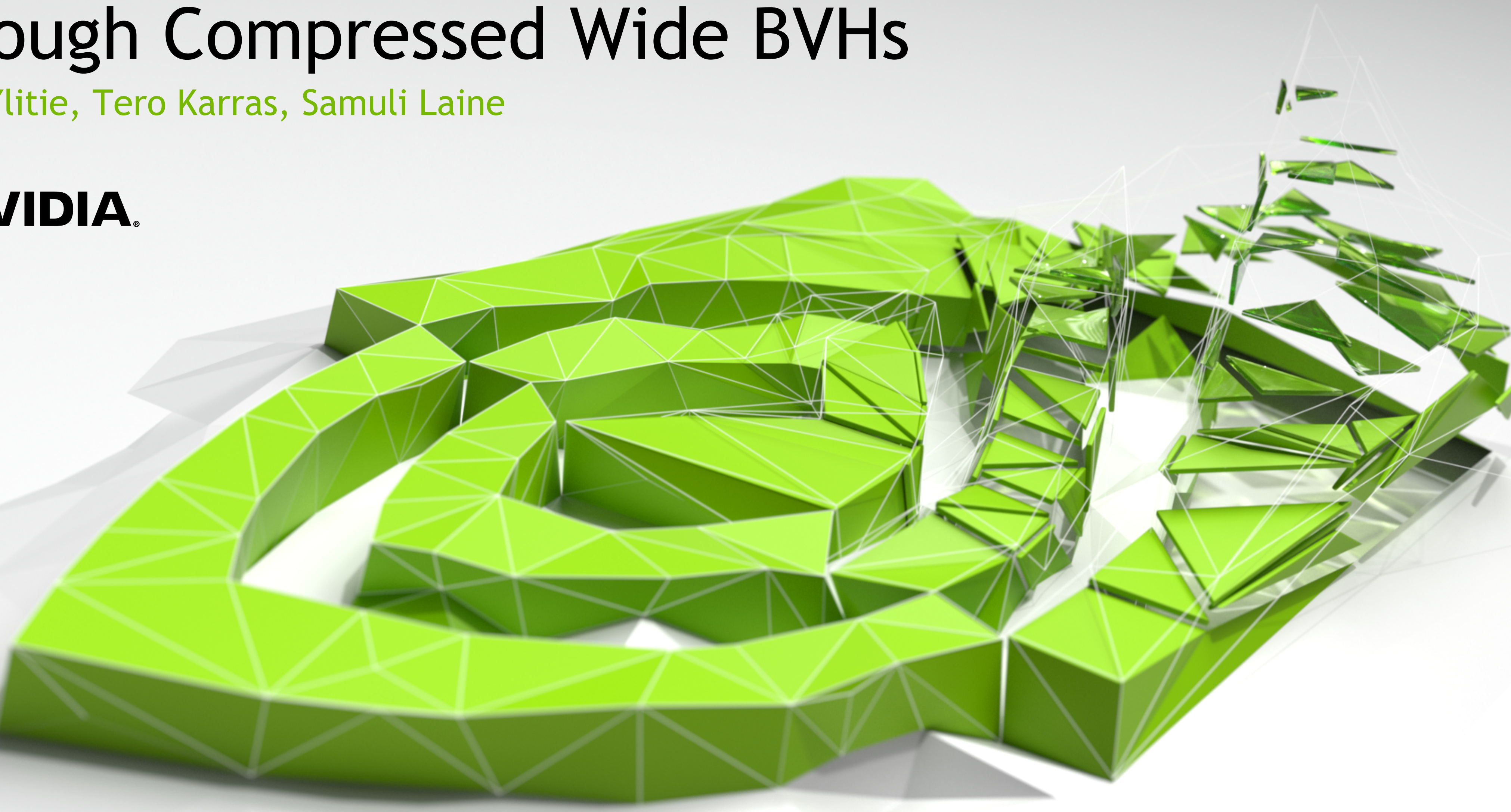
Efficient Incoherent Ray Traversal on GPUs Through Compressed Wide BVHs

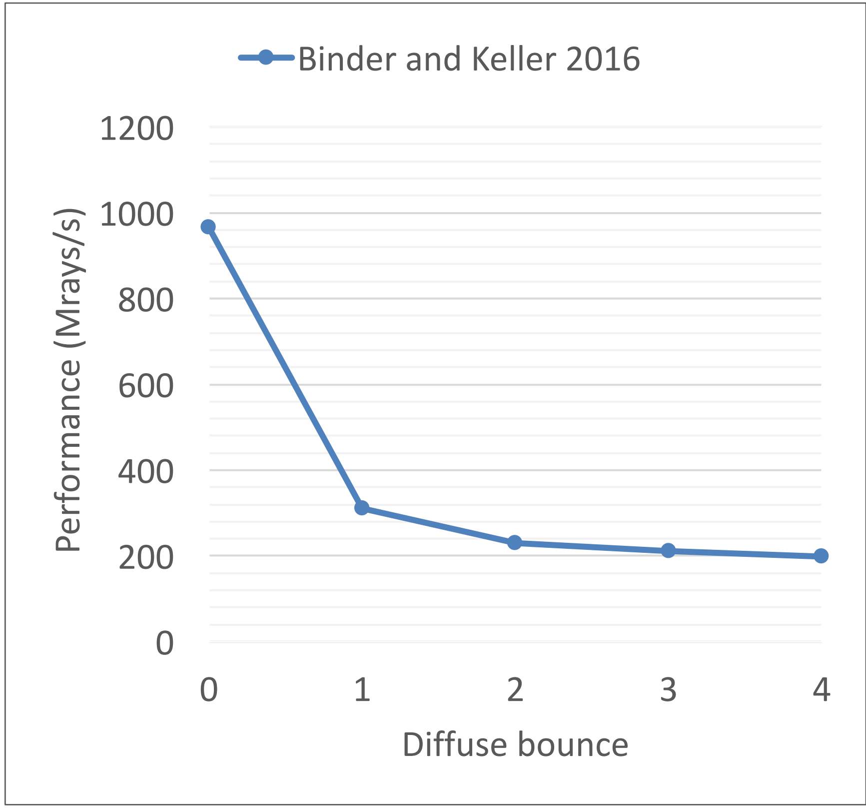
Henri Ylitie, Tero Karras, Samuli Laine



High Performance Graphics 2017; July 28th, 2017; Los Angeles, CA, USA

Inspiration

Rendered with NVIDIA Iray



Motivation

- GPU ray tracing performance limited by memory system
- Low SIMD utilization with incoherent rays
- Impressive results in CPU ray tracing using wide BVHs and compression
  - Full potential maybe not realized on GPUs yet?

Overview

Combination of new and existing techniques

- 8-wide BVH constructed with SAH-optimal widening
- Compressed node storage format
- Cheap octant-based fixed-order traversal
- Traversal stack traffic eliminated through compression and usage of shared memory
- Improved SIMD utilization through triangle postponing and dynamic ray fetching

- Starting point: BVH traversal kernels by Aila, Karras and Laine [2012]

Overview

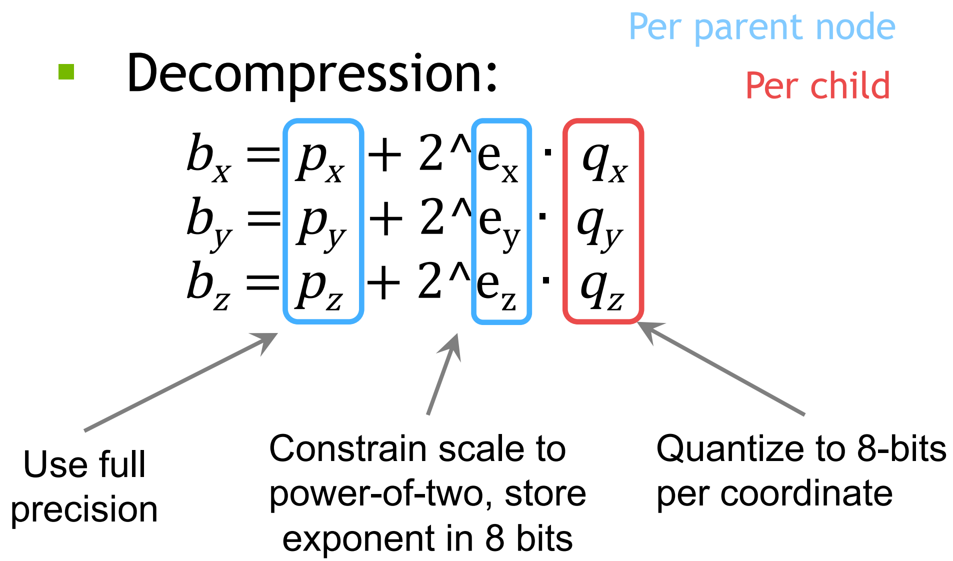
- 2x incoherent ray traversal performance
- 0.33x acceleration structure size

Compared to fastest previous method by Binder and Keller [2016]

Bounding box quantization

- Quantize child node AABBs to a local grid
  - Similar to [Mahovsky and Wyvill 2006; Segovia and Ernst 2010; Keely 2014; Vaidyanathan et al. 2016]
- Quantization grid position and scale stored in parent node

- Decompression:



