Megakernels Considered Harmful: Wavefront Path Tracing on GPUs

Samuli Laine
Tero Karras
Timo Aila
Path Tracing Overview

- Cast a ray from the camera through the pixel
- At hit point, evaluate material
  - Determine new incoming direction
  - Update path throughput
- Cast a shadow ray towards a light source
- Cast the extension ray and repeat

- At some depth, start using Russian roulette to terminate path probabilistically
  - Avoids infinitely long paths
Problems in Megakernel Path Tracer

- You can put all code in one kernel, even on a GPU
  - Ray casts, evaluators and samplers for all materials, evaluators and samplers for all light source types, path tracing logic, MIS, new path generation, etc.

**BUT:**

- Lots of code
  - Bad for instruction cache
- Lots of registers used (based on hottest spot)
  - Bad for latency hiding capacity
- Lots of execution divergence
  - Bad for SIMT execution model (warps of 32 threads)
Problems in Megakernel Path Tracer

You can put all code in one kernel, even on a GPU
- Ray casts, evaluators and samplers for all materials, evaluators and samplers for all light source types, path tracing logic, MIS, new path generation, etc.

**BUT:**
- Lots of code
  - Bad for instruction cache
- Lots of registers used (based on hottest spot)
  - Bad for latency hiding capacity
- **Lots of execution divergence**
  - Bad for SIMT execution model (warps of 32 threads)
Execution Divergence in Path Tracing

- Paths may hit different materials
  - Each material has its own code

- Paths may terminate at different lengths
  - Various reasons
  - This issue has been investigated before
    - Solutions, e.g., path regeneration are known, but they are not very effective

- Not all materials produce a shadow ray
  - BSDFs with Dirac distribution (mirror, glass)
Execution Divergence in Path Tracing

- Paths may hit different materials
  - Each material has its own code

- Paths may terminate at different lengths
  - Various reasons
  - This issue has been investigated before
    - Solutions, e.g., path regeneration are known, but they are not very effective

- Not all materials produce a shadow ray
  - BSDFs with Dirac distribution (mirror, glass)
Real-World Materials
or
How Bad Can It Be?
Materials Are Expensive

- Composed of multiple BSDF layers
- Non-trivial BSDFs
- Procedural noise everywhere
- Huge textures*

* Not addressed in this work
Example: Car Paint

- Four layers
  - Fresnel coat
  - Glossy flakes 1 & 2
  - Diffuse base
- Coat is a simple dielectric BSDF with Fresnel weight
- Flakes are Blinn-Phong BSDFs with procedural colors and normals
- Base is a diffuse BSDF with angle-dependent color
Example: Noise Evaluator

```c
// Compute noise correction as functions of input noise
// If N > 0.5dB, everything is a tensor product so ...  
unsigned int u0;

unsigned int noise_factors[10];
unsigned int noise_factors[20];
unsigned int noise_factors[30];
unsigned int noise_factors[40];
unsigned int noise_factors[50];
unsigned int noise_factors[60];
unsigned int noise_factors[70];
unsigned int noise_factors[80];
unsigned int noise_factors[90];
unsigned int noise_factors[100];

// Compute noise correction
// ...  

```

Example: The Entire Car Paint
And This Isn’t Even a Difficult Case

- Only four layers
- No textures
  - No procedural texcoords
  - No filtering
  - No out-of-core
- No iterative stuff

Still ~2x as expensive to evaluate as a ray cast

- In this scene, probably closer to 10x
Problem: How to Evaluate Materials Efficiently?

- Worst case: Every thread hits a different, expensive material
- Megakernel runs each sequentially with abysmal SIMD utilization
- We really need to do better than that
  - Otherwise the materials will dominate
Let’s Solve Everything
It’s Business Time

The recipe to reorganize path tracing to be more GPU-friendly:

1. Remove the loop around path tracer
   - Avoid consequences of variable path length
   - Also enables the two other optimizations

2. Place requests for operations into queues
   - Ray casts, material evaluations, etc.
   - Avoids threads idling when executing conditional code

3. Execute operations in separate kernels
   - Minimize register pressure and instruction cache usage
     - Avoid losing latency hiding capacity unnecessarily
Step 1: Remove the Loop

- Keep a pool of paths alive all the time
  - E.g., one million paths

- At each iteration, advance each of them by one segment
  - Cast rays, evaluate materials and lights, accumulate radiance, update throughput, run the roulette, etc.

