On the Efficient Implementation of a Real-time Kd-tree Construction Algorithm

Byungjoon Chang[†], **Woong Seo**, Insung Ihm

Department of Computer Science and Engineering
Sogang University, Korea

[†]Currently at Samsung Electronics, Digital Media & Communications R&D Center

Motivation

- **Kd-tree for ray tracing**
  - Known as one of the most effective structures for accelerating ray-object intersection calculations.
  - Constructed in a top-down manner by recursively subdividing the scene's bounding volume into two subvolumes using axis-aligned splitting planes.
  - **The construction cost is relatively higher** compared to other structures such as grids and bounding volume hierarchies.

- **Parallel kd-tree construction for real-time ray tracing**
  - Zhou et al. developed the first real-time kd-tree construction algorithm on the GPU [Zhou et al. 2008].
  - Their parallel algorithm is still effective, but their implementation details on the GPU are rather outdated.
  - Need **more efficient implementation techniques suitable for newer GPUs**.

Our Contributions

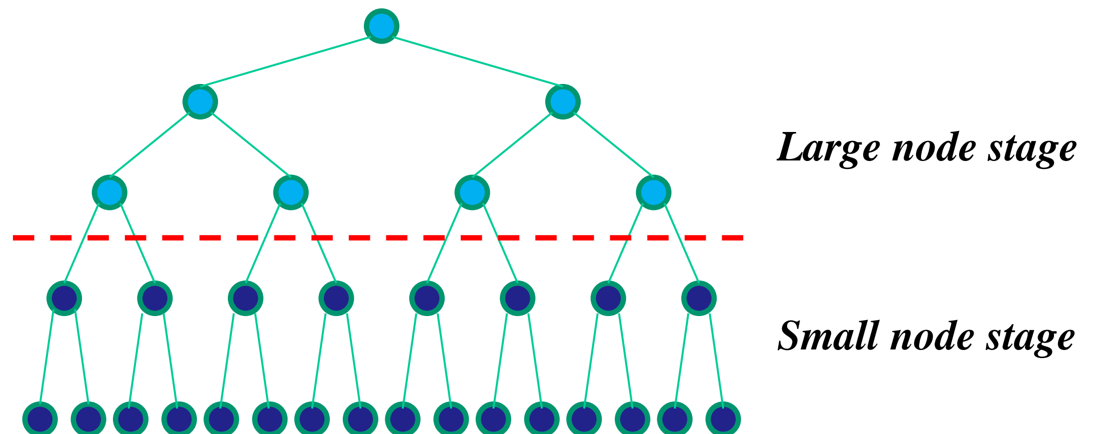
- Present **enhanced CUDA programming techniques** for implementing the Zhou et al.'s algorithm more efficiently on current GPUs.

  - Alleviate the overheads from multiple synchronous kernel calls by **reducing the number of necessary kernel functions markedly using simple per-block atomic operations**.

  - **Exploit recently introduced intrinsic functions of the CUDA API** for efficient implementation.

GPU-assisted KD-Tree Construction [Zhou et al. 2008]

- *Zhou et al., Real-time kd-tree construction on graphics hardware, ACM TOG, Vol. 27, No. 5, 2008.*
- **Two-stage kd-tree construction algorithm**
  - **Large node stage**
    - For the upper levels of kd-tree, apply a combination of **spatial median splitting** and **empty-space maximizing, parallelized over the triangles**.
  - **Small node stage**
    - When all nodes contain fewer than 64 triangles, apply **a variant of SAH-based node splitting, parallelized over the small nodes**.



Large Node Stage [Zhou et al. 2008]