Per parent node

Per child

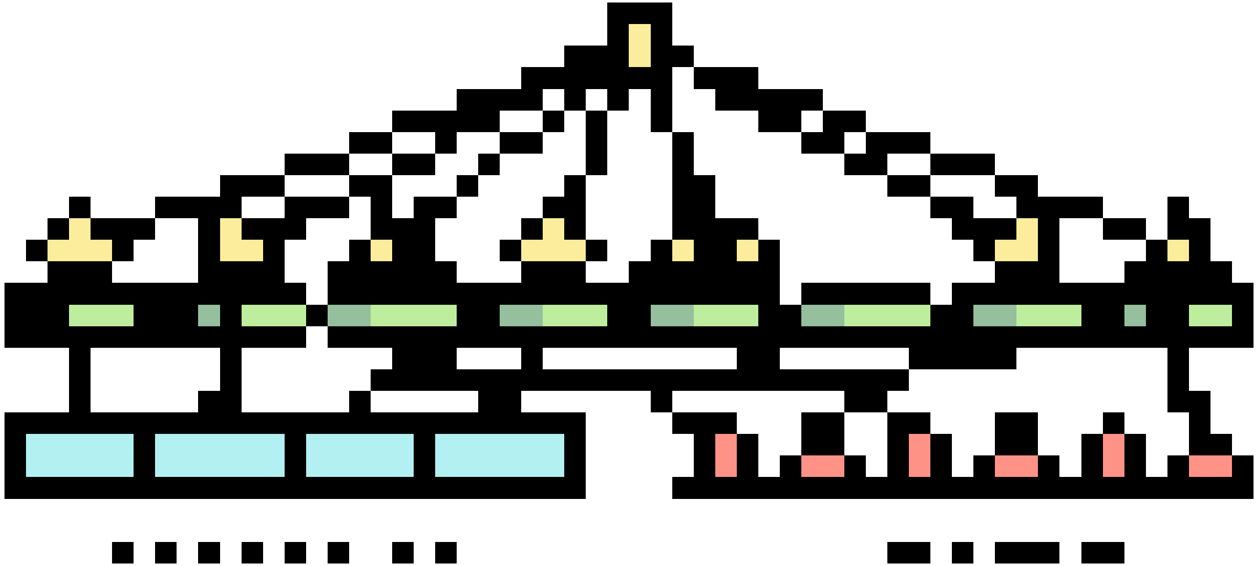
$$b_x = p_x + 2^{e_x} \cdot q_x$$
$$b_y = p_y + 2^{e_y} \cdot q_y$$
$$b_z = p_z + 2^{e_z} \cdot q_z$$

Use full precision

Constrain scale to power-of-two, store exponent in 8 bits

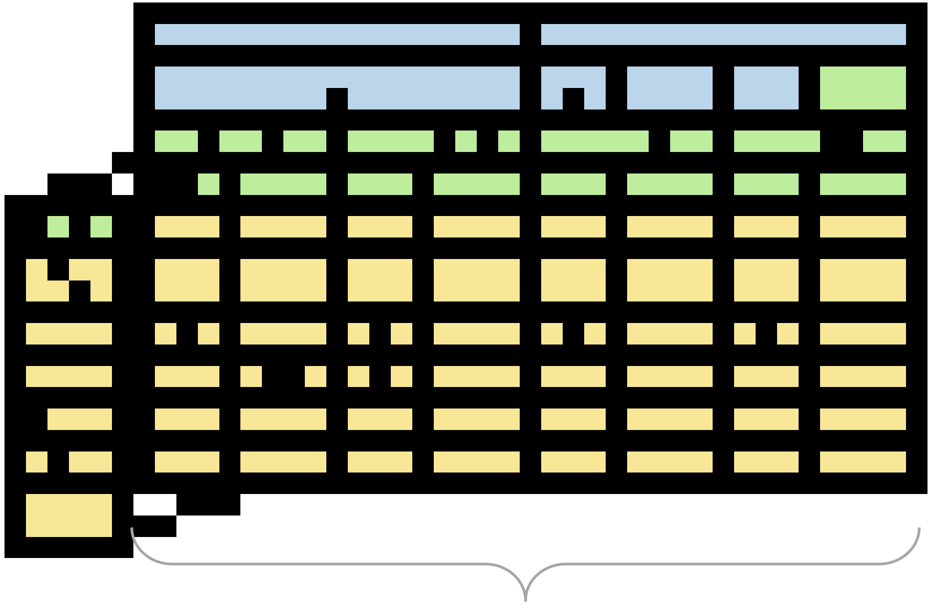
Quantize to 8-bits per coordinate

Child node index compression



- Child nodes, triangles stored contiguously in separate arrays
- Index of first child node, triangle stored in node
- 8-bit field per child to encode relative offset, child type
- Up to 3 triangles/leaf

Internal node memory layout



8 children / node

- 🟦 Quantization grid 15B
- 🟩 Indexing information 17B
- 🟨 Quantized bounding boxes 48B

= Total 80B
10B/child

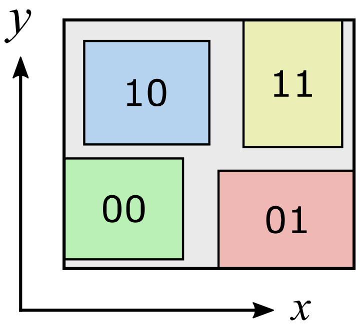
Aila et al. [2012]
32B/child

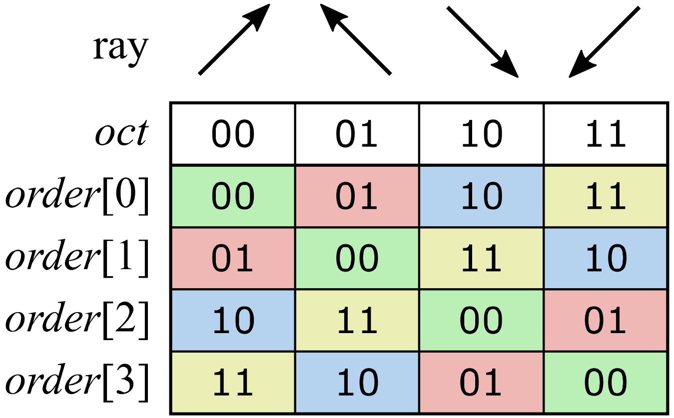
Traversal order

- Approximate near-to-far traversal order is important
    - Most approaches sort by distance
- 8-element distance sorts are expensive
    - Sorting network -> 19 compare-and-swap operations [Knuth 1998]
    - Sort hits only -> high divergence

Octant-based traversal order

[Garanzha and Loop 2010]





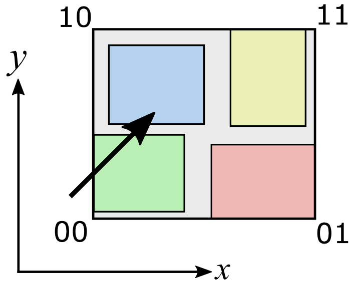
ray	↗	↖	↘	↙
oct	00	01	10	11
order[0]	00	01	10	11
order[1]	01	00	11	10
order[2]	10	11	00	01
order[3]	11	10	01	00

- Store child nodes to memory in Morton order of their AABB centers
    - Approximately assigns each child to closest parent box corner
- Traverse the nodes in order sortedChildren[i] = children[i ^ oct]
- Doesn't work well for partially filled nodes

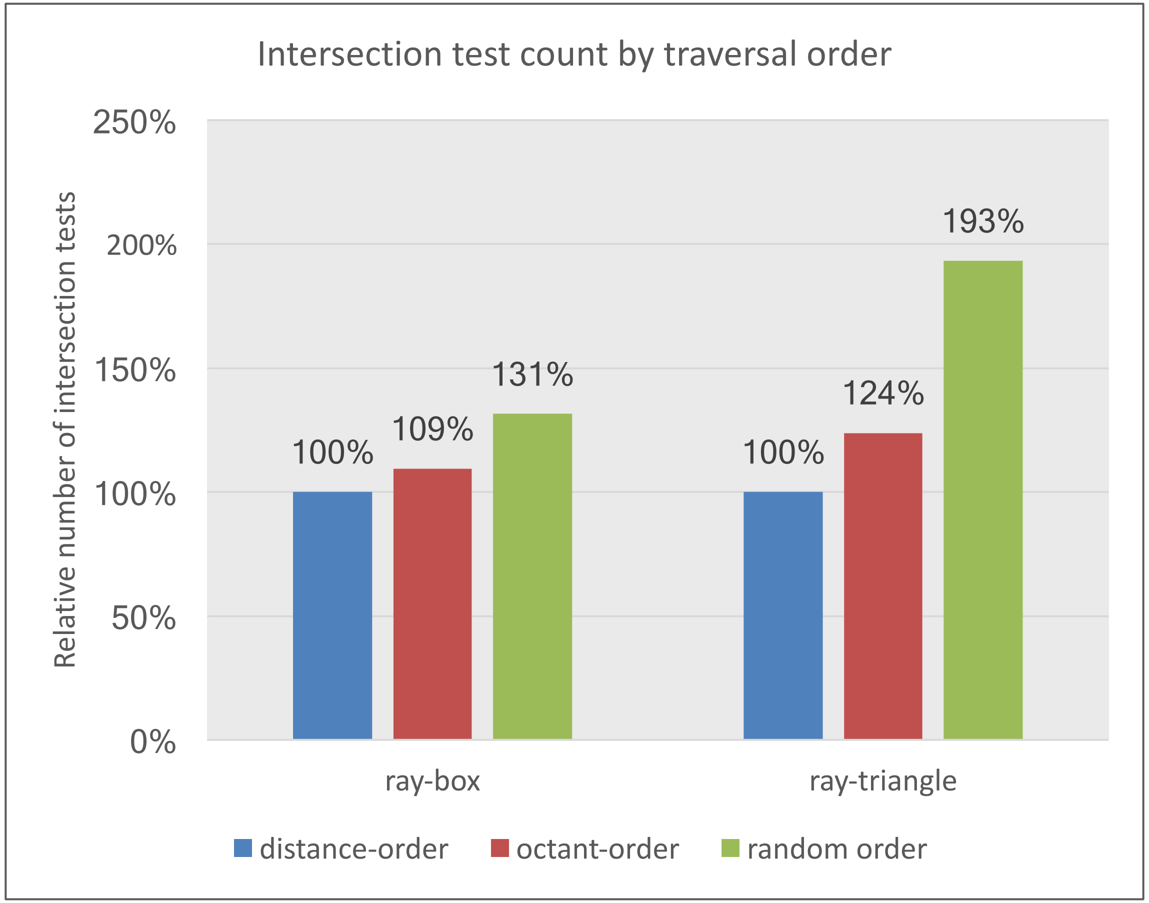
Octant-based traversal order

Idea: Optimize the child node assignment

- Enumerate corners of parent bounding box (child slots) in Morton order
- Optimize the way child nodes are assigned to the slots
  - Define a cost function for placing a child node with AABB center **c** in a slot s
  - Pick a diagonal ray with direction $\mathbf{d}_s$ = (±1, ±1, ±1) that traverses slot s first
    - 2D example: Slot 00 -> ray direction $\mathbf{d}_s$ = (1, 1)
  - Cost(**c**, s) = (**c** - **p**)·$\mathbf{d}_s$ ← 8x8 table
  - Distance from parent box center **p** projected on the ray direction
  - Minimize total cost using the auction algorithm [Bertsekas 1992]