- If path terminates, generate a new one in its place
  - Ensure there’s always paths to work on

- Similar to previous work
  [e.g. Wald 2011, Van Antwerpen 2011, etc.]
Pros and Cons

+ Variable path length is not an issue anymore
  As previously noted

+ Allows further optimizations
  Collecting requests in operation-specific queues and scheduling them individually
  This is the big one! Really hard to do in the megakernel approach

- Path state must reside in memory
  A simple loop-based method can keep it in registers
  Not as bad as it sounds if we use a good memory layout (SOA)

- Less “natural” to implement
  But only until you get used to it

- Doesn’t buy you much performance alone
Step 2: Per-Operation Queues

- Allocate a queue for each primitive operation request
  - Extension ray casts
  - Shadow ray casts
  - New path generation
  - Material evaluations
    - With separate queues for individual materials

- Place requests compactly (i.e., no gaps) into queues

- When executing, use one thread per request
  - Every thread will have an item to work on
  - Every thread will be doing the same thing, so there’s very little execution divergence!
Step 3: Individual Kernels for Each Operation

- We already have well-optimized kernels for ray casts from previous research
  - Now we can use them directly
  - Optimized for low register count and high perf

- Let each material have its own kernel
  - Some are simple and need few registers
    - Combining these into one kernel is sometimes a good idea
  - Some are complex and need many registers

- Smaller code $\rightarrow$ Won’t overrun instruction caches
Implementation
Always operates on all paths in pool

Each kernel has its own queue

Each type of ray has its own queue
The Logic Kernel

- Does not need a queue, operates on all paths
- Does “everything except rays and materials”
  - If shadow ray was unblocked, add light contribution
  - Find material and/or light source that ext ray hits
  - Apply Russian roulette if limit depth exceeded
  - If path terminated, accumulate to image
  - Apply depth-dependent extinction for translucent materials
  - Generate light sample by sampling light sources
  - Place path in proper queue according to material at hit
    - Or in “new path” queue if path terminated
New Path Kernel

- Generate a new image-space sample
  - Based on a global path index

- Generate camera ray
  - Place it into extension ray queue

- Initialize path state
  - Init radiance, throughput, pixel position, etc.
  - Initialize low-discrepancy sequence for the path, used when generating random numbers in samplers
Material Kernels

- Generate incoming direction
- Evaluate light contribution based on light sample generated in the logic kernel
  - Even though we haven’t cast the shadow ray yet
- Get the probability of acquiring the light sample from the sampling of incoming direction
  - Needed for MIS weights
- By evaluating all of these in one go, we can discard the BSDF stack immediately afterwards
- Generate extension ray and potential shadow ray
  - Place in respective queues
Ray Cast Kernels

- **Extension rays**
  - Find first intersection against scene geometry
  - Utilize optimized kernels from previous research
  - Store hit data into path state

- **Shadow rays**
  - We only need to know if the shadow ray is blocked or not
  - Cheaper than finding the first intersection
Results
# Test Scenes and Performance

![Carpaint](image1.png) ![City](image2.png) ![Conference](image3.png)

<table>
<thead>
<tr>
<th>scene</th>
<th>#tris</th>
<th>performance megakernel</th>
<th>(Mpaths/s) wavefront</th>
<th>speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>CARPAINT</td>
<td>9.5K</td>
<td>42.99</td>
<td>58.38</td>
<td>36%</td>
</tr>
<tr>
<td>CITY</td>
<td>879K</td>
<td>5.41</td>
<td>9.70</td>
<td>79%</td>
</tr>
<tr>
<td>CONFERENCE</td>
<td>283K</td>
<td>2.71</td>
<td>8.71</td>
<td>221%</td>
</tr>
</tbody>
</table>

Note: Megakernel has path regeneration
Execution Time Breakdown

<table>
<thead>
<tr>
<th>scene</th>
<th>logic</th>
<th>new path</th>
<th>materials</th>
<th>ray cast</th>
</tr>
</thead>
<tbody>
<tr>
<td>CARPAINT</td>
<td>2.40</td>
<td>0.86</td>
<td>2.31</td>
<td>4.31</td>
</tr>
<tr>
<td>CITY</td>
<td>3.42</td>
<td>0.86</td>
<td>5.47</td>
<td>12.53</td>
</tr>
<tr>
<td>CONFERENCE</td>
<td>3.01</td>
<td>0.79</td>
<td>6.37</td>
<td>9.62</td>
</tr>
</tbody>
</table>

(times in milliseconds / 1M path segments)

The most important takeaway: Ray casts are **not** the only expensive part!
- Optimizations yield diminishing returns already
Conclusions

- Path tracing can be reorganized to suit GPUs better
  - Bonus: Also works in practice

- Going to the future, there is no limit on the number and complexity of materials
  - Divergence will only get worse
  - Megakernels will only get bigger
  - The proposed approach becomes even more appealing

- Time to look beyond accelerating ray casts?
**Future Work**

- Look at other rendering algorithms
  - Bidir path tracing, MLT, etc.
  - There is already good work in this direction [Van Antwerpen, Davidovič et al.]
  - Should add complex materials in the mix!

- What to do about gigantic textures?
  - Run materials that have their textures resident while transferring missing textures on the GPU simultaneously?
  - Put paths “on hold” while their textures are being loaded, and let other paths continue?
  - Always run everything you can, try to make everything else runnable at the same time by async transfers?
Thanks!

Questions