- **The most time-consuming parts are**

```
Algorithm 2  Large Node Stage
procedure ProcessLargeNodes(
        in activelist:list;
        out smalllist, nextlist:list)
begin
    // group triangles into chunks
    for each node i in activelist in parallel
        Group all triangles in node i into fixed size chunks, store
        chunks in chunklist

    // compute per-node bounding box
    for each chunk k in chunklist in parallel
        Compute the bounding box of all triangles in k, using stan-
        dard reduction
    Perform segmented reduction on per-chunk reduction result to
    compute per-node bounding box

    // split large nodes
    for each node i in activelist in parallel
        for each side j of node i
            if i contains more than C_e empty space on
              side j then
               Cut off i's empty space on side j
        Split node i at spatial median of the longest axis
        for each created child node ch
            nextlist.add(ch)

    // sort and clip triangles to child nodes
    for each chunk k in chunklist in parallel
        i ← k.node()
        for each triangle t in k in parallel
            if t is contained in both children of i then
                t_0 ← t
                t_1 ← t
                Sort t_0 and t_1 into two child nodes
                Clip t_0 and t_1 to their respective owner node
            else
                Sort t into the child node containing it

    // count triangle numbers for child nodes
    for each chunk k in chunklist in parallel
        i ← k.node()
        Count triangle numbers in i's children, using reduction
    Perform segmented reduction on per-chunk result to compute
    per-child-node triangle number

    // small node filtering
    for each node ch in nextlist in parallel
        if ch is small node then
            smalllist.add(ch)
            nextlist.delete(ch)
end
```

Parallel triangle sorting w.r.t. splitting planes

Parallel counting of triangle numbers in child nodes

Triangle Sorting with Respect to Splitting Planes

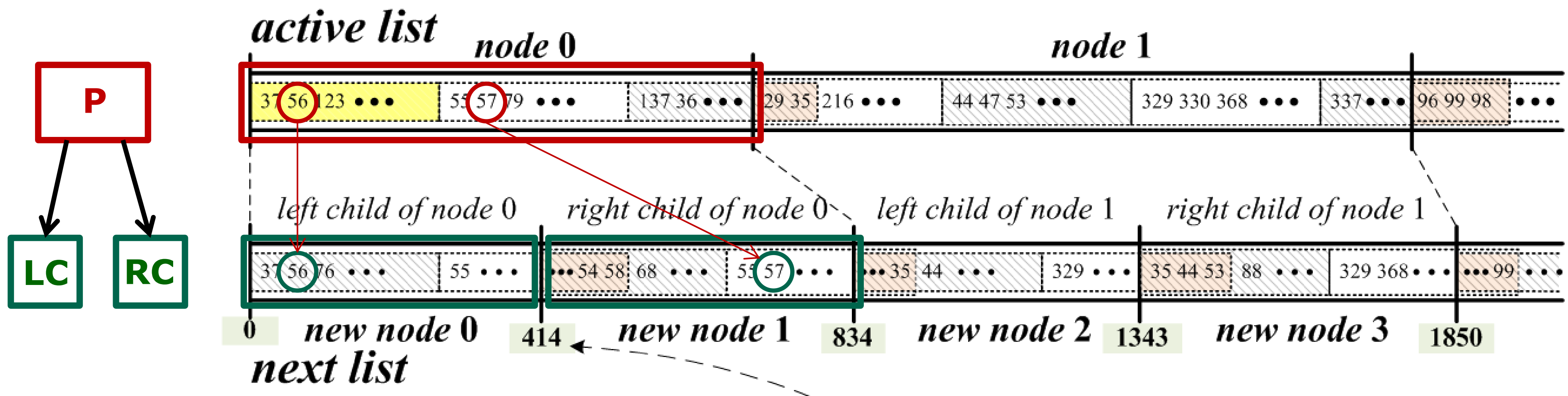
- **Implementation I**
  - A **four-kernel** implementation based on *standard (segmented) scan and reduction primitives*
  - Each segmented scan and reduction requires multiple kernel function calls.

- **Implementation II**
  - A **single-kernel** implementation based on *per-block atomic operations*
  - Enhance the GPU performance by minimizing the overheads caused by multiple synchronous kernel calls.

Implementation I: Using Standard Data-parallel Primitives

- For each triangle in a large node, how do we efficiently **calculate its address(es)** ***in parallel*** **in the new child node(s)** into which it is sorted?

- Can be **implemented in four kernel functions** using (segmented) scan and reduction.

