been considered as a promising technique to deliver more photo-realistic visual effects [1]. Recently, real-time ray tracing based rendering gains a new wave of momentum for its ability to generate more realistic effects at a lower level of power consumption. At the present time, the excessive computation time is the major hurdle for ray tracing to get wide acceptance in consumer level graphics platforms. In fact, A resolution of 2560×1440 requires a tracing throughput of over two billion rays per second [2], while today's leading desktop GPUs can only sustain 300M rays per second. The ray tracing algorithm requires an acceleration structures, which has to be constructed for each scene in dynamic applications, to store the graphics primitives for fast spatial lookup. Figure 1 shows the time distribution of ray tracing 6 benchmarks in both construction and traversal [3]. The results suggest that the construction process actually dominates the rendering time.

As a result, it is essential to develop ray tracing solutions to enable real-time performance but at an area and power budget that fits into a mobile platform. Hardware accelerators seem to be the only viable path towards such a goal. Most of the existing works, however, focus on accelerating ray traversal and leave the construction of acceleration structure to a CPU [4], [5]. One early work on expediting kd-tree construction was proposed by adopting several dedicated instructions for kd-tree construction [4]. Later works were more oriented to custom hardware. One recent work [6] developed customized hardware IP for BVH construction. It consumes a silicon estate of around $40mm^2$ at a 65nm process and achieves a construction speed that is 2-4 times slower than a leading-edge software constructor [7] running on an NVIDIA Fermi GPU. A light-weighted kd-tree construction hardware solution was proposed by Nah et al. [8]. It is based on a binning based approach that trades tree quality for construction time. The proposed construction unit has a clear advantage in die area, but the construction speed is still limited.

Targeting real-time ray tracing applications, the objective of this work is to develop a hardware accelerator that is advantageous in both performance and hardware cost. Now kd-tree and bounding volume hierarchy (BVH) are the two most popular acceleration structures, with the former offering better traversal efficiency at a longer construction time. Li et al. [9] introduced a fully parallel kd-tree construction algorithm, which for the first time allowed kd-trees to be built at a faster speed than BVHs without loss of traversal efficiency. We proposed FastTree, a hardware architecture using the kd-tree construction algorithm developed by Li et al. [9]. FastTree takes a scene and then builds the corresponding tree that can be used by ray traversing engines like T&I Engine [5] and RayCore [8]. To the best of the authors' knowledge, FastTree achieves the highest area- and power-efficiency. When backed up by the same dual-channel LPDDR3 memory [10], FastTree outperforms a cutting-edge ray tracing accelerator by a factor of nearly 4X at a similar area and power budget.

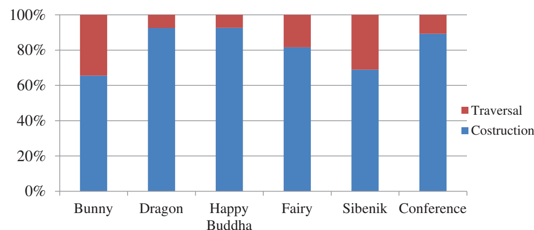

Fig. 1: Time Decomposition of Ray Tracing.

II. Path Compression Algorithm

The ray tracing algorithm emulates the vision formation process by shooting rays from an observer towards a 3-D scene. Given a scene consisting of graphic primitives (triangles in this work), a kd-tree is built by hierarchically partitioning the space according to the spatial distribution of the primitives. Given a ray, a traversal of the kd-tree will return the intersected triangles (if any). Such a kd-tree traversal has complexity of $O(\log(n))$, where n is the number of primitives. In a kd-tree, leaf nodes represent sub-spaces covered by a group of triangles, while internal nodes correspond to split planes. Interested readers please refer to references [1] and [11] for a detailed treatment.

Previous kd-tree construction algorithms typically adopted a top-down approach by recursively dividing the space and allocating the primitives accordingly. Such approaches suffer from an insufficient level of parallelism in upper levels. Li et al. developed a fully parallel kd-tree construction algorithm and reported the preliminary results in [9]. As illustrated in Figure 2, the algorithm starts from a conceptually complete binary tree, which corresponds to a regular division of the space into grid cells, and then concurrently working on all leaf nodes in a bottom-up manner. The algorithm uses the unique characteristics of the Morton code [12] to identify redundant spatial partitions as shown in Figure 2. As a result,

