Vision, Modeling, and Visualization (2017), pp. 1–8
M. Hullin, R. Klein, T. Schultz, A. Yao (Eds.)

C++ Compile Time Polymorphism for Ray Tracing

S. Zellmann[1] and U. Lang[1]

[1]Chair of Computer Science, University of Cologne, Germany

Figure 1: *San Miguel triangle scene geometry and reflective spheres rendered using BVHs. We evaluate the throughput of the primitive intersect routine when one BVH is used per primitive type, and when packing triangles and spheres into a single BVH and using compile time polymorphism to determine the primitive type during traversal.*

Abstract

Reducing the amount of conditional branching instructions in innermost loops is crucial for high performance code on contemporary hardware architectures. In the context of ray tracing algorithms, typical examples for branching in inner loops are the decisions what type of primitive a ray should be tested against for intersection, or which BRDF implementation should be evaluated at a point of intersection. Runtime polymorphism, which is often used in those cases, can lead to highly expressive but poorly performing code. Optimization strategies often involve reduced feature sets (e.g. by simply supporting only a single geometric primitive type), or an upstream sorting step followed by multiple ray tracing kernel executions, which effectively places the branching instruction outside the inner loop. In this paper we propose C++ compile time polymorphism as an alternative optimization strategy that does on its own not reduce branching, but that can be used to write highly expressive code without sacrificing optimization potential such as early binding or inlining of tiny functions. We present an implementation with modern C++ that we integrate into a ray tracing template library. We evaluate our approach on CPU and GPU architectures.

Categories and Subject Descriptors (according to ACM CCS): I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Raytracing

1. Introduction

Real-time ray tracing is an active research topic since the time that personal computers were equipped with multi-core CPUs with *single instruction multiple data* (SIMD) support and coprocessor cards for graphics operations. In the early days of real-time ray tracing, applications were hand tailored and employed algorithmic

optimizations and micro optimizations that were specific to the respective target platform. Typical optimizations for x86 platforms comprised the use of SIMD compiler intrinsics and intricate data layouts for types that are used in inner loops, such as triangles or surface properties [Wal04]. Early GPU ray tracers used the fragment stage of the rasterization pipeline to implement ray object traversal [PBMH02]. Due to hardware and software advances in areas like GPGPU, ray tracing libraries have since evolved and provide expressive APIs that help the application programmer to devise real-time ray tracing algorithms by supplying kernel operations such as acceleration data structure traversal [WWB*14].

Branching operations are however still costly on most contemporary hardware platforms. Unfortunately, many ray tracing algorithms involve branching in innermost loops. This happens e.g. at the *primitive intersection* stage when deciding which primitive type is being processed, or at the *shading* stage, when deciding for the appropriate surface property type pertaining to an intersection position. More examples include multi-channel ray casters for direct volume rendering, with support for different texture types per channel (e.g. 8-bit per channel vs. 16-bit per channel), or a path tracer with direct light sampling and support for different light types. Whenever the decision which instructions and data segments are loaded from main memory to registers for further processing depends on the type of a data object, we will in this paper refer to the operation as branching *based on type ids*.

CPUs contain branch prediction units that in general mitigate the costs of conditional statements in code. No matter how intricate the algorithm that implements the branch prediction logic, such an approach will however only work if branching behavior is coherent to some extent. Branching on CPUs will typically result in SIMD unit starvation. If packets of rays are traversed through a hierarchical data structure containing primitives, and if neighboring rays in the packet process different primitive types, the respective SIMD unit dedicated to the first primitive type will be deactivated using masking operations when the second primitive type is processed. GPUs typically employ threading models where program execution is not completely independent, but where multiple threads are combined to *warps* that execute code in lockstep [NBGS08]. This thread execution model implies that threads have to wait inactively for other threads in the warp to execute conditional code blocks. Hardware platforms that rely on caches will in general be sensitive to conditional branching. Conditional branching will have a negative impact on instruction cache utilization. Moreover, branching over type ids often implies that data objects need to be fetched from different regions of memory, which will typically result in low data cache utilization.

Object oriented programming provides *polymorphism* as an expressive means to encapsulate code for branching over type ids. Typical object oriented ray tracers supply abstract base types for primitives and surface properties, which can be extended by the application programmer using inheritance [Wil93, PJH16]. This has however several downsides from a runtime performance perspective. On the one hand, types that support virtual inheritance typically carry a vtable pointer with them, which is implementation specific and counterproductive when devising cache friendly data layouts. Furthermore, with runtime polymorphism, the application programmer can generally extend a ray tracing library from her own C++ module. This inhibits early binding, which is a crucial optimization for high performance C++ code.