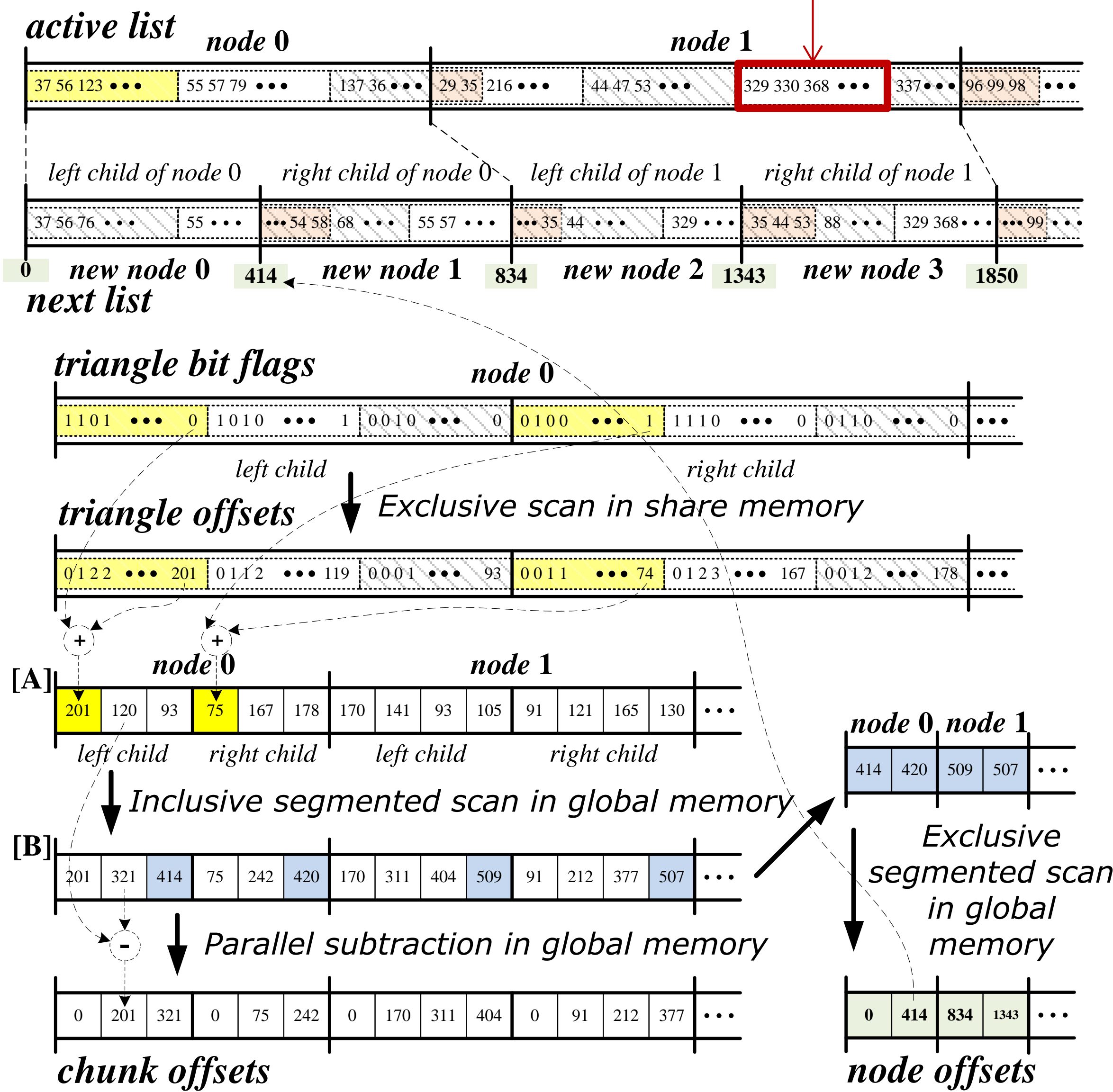


[Kernel 1] Classify each triangle w.r.t. the splitting plane, generating ***triangle bit flags***. Perform *an exclusive scan* and an addition to build ***triangle offsets*** and the array **[A]**.

[Kernel 2] Perform *an inclusive segmented scan* and parallel subtractions to build the array **[B]** and ***chunk offsets***.

[Kernel 3] Perform *an exclusive segmented scan* over the last element of each child node in **[B]** to build ***node offsets***.

[Kernel 4] For a triangle whose bit flag is on in ***triangle bit flags***, store its triangle index in ***next list***, whose address is calculated using ***node offsets***, ***chunk offsets***, and ***triangle offsets***.



Implementation II: Using Per-block Atomic Operations

- **Problems with Implementation I**
  - Require to **perform a segmented scan twice** on the data sequences in the global memory.
  - Need to **split the run-time execution into a sequence of synchronous kernels calls**.

