Path Tracing on Parallel Architectures

Morgan McGuire
CS 888 Fall'19
University of Waterloo

This content is described in more detail in The Graphics Codex version 2.17, which will be available after September 17, 2019 on http://graphicscodex.com

We'll study GPU and CPU parallel path tracing, as well as multi-node and out-of-core renderers later in the term. In addition to being fascinating graphics systems, these should be really interesting to any a computer scientist—they are cases of low-level systems issues directly interacting with high-level numerical theory. It is big-tent computer science in action and an argument for understanding the full breadth of our field.

To give you context assumed by the authors of those papers and appreciation of their solutions, in this lecture I will give a high level overview of parallel processor architecture and a brief example of how this affects the preferred implementation of the simple path tracers that I showed last week.

I'm not describing an actual processor architecture but a computational *model* of parallel processors. This model is more accurate for modern computers than the "random access machine" serial model assumed in your first programming courses, where n operations took c*n time [and where that constant was the same for all

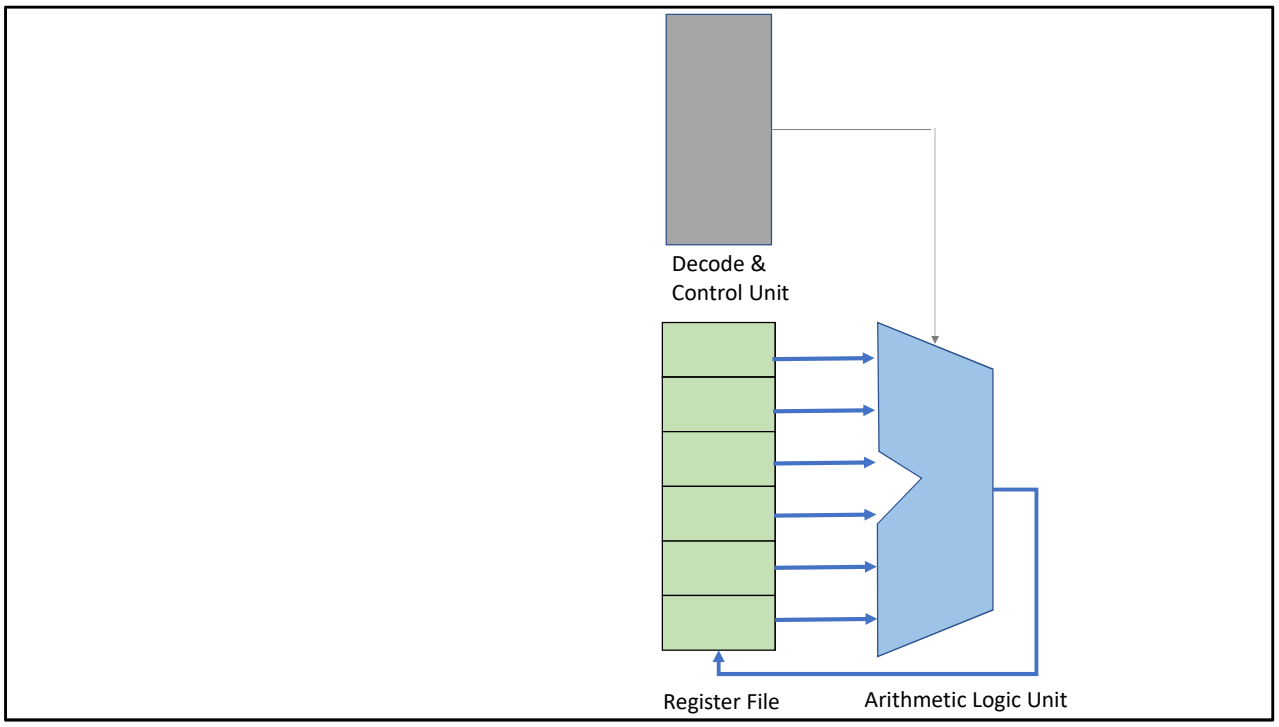
operations!]. Today's model is *also* more accurate than a naïve parallel model in which n operations on m processors take O(n / m) time.

However, the model I'm describing greatly simplifies many aspects of computer architectures. It also abstracts over many differences between CPUs and GPUs. I've chosen this level of detail because I think it is the best tradeoff between accuracy (so that it is actionable), robustness (so that what you've learned is still valid in a few years), and simplicity (so that you can learn it quickly).

I encourage you to learn more about processor architecture details. However, so that you don't feel deprived by the abstractions I'm making today, I should tell you that were you to read the latest Intel processor manual…you *still* wouldn't be learning how that architecture is implemented. You would only learn a more detailed *model* of that processor. The actual architectures of modern processors are fairly different from the interfaces that they present via microcode and drivers. This is partly because they are valuable business secrets, and partly because they change frequently and need to provide a stable higher-level model for developers to work with.

Part 1: A Processor Model

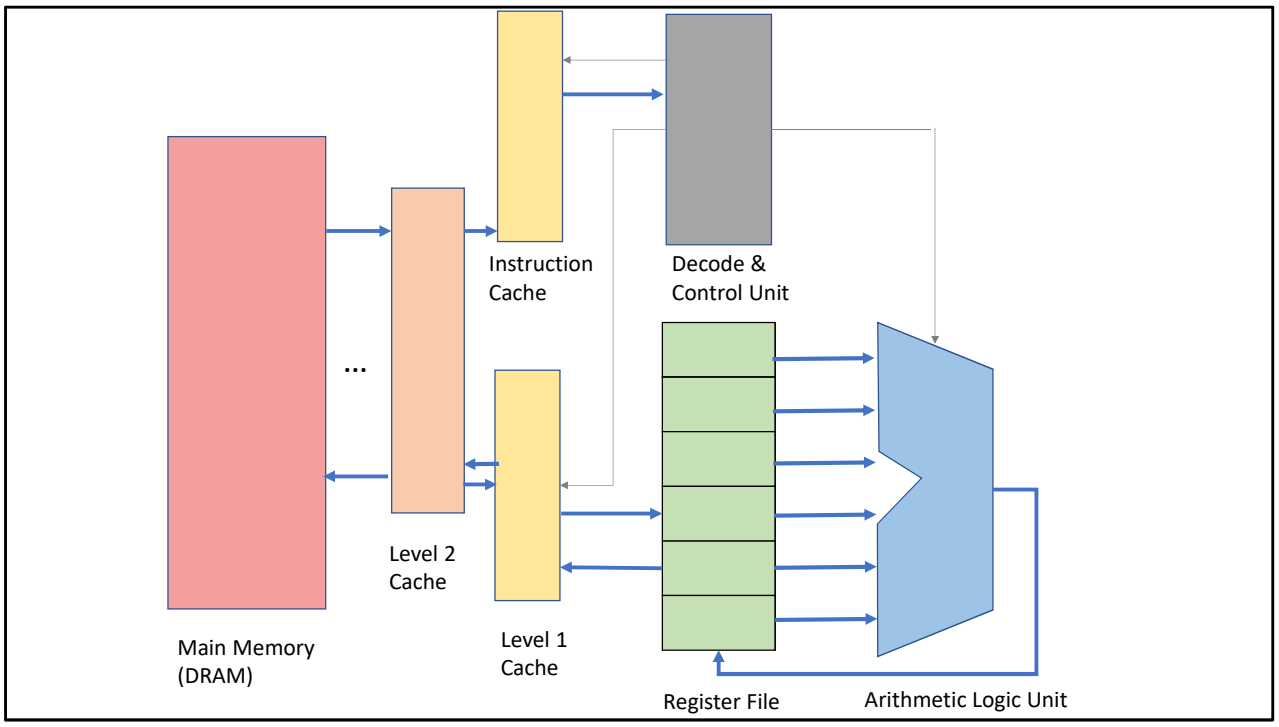
The following slides are derived from the “Production-scale Ray Tracing” section of the
McGuire, Shirley, and Wyman, Introduction to Real-Time Ray Tracing, SIGGRAPH ’19 course
http://rtintro.realtimerendering.com/



Here's a very high level view of how a processor core operates. This is a classic von Neumann architecture.

