SYCLcon
2021/04/28–29

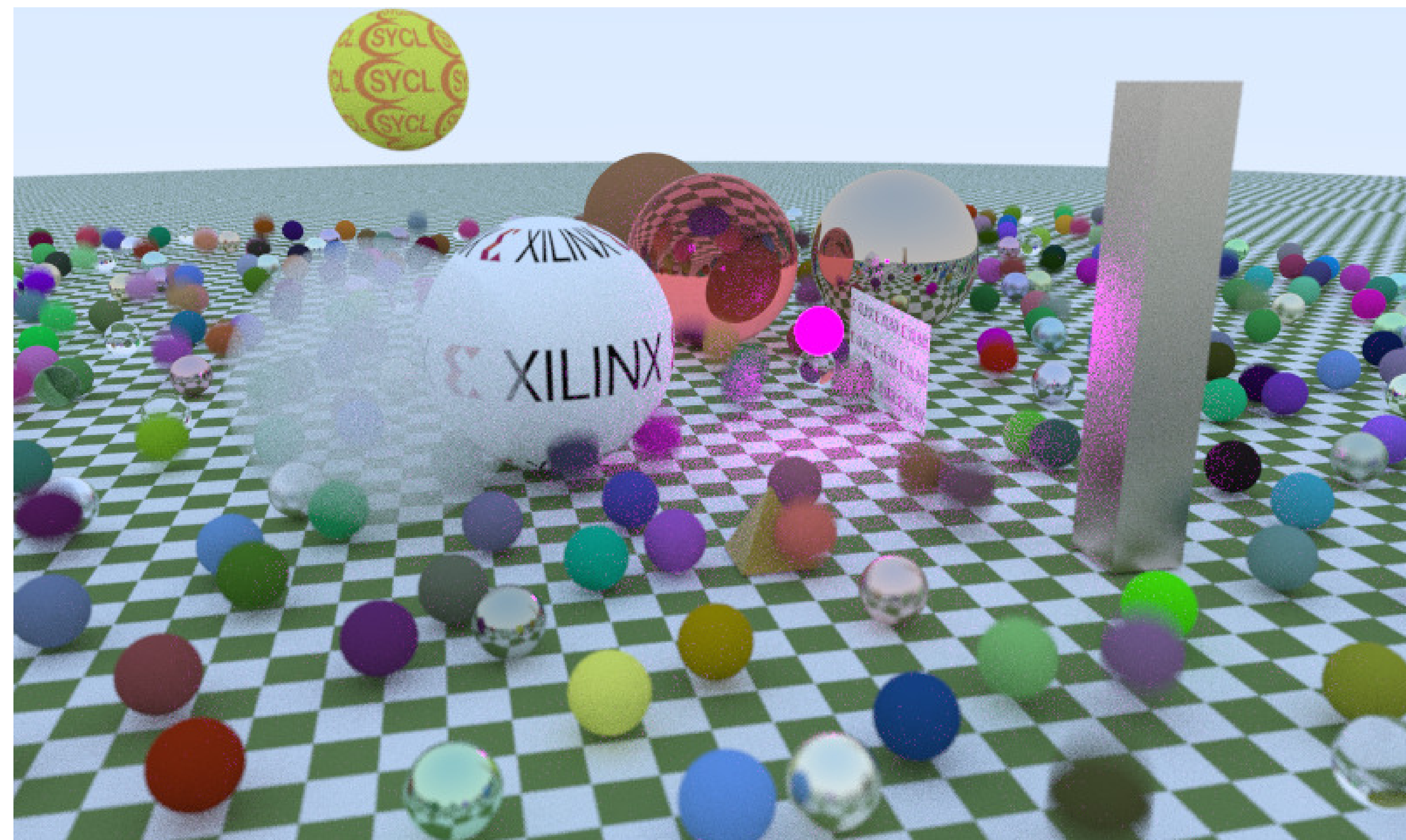
Path Tracing on FPGA with SYCL and C++20

Luc Forget[1] Gauthier Harnisch[1] Ronan Keryell[1] Krishna Kumar Ranipet Murugan[2]