  ➢ The kernel call overheads impact the run-time performance negatively.

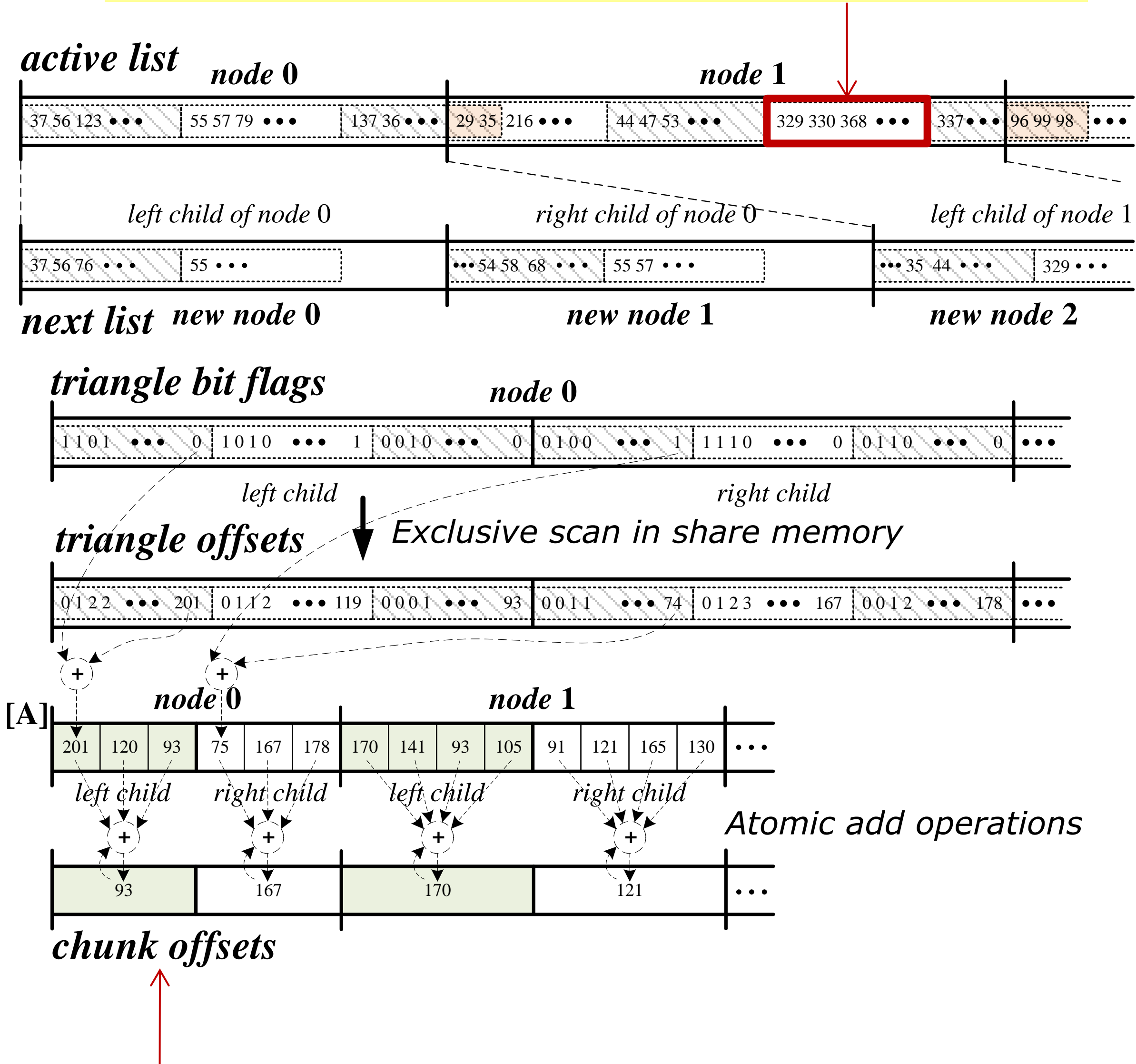
- **Observations**
  - By using a standard segmented scan, **the relative order of triangle indices within a node is preserved** in the respective child nodes.
  - This property **is not necessary in the kd-tree construction algorithm**!

