RTX-RSim

Accelerated Vulkan Room Response Simulation for Time-of-Flight Imaging

Peter Thoman, Markus Wippler,
Robert Hranitzky, and Thomas Fahringer

peter.thoman@uibk.ac.at

IWOCL
2020

tofmotion depth sensing and imaging est. 2016

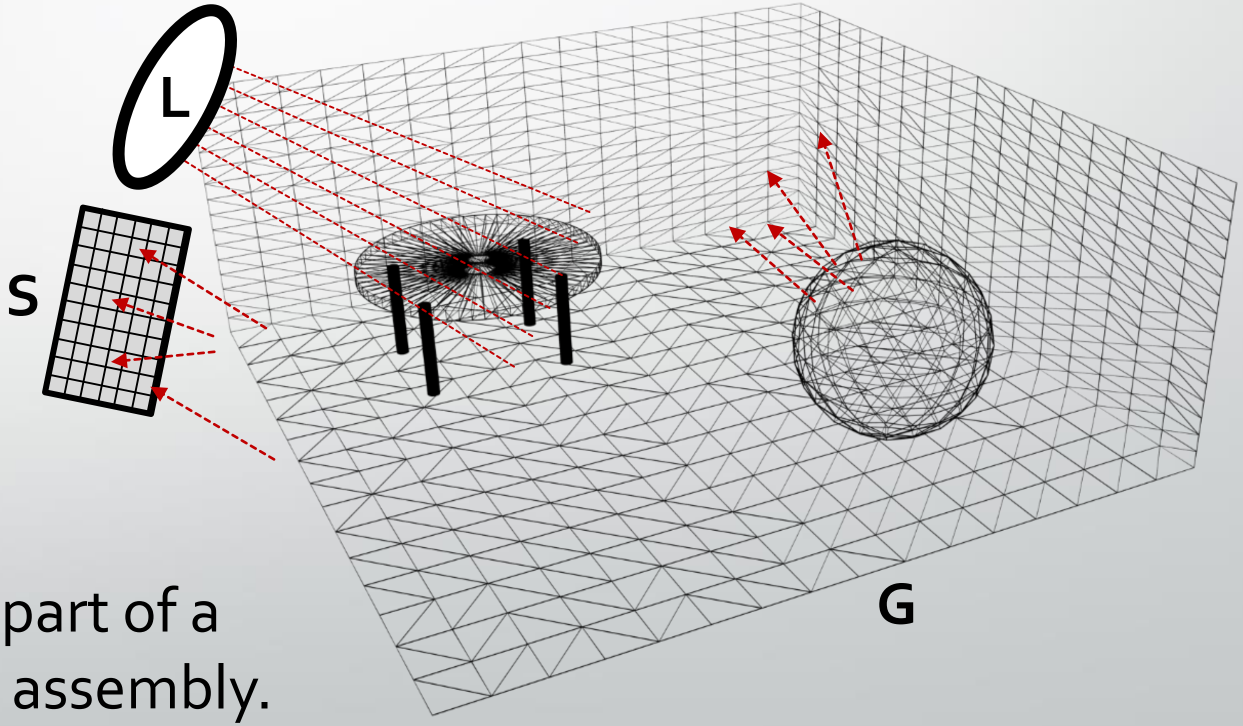
universität innsbruck

DPS Distributed and Parallel Systems

Background and Motivation

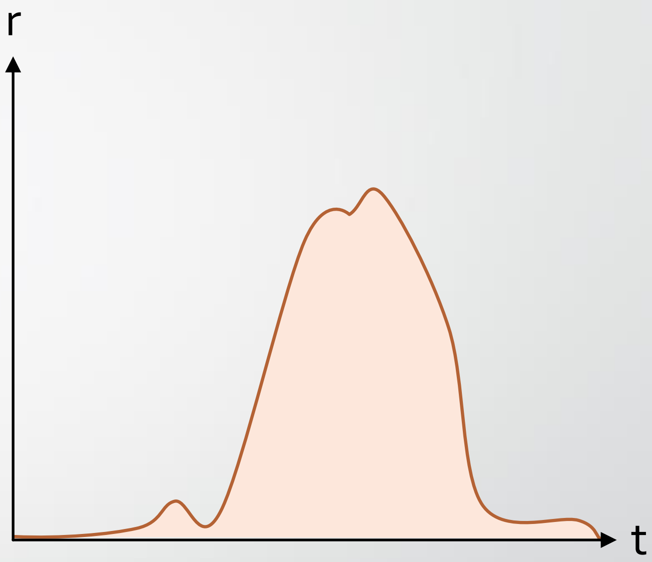
The Basic Idea

- In room response simulation for time of flight imaging, we are interested in computing the **propagation of light**
  - from a light source (**L**)
  - through a room (defined by some geometry and surface properties **G**)
  - to a sensor array (**S**)



In the real world, **L** and **S** are part of a **Time-of-flight (ToF) camera** assembly.

The Goal

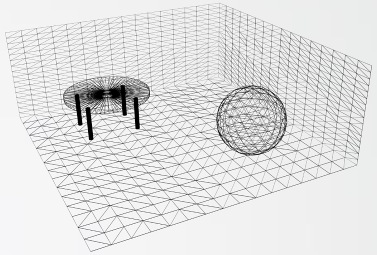


- Unlike in e.g. image rendering or lighting computations, the goal of the simulation is to compute a radiosity **time series** for **each geometric primitive**
- Based on this time series, which simulates the actual photons received by a ToF camera sensor, scene **depth** can be reconstructed
- With RSim, since the *exact* depth is known, different scenes and reconstruction schemes can be easily evaluated

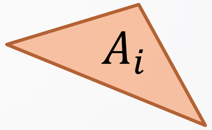
→ Use during development of better ToF hardware implementations or software algorithms