Compile time polymorphism (CTP) (which is sometimes also referred to as *static polymorphism* [MS00]) is a means to implement branching without having to explicitly specify every *possible* branch in library code. The latter would also not be practical because the application programmer then cannot extend the library code. CTP is a generic programming technique that allows for compiler optimizations such as *inlining* and *early binding*. CTP requires the application programmer to provide an instantiation at compile time. One downside of this approach when compared to runtime polymorphism is that the instantiation and thus every possible incarnation of the static polymorphic type needs to be known when the application (but not the library) is being compiled. We propose CTP as an expressive means for the application programmer to extend ray tracing library code and compare the runtime performance of this approach with traditional optimization approaches. Due to modern C++ support by APIs such as NVIDIA's CUDA or AMD's ROCm, CTP can also be used in GPGPU code. When using GPGPU APIs, runtime polymorphism is only available for locally created objects and not for objects that were created on the host, because CPU and GPU typically do not share address spaces.

The paper is structured as follows. In Section 2 we review related work from the field of ray tracing API design. In Section 3 we briefly review the CTP concept and modern C++ language features that help to devise expressive APIs. In Section 4 we describe how to integrate the CTP concept into a generic C++ ray tracing library. In Section 5 we evaluate our approach and in Section 6 we conclude this publication.

2. Related Work

Over the last couple of years, a variety of approaches to design ray tracing APIs have been proposed. The Embree ray tracing kernel library [WWB*14] e.g. basically only implements the primitive intersection stage of a typical ray tracing pipeline through an ANSI-C function interface and has builtin support for only a number of limited primitive types, which can be extended by the application programmer by means of a callback mechanism. NVIDIA's OptiX library [PBD*10] provides access to a static set of pipeline stages, e.g. the primitive intersection stage, and the shading stage, where custom programs can be supplied to define the behavior at the respective stage. PBRT [PJH16] and Mitsuba [Jak10] are research oriented, physically based rendering systems that follow an object oriented programming approach but are both not targeted towards real-time ray tracing. Generic ray tracing APIs like RTfact [SG08] or Visionaray [ZWL17] typically provide default implementations for types and functions that are often used, and allow for extending those by implementing so called *customization points*. Customization points are usually specified using *Concepts*, which in turn specify a set of traits a type must adhere to so that it can be used with the library interface. RTfact and Visionaray provide numerous customization points to extend the functionality of the libraries in a generic fashion. Those involve, amongst others, primitive intersection, shading, light sampling, and texturing. A diverging approach to using general purpose programming languages are domain spe-

cific languages (DSLs) [PM12, LBH*15], which may handle polymorphism and binding differently than C or C++. The choice of C++ API has a strong influence on the way that conditional branching in shading or intersection code translates to machine code. APIs that only facilitate querying rays for intersections with triangles in a bounding volume hierarchy completely avoid branching during primitive intersection and place the burden of handling branching in shading code on the shoulders of the application programmer. Object oriented, polymorphic APIs trade branching performance for better code readability. Generic libraries allow the application programmer to decide for herself how branching is best handled in code.