Octant-based traversal order



Reducing traversal stack traffic

Compressing stack entries

- Combine all sibling nodes of same type to a single 8-byte stack entry
  - 32-bit base index, bitmask for individual items
- Internal node test produces 0-2 stack entries
  - Up to 8 internal nodes in each *node group*
  - Up to 24 triangles from up to 8 leaf nodes in each *triangle group*

child node base index	hits	pad	*imask*
32	8	16	8

triangle base index	pad	triangle hits
32	8	24

Reducing traversal stack traffic

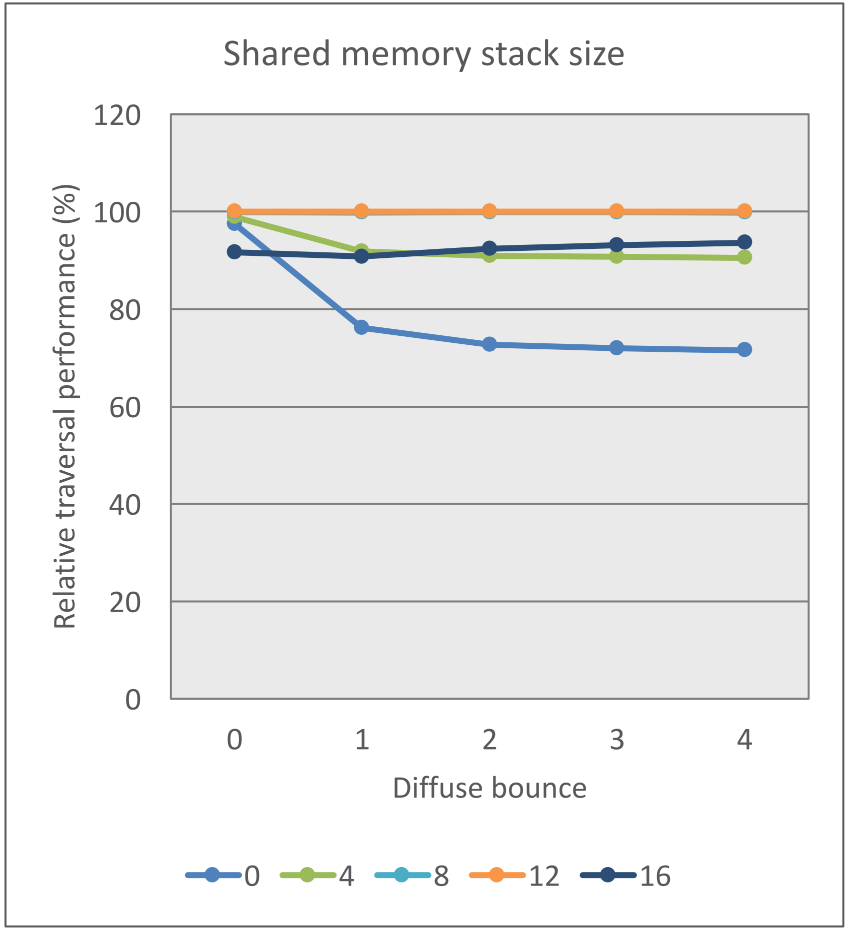
Compressing stack entries

- How to maintain the traversal order?
- Define a traversal priority as reverse of the traversal order
    - priority = slot_index ^ (7 – oct)
    - Traverse nodes with highest priority first
- Permute the *hits*-field: Internal nodes set bit corresponding to traversal priority
    - Find highest set bit to get node to traverse next
    - Reverse priority computation to obtain child slot index

Reducing traversal stack traffic

Using shared memory

- Store as many stack entries to shared memory as possible
  - 12 in our kernel
- Spill rest of the entries to local memory
  - Happens very rarely
- Eliminates practically all external memory traffic



Improving SIMD utilization

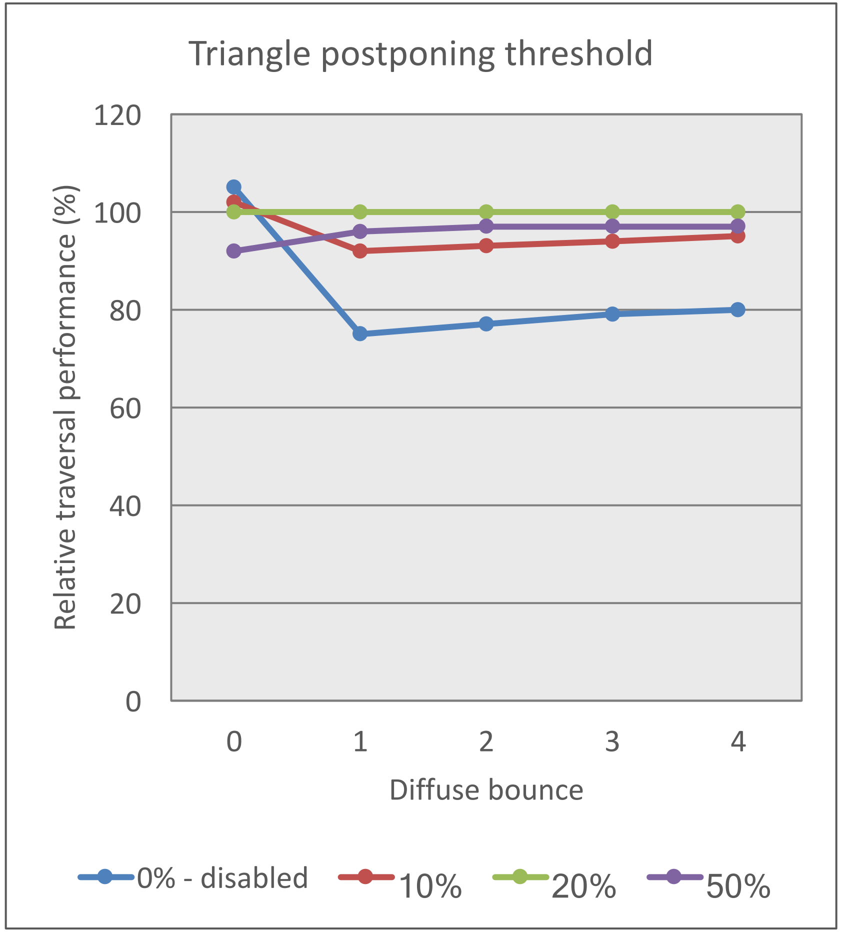
Postponing triangle intersection tests

- Threads follow different paths in the tree
    - Especially with incoherent rays
- Internal nodes traversed more often than leaves
- Only a few threads in a 32-lane warp active in ray-triangle intersection test

Improving SIMD utilization

Postponing triangle intersection tests

- Postpone triangle intersections by pushing triangle groups to stack
- Do this whenever less than 20% of active threads want to intersect triangles



Constructing wide BVHs

- Start with a binary SBVH with one triangle per leaf [Stich et al. 2009]
- Form a wide BVH by collapsing nodes in a SAH-optimal fashion

- Greedy top-down collapsing and splitting [Wald et al. 2008 ; Afra et al. 2013]

- Our: Jointly optimize both internal nodes and leaves at the same time

Constructing wide BVHs

- Moving from bottom to top, process each node in the binary BVH
  - Compute and store optimal SAH cost for all configurations the node could have in the final wide BVH:
    - Leaf
    - Wide internal node
    - Eliminated – subtree is represented as forest with 2-7 roots, placed as children of the node's parent. Ask child nodes how to optimally divide roots between them.
  - Backtrack from root and create wide nodes so that optimal cost is realized.