Algorithm Overview

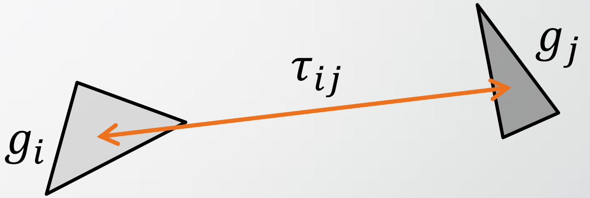
1. Read input data, including geometric primitives (G), their surface material information (ρ), and initial impulse



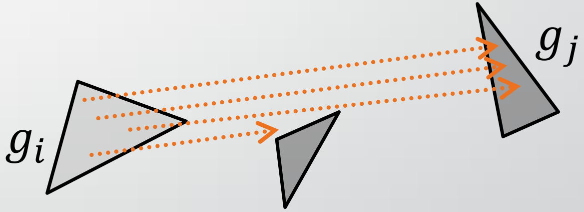
2. Pre-computation of the per-triangle area (A_i)



3. Mutual signal delay computation, storing the signal delay for each triangle pair (g_i,g_j) in τ_{ij}



4. Mutual visibility computation, evaluating the energy transfer between each triangle pair stochastically and storing in K_{ij}



5. For each timestep $t \in [0,T)$:
    - Propagate radiosity, computing $rad_{t,i}$ for each triangle g_i in all pairs (g_i,g_j) based on K_{ij} and $rad_{t-1,i}$
6. Compute the distance from the light/sensor position to each triangle g_i, based on $rad_{[0,T),i}$

Algorithm Performance and Data Requirement Analysis

Algorithm Steps

1. Input data prep.
2. Pre-compute A_i
3. Pre-compute τ_{ij}
4. Mutual visibility comp. → K_{ij}
5. Radiosity propagation → $rad_{[0,T),i}$
6. Compute distance

Analyse time complexity for each step of the algorithm.

Algorithm Steps

1. Input data prep.
2. Pre-compute A_i
3. Pre-compute τ_{ij}
4. Mutual visibility comp. → K_{ij}
5. Radiosity propagation → $rad_{[0,T),i}$
6. Compute distance

Steps 1 and 2 iterate over $\boldsymbol{N}$ triangles, with simple I/O operations and area computation for each element.

Readily identified as $\boldsymbol{O}(\boldsymbol{N})$ complexity.

Algorithm Steps

1. Input data prep.
2. Pre-compute A_i
3. Pre-compute τ_{ij}
4. Mutual visibility comp. → K_{ij}
5. Radiosity propagation → $rad_{[0,T),i}$
6. Compute distance

Computing propagation delay for each pair of triangles → $\boldsymbol{O(N^2)}$

However, the fixed factor is low, and compared to the remaining phases, even $\boldsymbol{N^2}$ complexity is largely negligible.

Algorithm Steps

1. Input data prep.
2. Pre-compute A_i
3. Pre-compute τ_{ij}
4. **Mutual visibility comp. → K_{ij}**
5. Radiosity propagation → $rad_{[0,T),i}$
6. Compute distance

Stochastically evaluate the visibility between every pair of triangles – in naïve implementation requires a ray-triangle intersection check against *all* other triangles in the scene. With $\boldsymbol{S}$ stochastic samples: → $O(N^3 * S)$.

In practice, use geometric acceleration structure. Current RSim on CPU uses octrees, resulting in a reduction of average-case query complexity from $O(N)$ to $O(\log(N))$.

➔ $\boldsymbol{O(N^2 * log(N) * S)}$

Algorithm Steps

1. Input data prep.
2. Pre-compute A_i
3. Pre-compute τ_{ij}
4. Mutual visibility comp. → K_{ij}
5. Radiosity propagation → $rad_{[0,T),i}$
6. Compute distance

Uses signal delay τ_{ij} and mutual visibility information K_{ij}, as well as the previous radiosity up to the currently computed timestep $rad_{[0,t),i}$.

For each timestep t and each pair (g_i,g_j):

Propagate energy between triangles in the pair from time $t - \tau_{i,j}$ according to mutual visibility as well as their surface properties.