The main action is between the REGISTER FILE and the ALU.
The CONTROL UNIT sends one or more registers to the ALU, and then selects an operation. The output goes back to the register file.
The "operations" might be +, -, *, /, >, <, sin, cos, etc.

The CONTROL UNIT also decides which instruction to fetch next from memory, which is either the next instruction or one elsewhere if a BRANCH or JUMP has occurred.



There usually aren't enough registers to hold the state of a program, so only the state being directly operated on or used very frequently is in the registers, and everything else is stored in the memory hierarchy (this is the "stack" and the "heap" of general data, as well as programs/textures/shaders/vertex arrays)

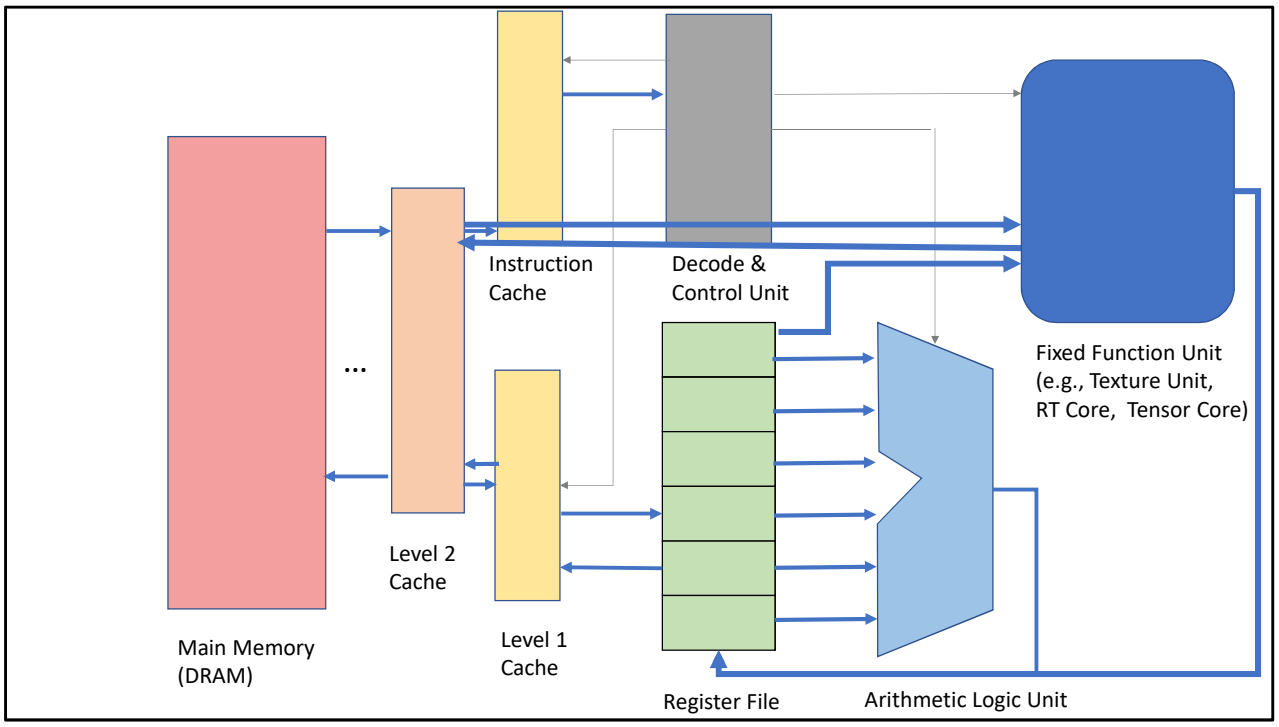
When reading from memory, there's a hierarchy that has typically one to four layers of caches ending in main memory (and of course, disks and network disks behind that). As a rule of thumb, assume a 2x – 10x decrease in bandwidth and 10x-100x increase in storage for moving up the memory hierarchy from registers to L1 to L2, etc.

Your first observation should be that programs which can operate with a small working set have a huge advantage in this computing model.

That includes not just the "data" but the instructions themselves…there's a cost for fetching and decoding new instructions, so long sets of instructions and programs that jump around in memory could be limited by the instruction cache size. This is a serious consideration on a GPU. It is less important on CPUs.

Your second observation should be that if those “registers” and ALU worked with vectors instead of scalars, then you could amortize the cost of instruction decode. More specifically, you could amortize the *space* in the processor spent on actual computation vs. support for the computation, which means more computational units would be available. This is exactly what modern processors do. CPUs present them as SIMD instructions, and GPUs model each vector element as if it was being processed on a separate thread in the programming model.

Main memory is definitely not owned by the core, and the L1 cache definitely is. At which point in a multicore system the memory hierarchy is shared between cores varies.

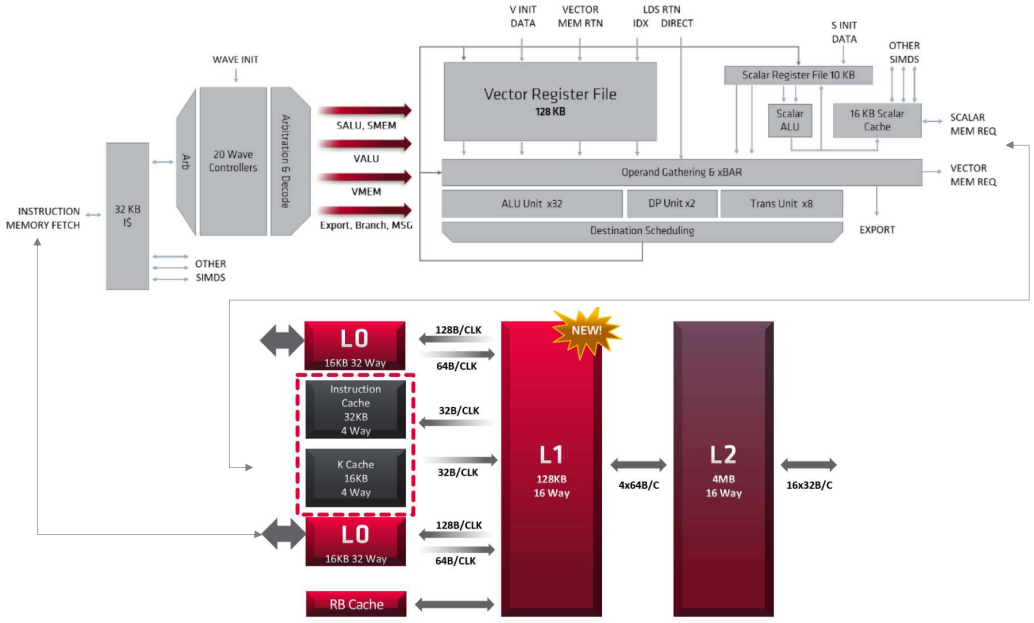


GPUs in particular also have fixed function units that act as coprocessors to the general purpose cores.

These perform specific functions that might take hundreds of instructions if executed on the ALU. They have direct access to the memory hierarchy and execute asynchronously from the main core.