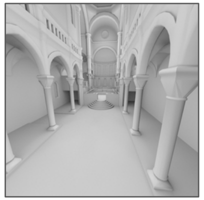
Constructing wide BVHs

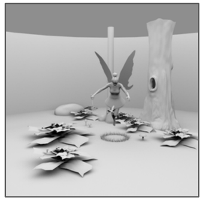
- Improves the tradeoff between performance and memory usage
- Compared to node collapsing method by Afra et al. [2013]
  - 1 – 4% higher traversal performance
  - Lowers memory consumption, 1.18x as many children per node (7.51 vs. 6.39),

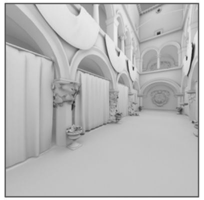
Results

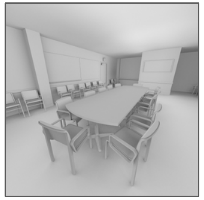
Benchmark setup

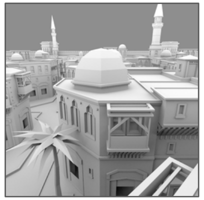
- Diffuse path tracing, measure ray cast time for each bounce separately
- 2048x2048 resolution
- 15 scenes with 1-5 viewpoints each.
- Hardware: NVIDIA Titan X (Pascal)