→ $\boldsymbol{O}(\boldsymbol{N}^{\mathbf{2}} * \boldsymbol{T})$

Algorithm Steps

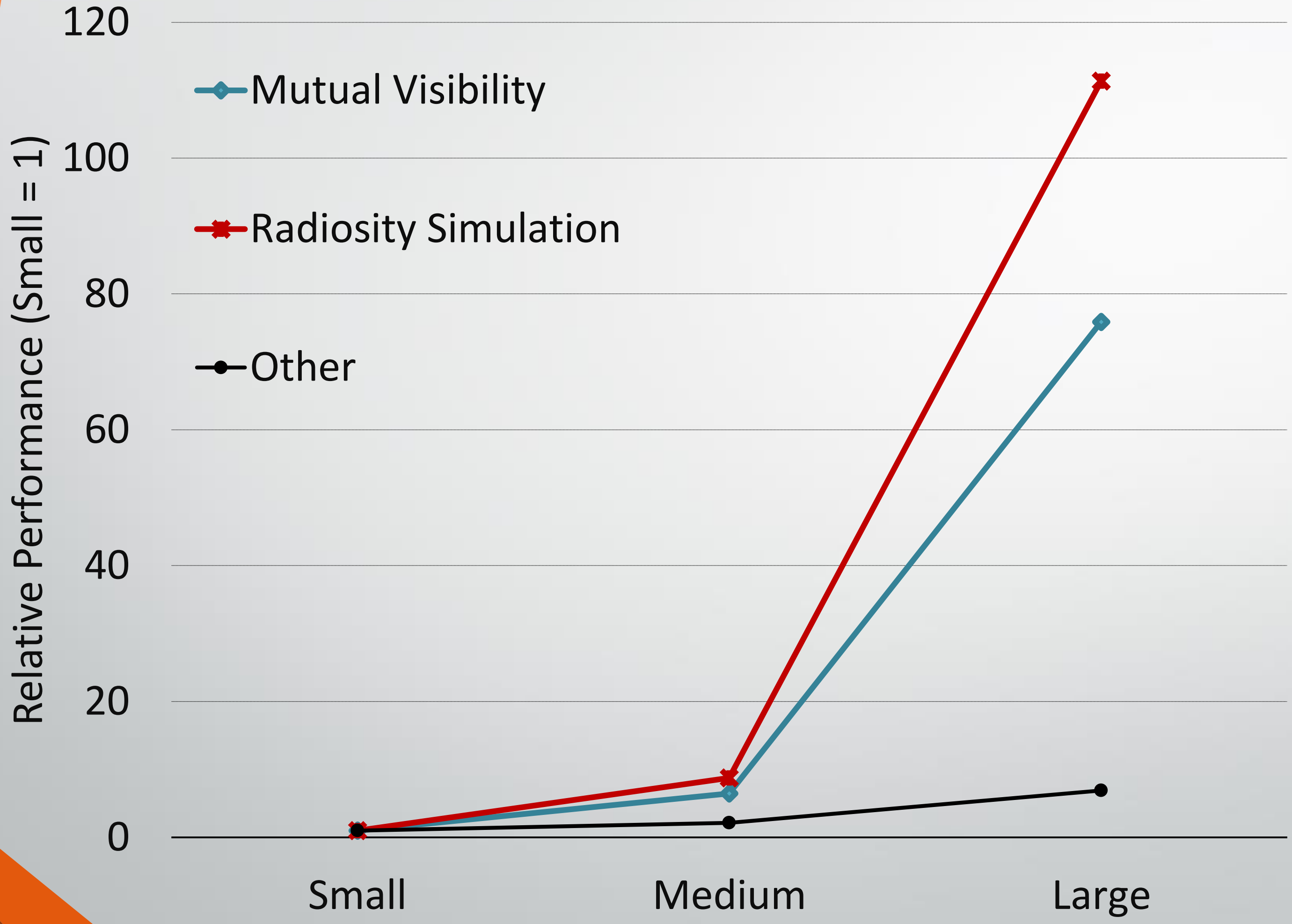
1. Input data prep.
2. Pre-compute A_i
3. Pre-compute τ_{ij}
4. Mutual visibility comp. → K_{ij}
5. Radiosity propagation → $rad_{[0,T),i}$
6. Compute distance

Distance computation usually based on cross-correlation of radiosity time series.

→ $\boldsymbol{O(N * T^2)}$

T is usually much smaller than N, and fixed factor is very small as well. Usually negligible overall, similar to step 3.

Measured Performance



- Scaling trend **matches observations** on algorithmic complexity
- Clearly **mutual visibility computation** and **radiosity simulation** are main priority

Vulkan Raytracing and Compute for Room Response Simulation

Data Management

- A Vulkan implementation needs to be massively data-parallel to be efficient
- And we are constrained in the amount of data we can store on a GPU

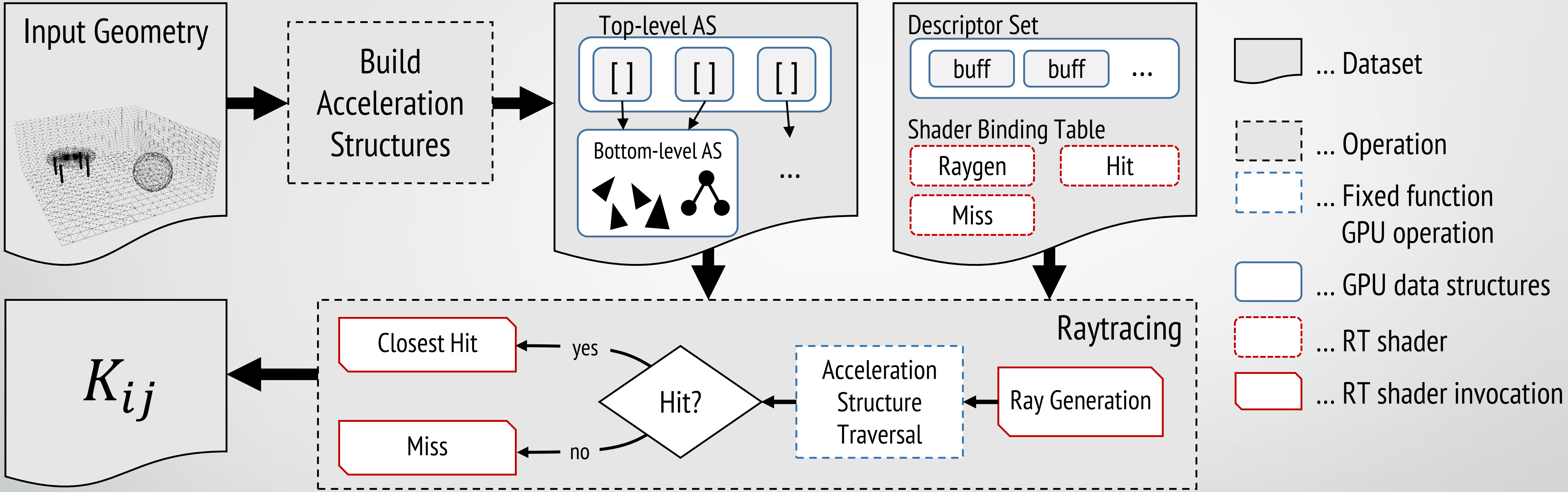
→ **Data-centric** view of the algorithm

Data Management

Contents	Format	Size
Triangles (G)	Indexed vertex buffer	N
Material information (ρ)	$3 * \text{FP32}$	N
Raytracing Buffers	Internal / opaque	$O(N)$
Sample Coordinates	$2 * \text{FP32}$	S
Mutual Visibility (K_{ij})	FP16	N^2
Radiosity (rad)	$4 * \text{FP32}$	$N * T$
Distance	FP32	N