  ➢ Replace the segmented-scan operations with **simpler per-chunk atomic operations for a single faster-running kernel**.

[Step 1] Classify each triangle w.r.t. the splitting plane, generating ***triangle bit flags***. Perform *an exclusive scan in shared memory* and an addition to build ***triangle offsets*** and the array **[A]**.

[Step 2] A representative thread performs *two atomic operations*, respectively fetching the local offsets, one for each child node, from the corresponding atomic variables in ***chunk offsets***. At the same time, add the triangle counts to them.

[Step 3] Once per child node, for a triangle whose bit flag is on in ***triangle bit flags***, store its triangle index in ***next list***, whose address is calculated using the fetched offset and ***triangle offsets***.



An integer-valued array to hold the current local offset per each child node.

- In the new implementation,
  - **Two segmented scans** over the arrays **in the global memory** are replaced by **two atomic-add operations per thread block**.
- Still, **two per-block (per-chunk) scans** must be performed **in the shared memory**, one for each child.
  - The recent GPUs offer several intrinsic operations:
    - `__ballot()` for warp voting, `__popc()` for bit counting, `__shfl()` for warp shuffling
  - Use them for efficient implementation of the per-block scan.

```
/* Perform intra-warp scan. */
unsigned int LeftMask = ballot(LChildTriBitFlag), LaneMaskLT = 0;
unsigned int RightMask = ballot(RChildTriBitFlag), LaneMaskLE = 0;

asm("mov.u32 %0, %%lanemask lt;" : "=r"(LaneMaskLT));
asm("mov.u32 %0, %%lanemask le;" : "=r"(LaneMaskLE));

LChildTriOffsets = popc(LeftMask & LaneMaskLT);
RChildTriOffsets = popc(RightMask & LaneMaskLT);

if (LaneID == 31) {
   LChildTriOffsets2[(threadIdx.x >> 5) + 1] = popc(LeftMask & LaneMaskLE);
   RChildTriOffsets2[(threadIdx.x >> 5) + 1] = popc(RightMask & LaneMaskLE);
}
syncthreads();
```

```
/* Perform inter-warp scan. */
float Scan8[2];

if (threadIdx.x < 8) {
   Scan8[0] = LChildTriOffsets2[threadIdx.x + 1];
   Scan8[1] = RChildTriOffsets2[threadIdx.x + 1];
   for (int i = 1; i <= 4; i *= 2) {
      float n0 = shfl up(Scan8[0], i, 8);
      float n1 = shfl up(Scan8[1], i, 8);
      if (LaneID >= i) {
         Scan8[0] += n0; Scan8[1] += n1;
      }
   }
}
if (threadIdx.x < 8) {
   LChildTriOffsets2[threadIdx.x + 1] = Scan8[0];
   RChildTriOffsets2[threadIdx.x + 1] = Scan8[1];
}
```

Optimization for AABB Computations

- For every large node step, **the AABB of all triangles in each node** must be calculated.

- **The original implementation** requires **per-block reductions** in the shared memory and **six segmented reductions** over the array in the global memory.

chunks in *chunklist*

// compute per-node bounding box
for each chunk k in *chunklist* **in parallel**
  Compute the bounding box of all triangles in k, using standard reduction
Perform segmented reduction on per-chunk reduction result to compute per-node bounding box

// split large nodes

- **In our implementation**,
  - Use the intrinsic shuffle function **`__shfl_up`**`()` for efficient per-block reductions.
  - Use **three pairs of atomic min and max operations** per chunk to replace the six segmented reductions.

Optimization for Small Node Stage

- To lower the memory latency on the GPU, use **efficient memory layout for the representation of small root nodes**.
  - 4-byte access from global memory for the splitting plane location
  - 16-byte access from texture memory for the triangle sets
  - ➢ Details in the paper.

- The small-node-splitting step is performed using simple bitwise operations.
  - **The original implementation** suggests to use **the method [Manku 2002] for a repeated parallel bit-counting operation**.
  - **In our implementation**, use **the __popc() intrinsic function** for efficient bit-counting.

- Also **accelerate the intrablock scan** using the intrinsic functions.