Regardless of the API, it is generally best to avoid unnecessary branching at all. With ray tracing, there is typically a trade-off between branching in innermost loops, and the costs of introducing an additional constant overhead e.g. for sorting rays or material IDs for coherence. Garanzha and Loop [GL10] propose a framework where secondary rays are sorted by projecting ray origins to cells of a virtual regular grid, and by quantizing ray directions into sectors on the unit sphere. Áfra et al. [ÁBWM16] concentrate on coherence for shading calculations and therefore process ray bounces in a path tracing pipeline in batches. Between bounces, they sort ray IDs by the material IDs of the hit points from the preceding primitive intersection phase. Breadth first processing of incoherent ray paths is especially useful on GPUs. GPUs have no branch prediction units and execute threads in a SIMD fashion with very wide vectors (those are referred to as *warps* in CUDA related documents and as *wavefronts* in documents related to the OpenCL programming language). Laine et al. [LKA13] compared breadth first and depth first path tracing and proved the superiority of the *wavefront approach* over *megakernels* on contemporary GPU architectures. Wavefront approaches are especially beneficial on GPUs because they reduce the register pressure on individual compute kernels and thus allow for higher occupancy of the GPU cores. Both Laine et al. as well as Áfra et al. concentrate on scenarios where the shading phase makes up for a significant portion of the rendering algorithm, which is typically true with complex, layered materials [JdJM14] or texture types that are costly to evaluate. Davidovič et al. [DKHS14] conduct a thorough survey of several global illumination algorithms implemented with CUDA and the freely available bounding volume hierarchy (BVH) intersection framework that was published in conjunction with [AL09]. They systematically evaluate several implementations and compare megakernel approaches against approaches that involve multiple kernel calls to render a frame. They observed that separate compute kernels for the shading phase and possible coherency sorting are beneficial with complex materials, but impose too high an overhead when only simple materials are involved. Coherence is also important to avoid *starvation* of large vector units. Interactive thread compaction of active paths in a GPGPU kernel to avoid starvation of individual warps was investigated in [Wal11]. The author concluded that the benefit was negligible and the strategy in some cases even resulted in a slowdown. Davidovič et al. [DKHS14] also investigate this matter in their survey paper and use atomic operations to reassign work units to vector lanes during the execution of general ray tracing kernels. With wavefront approaches, it is possible to perform the reassignment *inbetween* kernel calls.

3. Compile Time Polymorphism

In this section we briefly deduce the CTP concept and how to implement a static polymorphic type based on that. Our implementation relies on support for *variadic templates*, which were introduced with the (now superseded) ISO/IEC 14882:2011 standard (short: ISO C++11) [ISO11]. In this paper, we refer to core language versions that adhere to the ISO C++11 or newer standard as "modern C++". Core language support for variadic templates is necessary to implement a *variant* type as it is proposed to be included [Nau16] in the upcoming ISO C++17 standard, which is in draft status at the time of writing.

Variants derive from *tagged unions*, which basically extend the concept of ANSI-C unions by storing a tag along with them:

```
union Data {
    Type1 t1;
    Type2 t2;
    Type3 t3;
};

enum Tag { Tag1, Tag2, Tag3 };

struct TaggedUnion {
    Data data;
    Tag tag;
};
```

The tag can then be used at runtime to deduce the correct type id of the data object:

```
auto tu = makeTaggedUnion(···);
if (tu.tag == Tag1)
    treatAsT1(tu.data);
else if (tu.tag == Tag2)
    treatAsT2(tu.data);
else if (tu.tag == Tag3)
    treatAsT3(tu.data);
```

Tagged unions are not useful to design the API of a library because the number of supported types is static. Modern C++ variadic templates provide a means to implement a variant type that has a bit representation for a variable number of internal types. The data representation for a simple variant type looks as follows:

```
template <typename ...Ts>
union VariantStorage {};

template <typename T, typename ...Ts>
union VariantStorage<T, Ts...> {
    T element;
    /* Recursion stops when the empty
       VariantStorage<> template is
       being instantiated. */
    VariantStorage<Ts...> nextElements;
};
```

The variant type itself extends the variant storage type with a simple integer type id. At runtime, the variadic template parameter is unfolded to retrieve the index of the type (if present) in the parameter pack corresponding to type id:

```
template <typename ...Ts>
struct Variant {
    VariantStorage<Ts...> storage;
    int type_id;

    template <typename T>
    T* as() {
        // Reinterpret as type T if type id matches.
        if (type_id == index_of(T, Ts))
            return reinterpret_cast<T*>(&storage);
        else
            return nullptr;
    }
};
```

(For brevity, we omit the definition of the function `index_of()`, which returns the index of type `T` in the parameter pack `Ts`.) With this scheme a list of types is set up that can be statically traversed with zero runtime overhead using a compile time visitor pattern to unfold the parameter pack. Application code using variants structurally looks as follows:

```
struct Visitor {
    auto operator()(Type1 t1) {
        treatAsT1(t1);
    }

    auto operator()(Type2 t2) {
        treatAsT2(t2);
    }

    auto operator()(Type3 t3) {
        treatAsT3(t3);
    }
};

Variant<Type1, Type2> var = makeVariant(···);
applyVisitor(Visitor(), var);
```

(We omit the definition of `applyVisitor()`, which traverses the parameter pack, reinterprets the variant's `storage` member bitwise using `Variant::as()`, and then calls the appropriate `Visitor::operator()` overload.) A variant implementation is e.g. provided as part of the Boost C++ libraries [boo]. In order for compatibility with NVIDIA's CUDA or AMD's ROCm, and because support libraries such as Thrust [BH11] do not provide a variant type as of yet, it may in general be necessary to implement such a type as part of a utility library module.