[1]Xilinx [2]North Carolina State University

Abstract

Path tracing is a global illumination method used in computer graphics to render photorealistic images by using a Monte Carlo integration. While it is very compute intensive, it is nowadays commonly used in movie production because of its image quality.
We present a straight-forward implementation in modern C++ using SYCL for the offloading of the compute intensive part on accelerators.
Usually ray tracer and path tracer implementations rely on dynamic polymorphism to handle objects with different shapes and different materials but this is currently unsupported in SYCL since often accelerators cannot handle function pointers. Instead, we do not use polymorphism but rely on C++17 `std::variant` and `std::visit` to dispatch operations with duck-typing in a type-safe way.
`std::visit` can be executed in $\mathcal{O}(1)$ on FPGA because the dispatch is spacialized on the architecture.
This is an ongoing open-source project available on `https://github.com/triSYCL/path_tracer`

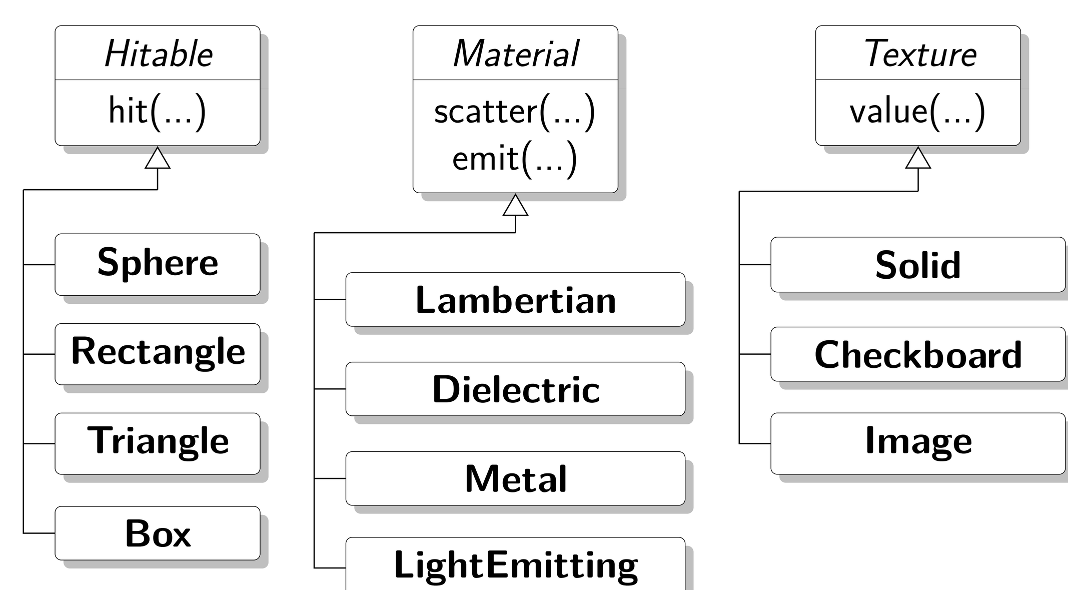


Path tracing

1. Set a **viewport** with **camera** properties : **position**, **view angle**, **resolution**, **aperture**
2. For each **viewport pixel**, many rays starting from the point of view and having approximately the direction of the center of the pixel are thrown in the **scene** (Monte Carlo)
3. The ray can hit a **hitable** scene object, in which case it can be reflected, refracted or whatever according to the material behaviour
4. When no more scattering occurs, or after a maximum scattering depth is reached, the **color** of the last intersected object is associated to the ray
5. The output color for a given viewport pixel is the average of the color of all the ray that are casted in its direction (Monte Carlo)

Algorithm 1: FPGA kernel
input : A collection H of **hitable** objects, C a **camera**, a maximum scattering number $D_{\max}$, a number of samples S
output: A **color** map I
foreach *Pixel coordinate* $(x, y) \in C.\text{range}()$ **do**
  out $:= (0., 0., 0.)$ **foreach** *Sample* s **do**
    ray $:= C.\text{get_ray}(x, y)$;
    hit_record := no_hit_record;
    shortest_hit_distance $:= +\infty$;
    bounce := 0;
    scatter := true;
    while *scatter and bounce* $< D_{\max}$ **do**
      foreach *hittable* $h \in H$ **do**
        intersec_record $:= h.\text{hit}(\text{ray})$ **if** *intersec_record.distance* $<$ *shortest_hit_distance* **then**
          hit_record := intersec_record;
          shortest_hit_distance := intersec_record.distance;
      bounce $+= 1$;
      scatter = hit_record.scatter();
    out $+=$ hit_record.get_color();
  $\text{I}[y, x] = \text{out}/S$;

Initial project class diagram



- **Hitable** describes object geometry & determines ray-object intersection point
- **Material** describes and compute incident ray reflection or deviation
- **Texture** gives object albedo at any point of its surface