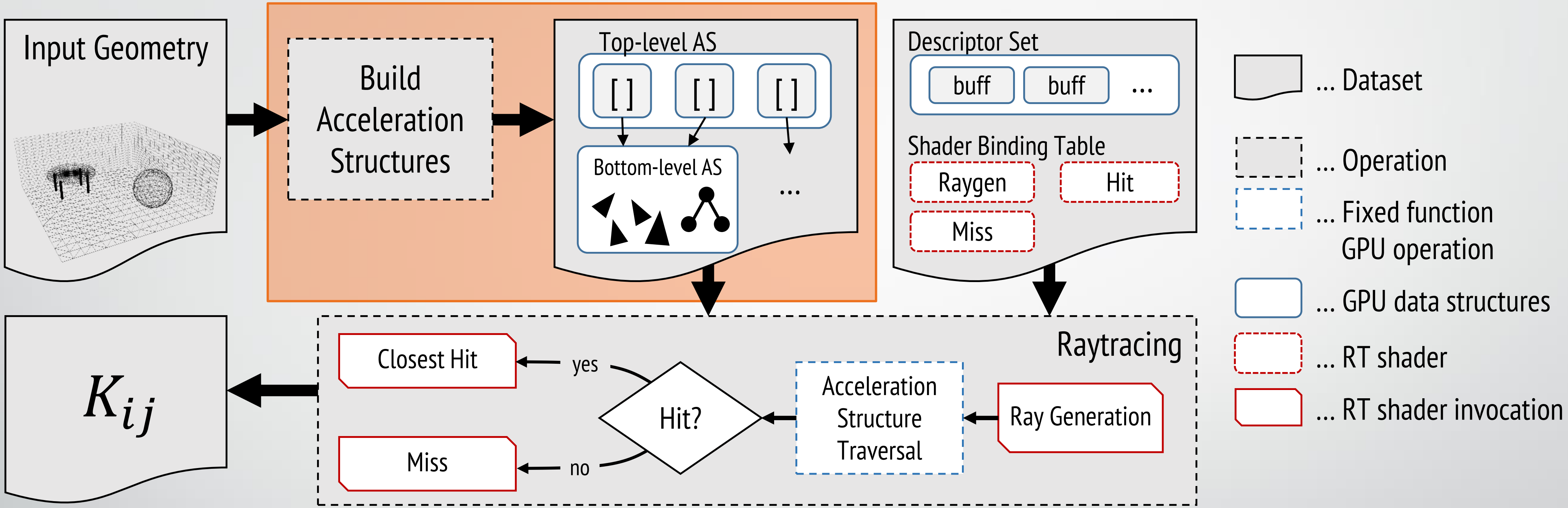
- Generally, $S \ll T \ll N$, therefore K_{ij} dominates. → FP16 sufficient!
- Signal delay τ_{ij} recomputed instead of stored.

Hardware Raytracing for Mutual Visibility



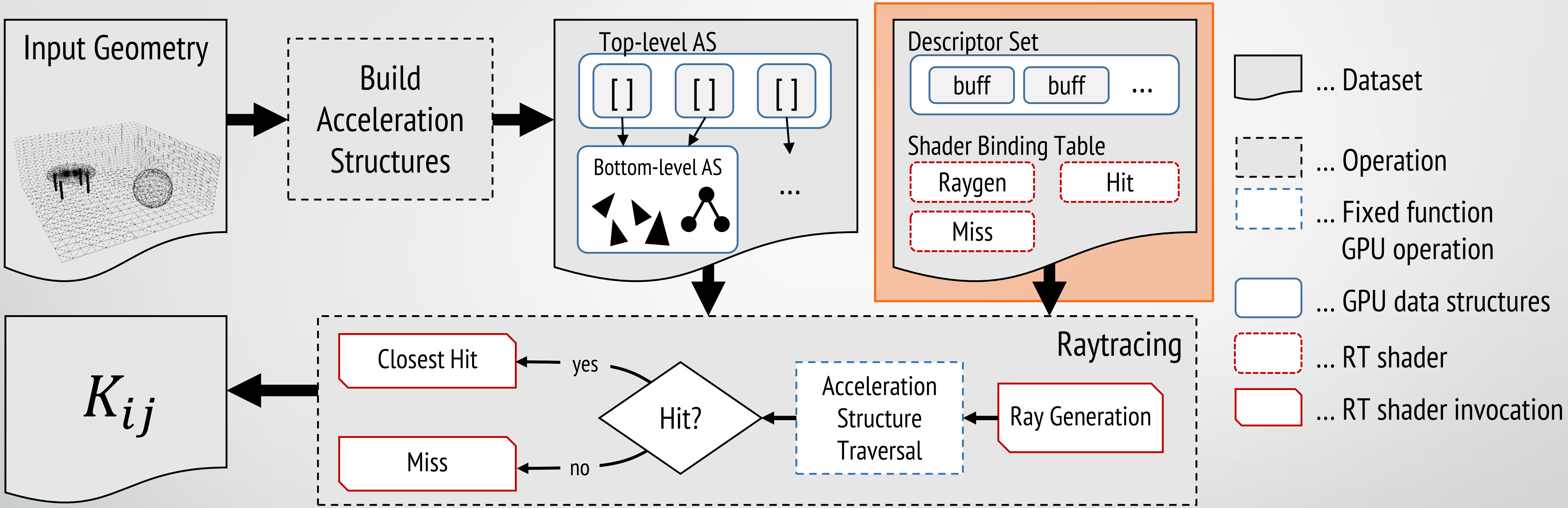
- Schematic representation of HW raytracing process