80k

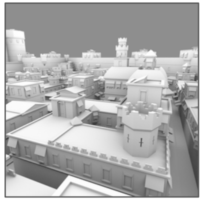

174k


262k


283k


407k

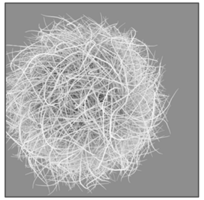

606k


762k


1.3M


1.9M


2.2M

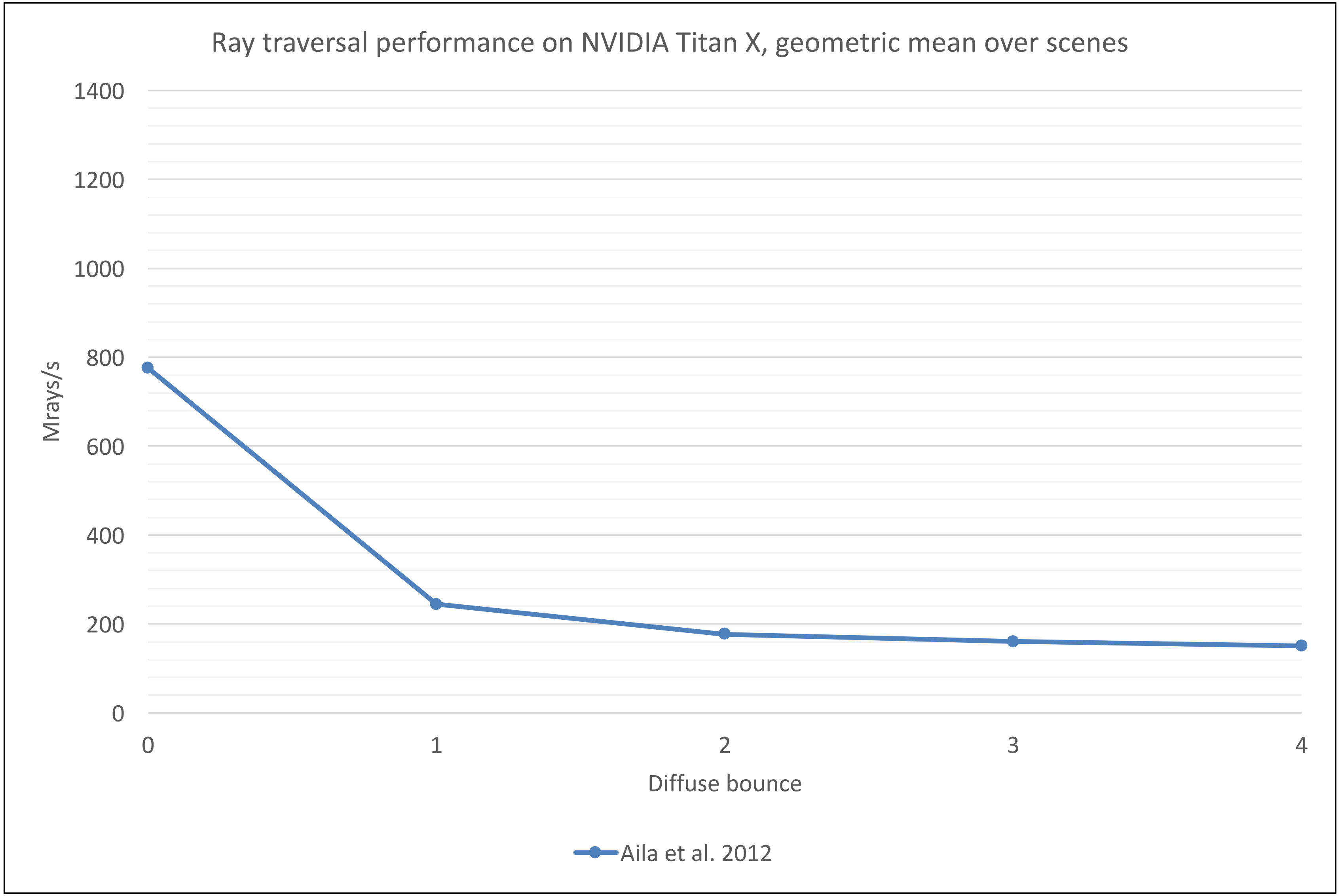

2.9M

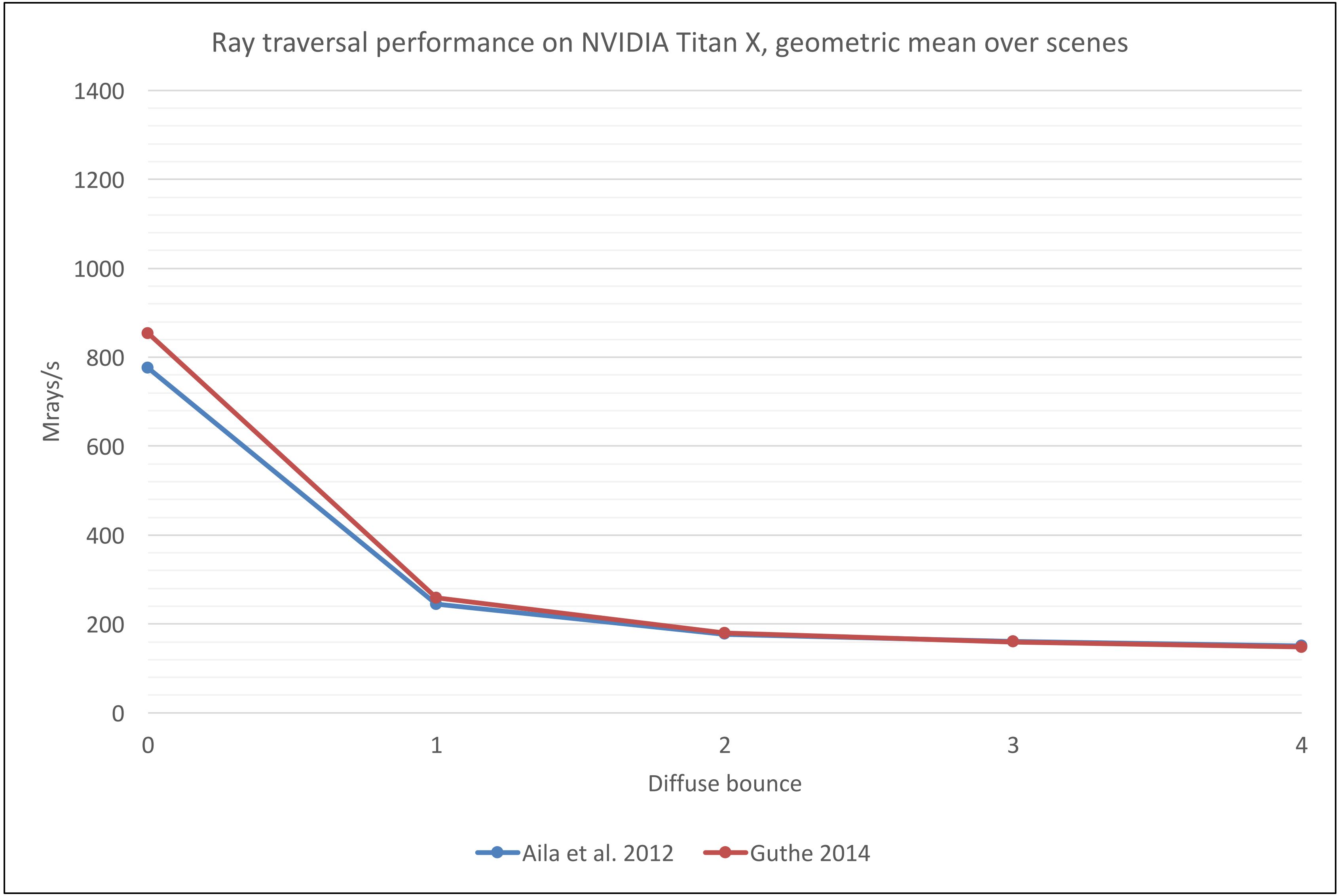

4.1M

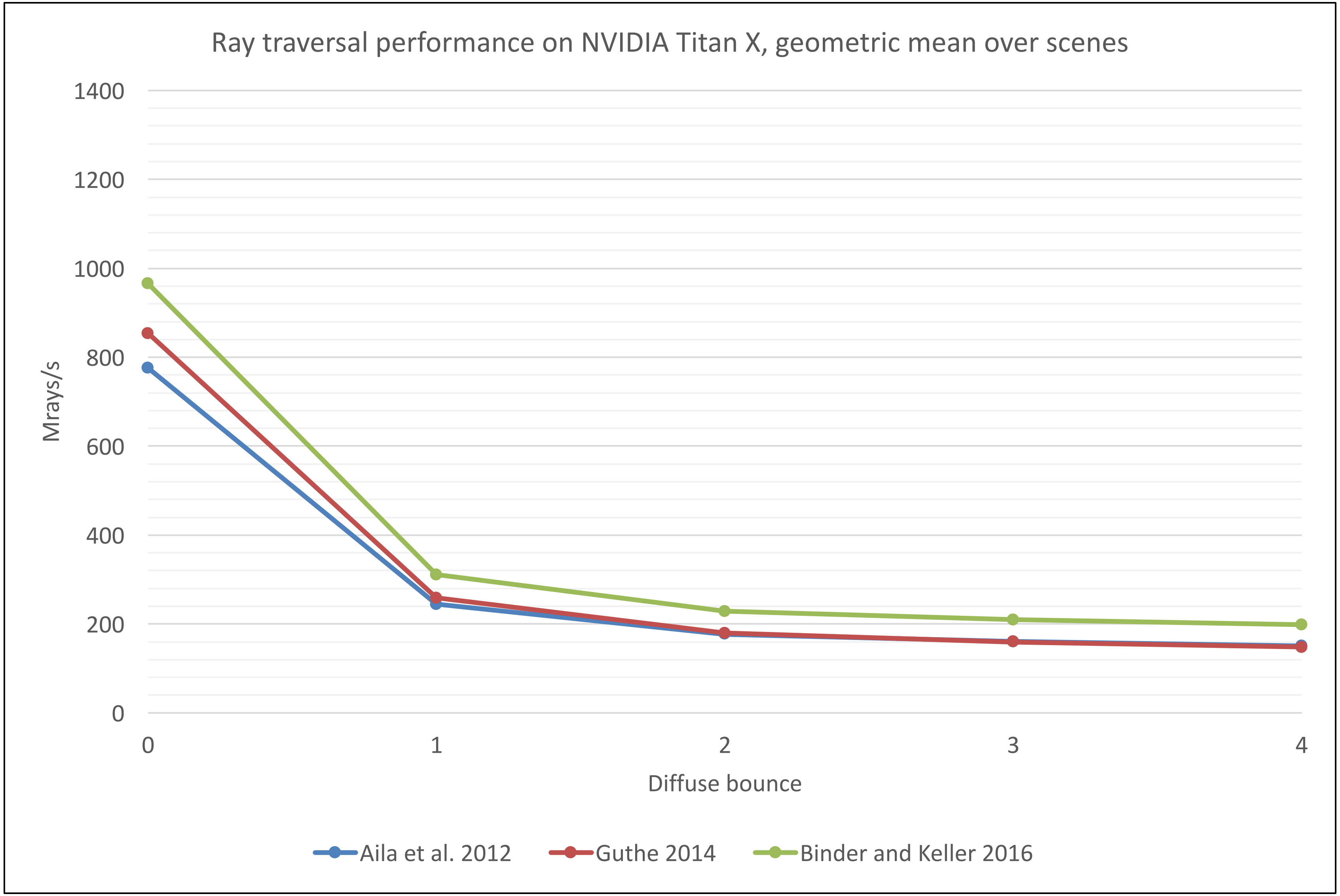

7.5M

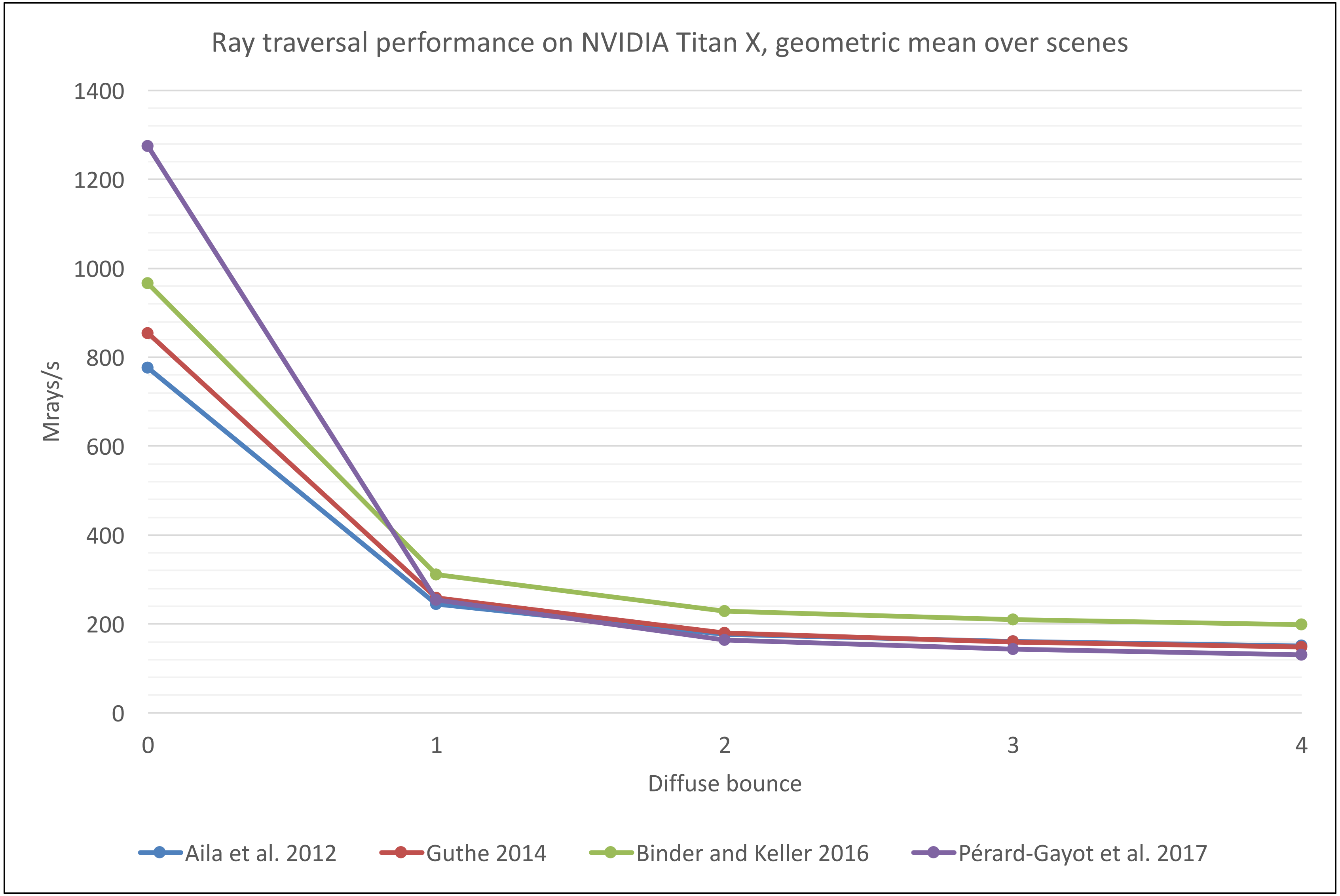

10.5M

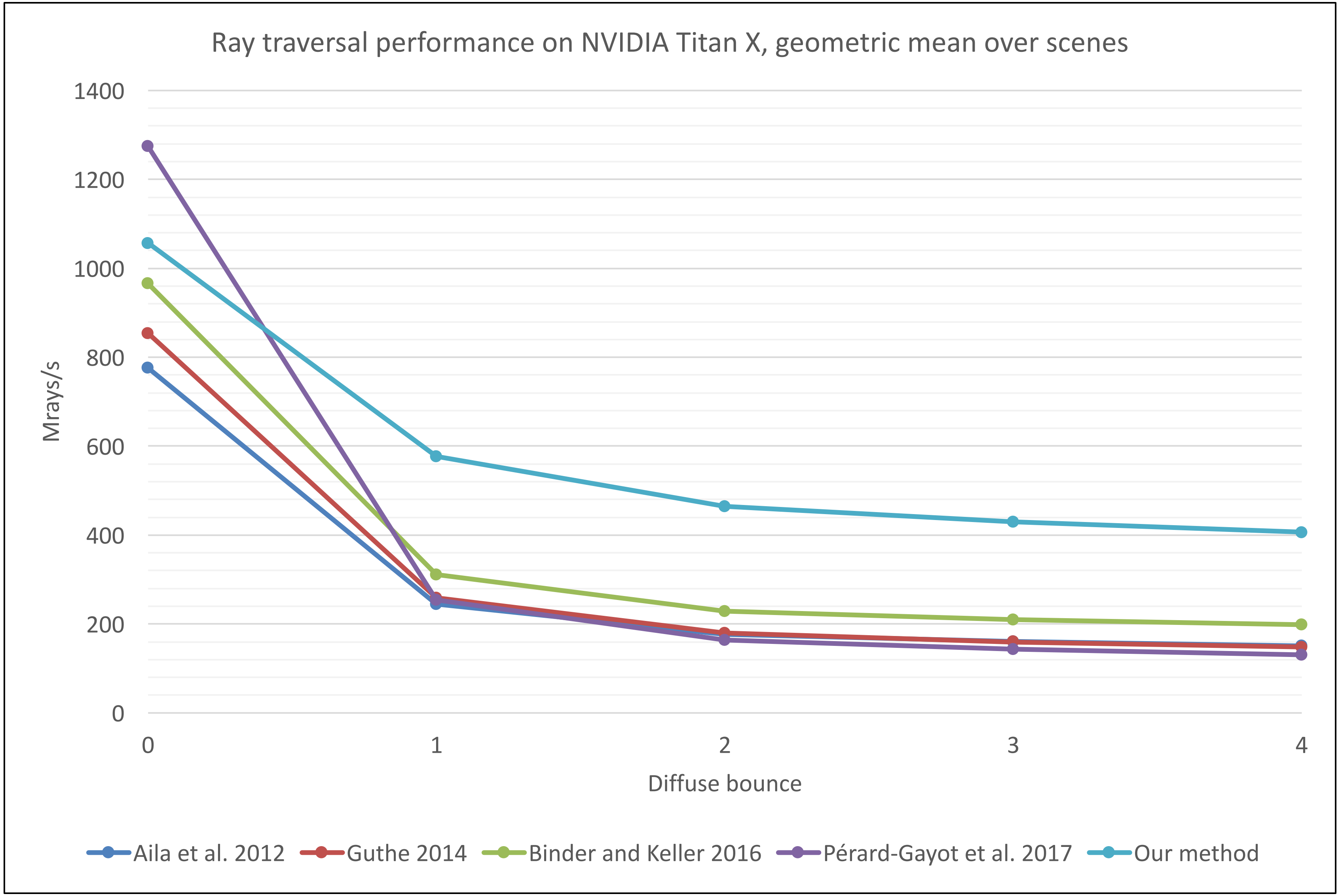

12.8M

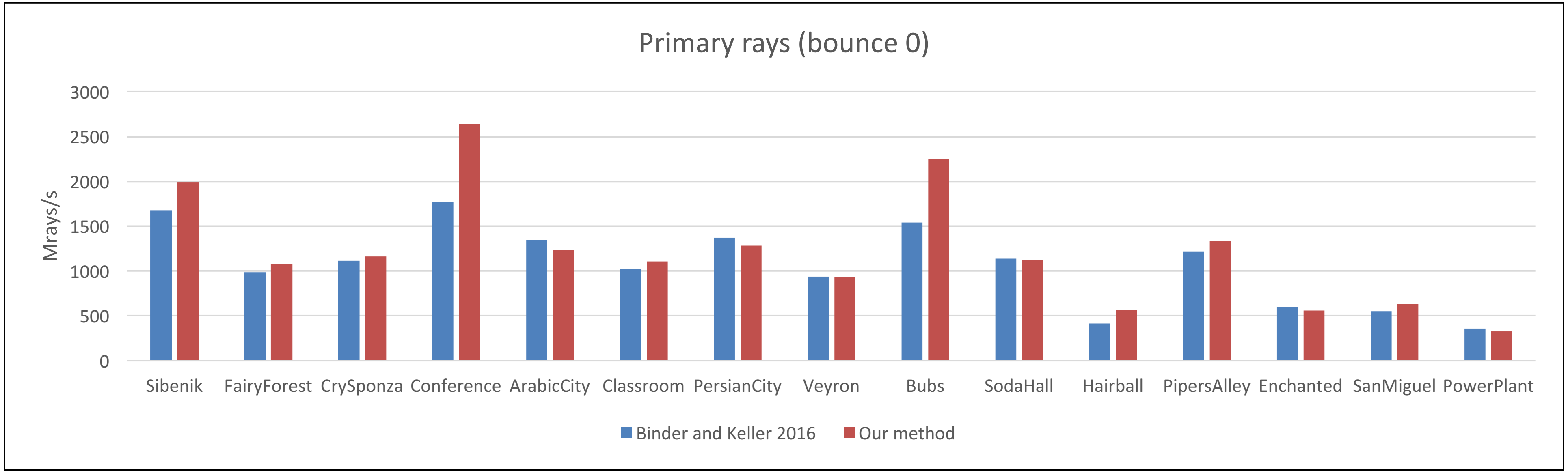

Ray traversal performance on NVIDIA Titan X, geometric mean over scenes
Mrays/s
1400
1200
1000
800
600
400
200
0
0
1
2
3
4
Diffuse bounce
Aila et al. 2012

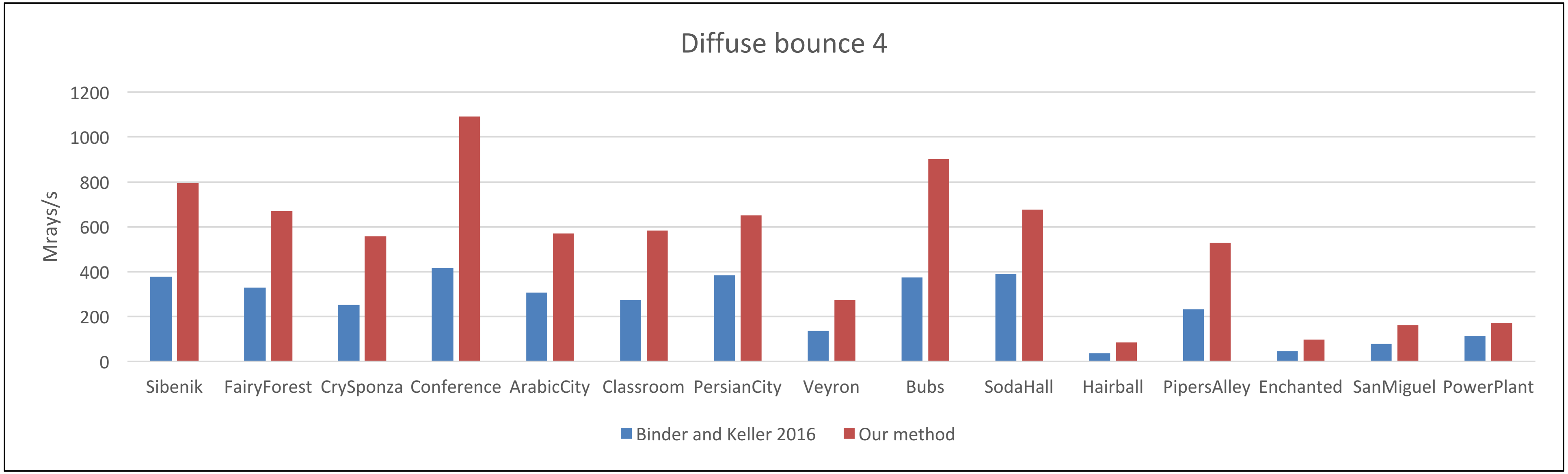

Ray traversal performance on NVIDIA Titan X, geometric mean over scenes
1400
1200
1000
800
600
400
200
0
Mrays/s
0
1
2
3
4
Diffuse bounce
Aila et al. 2012
Guthe 2014


Ray traversal performance on NVIDIA Titan X, geometric mean over scenes