Problem: no dynamic polymorphism possible in SYCL kernel code

- No function pointer support: dynamic polymorphism based on inheritance will not work
- Replaced by static polymorphism based on C++17 `std::variant`
- Static function dispatch uses *visitor pattern* using C++17 `std::visit`

```
// Before: with dynamic polymorphism
class hitable {
  void whoami() = 0;
};

class sphere : public hitable {
  void whoami() override {
    cout << "sphere";
  }
};

class triangle : public hitable {
  void whoami() override {
    cout << "triangle";
  }
};

hitable* t = new triangle();
hitable* s = new sphere();
t->whoami();
s->whoami();
```

```
// After: static dispatching based on
// std::variant and std::visit
class sphere {
  void whoami { cout << "sphere"; }
};

class triangle {
  void whoami() { cout << "triangle"; }
};

using hitable = std::variant<triangle, sphere>;
hitable t{triangle{}};
hitable s{sphere{}};

std::visit([&](auto&& h) {
    h.whoami();
  }, t);
std::visit([&](auto&& h) {
    h.whoami();
  }, s);
```

- Unfortunately `std::visit` from standard C++ library based on function pointers will not work...
- Custom interface-compliant alternative `dev_visit` developped
- Template unrolling for generating `if` statement to select correct method based on variant id

Problem: stateful shared pseudo random number generator

Original program uses a pseudo-random number generator (RNG) shared in a global variable ⇒ not available to device code
First solution: create RNG on device, propagate reference via context parameter
Problem: RaW dependencies on RNG internal state that prevent automatic parallelization
Second alternative: Hash function input as local RNG seed, or create multiple RNG deeper in call tree to break dependency chain

Problem: image texture bitmaps

- Image bitmaps loaded on host from sprites files
- On original code: dynamic allocation for bitmap buffer & `ImageTexture` stores this pointer

Problem: However, pointer to host memory invalid on device without USM support
Alternative: Use offset in a global buffer

- On image texture creation, the image bitmap is appended to a `std::vector`
- When all the textures are created, the vector is frozen and copied to a `sycl::buffer`
- Transfer accessor to this buffer through the kernel call stack in a context object
- `ImageTexture` uses this accessor to retrieve a color from the bitmap based on the stored offset

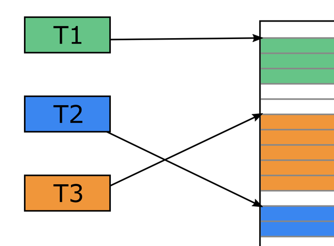


Figure 1: Base code image texture architecture

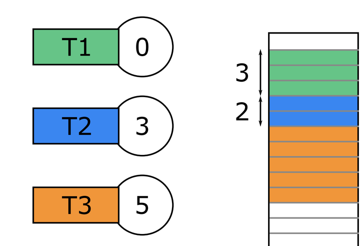


Figure 2: Alternative architecture

Contrary to RNG case, does not create dependencies as there is only read access in device code

Future work

Project goal: port an existing code to an FPGA using SYCL & C++20, with minimal changes.
However, current design is sub-optimal:

- Monolithic behemothian kernel with unexplicit parallelism
- Current algorithm formulation far from having an obvious efficient hardware mapping

Next step: use *SYCL as a hardware co-design tool*:

1. Split the path-tracing in sub-problems
2. Sketch an efficient architecture for solving each of these problems on FPGA
3. Formulate each of these high-level architecture as a SYCL kernel
4. Let the compiler handle all the low level gory details

Besides,

- Current code uses IEEE-754 binary32 format for representing real numbers
  - *Fixed-point* arithmetic hardware implementation is smaller and faster than floating-point's on FPGA
  - Experiment path-tracing with fixed-point numbers with custom width formats
- As SYCL is plain C++, software emulation is just compilation for host device
- Compilation for CPU faster than RTL simulation: numerical format tuning is faster and more iteration can be done before starting costly hardware synthesis and routing

Conclusion

While still work in progress, this project has already allowed to discover some SYCL and HLS compiler and runtime issues with standard mathematical library. Besides, it showed that porting an existing code to SYCL can be done without complex code refactoring. Except for the lack of dynamic polymorphism, the biggest issue comes from shared data on kernel code. This issue no longer exists with unified shared memory. Finally, SYCL is a promising tool for FPGA architecture experimentation and co-design.

XILINX. SYCL NC STATE UNIVERSITY