The Texture Unit is a classic example. A single texture instruction can cause the texture unit to go off independently for hundreds of cycles without tying up the main core. It will fetch multiple addresses from memory, decode compressed representations to full floating point, filter them appropriately, and then deliver the results back to the main core's register file.

Tensor Cores and Ray Tracing cores in the NVIDIA Turing architecture are examples of newer fixed function cores.

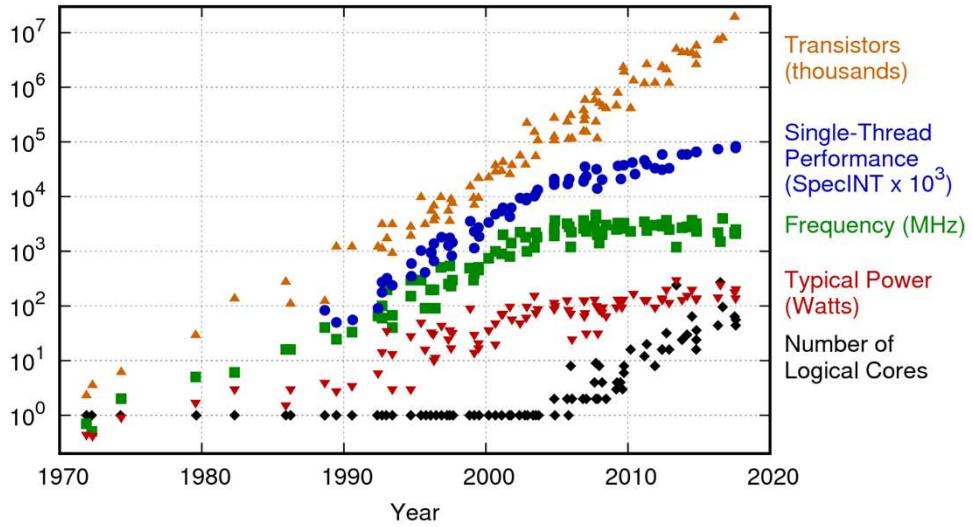


https://www.amd.com/system/files/documents/rdna-whitepaper.pdf

Here's an AMD diagram of a core and the memory hierarchy in their RDNA architecture. You can see that there are a few more details for handling vector operations and a "L0" cache, but it is basically the same as the model we just discussed.

A modern processor has many copies of those cores. The reason for this is that simply running a scalar processor at a higher clock rate

Clock Scaling Ended in 2008



Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2017 by K. Rupp https://www.karlrupp.net/2018/02/42-years-of-microprocessor-trend-data/

The reason for multiprocessing is that for the last 20 years, there have been diminishing returns on clock scaling.

In this graph, the horizontal axis is time and the vertical axis is several metrics of CPU design on a LOGARITHMIC scale.

The top curve is basically linear—that's "Moore's Law", where economic forces drive transistor count to grow exponentially in time. This is widely expected to level off in another ten years, but so far is still tracking.

The 1975 to 2000 portion of the curves is where single-core CPUs were tracking Moore's law and all metrics were growing exponentially (except for cores per processor)

The green and blue lines show that between 2000 and about 2008, single core performance driven by clock frequency began to flatline, and by 2008, it as even beginning to reverse, because it made more sense to have more, slower cores than few fast ones in that regime.

You'll also see that CPUs start going multicore around 2005 and that as we move towards the present, nearly all of the transistor count is being consumed by multiple cores instead of more complicated cores.

Modern GPUs started around 2000 as natively multicore processors in anticipation of this trend and have been growing rapidly. Here's what a recent GPU looks like…

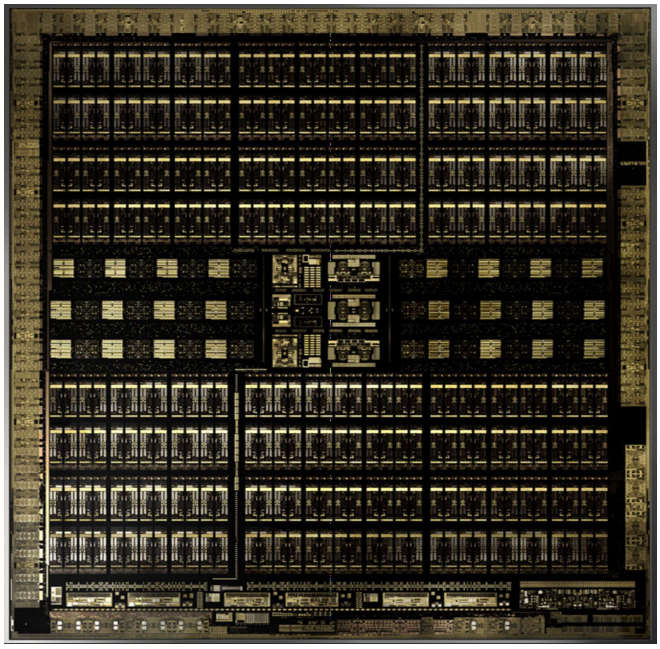
https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf

A modern processor has many copies of those cores.

Here's a diagram of the NVIDIA Turing TU102 GPU.

This has 144 cores, each of which is 32-way SIMD
72 ray tracing cores (each shared between two general cores)
576 tensor cores
288 texture units
And 12 GDDR6 memory units, with 512 kB L2 cache for each one, making a total of 6 MB of L2 cache across the processor.

The layout here is abstracted, but at a high level, not too far from reality…



https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf

Here's a photograph of the same GPU die, which is a few centimeters on a side. You can clearly see the individual cores, the cache, and the memory controllers.

There are a few different kinds of parallelism going on in these processors. The first is task parallelism. You see the duplicated units.

A Construction Crew is **Task Parallel**

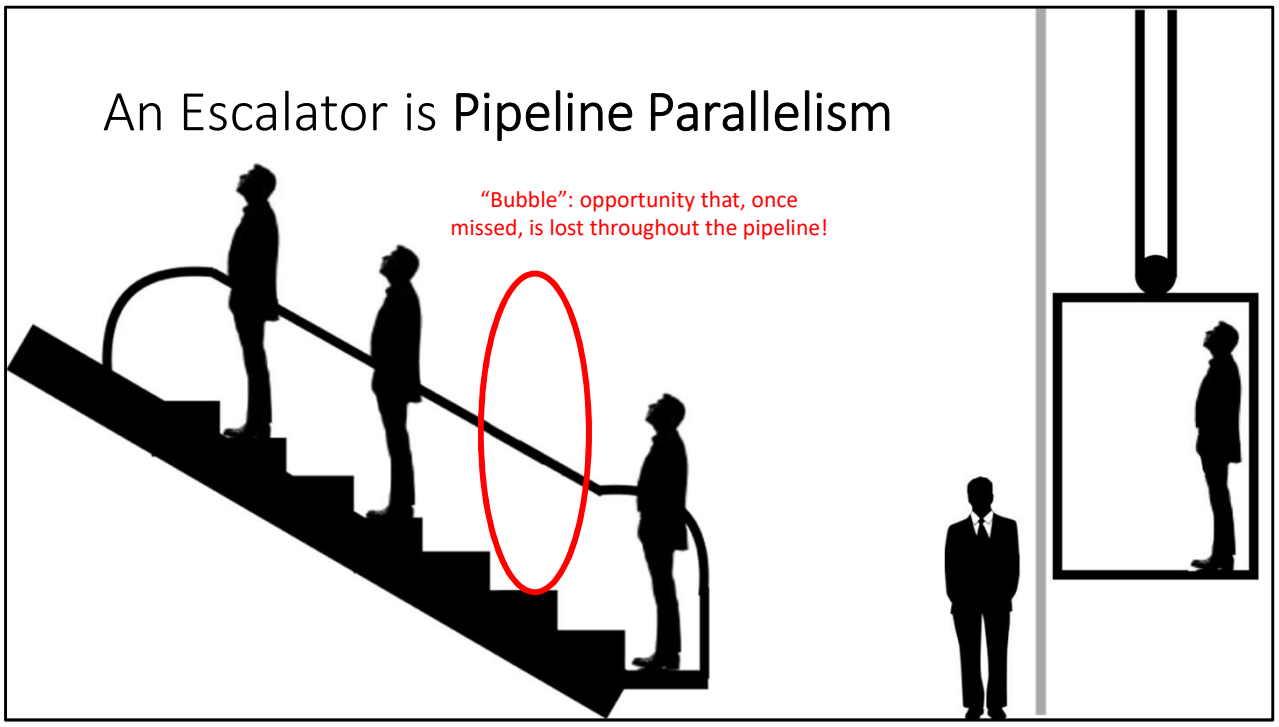
Each core can perform its own task, such as evaluating a material scattering function or generating a primary ray. That's task parallelism. It is like a construction crew. There are many different tasks going on simultaneously.