Hardware Raytracing for Mutual Visibility



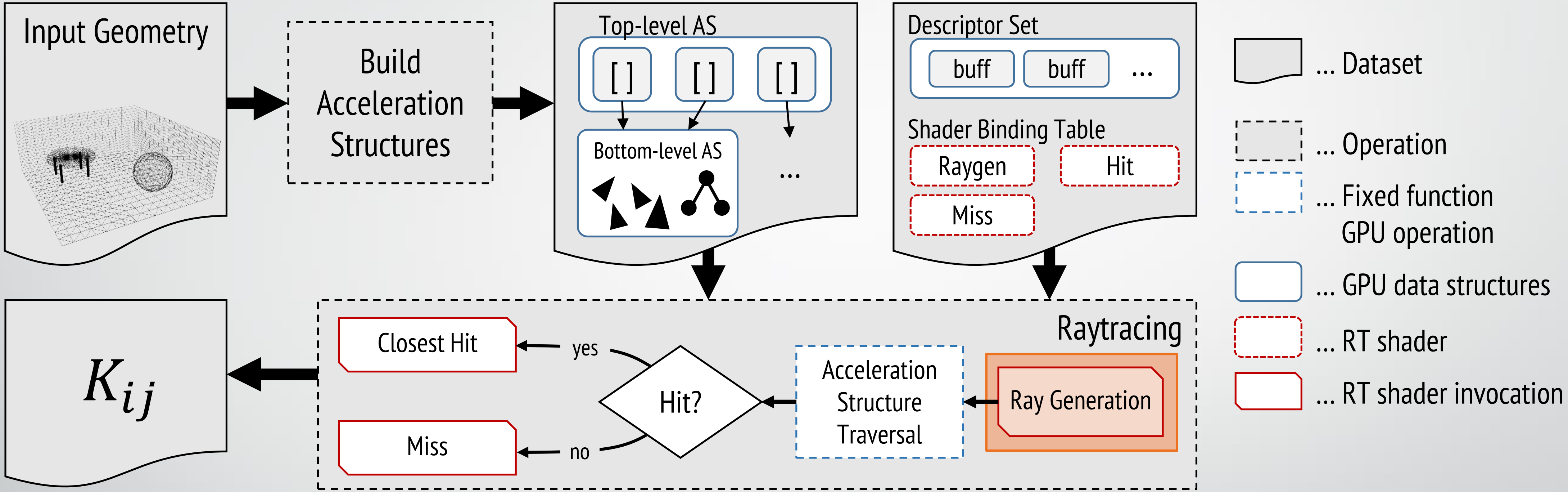
- Geometry is static → we can optimize AS build for traversal speed rather than build/update performance

Hardware Raytracing for Mutual Visibility



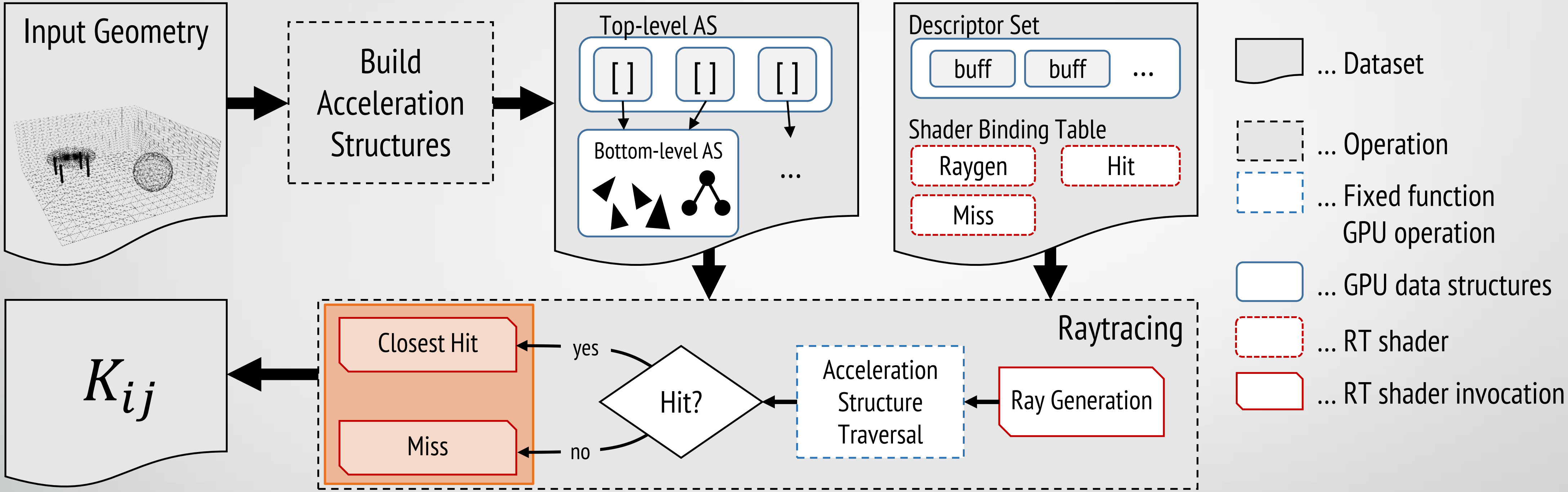
- Descriptor Set: our RT shaders require read-only access to G, ρ, and the Sample Coordinates buffer, as well as write access to K_{ij}
- Shaders: only require ray generation and a single hit and miss shader

Hardware Raytracing for Mutual Visibility



- Ray generation: generate S rays for every pair of triangles (order independent, thus $N^2/2 - N$ required size, 1D grid)
- Aggregate results and write to K_{ij}

Hardware Raytracing for Mutual Visibility



- Miss shader: trivial, simply set visible=false for use in raygen shader
- Closest hit: check if expected triangle hit

Compute Shader Radiosity Simulation

- Second compute-intensive phase, based on mutual visibility result from HW raytracing
- Implemented using Vulkan compute shaders
  - One shader invocation per time step
  - Important: parallelized in 1D over N, not 2D over N^2
    → slightly lower potential at small sizes, but less synchronization

Simplified Radiosity Compute Shader

- Excerpt of core loop over destination triangles
- Note data-dependent access to previous radiosity buffer