4. Application to Ray Tracing

With variants it is easily possible to provide container types that mimic the behavior of abstract base classes at compile time. We propose a software interface and have integrated it into the ray tracing template library *Visionaray*. A casual description of CTP for ray tracing can be found in [ZWL17], but is however not thoroughly evaluated with regard to runtime performance overhead. Visionaray provides sets of default implementations for primitive intersection (triangles using the intersection test from [MT97], simple quadric types, BVHs which act as compound primitives), material shading, light sampling, and texture filtering. Visionaray requires primitives to implement the *intersect(Ray, Primitive)* customization point. Material shading can be customized by supplying custom material types that encapsulate *reflectance distribution functions* (BRDFs). Custom materials must supply member functions *shade()* and *sample()*, which invoke BRDF evaluation and numerical sampling, respectively. Customization of light and texture types works conceptually similar. We extended Visionaray with custom types that derive from variants, provide the interface that the respective customization point mandates, and implement the interface functions using the compile time visitor pattern we introduced above. We exemplarily show pseudo code for the implementations of a generic material type, which implements a member function interface to adhere to Visionaray's material shading and sampling interface. Note that those types are generic and can e.g. be compiled into a GPGPU kernel with the NVIDIA CUDA compiler or AMD's HCC compiler.

```
template <typename ...Ts>
class GenericMaterial<Ts...>
     : public Variant<Ts...> {
public:
    // Construct variant from concrete material.
    template <typename Mat>
    GenericMaterial(Mat mat)
        : Variant<Ts...>(mat) {}

    // Shade interface function.
    Spectrum shade(Intersection) {
        // ShadeVisitor visits all types in
        // parameter pack Ts... and calls
        // T::shade(Intersection)
        applyVisitor(ShadeVisitor(Intersection),
                     *this);
    }

    // Sample interface function.
    Spectrum sample(Intersection, Wi, Wo) {
        applyVisitor(SampleVisitor(Intersection,
                                   Wi, Wo),
                     *this);
    }
};
```

The *GenericMaterial* type resembles an abstract base type with an interface that can be extended by the application programmer, who instantiates the generic type with custom types as follows.

```
struct CustomMat1 {
    Spectrum shade(Intersection) {···}
    Spectrum sample(Intersecion, Wi, Wo) {···}
};

struct CustomMat2 {
    Spectrum shade(Intersection) {···}
    Spectrum sample(Intersecion, Wi, Wo) {···}
};

CustomMat1 cm1;
CustomMat2 cm2;

// Variant containing CustomMat1
GenericMaterial<CustomMat1, CustomMat2> gm(cm1);
```

```
// Variant containing CustomMat2
GenericMaterial<CustomMat1, CustomMat2> gm(cm2);
```

Builtin or custom functions for material shading can now be instantiated with the generic material type and will choose the right material type to shade or sample at runtime. Branching is executed in the visitor implementation, which unfolds the respective template types and invokes the material interface if types match. A type implemented in that way will, similarly to an ordinary polymorphic type, perform branching to deduce the type stored in the variant at runtime. The number of types stored in the variant is however known at compile time, so that early binding optimization is possible. In contrast to static approaches like tagged unions, the supported types need not to be known by the developer of the ray tracing library, but by the application programmer using the ray tracing library.

5. Results

We evaluate the performance impact of using CTP for ray tracing with incoherent workloads. We are interested in the performance with regard to the shading operation in a wavefront path tracing pipeline. GPGPU APIs like NVIDIA CUDA have limited support for virtual function calls. It is however not possible to create polymorphic objects in host memory and then copy those to GPU DDR memory, because CPU and GPU in general do not share address spaces. Because of that, shading in a GPU wavefront path tracing pipeline involves either sorting to avoid virtual function calls (because then the material type is known a priori) or shading on the CPU. With CTP, it is possible to perform shading on the GPU without having to sort intersection hit records. We would also like to know how CTP impacts the intersect operation with bounding BVHs that contain multiple primitive types for setups like the one depicted in Figure 1. An alternative strategy involves consecutively intersecting the ray buffer with one BVH per primitive type. This reduces branching at the cost of many additional full BVH traversal operations. When testing on the CPU, we are also interested in the performance of CTP compared to object oriented polymorphism (OOP) evaluated at runtime.