Two Implementations of the Kd-tree Construction Algorithm

- **Implementation Environment:**
  - An NVIDIA GeForce GTX 680 GPU & CUDA v5.0

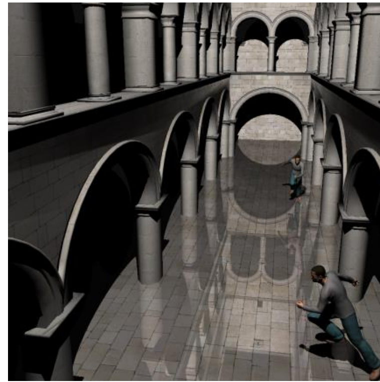
- **Original:** as described in [Zhou et al. 2008]
  - **Intra-block scans and reductions**: the techniques in [Sengupta et al. 2011]
  - **Segmented scans and reductions**: the CUDPP primitives in [CUDPP 2011]
  - **Bit counting**: the technique in [Manku 2002]
- **Ours:**

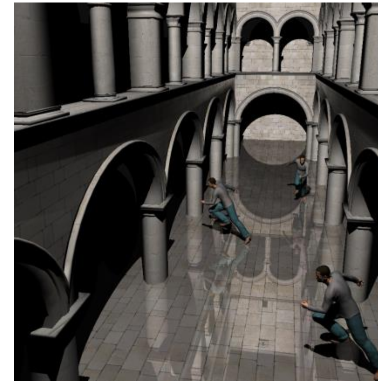
	Stage	Operations	Our implementation
[A]	Large node stage: *triangle sorting*	Two segmented scans	Two atomic operations per block
[B]		Intra-block scans	`__ballot()/__popc()/__shfl_up()`
[C]	Large node stage: *AABB computation*	Six segmented reductions	Six atomic operations per block
[D]		Intra-block reductions	`__shfl_up()`
[E]	Small node stage	Bit counting	`__popc()`
		Intra-block scans	same as **[B]**

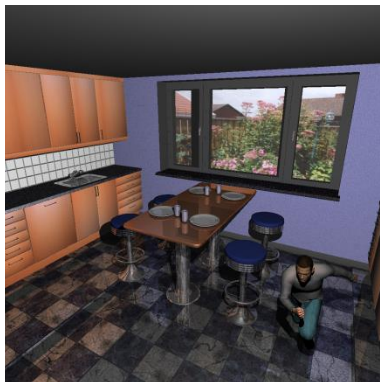
Tested Example Scenes

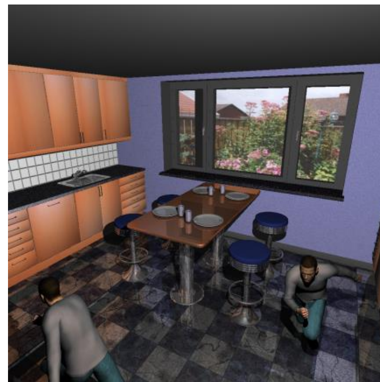
- **SRi** (Sponza with i Runners): 66,454 + i*78,029 triangles
- **KRi** (Kitchen with i Runners): 101,015 + i*78,029 triangles
- **FF** (Fairy Forest): 174,117 triangles

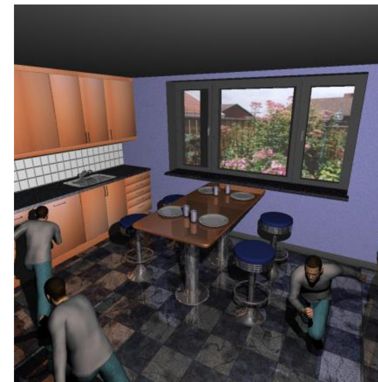
SR1 (144,483)

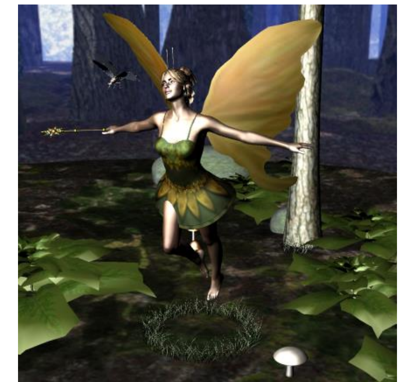

SR2 (222,512)


SR3 (300,541)


KR1 (179,044)


KR2 (257,073)


KR3 (335,102)


FF (174,117)

Experimental Results: Kd-tree Construction

- **Average construction time** (in milliseconds)
  - Averaged for given animation sequences
  - **The replacement of segmented scan and reduction by per-block atomic operations** produced **significant improvements**.