978-3-9815370-4-8/DATE15/©2015 EDAA

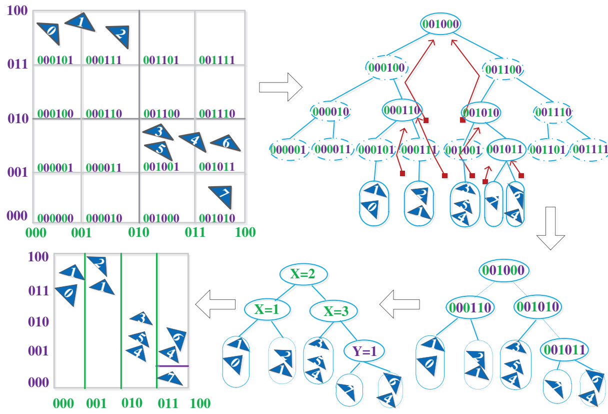

Fig. 2: The procedures of the algorithm.

the original complete tree is compressed by removing the redundant partitions (i.e., internal nodes of kd-tree).

The proposed algorithm consists of 6 steps: 1) Scene Bound Calculation, 2) Morton Code Generation and Triangle Duplication, 3) Morton Code Sort, 4) Leaf Node Generation, 5) Internal Node Generation, 6) Path Compression. Interested readers please refer to [9] for algorithmic details.

III. Overall Architecture and Module Implementation

Figure 3 on the next page illustrates the overall architecture of the proposed FastTree accelerator. It consists of eight types of data-processing modules, which are coordinated by a global control unit. These modules exchanged data via a high-speed bus. An off-chip LPDDR3 memory is also attached to the bus. All operands in FastTree have a width of 32 bits.

A. Scene Bound Calculation Unit (SBCU)

SBCU takes eight 32-bit floating point values as input. It consists of three fully-pipelined stages of 2-input floating-point comparator and 2-to-1 multiplexor (MUX) and performs 8-to-1 selection to select the largest or smallest value among 8 inputs. Then SBCU compares the selected value with current upper or lower bound and update bound value. We implemented two similar SBCUs for computing the lower and upper bounds in parallel.

B. Prefix Sum Unit (PSU)

Computing the prefix sum of an array is a frequent operation of the construction process and efficient PSUs are crucial for the overall performance. PSU chops a long array into shorter sections, performs prefix sum operations on each section in parallel and then merges these partial results by adding a specific offset value to all elements in the section. We implemented four PSUs operating in parallel. Following the computing flow proposed in [13], the operation of PSU consists of two pipeline stages, up-sweep and down-sweep, as illustrated in Figure 4. We chose the length of a section to be 32, so each stage of the pipeline consists of thirty-two 32-bit registers.

C. Morton Code Generation Unit (MCGU)

MCGU unit generates Morton code and copies triangles if necessary. MCGU takes nine 32-bit coordinates of a triangle and computes the bounding box of the triangle using three

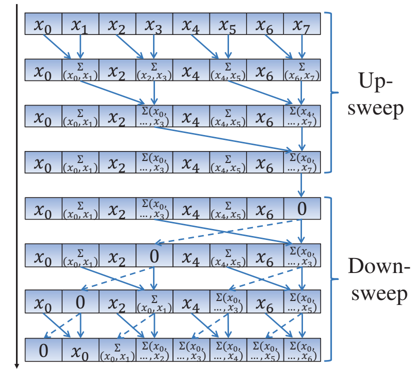

Fig. 4: Computing flow of prefix sum of an 8-element array

3-input bound calculation units similar to SBCU. The six bounding values are translated into partial Morton codes. MCGU iterates all cells inside the bounding box to derive all Morton code words. In this work, we use four MCGUs that operate in parallel. The generation of new partial Morton code of bounding coordinate values will be stalled when MCGU iterates the bounding box. Its output are directly sent to PSUs for radix sorting.

D. Radix Sort Unit (RSU)

The sorting of the Morton code is a bottleneck of the algorithm in its GPU implementation [9]. We designed a radix sort unit to order the array of Morton code words. The parallel radix sort process is based on the algorithm introduced in [14] and [15]. It is designed as several rounds of counting sort with each round handling a different digit of the key. In these rounds, we move from the least significant digit to the most significant digit. In this work, a radix value of 16 is selected. So we chose a digit width of 4 bits and eight rounds of counting sort. According to [14] and [15], the counting sort can be realized as prefix sum computation. The PSUs are reused and the RSU actually serves as the control logic.