5.1. Wavefront path tracer

We evaluate our approach using a wavefront path tracer similar to the one proposed by Áfra et al. [ÁBWM16]. The path tracer can be compiled for NVIDIA GPUs with CUDA nvcc, and for CPUs with a conventional C++ compiler. Ray generation, primitive intersection, sorting, shading, active path compaction, and blending are implemented in separate compute kernels on the GPU and thread-parallel functions on the CPU. In contrast to Áfra et al., our kernels process single rays in parallel instead of combining rays to streams. We maintain and update buffers in DDR memory for rays, hit records from primitive intersection, and shaded pixel colors in a structure of array (SoA) fashion. We use the Visionaray ray tracing API to implement common tasks such as primitive intersection with an SBVH [SFD09] and BRDF sampling. Reordering operations (sorting and compaction) are executed on an index buffer into the ray, hit record, and pixel buffers. Depending on the modality we would like to test, the shade kernel is further divided into one kernel per material type. We are interested in how a single shade kernel with CTP compares to ordinary OOP using a single kernel, and to one shade kernel per material type. The latter approach requires a priori subdividing the materials into separate lists (one per material type) and reordering paths according to the material type attached to the surface that was hit during primitive intersect before shading. The single shade kernel cases involve no sorting. For the case that involves multiple shade kernels, like Áfra et al. we use counting sort [Knu98] to reorder intersection hit records by material types. We use the prefix sum array that is calculated during the counting sort algorithm execution as offsets into the index buffer so we can identify the workloads to be processed by the shade kernel in constant time. When compiling for the GPU, we assign each shade kernel to a separate CUDA stream so that the kernels can be scheduled concurrently. We found concurrent kernel execution vital for shading throughput. Note that contemporary NVIDIA compute architectures limit the number of concurrently executed kernels to 32 [NVI17], which may result in an execution bottleneck in the event that many shade kernels (e.g. one per material ID) are executed. See Figure 2 for an overview of the control flow of the two

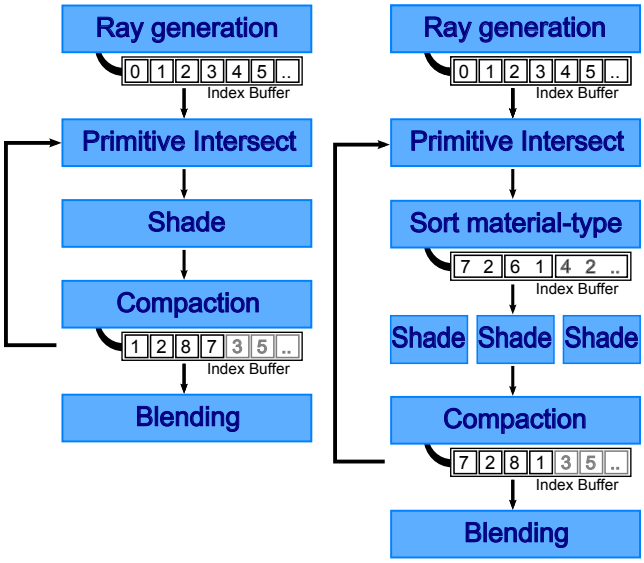


Figure 2: *Compute kernels used to implement the wavefront path tracer that we use to evaluate shading throughput. Left: single shade kernel, uses CTP or OOP for materials. Right: one shade kernel per material type, requires a priori sorting.*

implementations. In contrast to Laine et al. [LKA13] and Áfra et al. we are interested in real-time rather than offline rendering and thus limit our tests to single layer materials with textures for diffuse color only. A typical workload will comprise shading costs to be approximately 10% compared to those for primitive intersection. Such a scenario is untypical for offline rendering involving complex multi layer materials, but makes sense e.g. to improve rendering fidelity in the context of scientific visualization. We acknowledge the findings by Davidovič et al. [DKHS14] that a wavefront approach with a separate shading kernel is not optimal for simple materials, however, in order to be able to faithfully compare with a sorting pipeline, we opted to employ such a setup for all our tests.