The total time spent on the construction

The sum of each kernel's pure execution time

	SR1	SR2	SR3	KR1	KR2	KR3	FF
Original	**120.9(46.3)**	**146.2(59.1)**	**181.8(77.7)**	**130.9(47.1)**	**149.7(60.0)**	**185.5(77.8)**	**107.4(42.1)**
[A]	88.6(34.8)	93.0(49.9)	109.7(64.2)	91.8(35.1)	109.1(50.0)	118.9(64.9)	76.2(36.8)
[A] to **[B]**	76.3(34.1)	89.2(49.0)	107.4(62.8)	85.7(34.3)	94.7(48.6)	104.4(62.6)	67.1(36.2)
[A] to **[C]**	52.3(29.9)	67.5(44.7)	79.8(59.0)	62.3(31.1)	67.6(44.9)	90.6(59.3)	58.5(34.0)
[A] to **[D]**	51.6(29.2)	66.4(43.3)	78.1(56.9)	60.1(29.8)	67.1(43.6)	88.6(57.4)	54.5(32.8)
Ours(All)	**48.5(26.7)**	**64.3(39.1)**	**74.1(52.4)**	**48.8(27.4)**	**64.0(40.6)**	**72.5(52.8)**	**48.1(30.4)**

2 segmented scans → 2 atomic operations per block

6 segmented reductions → 6 atomic operations per block

- **Average number of launched CUDA kernels**
  - ➢ The extra calls within the CUDPP functions were not counted here.

	SR1	SR2	SR3	KR1	KR2	KR3	FF
Original	1,031	1,041	1,069	9,77	995	1,034	681
Ours(All)	221	229	232	214	219	223	153

- **Our implementation** called **much fewer numbers of kernels**, leading to the significant reduction of the overheads from multiple synchronous kernel calls.
  - **Example: the triangle-sorting process**
    - **Original**: four kernel calls plus those necessary for two segmented-scan calls
    - **Ours**: one kernel call
- **The intrinsic functions** of the recent CUDA compute capability also provided **a meaningful reduction in the construction time**.

Experimental Results: Interactive Rendering

- **Average rendering time** (in milliseconds)
  - Rendered **a 1024x1024 image** by **full ray tracing** with shading, textures, reflection, and shadows.

		SR1	SR2	SR3	KR1	KR2	KR3	FF
Triangle numbers		144,483	222,512	300,541	179,044	257,073	335,102	174,117
Original	Ray-tracing time	92.5	83.9	95.4	90.0	83.5	88.7	92.1
	Total rendering time	**213.4**	**230.1**	**277.2**	**220.9**	**233.2**	**274.2**	**199.5**
Ours	Ray-tracing time	90.1	92.7	93.5	88.2	94.3	100.5	103.3
	Total rendering time	**138.6**	**157.0**	**167.6**	**137.0**	**158.3**	**173.0**	**157.8**

Kd-tree construction time + ray-tracing time

- **The efficient implementation of the kd-tree construction algorithm** resulted in **a significant reduction in the interactive rendering time** for dynamic scenes with nontrivial complexity.

Conclusion

- This paper presented **efficient GPU programming techniques** for implementing **the kd-tree construction algorithm** [Zhou et al. 2008].
- **Much fewer numbers of kernels calls** were made during the construction, which resulted in **a markedly more efficient execution on the GPU**.
  - With current GPUs, it is important to minimize the number of kernel calls made!

Kitchen with 3 Runners

GPU: NVIDIA GeForce GTX 680
Resolution: 1024x1024
Triangles: 335,102
Lights: 1
Reflection bounces: 1
(Video resolution was reduced due to file size limit.)