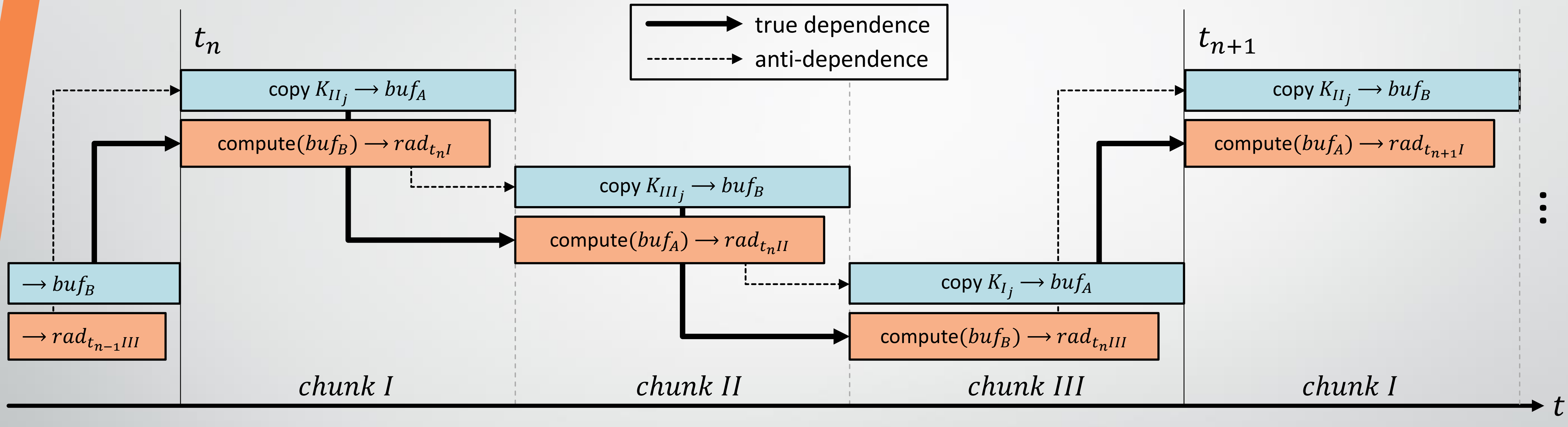
```
for(uint dstTriIdx = dstTriStart;
    dstTriIdx < dstTriEnd; ++dstTriIdx) {

  if(srcTriIdx == dstTriIdx) continue;

  const mat3 dstTri = getTriangle(dstTriIdx);
  const int tauij = calcTauij(srcTri, dstTri);

  // Skip if wave hasn't yet propagated between ←↩
       triangle i and triangle dstTriIdx.
  if(timestep < tauij) continue;

  // Use radiosity from the point in time where ←↩
       emission actually took place.
  const Radiosity link =
    radBuffer.r[(timestep - tauij) * N + dstTriIdx];

  if(link.B == 0.0f && link.Z == 0.0f) continue;

  const uint index = (dstTriIdx - dstTriStart)
                     * N + srcTriIdx;
  const kij_t kij = curKijBuffer.f[index];
  if(kij == 0.0f) continue;

  radBuffer.r[timestep * N + srcTriIdx].B +=
    clamp01(kij * calcArea(dstTri)) * link.B;
  radBuffer.r[timestep * N + srcTriIdx].Z +=
    clamp01(kij * calcArea(srcTri)) * link.Z;
}
```

Data Streaming with Latency Hiding

Streaming Motivation

- Recall that mutual visibility buffer K_{ij} requires N^2 entries
- Therefore GPU memory limited to low triangle counts
- Recomputation is not desirable → slowdown by at least factor 10
- Solution: asynchronous streaming
  - Minimize performance impact by suitable chunking and latency hiding

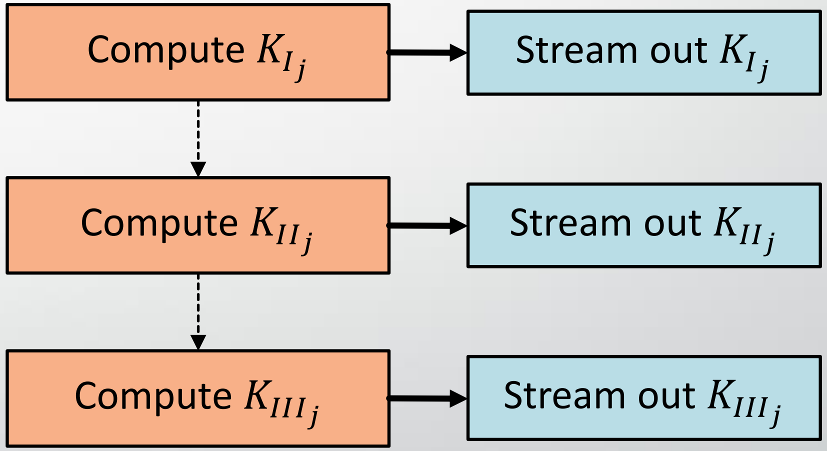
RTX-RSim Streaming Scheme



- Requires two extra chunk buffers for double buffering
  - Linear rather than quadratic in size!

Streaming for Mutual Visibility

- Mutual Visibility step generates K_{ij} → also requires streaming
- Implementation simpler, only need to stream the finished data out once
- Also less performance critical, since mutual visibility computation has higher per-element cost
  - → We actually see speedup with streaming in some results!