1400
1200
1000
800
600
400
200
0
Mrays/s
0
1
2
3
4
Diffuse bounce
Aila et al. 2012
Guthe 2014
Binder and Keller 2016

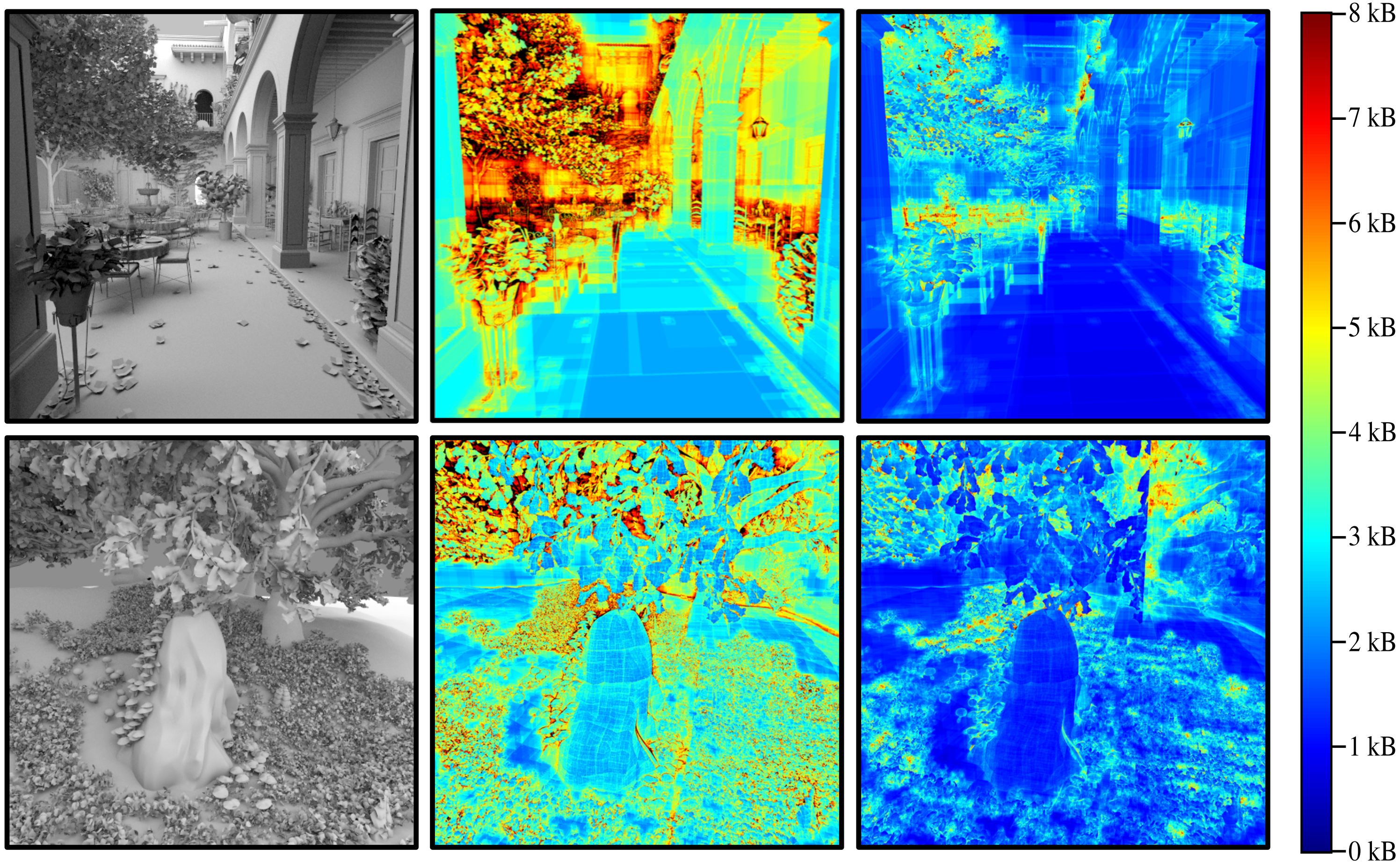

Ray traversal performance on NVIDIA Titan X, geometric mean over scenes
Mrays/s
1400
1200
1000
800
600
400
200
0
0
1
2
3
4
Diffuse bounce
Aila et al. 2012
Guthe 2014
Binder and Keller 2016
Pérard-Gayot et al. 2017


Ray traversal performance on NVIDIA Titan X, geometric mean over scenes
1400
1200
1000
800
600
400
200
0
Mrays/s
0
1
2
3
4
Diffuse bounce
Aila et al. 2012
Guthe 2014
Binder and Keller 2016
Pérard-Gayot et al. 2017
Our method


Primary rays (bounce 0)
Mrays/s
3000
2500
2000
1500
1000
500
0
Sibenik
FairyForest
CrySponza
Conference
ArabicCity
Classroom
PersianCity
Veyron
Bubs
SodaHall
Hairball
PipersAlley
Enchanted
SanMiguel
PowerPlant
Binder and Keller 2016
Our method