A Single Kernel CUDA Implementation for the Triangle Sorting Process

```
/* This kernel corresponds to the fourth and fifth steps of the large node stage described in [11]. */

__global__ void MedianSplitChunk(float *TriAABB, int *ChunkNodeIDs, int *NodeTriOffsets, int *NodeTriNums,
                                 int *ChunkStartIndices, int *ActiveNodeList, char *NodeSplitAxes,
                                 float *NodeSplitPoss, int *ChunkOffsets, int *NextNodeList) {

   __shared__ volatile int LChildTriOffsets2[9], RChildTriOffsets2[9];
   __shared__ int LChildNodesID, RChildNodesID, LOffset, ROffset;

   int LaneID = threadIdx.x & 0x0000001f;
   int NodeID = ChunkNodeIDs[CurBlockIndex];
   int CurBlockIndex = blockIdx.x
   int TriNum = NodeTriNums[NodeID], TriOffset = NodeTriOffsets[NodeID];
   int LChildTriOffsets, RChildTriOffsets;

   if (threadIdx.x < 9)
      LChildTriOffsets2[threadIdx.x] = RChildTriOffsets2[threadIdx.x] = 0;

   int TriIndex, StartPos = ChunkStartIndices[CurBlockIndex];
   int CurPos = StartPos + threadIdx.x;
```

```
/* Classify the current triangle w.r.t. splitting plane. */
unsigned int LChildTriBitFlag = 0, RChildTriBitFlag = 0;

if (CurPos<TriNum) {
   /* The last chunk may have fewer than 256 triangles. */
   int SplitAxis = NodeSplitAxes[NodeID];
   float SplitPos = NodeSplitPoss[NodeID];

   TriIndex = ActiveNodeList[TriOffset+ CurID];
   float MinPos = TriAABB[TriIndex + SplitAxis * TRI OFFSET];
   float MaxPos = TriAABB[TriIndex + (SplitAxis + 3) * TRI OFFSET];
   LChildTriBitFlag = (MinPos < SplitPos);
   RChildTriBitFlag = (MinPos >= SplitPos);
   if (LChildTriBitFlag) RChildTriBitFlag = (SplitPos < MaxPos);
}

/* Perform intra-warp scan. */
unsigned int LeftMask = ballot(LChildTriBitFlag), LaneMaskLT = 0;
unsigned int RightMask = ballot(RChildTriBitFlag), LaneMaskLE = 0;

asm("mov.u32 %0, %%lanemask lt;" : "=r"(LaneMaskLT));
asm("mov.u32 %0, %%lanemask le;" : "=r"(LaneMaskLE));
LChildTriOffsets = popc(LeftMask & LaneMaskLT);
RChildTriOffsets = popc(RightMask & LaneMaskLT);

if (LaneID == 31) {
   LChildTriOffsets2[(threadIdx.x >> 5) + 1] = popc(LeftMask & LaneMaskLE);
   RChildTriOffsets2[(threadIdx.x >> 5) + 1] = popc(RightMask & LaneMaskLE);
}
syncthreads();
```

```
    /* Perform inter-warp scan. */
    float Scan8[2];
    if (threadIdx.x < 8) {
        Scan8[0] = LChildTriOffsets2[threadIdx.x + 1];
        Scan8[1] = RChildTriOffsets2[threadIdx.x + 1];
        for (int i = 1; i <= 4; i *= 2) {
            float n0 = shfl up(Scan8[0], i, 8); float n1 = shfl up(Scan8[1], i, 8);
            if (LaneID >= i) { Scan8[0] += n0; Scan8[1] += n1; }
        }
    }

    if (threadIdx.x < 8) {
        LChildTriOffsets2[threadIdx.x + 1] = Scan8[0]; RChildTriOffsets2[threadIdx.x + 1] = Scan8[1];
    }

    /* Fetch start positions for the current chunk. */
    if (threadIdx.x == 0) {
        LChildNodesID = 2*NodeID; RChildNodesID = 2*NodeID + 1;
        LOffset = atomicAdd(&ChunkOffsets[LChildNodesID], LChildTriOffsets2[8]);
        ROffset = atomicAdd(&ChunkOffsets[RChildNodesID], RChildTriOffsets2[8]);
    }
    syncthreads();

    LChildTriOffsets += LChildTriOffsets2[(threadIdx.x >> 5)];
    RChildTriOffsets += RChildTriOffsets2[(threadIdx.x >> 5)];

    if (LChildTriBitFlag != 0) NextNodeList[LOffset + LChildTriOffsets] = TriIndex;
    if (RChildTriBitFlag != 0) NextNodeList[ROffset + RChildTriOffsets] = TriIndex;
}
```