In order to test the rendering throughput when more than one primitive type is involved, we either employ multiple consecutive primitive intersection kernels, or we compile a compute kernel that intersects the ray buffer with a list of generic primitives organized using variants. With that scenario we are only interested in intersect throughput because no ray reordering is involved and so the throughput of the remaining operations is unaltered.

5.2. Test setup

We test on an NVIDIA GeForce GTX 1080Ti GPU system, and on an Intel Xeon E5-2637 v2 dual CPU system with a total of eight cores and sixteen threads. We use the NVIDIA Visual Profiler, the NVIDIA CUDA command line profiler and the Intel vTune profiler to measure parallel compute throughput, and to ensure that *GPU* kernels assigned to separate streams actually execute in parallel. We use the gcc 5.4.0 and CUDA 8.0 Linux toolchains to compile our test program.

5.3. Shading throughput

Figure 3: *Conference Room and Crytek Sponza Atrium test scenes comprised of simple, optionally textured, materials: matte material with Lambertian BRDF, plastic material with a Lambertian and a Blinn microfacet term, mirror material with perfect specular reflection, emissive material with a constant emissive term. We test if it is beneficial to pack all materials in a single list using CTP or OOP, or to sort hit points and use one parallel compute kernel per material type.*

In order to measure shading throughput, we render progressive frames that converge to the 1024 × 1024 pixel images (Conference Room and Crytek Sponza Atrium) shown in Figure 3. The setup involves four material types: a diffuse matte material with a Lambertian BRDF, a plastic material with a diffuse Lambertian and a glossy Blinn microfacet term, a mirror material with a perfectly specular term, and an emissive material so geometry can behave like light sources. We test the throughput of the wavefront path tracing pipeline described above by either using a single parallel compute kernel and CTP (or OOP on the CPU) for shading, or by using a separate compute kernel for each of the four materials. We report the throughput of the involved compute kernels in Table 1 and execution time for ten reflective bounces in Figure 4. Since rays are potentially reordered throughout the whole pipeline, and because reordering for shading during one bounce may affect the performance of the intersect execution of the ensuing bounce, we opted to provide results for all *per bounce* kernels.

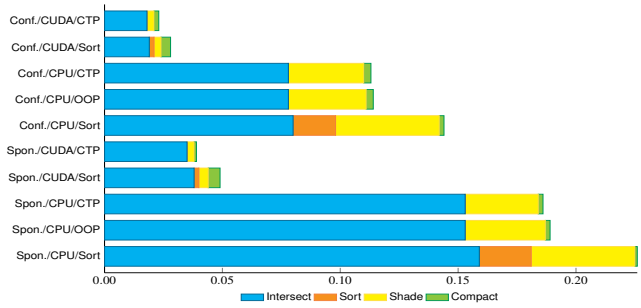


Figure 4: *Times in seconds needed to execute the* per bounce *parallel compute kernels from Figure 2 in order to render one progressive frame, one of many that eventually converge to obtain the images from Figure 3 . We report the accumulated time it takes to perform ten reflective bounces with the intersect, sort, shade, and compaction kernels. Throughput results with respect to a single bounce can be found in Table 1.*

5.4. Primitive intersection throughput

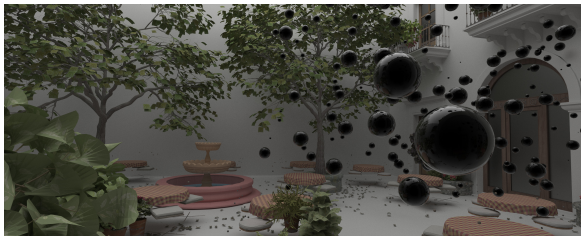
Spheres	San Miguel (7,842K triangles)	Conference (331K triangles)
1K	5,976K nodes	253K nodes
10K	5,986K nodes	260K nodes
100K	6,047K nodes	328K nodes

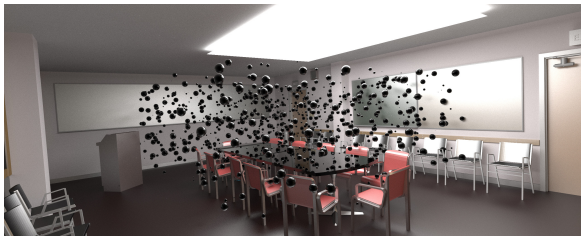
Table 2: *Footprint for SBVHs containing both triangles and spheres. For comparison, SBVHs containing only the triangle geometry of the San Miguel and Conference room scene are comprised of 5,971K and 252K nodes, respectively. SBVHs containing 1K, 10K, or 100K spheres are comprised of 675, 6,827, and 68,233 nodes, respectively.*