E. Leaf Node Generate Unit (LNGU)

LNGU contains thirty-two pairs of 2-input integer comparator and 2-to-1 MUX operating in parallel to obtain the base address The integer comparator fetches two Morton code words that are adjacent in the sorted array from external memory. Their indices in the array are loaded together. If two input Morton code words are different, the index and value of the larger Morton code word will be written to the buffer pool. To obtain the size of a segment (i.e., the number of triangles in a leaf node), LNGU also has thirty-two 32-bit integer subtraction units. The inputs of a subtraction unit are the base indices of two adjacent chunks from the buffer pool.

F. Internal Node Generate Unit (INGU) and Bit Encoding Unit (BEU)

In INGU, a right shifter working iteratively is used to indicate the left-most different bit in the two input Morton codes and then identify a split plane. Since the data width is 32 bits in our system and only one bit is needed to mark whether a node is *essential*, i.e. whether it appears in the final compressed kd-tree, one operand in FastTree is actually the union of 32 marks. We implemented a Bit Encoding Unit (BEU) for the related bit operations. The output of INGU is sent to BEU directly. The encoded results are written to off-chip memory via an on-chip cache.

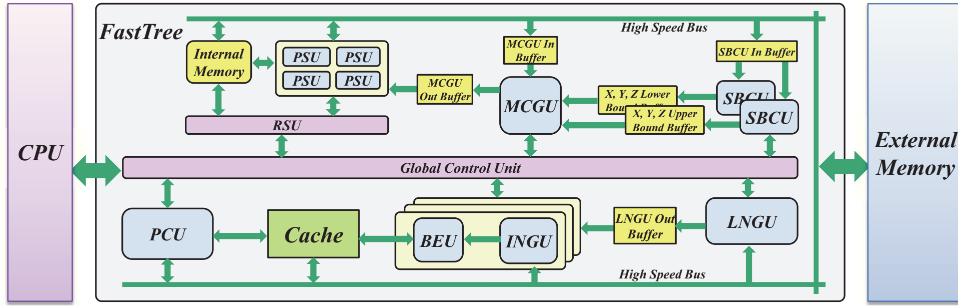

Fig. 3: The overall architecture of FastTree

G. Path Compression Unit (PCU)

PCU takes the Morton code value of a node and calculates its parent in the compressed kd-tree. A left shifter is used to obtain the right-most non-zero bit of the Morton code. Suppose it is the i-th bit. At the i-th bit, PCU judges if the current node is the left or right child of its parent by looking up its (i+1)-th bit. PCU also checks if its direct parent is essential by looking up the encoded bit produced by INGU and BEU in the cache. If not, PCU continues to check the ancestor nodes upwards until reaching an essential node, which is the actual parent node in the compressed tree.

H. On-chip Cache

FastTree is equipped with a 16KB on-chip cache to mark if a node is essential. Deployed between BEU and PCU, it is configured as a 16-way set associative cache with a block size of 8 bytes and a least-recently-used update policy.

IV. Hardware Implementation

In this section, we describe the hardware implementation of FastTree. FastTree takes a scene from PC as input and the constructed kd-tree will be sent back to a PC for validation. Note that the resultant kd-trees can be fed to ray traversal engines like T&I Engine [5] and RayCore [8] in real-world applications. We follow the methodology used by Nah et al. ([5] and [8]) to assess the performance by first developing an FPGA prototype and then performing an ASIC evaluation.

A. FPGA Prototype

FastTree was deployed on a Virtex-7 VC707 evaluation board equipped with a Xilinx Virtex-7 XC7VX485 FPGA and 1GB DDR3 memory. The Virtex-7 FPGA used in this work is equipped with 1,030 36Kb block RAM modules and supports up to 8,175 Kb of distributed RAMs. The evaluation board provides an Ethernet link for the FPGA to communicate with a host PC. We use the Ethernet interface to download scene data to FPGA and collect constructed kd-tree for verification. The implementation of the proposed kd-tree constructor runs at a speed of 163MHz. The usage of hardware resources is listed in Table I.

B. ASIC Evaluation

We use TSMC 65nm process and Synopsys Design Compiler to evaluate the ASIC evaluation. FastTree can be synthesized with a clock frequency of up to 720 MHz at a voltage of 1 V. We set the frequency to 500 MHz for a conservative evaluation. The total area of FastTree is 5.4 mm^2. The total power consumption is 3.9 W.

TABLE I: Hardware Resources Usage Result

Module	LUT	Slice Register	Distributed RAM (kb)	Block RAM (kb)
SBCU	2485	1415	0	0
PSU	11454	5702	0	0
MCGU	12454	7561	0	0
RSU	16530	50282	0	0
LNGU	8220	4641	0	0
INGU	9093	5727	0	0
BEU	1157	1086	0	0
PCU	7251	5562	0	0
Cache	21744	34295	2	172
Overall	173391	180057	2	172

V. Experiment and Results

In this section, we report the kd-tree construction performance of the FPGA prototype and the ASIC version respectively. We also compare the ASIC version of FastTree with previous kd-tree construction works.