Compute K_{I_j}

Stream out K_{I_j}

Compute K_{II_j}

Stream out K_{II_j}

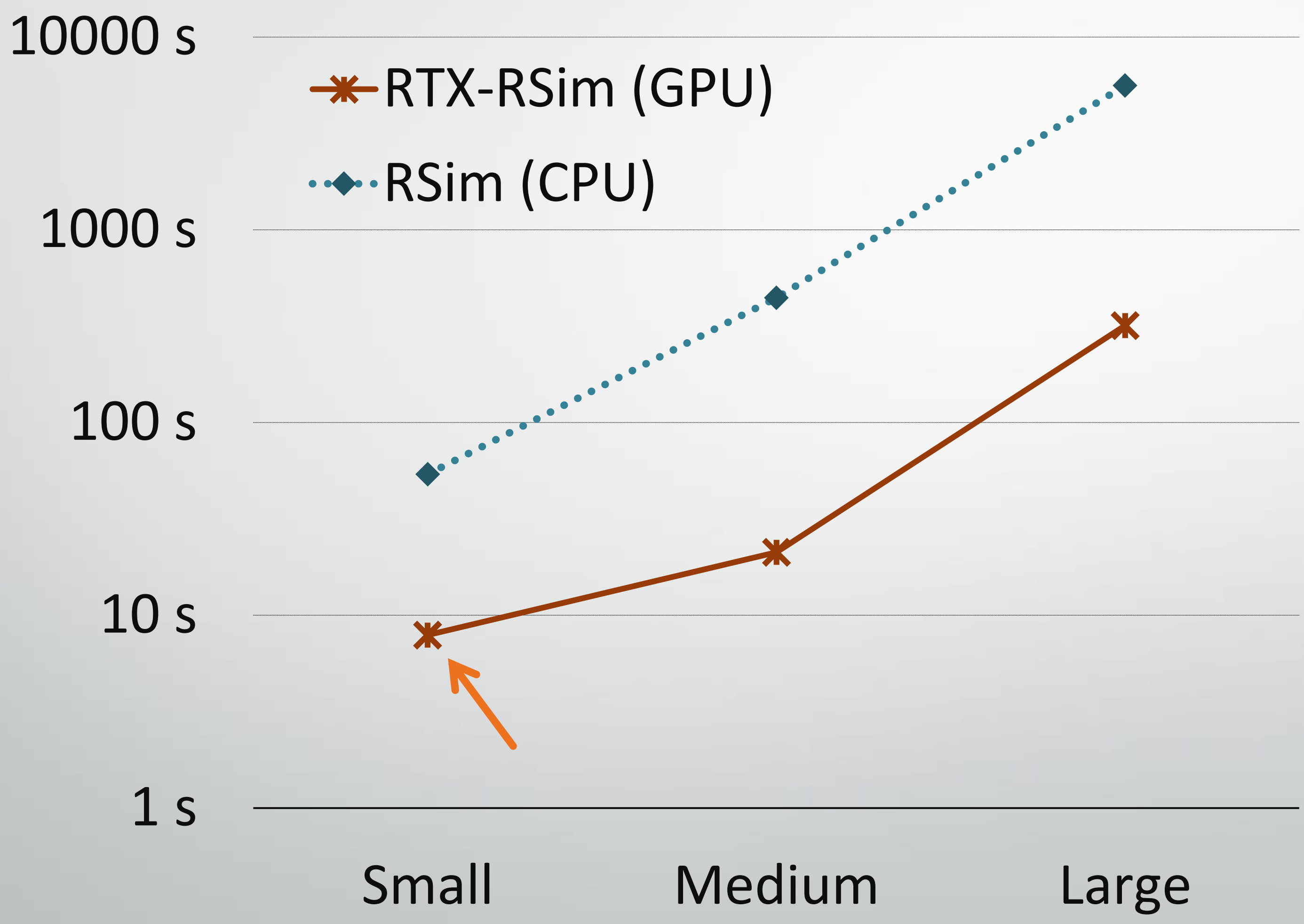
Compute K_{III_j}

Stream out K_{III_j}

Performance Evaluation

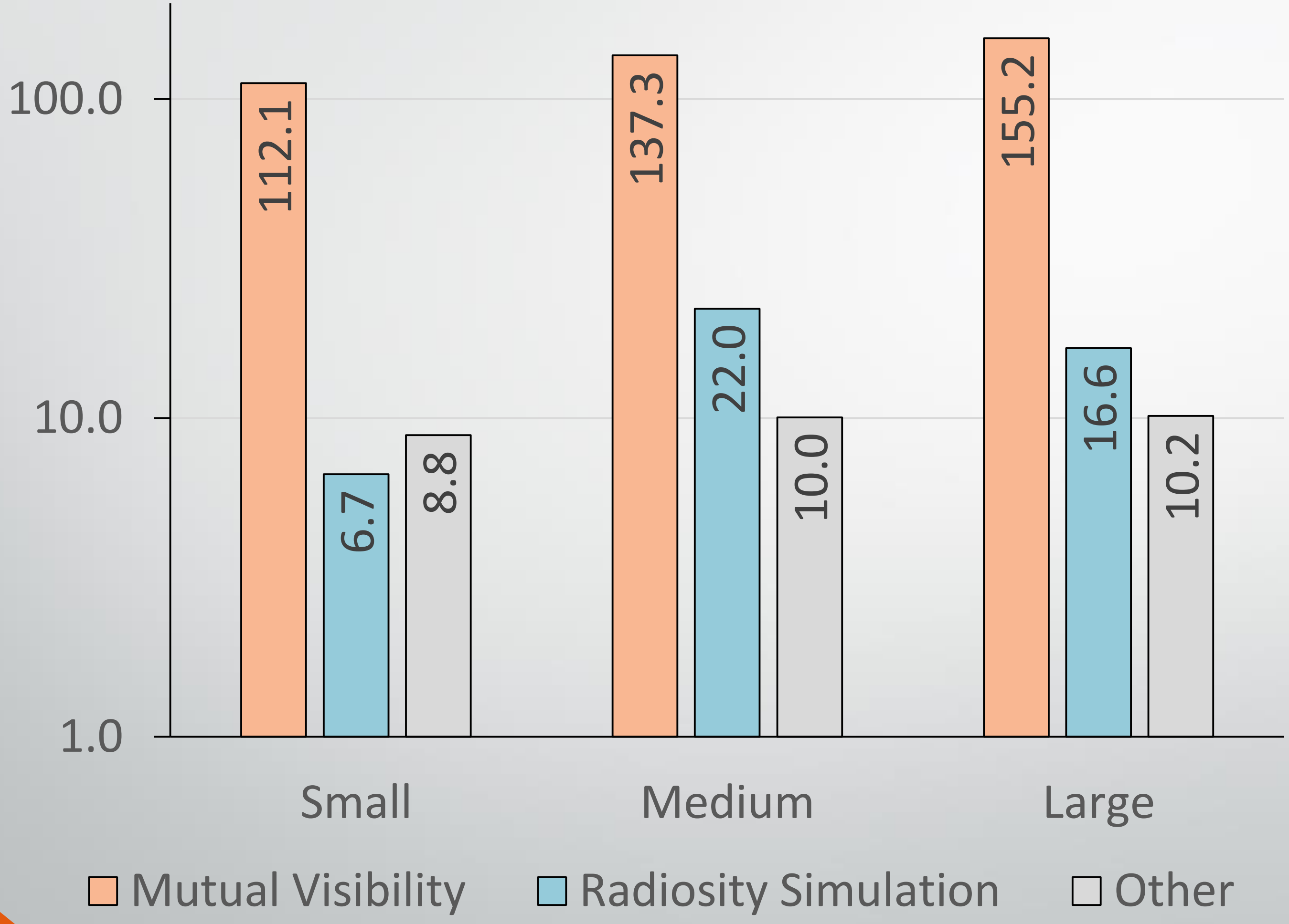
All results on an AMD Ryzen TR 2920X + NVIDIA GeForce RTX 2070 system
Note that CPU results are fully parallelized