We test the throughput of the intersect kernel when either using CTP, OOP on the CPU, or an approach with one kernel per primitive type (we refer to this as "kernel restart"). We measure rendering single progressive frames that eventually converge to provide the images shown above Table 3. We therefore place uniformly distributed, reflective spheres inside the San Miguel scene (7,842K triangles) and Conference Room scene (331K triangles). We render images with a resolution of 2,560 × 1,024 pixels. We run our test with 1K, 10K, and 100K spheres. See Table 2 for an overview of the footprint of the SBVHs in memory (a tree node consists of an axis aligned bounding box and integer indices and requires 32 bytes of aligned memory. Note that the number of nodes is not necessarily linearly related to the number of primitives contained in the SBVH. For the San Miguel model, the construction algorithm finds a configuration with even fewer nodes for the combined triangle and sphere setup than for the mere triangle geometry. Table 3 presents throughput results for the intersect routine and the various configurations. We can see that using CTP for intersect on the GPU

	Conference					Crytek Sponza				
	CUDA		CPU			CUDA		CPU		
Kernel	CTP	Sorting	CTP	OOP	Sorting	CTP	Sorting	CTP	OOP	Sorting
Intersect	56.2	51.8	12.9	12.8	12.5	28.6	26.5	6.55	6.53	6.32
Sort	n/a	359	n/a	n/a	45.4	n/a	372	n/a	n/a	41.6
Shade	362	342	31.6	30.4	22.6	307	272	32.3	29.5	22.9
Compact	586	234	389	395	453	737	183	542	581	685

Table 1: *Throughput of the* per bounce *parallel compute kernels from Figure 2 when either using CTP, OOP, or hitpoint sorting and one kernel per material type for shading, in million rays per second (Mrays/s). Figure 4 presents accumulated results over ten bounces for the same kernels in units of time.*





	San Miguel					Conference				
	CUDA		CPU			CUDA		CPU		
Spheres	CTP	Kernel Restart	CTP	OOP	Kernel Restart	CTP	Kernel Restart	CTP	OOP	Kernel Restart
1K	9.48	27.3	4.56	3.91	4.33	17.8	81.5	12.6	11.8	10.9
10K	8.78	25.8	4.25	3.63	4.05	13.1	66.5	10.2	9.46	8.72
100K	7.59	21.3	3.97	3.32	3.58	9.52	36.8	7.39	6.67	6.51

Table 3: *Throughput in Mrays/s of the intersect parallel compute kernel when combining triangle and sphere primitives in the same ray tracing scene. We compare CTP, OOP (only on the CPU), and storing all primitives in the same SBVH with an approach where we consecutively intersect ray wavefronts with one BVH per primitive type. We test on the CPU and on the GPU, and with either 1K, 10K, or 100K uniformly distributed spheres placed inside the boundaries of the triangle geometry.*

is prohibitive. Performance decreases by a factor three to four compared to a kernel restart approach. The opposite is however true on the CPU, where some test cases show a performance increase of about 10 to 15% compared to kernel restart. CTP is always beneficial compared to OOP, and our results indicate that the impact of using OOP increases with the number of primitive intersection tests. The fact that intersect throughput drops so drastically when using CTP on the GPU is not surprising since this architecture is particularly ill suited for conditional branching. We find it noteworthy that the opposite behavior can be observed on the CPU.

6. Conclusions

We have presented compile time polymorphism with C++ variants as an elegant way to provide an extensible library interface for small data types that are used in the innermost loops of ray tracing algorithms. Due to the support for modern C++ on a variety of architectures, this approach is also applicable to GPGPUs. Data types implemented with CTP can be "plain old data types" and can thus be extended to an accelerator over PCI Express using memory copy operations. The most important benefit for a library implementation in our opinion is that the number of types the variant is instantiated with does not need to be known a priori to the library programmer. We evaluated CTP and tested it against traditional strategies that involve additional sequential or concurrent compute kernel executions. We come to the conclusion that CTP is an option to redesign ray tracing algorithms that are dominated by memory operations and where additional indirection e.g. due to sorting outweighs the benefit of more coherent operations and reduced branching. In certain cases and especially on GPUs, we however found CTP to reduce throughput by a significant amount. When to use CTP and when to apply an alternative optimization strategy that reduces branching in inner loops should in our opinion be tested from case to case. Our tests especially revealed that on a CPU, in cases where the most time of the ray tracing algorithm is spent for intersect, preferring CTP over object oriented programming or kernel restart even resulted in a performance increase.