Of course, whenever those workers have to hand off a task between them or coordinate for a shared resource, they will slow down because they are no longer independent. So, task parallelism has limitations.

A Carpool is Instruction Parallel

Within the ALU, we can process small numbers of operations in lockstep with vector operations.
For example, adding 32 numbers to 32 other numbers at the same time. This is like a carpool.

It is very efficient when everyone is going to the same place. It becomes less efficient when their paths DIVERGE.



The third kind of parallelism you can't see in the processor structure because it isn't spatial. It is temporal.

Consider an elevator. It is a bottleneck in a building because once it has departed, everyone else has to wait for it to finish its first task before they can use it for their own task.

The solution is an escalator. It is pipelined…it takes the same amount of time for *one task*; in the best case, you still have to travel the distance between floors on an escalator or an elevator with no wait. But it has great throughput. The second person can start immediately after the first person instead of waiting for the first task to complete.

Modern processors (especially GPUs) and memory systems have deep pipelines that allow many tasks to be in flight at once, and fairly complicated logic for ensuring that all of the data are routed appropriately to these tasks in flight.

Part 1 Summary & Conclusions

Goals:

- Small working set size (for caches)
- Memory coherence (for caches)
- Instruction coherence (for instruction parallelism)
- Avoid bubbles (for pipeline parallelism, and fixed function tasks…)

Other Issues: (see the reading)

- Memory coalescing (for instruction parallelism in memory)
- Occupancy (for task parallelism)

Many of the principles of designing efficient parallel algorithms follow from the model I've just described

To use the caches efficiently, we want a small working set size, minimal memory traffic, and lots of coherence in addresses. This avoids memory becoming the bottleneck.

We need instruction coherence for SIMD instruction parallelism. This means that all branches should be taken the same way by threads in the same "warp" or SIMD vector, and that generally we should have large groups of threads that are "doing the same thing"

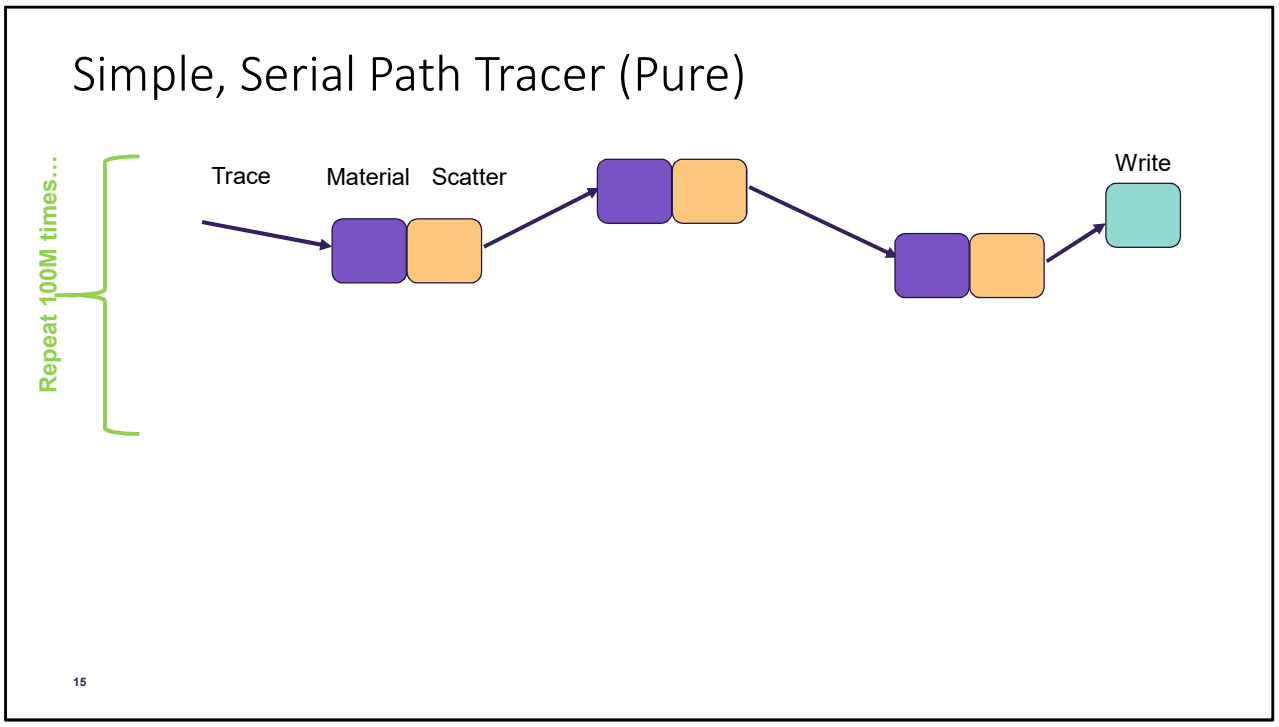
We need to avoid bubbles in the pipeline. For path tracing, this will be closely tied to instruction coherence. If one thread is tracing a ray while another one is shading, then we can't use either a general core or RT Core effectively.

The reading goes deeper on all of this and also addresses two other critical issues on GPUs: memory coalescing and occupancy. Those aren't as critical on CPUs.