A. Performance of the FPGA Prototype

The performance of FastTree's FPGA prototype is illustrated in Table II. The scenes for benchmarking FastTree as well as their rendering results are shown in Figure 5. These scenes have varying complexity from 69K triangles to over 1M triangles.

B. Performance Evaluation of ASIC Version and Comparison with Previous Work

The FPGA results proved that, when working with a traversal engine in parallel, the ASIC version of FastTree at 500 MHz could sustain a frame rate of at least 58 FPS on scenes with less than 300K triangles. For large benchmark Dragon and Buddha, it is still feasible to enable a frame rate of more than 19 and 15 FPS respectively. The frame rate results are listed in the 5th column of Table II. The resultant frame rates require a memory bandwidth of 1.15 GB/s, only 9% of the peak bandwidth of mobile dual-channel LPDDR3 (12.8 GB/s) [10].

We compared the performance of FastTree with that of RayCore. The kd-tree construction performance of RayCore was evaluated on small benchmarks (0.6K−64 K triangles) that are not publicly available. In addition, the kd-tree construction time of RayCore grows approximately linearly with the number of triangles [8]. Accordingly, we selected public benchmark scenes with similar triangle counts (Bunny, 69K) for comparison. Table III compares FastTree with RayCore and previous kd-tree construction works on CPUs and GPUs.

FastTree enables a substantial advantage in performance

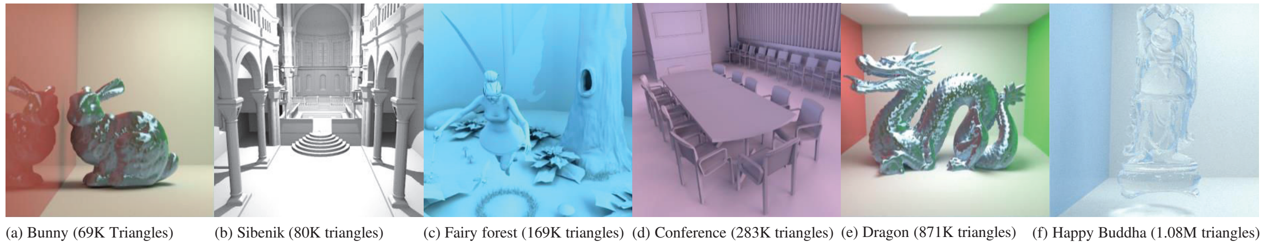

Fig. 5: Test scenes for kd-tree construction. The generated trees are validated by a ray tracing rendering engine [16]

TABLE III: Comparison with previous works on kd-tree construction

	CPU approaches		GPU approach	Dedicated H/W		
	Shevtsov et al. [17]	Choi et al. [18]	Wu et al. [3]	RayCore [8]	**FastTree (ours)**	
Scene (triangle count)	Bunny (69K)	Bunny (69K)	Bunny (69K)	Transparent Shadows (64K)	Bunny (69K)	
tree construction time (ms)	27	50	59	20	**5.1**	
Platform	Intel Core2 Duo ×2	Intel Xeon X7550 ×4	NVIDIA Geforce GTX280	ASIC	ASIC 65nm	ASIC scaled to 28nm
Clock (MHz)	3000	2000	48 (core)	500	500	500
Process (nm)	16	45	55	28	65	28
Area (mm^2)	143 × 2	684 × 4	576	1.6	**5.4**	**≤1.4**
Power (W)	65 ×2	130 ×4	236	1	**3.9**	**0.8**

TABLE II: Kd-tree construction performance

Module	Number of triangles	Kd-tree build time (ms)		FPS on ASIC
		FPGA Prototype (163MHz)	ASIC (500 MHz)	
Bunny	69451	15.6	5.1	196
Sibenik	79983	20.1	6.6	152
Fairy forest	169711	33.0	10.8	92
Conference	282751	52.7	17.2	58
Dragon	871414	160.2	52.2	19
Happy Buddha	1087716	200.9	65.5	15

per unit area and per unit power over CPU- or GPU-based software implementations. It also outperforms RayCore by a factor of 3.9. We scaled FastTree to a TSMC 28nm process so that it can be compared with RayCore at the same technology node. TSMC suggested that a total down scaling factor of at least 4 in area is available when migrating from 65nm to 40nm [19] and then to 28nm [20]. The power was scaled down similarly but with the additional consideration of the reduction of supply voltage (1V in 65nm vs. 0.9V in 28nm). The resultant area and power of FastTree are less than $1.4mm^2$ and approximately 0.8 W, respectively.