Overall CPU vs. RTX-RSim Comparison



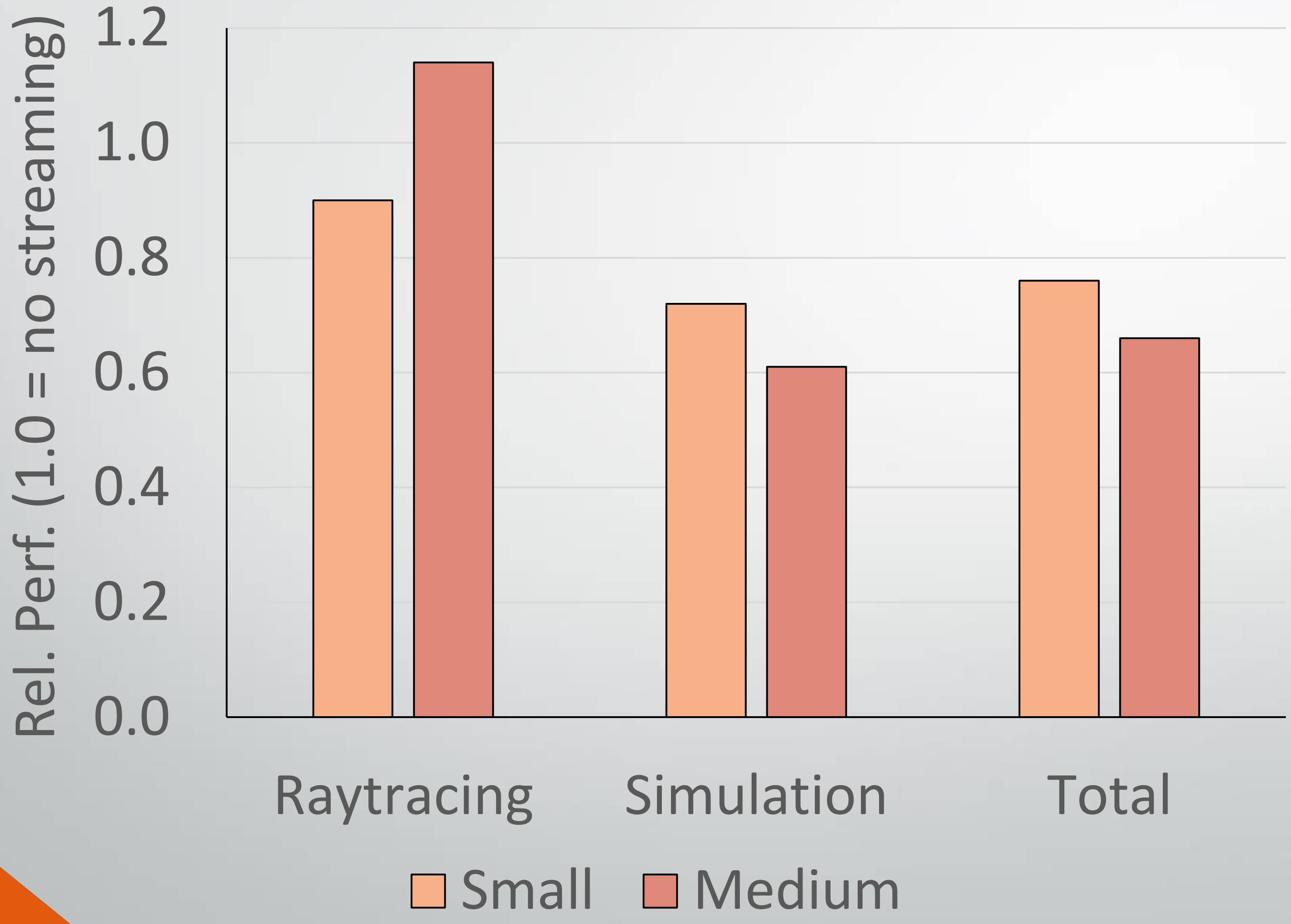
- CPU results roughly linear on logarithmic scale
- GPU result worse at “Small” size (insufficient parallelism in radiosity comp.)
- Factor ~20 improvement over CPU at “Medium” and larger

Speedup of individual phases



- Very high speedup in mutual visibility phase with hardware raytracing
- Radiosity simulation limited by:
  - lack of parallelism at “Small” size
  - streaming requirements at “Large” size

Streaming Performance Impact



- Raytracing actually benefits from streaming (hiding some transfer latency)
- Roughly 40% performance impact on radiosity simulation due to streaming
  - Not ideal, but order of magnitude better than recomputation

Summary & Conclusion

Conclusion

- Using new raytracing hardware for accelerating room response simulation is both viable and effective
  - → Over factor 100 improvement in raytracing-heavy phases compared to CPU
- Vulkan compute shaders are a good cross-platform and cross-vendor alternative to e.g. CUDA, OpenCL and SYCL if direct interaction with graphics features is required
- Streaming with full latency hiding allows overcoming GPU memory limits for this algorithm with moderate performance impact
  - But is still limited by PCIe bandwidth

Thank you for your attention!

peter.thoman@uibk.ac.at

Partially funded by the FFG INPACT project.

tofmotion
depth sensing and imaging
est. 2016

universität
innsbruck

DPS
Distributed and Parallel Systems