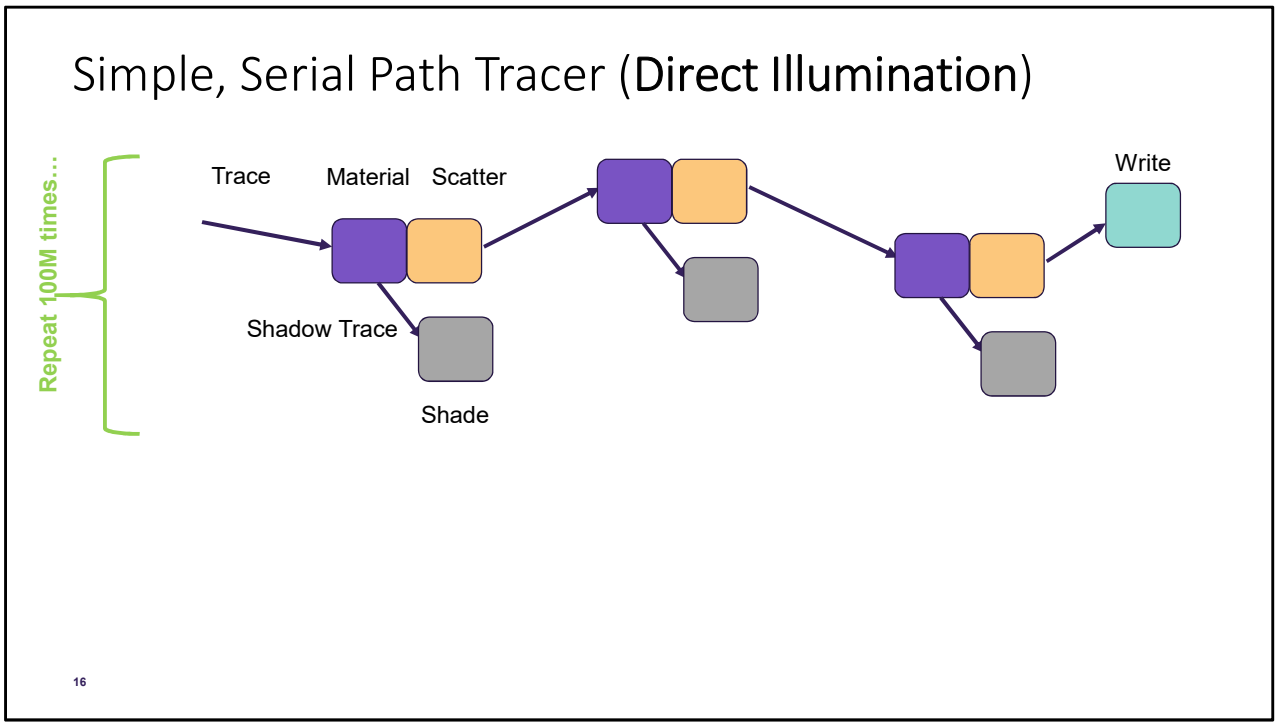
Now, with

Part 2: Parallel Path Tracer Case Study

The following slides are derived from the “Production-scale Ray Tracing” section of the
McGuire, Shirley, and Wyman, Introduction to Real-Time Ray Tracing, SIGGRAPH ’19 course
http://rtintro.realtimerendering.com/



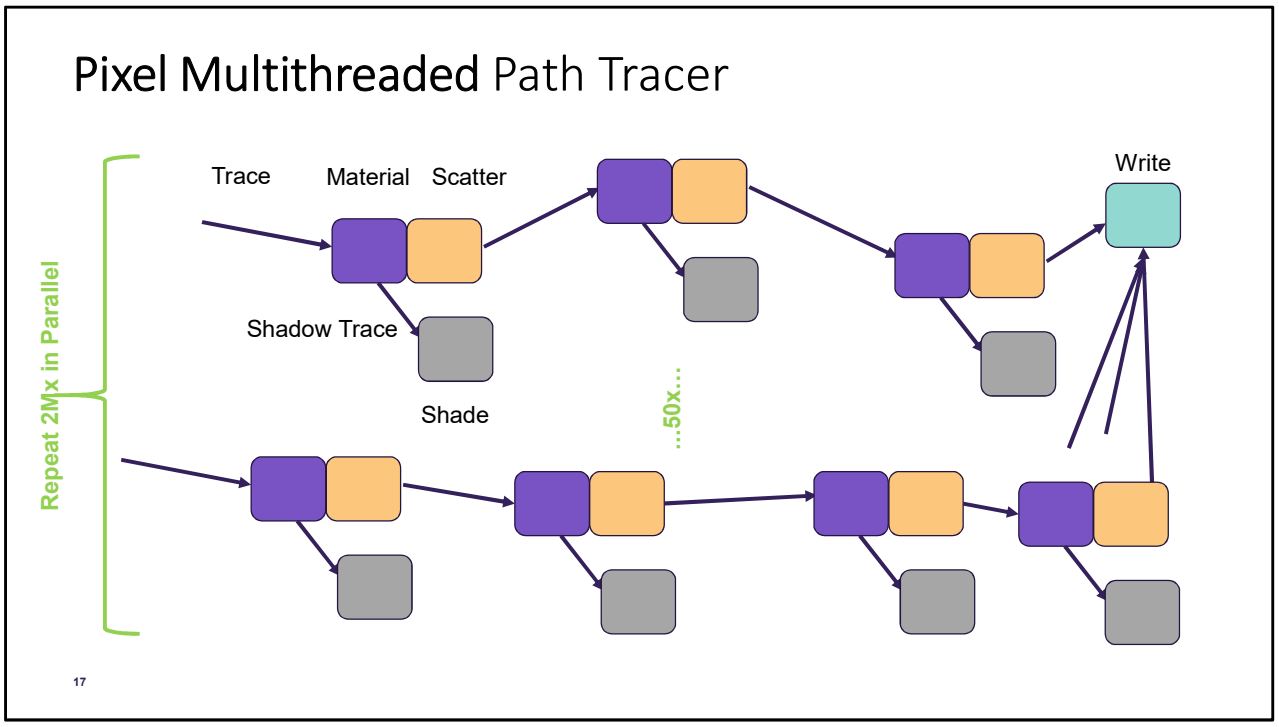
We start with work that looks kind of like this. That's the simplest path tracer I can write.



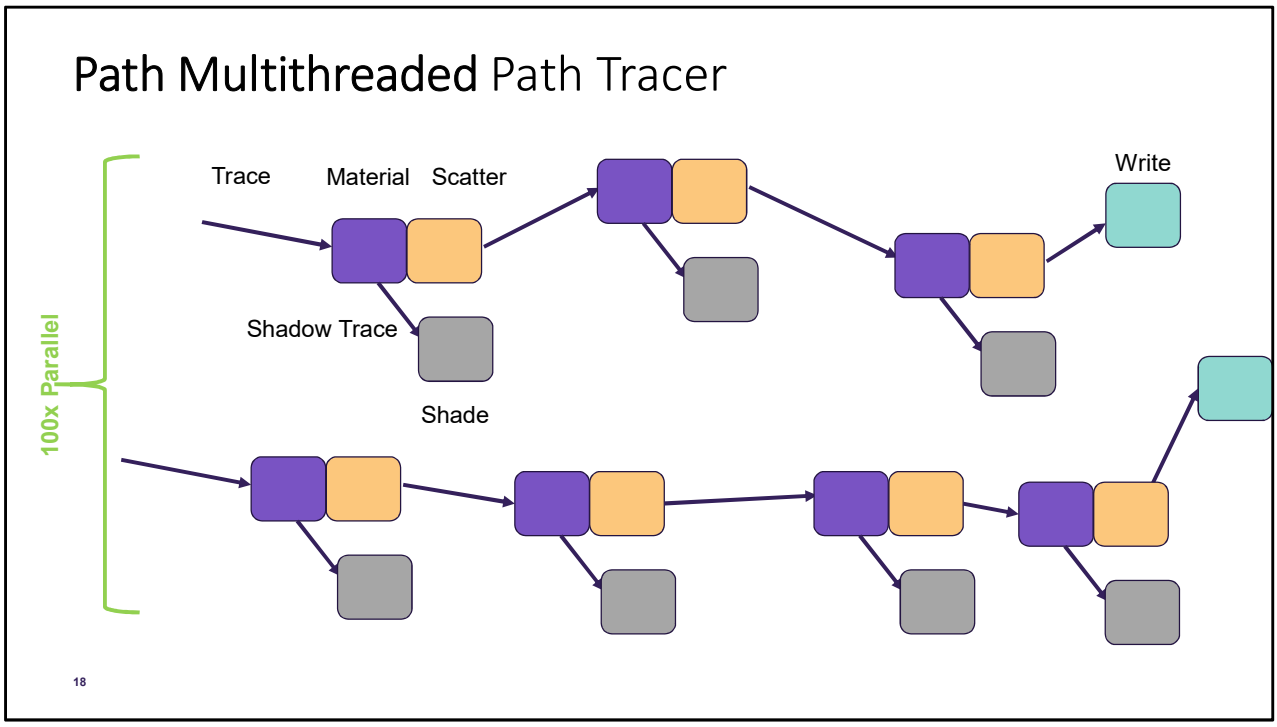
Then, we add direct illumination to cut down on noise. We now have Kajiya's original path tracer, plus some clever importance sampling.