References

[1] M. Pharr and G. Humphreys, *Physically Based Rendering, Second Edition: From Theory To Implementation*, 2nd ed. Morgan Kaufmann, 2010.

[2] Y. Deng, “Mining Hidden Coherence for Parallel Path Tracing on GPUs,” In Proceedings of GPU Technology Conference, 2014.

[3] Z. Wu, F. Zhao, and X. Liu, “SAH KD-tree construction on GPU,” in *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics (HPG 11)*, 2011, pp. 71–78.

[4] S. Woop, J. Schmittler, and P. Slusallek, “RPU: A Programmable Ray Processing Unit for Realtime Ray Tracing,” in *Proceedings of ACM SIGGRAPH*, 2005, pp. 434–444.

[5] J.-H. Nah, J.-S. Park, C. Park, J.-W. Kim, Y.-H. Jung, W.-C. Park, and T.-D. Han, “T&I Engine: Traversal and Intersection Engine for Hardware Accelerated Ray Tracing,” *ACM Transactions on Graphics - Proceedings of ACM SIGGRAPH Asia 2011*, vol. 30, no. 160, 2011.

[6] M. J. Doyle, C. Fowler, and M. Manzke, “A hardware unit for fast SAH-optimised BVH construction,” *ACM Transactions on Graphics*, vol. 32, no. 4, 2013.

[7] K. Garanzha, J. Pantaleoin, and D. Mcallister, “Simpler and faster HLBVH with work queues,” in *High Performance Graphics*, 2011, pp. 59–64.

[8] J.-H. Nah, H.-J. Kwon, D.-S. Kim, C.-H. Jeong, J. Park, T.-D. Han, D. Manocha, and W.-C. Park, “RayCore: A ray-tracing hardware architecture for mobile devices,” *ACM Transactions on Graphics*, vol. 30, no. 6, 2014.

[9] Z. Li, T. Wang, and Y. Deng, “Fully Parallel Kd-Tree Construction for Real-Time Ray Tracing,” in *Proceedings of ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, 2014.

[10] Micron Technology Inc, “Point-to-Point System Design: Layout and Routing Tips for LPDDR2 and LPDDR3 Devices,” Technical Note, http://www.micron.com/-/media/documents/products/technical%20note/dram/lpddr3/tn_5202_lpddr3_design_layout.pdf, 2013.

[11] A. S. Glassner, Ed., *An Introduction to Ray Tracing*. London, UK, UK: Academic Press Ltd., 1989.

[12] G. Morton, “A computer oriented geodetic data base and a new technique in file sequencing,” Tech. Rep. Ottawa, ON, Canada, 1966.

[13] M. Harris, S. Sengupta, and J. D. Owens, *Parallel Prefix Sum (Scan) with CUDA*. In GPU Gems 3, 2007.

[14] L. Ha, J. Krueger, and C. Silva, “Fast 4-way parallel radix sorting on GPUs,” *Computer Graphics Forum*, vol. 28, no. 8, 2009.

[15] M. C. Delorme, T. S. Abdelrahman, and C. Zhao, “Parallel radix sort on the amd fusion accelerated processing unit,” in *Proceedings of International Conference on Parallel Processing*, 2013, pp. 339–348.

[16] Y. Wang, C. Liu, and Y. Deng, “A Feasibility Study of Ray Tracing on Mobile GPUs,” accepted by SIGGRAPH Asia Symposium On Mobile Graphics And Interactive Applications, 2014.

[17] M. Shevtsov, A. Soupikov, and A. Kapustin, “Highly parallel fast kd-tree construction for interactive ray tracing of dynamic scenes,” in *Proceedings of EUROGRAPHICS 2007, Computer Graphics Forum*, 2007, pp. 395–404.

[18] B. Choi, R. Komuravelli, V. Lu, H. Sung, R. L. Bocchino, S. V. Adve, and J. C. Hart, “Parallel sah k-d tree construction,” in *Proceedings of the Conference on High Performance Graphics*, 2010, pp. 77–86.

[19] TSMC, “40nm Technology,” Technical Report. http://www.tsmc.com/english/dedicatedFoundry/technology/40nm.htm.

[20] ——, “28nm Technology,” Technical Report. http://www.tsmc.com/english/dedicatedFoundry/technology/28nm.htm.