Diffuse bounce 4
Mrays/s
1200
1000
800
600
400
200
0
Sibenik
FairyForest
CrySponza
Conference
ArabicCity
Classroom
PersianCity
Veyron
Bubs
SodaHall
Hairball
PipersAlley
Enchanted
SanMiguel
PowerPlant
Binder and Keller 2016
Our method

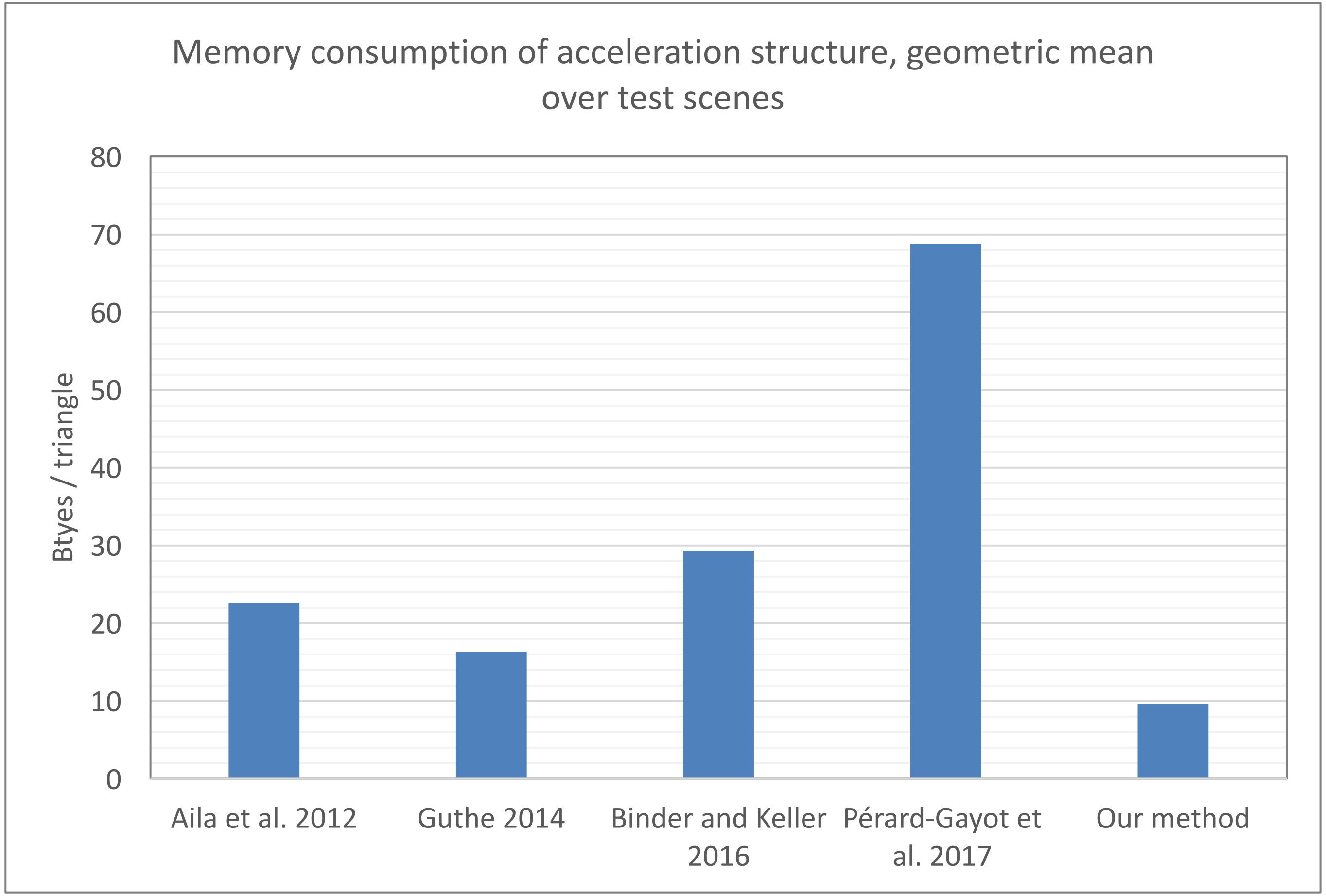
Memory bandwidth – node and triangle fetches



Scene image [Aila et al. 2012] Ours

- ≈ 0.5x on average

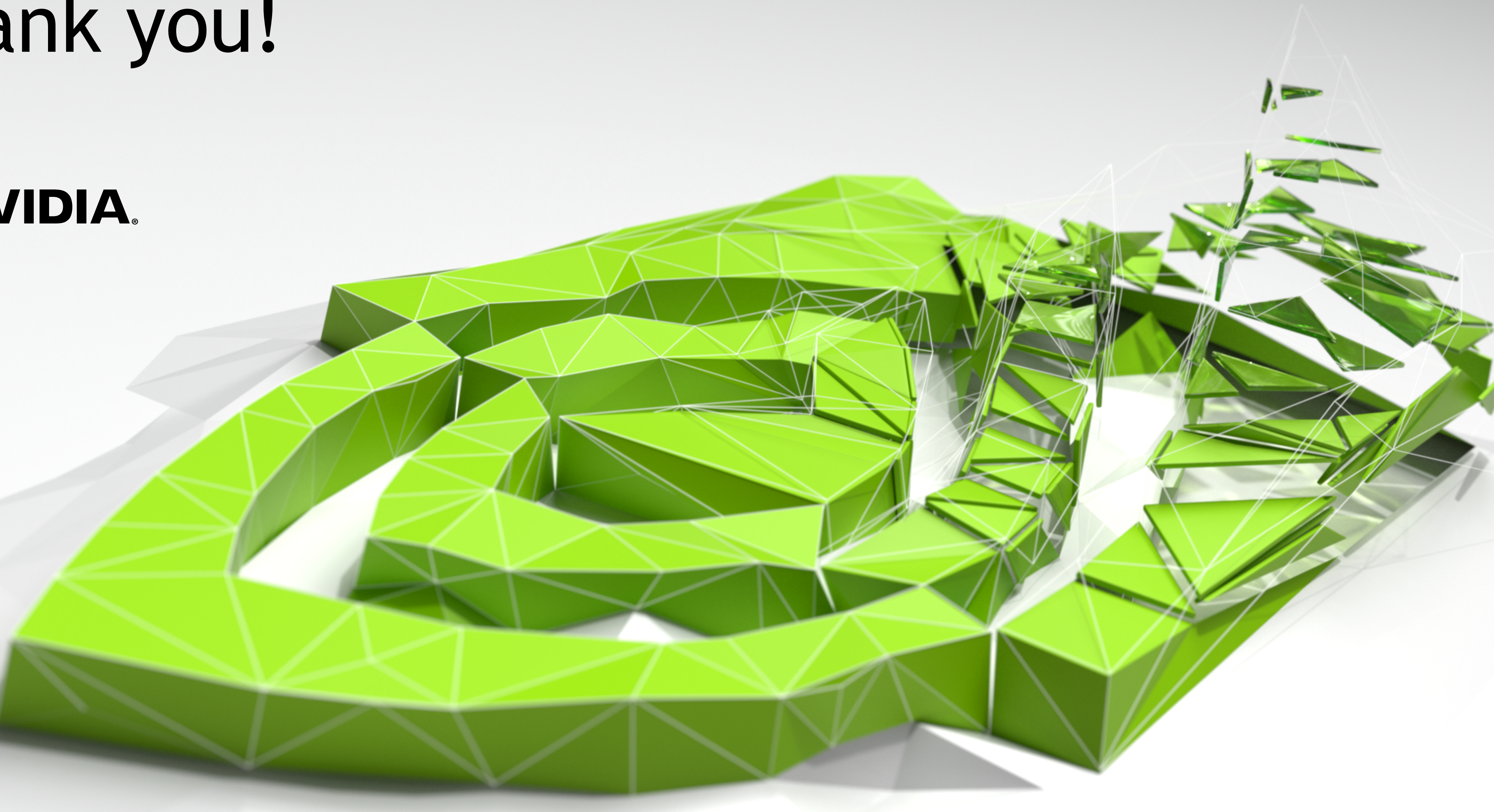
Memory usage



- 0.27 - 0.47x compared to fastest previous method [Binder and Keller 2016]

Questions?

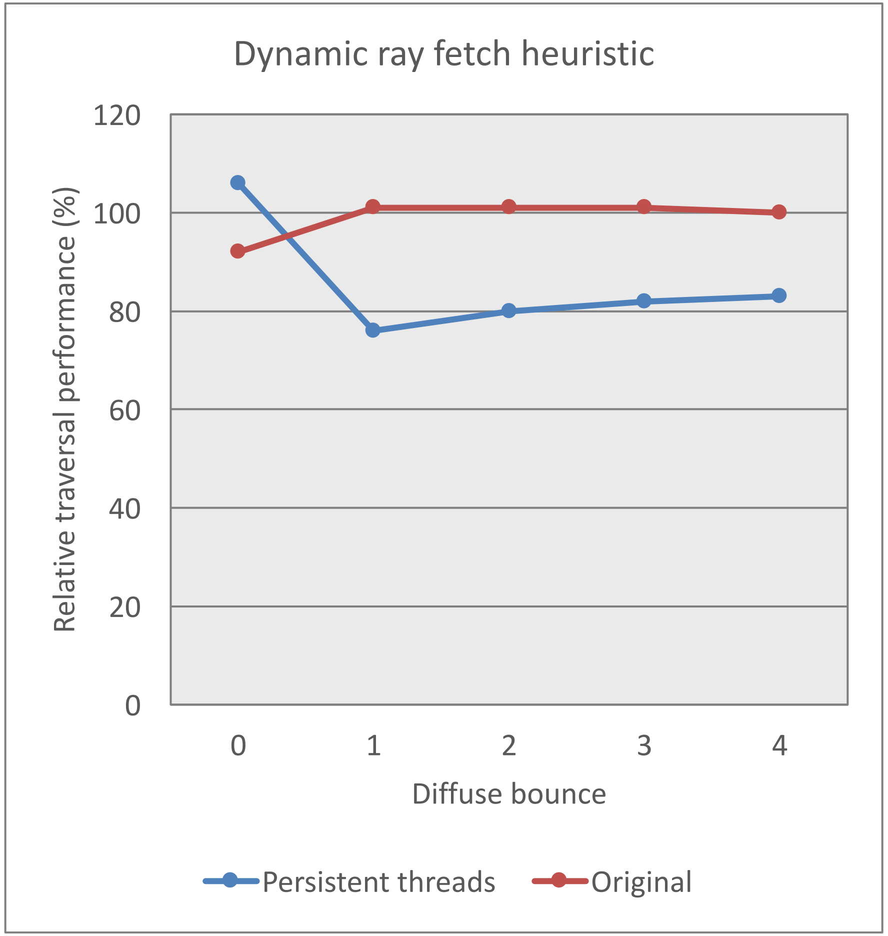
Thank you!