But it only takes advantage of one core. On that TU102 GPU I have 144 cores with 32 SIMD lanes each, so there's a factor of 4,608 in task and instruction parallelism just sitting there.

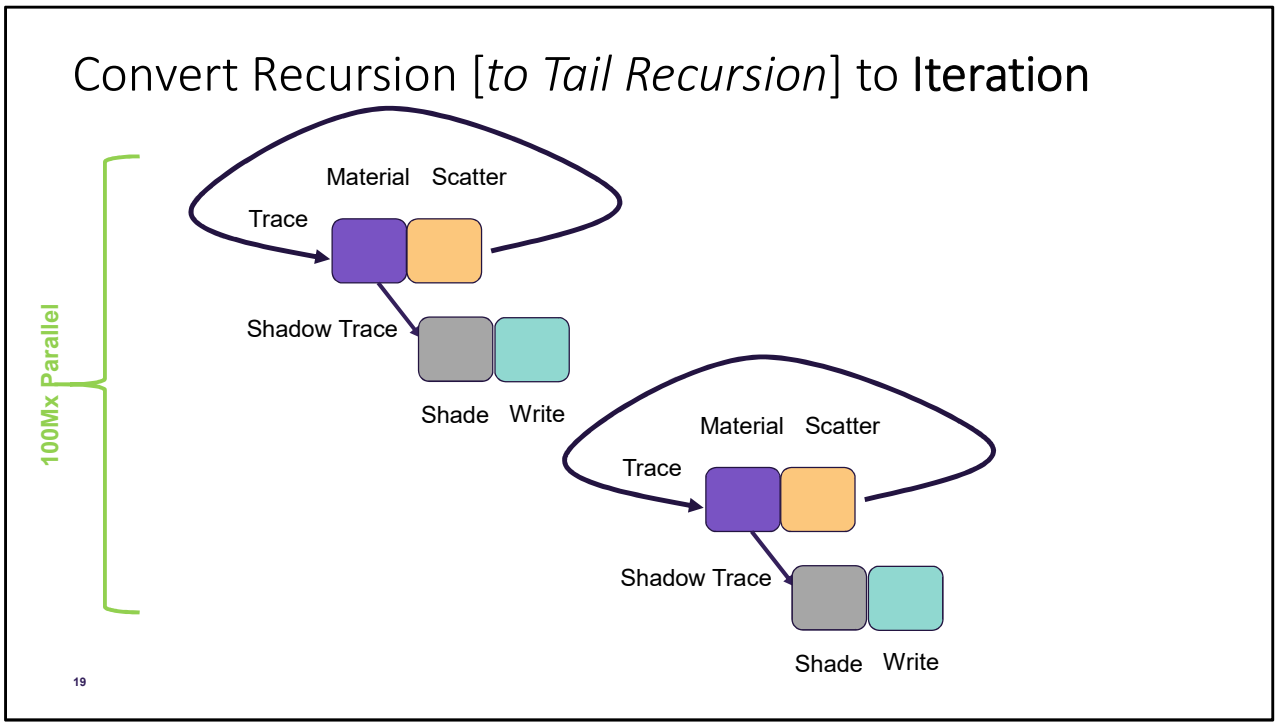
The first obvious transformation is to process each pixel independently on a separate thread. We have about 2M pixels in a 1920x1080 image, so that's plenty of work to fill a processor…



So, we launch 2M threads, each of which runs 50 paths. There's obviously a lot more parallelism that we can exploit here.
So, let's switch from processing each pixel on a thread to processing each path on a thread…



If you look horizontally, there's a lot of repetition. That's the recursive path. If we change that to iteration, it is just a tight loop…

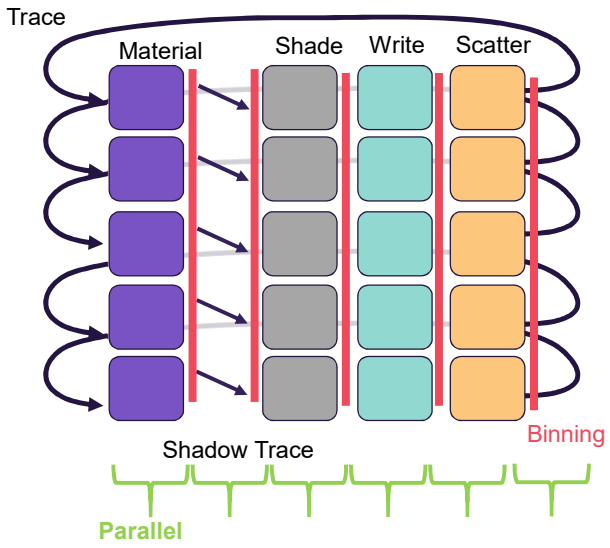


(along the way to here I had to make it tail recursive, which involved some restructuring)

Now we've captured a lot of redundancy. But there's a problem. We have lost all coherence of both instructions and memory. Every node of every path can be in a completely different point in its processing, which is terrible for vector ALUs, caches (and coalescing and occupancy, for other reasons).

So, what we want is to process each stage in lockstep across the whole processor. This is the final optimization. It is called "wavefront tracing":

Convert to Arrays and **Wavefront** Tracing



20

Here, we invert the parallelism. We use the whole machine (or at least a group of cores) to ALL trace together. When they've finished, they all evaluate materials together. And then trace shadows.

(this is another reason it is good to trace exactly one shadow ray, regardless of the light count: no matter where those rays hit in the scene, they're all going to do exactly the same amount of work in parallel). Then they all shade, write back to the frame buffer, scatter, and repeat the trace.

Doing this means that instead of passing values along between stages in registers, we have to dump the entire state out as a buffer between stages, since we have millions of values instead of one for each thread […or be very clever in the program structure using something called a megakernel…we'll see papers on all of this!]

Several good things happen in this structure.

1. we have great instruction cache usage. Each core is going to do a small operations many times until all of the work is processed at that stage: just trace a ray | just shade | just write…

2. We have a LOT of work at each stage. We can schedule it to take advantage of the architecture. Assuming that we group threads by binning/"sorting" between steps so that similar work goes to each processor, we can extract **coherent execution for instruction parallelism** and **cache efficiency**
3. Keeping each stage of the program short means fewer registers and thus more **occupancy for task parallelism**
4. Not shown is a compaction step after each write, where paths that have been terminated are squeezed out of the workload for the next iteration across the processor to maintain full **occupancy for task parallelism**

The bin/sort/compaction is very fast on a GPU in particular. It is a radix-based process that is extremely parallel and asymptotically linear time, and there are highly optimized GPU implementations of this.

I'm not going to present the following slides walking you through the code, but you can read them offline and look at the complete solution that I coded in C++ as an example.

Naive Multithreaded Path Tracer 1/2

```
image->forEachPixel<Radiance3>([&](Point2int32 pixel) {
        Random& rng = Random::threadCommon();
        Radiance3 sum;
        for (int p = 0; p < pathsPerPixel; ++p)
                sum += L_i(camera->worldRay(
                        pixel.x + rng.uniform(), pixel.y + rng.uniform(),
                        image->bounds()));
        return sum / float(pathsPerPixel);
});
```

21

The path tracing algorithm has two pieces.

The first is an iterator over pixels, which generates a bunch of primary rays for each pixel and averages the light along them.

The more interesting function is the one that evaluates the light coming *back* along a ray towards its origin.
I've labeled this function L_i, for "incoming light" following academic notation.

Naive Multithreaded Path Tracer

```
// Trace this (world space) ray and return the radiance it encounters
Radiance3 L_i(const Ray& ray) {
    const shared_ptr<Surfel>& surfel = tree->intersectRay(ray);
    const Vector3& w_o = -ray.direction();
    const Radiance3& L_e = surfel->emittedRadiance(w_o);

    Vector3 w_i;
    Color3 weight;
    if (! surfel->scatter(PathDirection::EYE_TO_SOURCE, w_o, true,
        Random::threadCommon(), weight, w_i))
        return L_e;

    const Point3& X = surfel->position();
    const Ray& nextRay = Ray(X, w_i, epsilon, finf()).bumpedRay(epsilon, surfel->geometricNormal);

    return L_e + L_i(nextRay) * weight;
}
```

22

The incoming light function does three interesting things.

It intersects the ray with the scene
It computes the amount of light (maybe none) that is emitted at that intersection
And then it either terminates abruptly *or* recursively scatters and evaluates a new ray starting at the hit point.

Everything about the “material” is encoded in the emittedRadiance and scatter functions. You’ve seen simple ones. They’re about to get more complex.

One pixel → one path per "thread"

Before

```
// Compute the average radiance into this pixel
image->forEachPixel<Radiance3>([&](Point2int32 pixel) {…
    for (int p = 0; p < pathsPerPixel; ++p) {
        const Point2 pixelPos(pixel.x + rng.uniform(), pixel.y + rng.uniform());
        sum += L_i(camera->worldRay(pixelPos.x, pixelPos.y, image->bounds()));
    }

    return sum / float(pathsPerPixel);
});
```

After

```
for (int p = 0; p < pathsPerPixel; ++p) {
    image->forEachPixel<Radiance3>([&](Point2int32 pixel) {…
        const Point2 pixelPos(pixel.x + rng.uniform(), pixel.y +rng.uniform())
        const Radiance3& L = L_i(camera->worldRay(pixelPos, image->bounds()));
        rawImage->bilinearAdd(pixelPos, L);
        weightImage->bilinearAdd(pixelPos, 1);
    }}); }
// Normalize by weights later…
```

Instead of processing each PIXEL in a "thread", we're going to process each PATH on a thread.

(These aren't real operating system threads, which have high overhead…they are CPU thread-pooled work or GPU SIMD warp lanes, but the TBB/CUDA/DirectX/Vulkan APIs make them *look* like threads, so we'll keep pretending that they are real threads).

Each path will remember its source location on the image plane and bilinearly interpolate into it.
This gives us a lot more parallelism to work with, which will help to "fill the machine". It also sets up the following steps because we can now deal with paths individually. Instead of *returning* a value recursively, we'll now write directly back to the image at each iteration step.

In practice you don't even need that outer FOR loop…just spawn a total number of threads equal to pixels times paths per pixel.

Bilinear add uses atomic addition, so you don't have to worry about the race condition.

Recursion → Tail recursion

Before

```
/* Trace this  ray and return the radiance it encounters */
Radiance3 L_i(const Ray& ray) const {
    const shared_ptr<Surfel>& surfel = tree->intersectRay(ray); …
    if (surfel->scatter(…, weight, w_i))  …
        return L_e + L_i(Ray(X, w_i)) * weight;
    else
        return L_e;
}
```

After

```
/* Trace this ray and write to the image */
void trace(const Ray& ray, const Point2& pixelPos, const Color3& modulate) const {…
    const shared_ptr<Surfel>& surfel = tree->intersectRay(ray); …
    image->bilinearAdd(pixelPos, L_e * modulate); …
    if (surfel->scatter(…, weight, w_i)) …
        return trace(Ray(X, w_i), pixelPos, weight * modulate);
}
```

24

Next, we're going to eliminate the recursion.

Step 1: To make this tail recursive, we have to carry through the value to modulate by when writing back to the original pixel.

Recursion → Iteration

Before

```
/* Trace this  ray and return the radiance it encounters */
Radiance3 L_i(const Ray& ray) const {
    const shared_ptr<Surfel>& surfel = tree->intersectRay(ray); …
    if (surfel->scatter(…, weight, w_i))  …
        return L_e + L_i(Ray(X, w_i)) * weight;
    else
        return L_e;
}
```

After

```
/* Trace this ray and write to the image */
void trace(const Ray& ray, const Point2& pixelPos) const {…
    while (modulate > 0) {
        const shared_ptr<Surfel>& surfel = tree->intersectRay(ray); …
        image->bilinearAdd(pixelPos, L_e * modulate); …
        surfel->scatter(…, weight, w_i));
        modulate *= weight;
    }
}
```

Step 2: Convert the tail recursion to iteration

Fire and forget…each step deeper into the transport graph carries the information to write back to the pixel. Never “return”.

We’ve now done two important things:

1. We eliminated the stack and reduced the working state. Essential for a GPU and good for a CPU.
2. Most importantly, we now have each RAY of each PATH treated equally. They don’t know their transport graph depth. That means we can do the next step and process all depths in exactly the same way, in parallel.

Recursion → Iteration

Before

```
/* Trace this  ray and return the radiance it encounters */
Radiance3 L_i(const Ray& ray) const {
    const shared_ptr<Surfel>& surfel = tree->intersectRay(ray); …
    if (surfel->scatter(…, weight, w_i))  …
        return L_e + L_i(Ray(X, w_i)) * weight;
    else
        return L_e;
}
```

After

```
/* Trace all rays and write to the image */
image->forEachPixel<Radiance3>([&](Point2int32 pixel) {…
    while (modulate > 0) {
        const shared_ptr<Surfel>& surfel = tree->intersectRay(ray); …
        image->bilinearAdd(pixelPos, L_e * modulate); …
        surfel->scatter(…, weight, w_i));
        modulate *= weight;
    }
}
```

To make this clearer, I'll get rid of the trace function that we only call once per path and just inline the body into the thread launch.

OK, here's the big one. We're going to invert the structure and make EACH line of code process all of the rays and then hits in parallel. This eliminates the big thread launch and makes a bunch of little launches, one per line:

Asynchronous → Wavefront processing

Every variable & parameter becomes an ARRAY:

```
Array<Ray> ray;
Array<Color3> modulation;
Array<shared_ptr<Surfel>> surfel;  // hits
Array<Radiance3> direct; // used if lightShadowed = false
Array<Ray> shadowRay;
Array<bool> isLightShadowed;
Array<bool> isImpulseRay; // avoids double-counting of emissives
Array<Point2> pixelPos;
```

27

First, I'm going to replace all of the state in the program with arrays of state. This is so that we can process all of the

Asynchronous → Wavefront processing

Before

```
image->forEachPixel<Radiance3>([&](Point2int32 pixel) {…
    while (modulate > 0) {
        const shared_ptr<Surfel>& surfel = tree->intersectRay(ray); …
        image->bilinearAdd(pixelPos, L_e * modulate); …
        surfel->scatter(…, weight, w_i));
        modulate *= weight;
    }
}
```

After

```
image->forEachPixel<Radiance3>([&](Point2int32 pixel) { ray.push(…); });
while (modulate.length() > 0) {
    tree->intersectRays(ray, hit);
    sampleHits(hit, surfel);
    modulateAndAddEmissive(image, surfel, modulate);
    scatterSurfels(ray, surfel, weight, w_i, modulate);
    compact(ray, surfel, …);
}
```

All variables are now ARRAYS

Now, you need to know that every variable on the top is a single number, or vector, or color, etc. but every variable on the bottom is an ARRAY of numbers, or vectors, or colors, etc.