Improving SIMD utilization

Replacing terminated rays

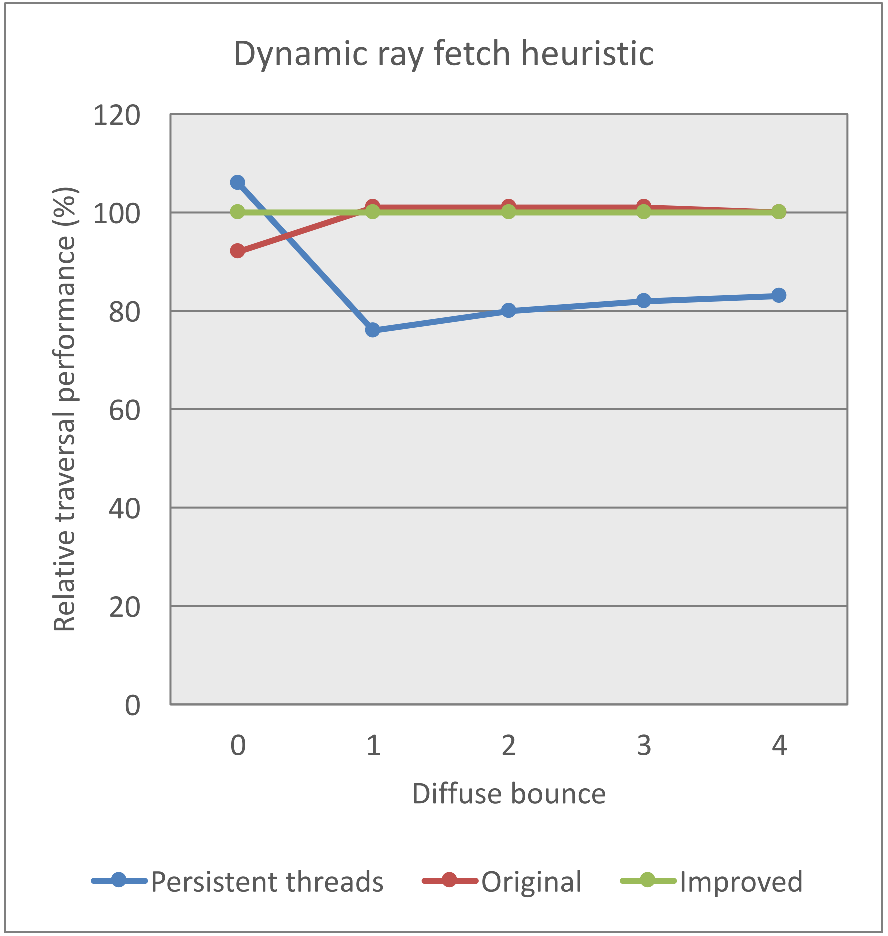
- Rays in a warp finish traversal at different times
- Low SIMD utilization with incoherent rays
- Fetch new rays to replace terminated ones:
    - Persistent threads: Fetch when entire warp is out of work [Aila and Laine 2009]
    - Original: Fetch when more than 8 lanes inactive [Aila and Laine 2009]



Improving SIMD utilization

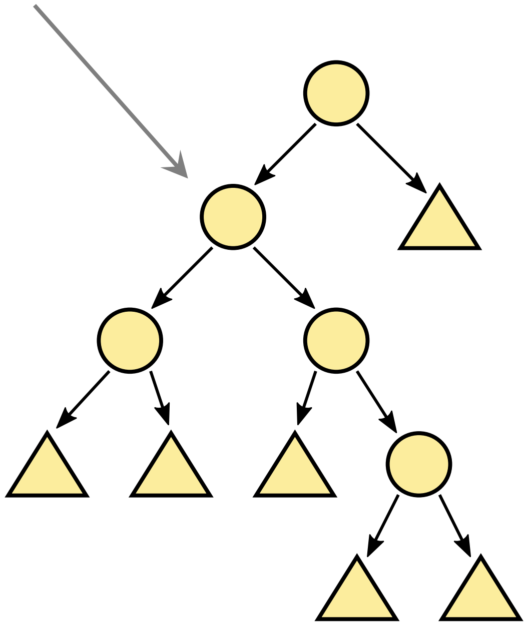
Replacing terminated rays

- Fetch new rays to replace terminated ones:
  - Persistent threads: Fetch when entire warp is out of work [Aila and Laine 2009]
  - Original: Fetch when more than 8 lanes inactive [Aila and Laine 2009]
  - Improved: Keep track of lost work in the warp since last ray fetch, fetch when a threshold is exceeded

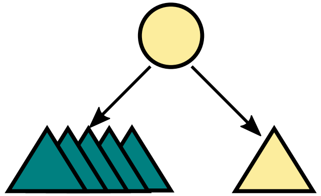


Constructing wide BVHs

- For each node in the binary BVH, starting from bottom:
    - Compute optimal SAH cost for subtree, 3 options



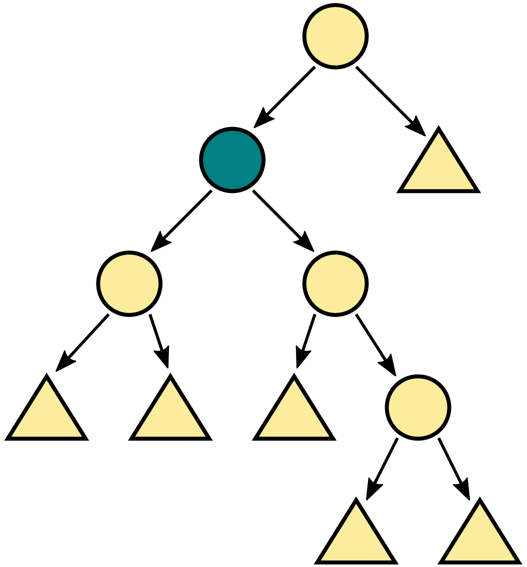
Constructing wide BVHs



- For each node in the binary BVH, starting from bottom:
  - Compute optimal SAH cost for subtree, 3 options
    - Create leaf

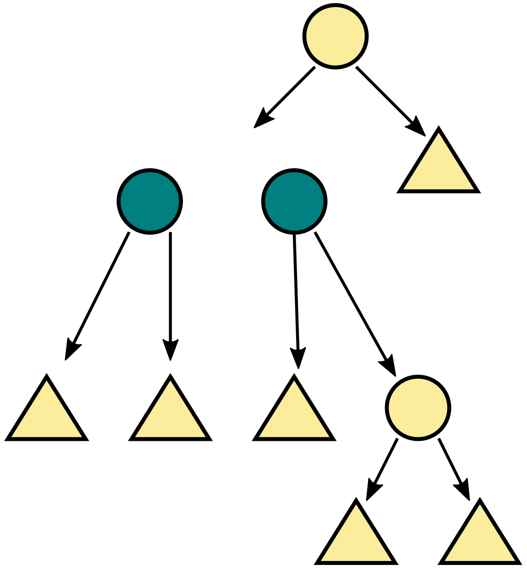
Constructing wide BVHs

- For each node in the binary BVH, starting from bottom:
  - Compute optimal SAH cost for subtree, 3 options
    - Create leaf
    - Create internal node



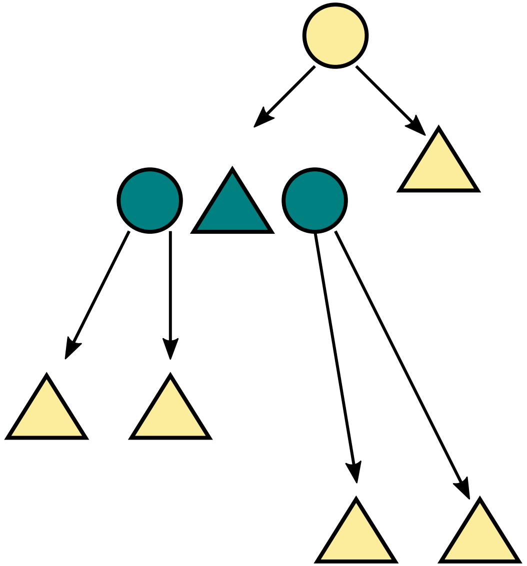
Constructing wide BVHs

- For each node in the binary BVH, starting from bottom:
  - Compute optimal SAH cost for subtree, 3 options
    - Create leaf
    - Create internal node
    - Create forest with 2 - 7 roots



Constructing wide BVHs

- For each node in the binary BVH, starting from bottom:
  - Compute optimal SAH cost for subtree, 3 options
    - Create leaf
    - Create internal node
    - Create forest with 2 - 7 roots



Constructing wide BVHs

- For each node in the binary BVH, starting from bottom:
  - Compute optimal SAH cost for subtree, 3 options
    - Create leaf
    - Create internal node
    - Create forest with 2 - 7 roots
- Backtrack decisions starting from root and create wide nodes so that optimal cost is realized.