Allows us to separate the closest-hit intersector, material evaluation, and scatter code into separate kernels

Fill the machine with coherent programs

Megakernel is actually doing something similar, it is just a question of how much we schedule and how much the driver/OS/API schedules…I think eventually programmers will not have to do this for performance.

Fit in I$

Better data cache via locality

Better CPU branch prediction

Better GPU SIMD for the trace

Better GPU SIMD for everything

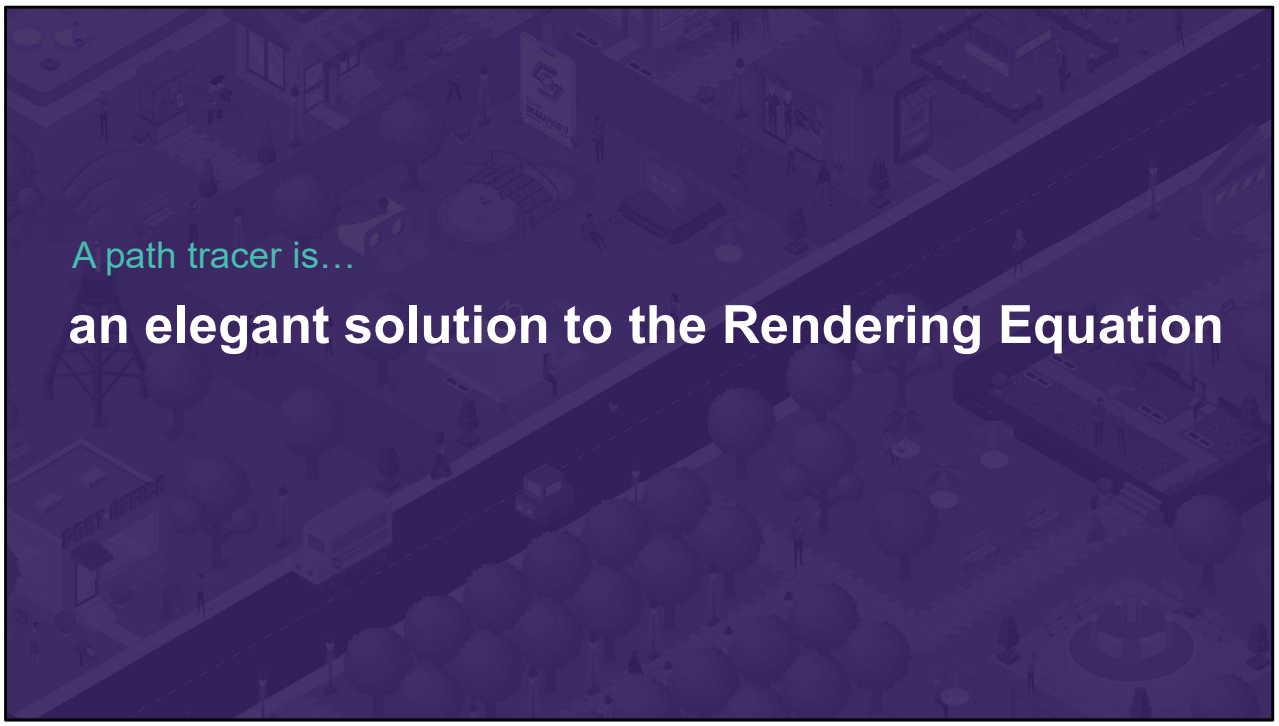
Gives us compaction and binning points

Drawback: LOT more memory (“deferred”)

Full Details

- Before:
  - https://casual-effects.com/g3d/G3D10/samples/simplePathTracer/main.cpp
- After parallel refactor:
  - https://casual-effects.com/g3d/G3D10/G3D-app.lib/include/G3D-app/PathTracer.h
  - https://casual-effects.com/g3d/G3D10/G3D-app.lib/source/PathTracer.cpp

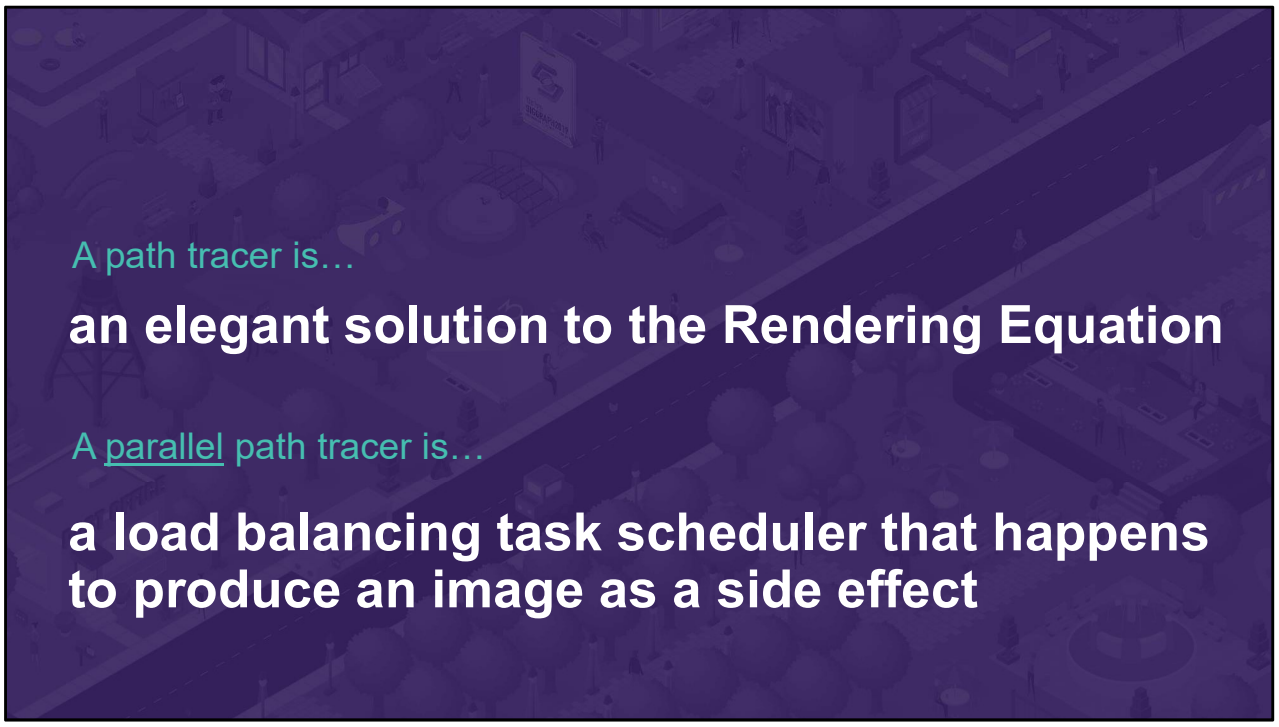
29



The path tracer is pretty unrecognizable after all of this optimization.
In last week's lecture, I showed you the beauty of Path Tracing.

This week I destroyed it.

To paraphrase a quote from Larry Gritz,


A path tracer is…
an elegant solution to the Rendering Equation
A parallel path tracer is…
a load balancing task scheduler that happens to produce an image as a side effect