C++ has changed significantly over the last couple of years by adding elements typical to functional and imperative programming languages that allow for writing highly expressive code. Modern C++ compiler infrastructures like LLVM [LA04] are open source, highly extensible and lay the ground for standards-compliant C++ implementations even on exotic hardware architectures. We have shown that modern C++ language features can enrich the way real-time graphics libraries are designed. While such features allow for writing most expressive code, they however do not make up for explicitly having to test for runtime performance and can complement platform specific optimizations, but do not generally render them unnecessary.

Acknowledgements

The authors wish to thank Arsène Pérard-Gayot for the helpful discussion about wavefront path tracing and hitpoint sorting. The Crytek Sponza, Conference Room, and San Miguel 3-D meshes we used for the evaluation are freely available thanks to Morgan McGuire [McG11].

References

[ÁBWM16] ÁFRA A. T., BENTHIN C., WALD I., MUNKBERG J.: Local Shading Coherence Extraction for SIMD-Efficient Path Tracing on CPUs. In *Eurographics/ ACM SIGGRAPH Symposium on High Performance Graphics* (2016), Assarsson U., Hunt W., (Eds.), The Eurographics Association. doi:10.2312/hpg.20161198. 3, 5

[AL09] AILA T., LAINE S.: Understanding the efficiency of ray traversal on GPUs. In *Proceedings of the Conference on High Performance Graphics 2009* (New York, NY, USA, 2009), HPG '09, ACM, pp. 145–149. URL: http://doi.acm.org/10.1145/1572769.1572792, doi:10.1145/1572769.1572792. 3

[BH11] BELL N., HOBEROCK J.: Thrust: A productivity-oriented library for CUDA. In *GPU Computing Gems Jade Edition*, Hwu W.-m. W., (Ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2011, ch. 26, pp. 359–373. 4

[boo] Boost C++ libraries. http://www.boost.org/. Accessed: 2017-05-30. 4

[DKHS14] DAVIDOVIČ T., KŘIVÁNEK J., HAŠAN M., SLUSALLEK P.: Progressive light transport simulation on the GPU: Survey and improvements. *ACM Trans. Graph. 33*, 3 (June 2014), 29:1–29:19. URL: http://doi.acm.org/10.1145/2602144, doi:10.1145/2602144. 3, 5

[GL10] GARANZHA K., LOOP C.: Fast Ray Sorting and Breadth-First Packet Traversal for GPU Ray Tracing. *Computer Graphics Forum* (2010). doi:10.1111/j.1467-8659.2009.01598.x. 3

[ISO11] *ISO/IEC 14882:2011, Information technology – Programming languages – C++*. Standard, International Organization for Standardization, Geneva, CH, Sept. 2011. 3

[Jak10] JAKOB W.: Mitsuba renderer, 2010. Accessed: 2017-05-30. URL: http://www.mitsuba-renderer.org. 2

[JdJM14] JAKOB W., D'EON E., JAKOB O., MARSCHNER S.: A comprehensive framework for rendering layered materials. *ACM Trans. Graph. 33*, 4 (July 2014), 118:1–118:14. URL: http://doi.acm.org/10.1145/2601097.2601139, doi:10.1145/2601097.2601139. 3

[Knu98] KNUTH D. E.: *The Art of Computer Programming, Volume 3: (2nd Ed.) Sorting and Searching*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1998. 5

[LA04] LATTNER C., ADVE V.: LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization* (Washington, DC, USA, 2004), CGO '04, IEEE Computer Society, pp. 75–. URL: http://dl.acm.org/citation.cfm?id=977395.977673. 7

[LBH*15] LEISSA R., BOESCHE K., HACK S., MEMBARTH R., SLUSALLEK P.: Shallow embedding of DSLs via online partial evaluation. In *Proceedings of the 14th International Conference on Generative Programming: Concepts & Experiences (GPCE)* (10 2015), ACM, pp. 11–20. 3

[LKA13] LAINE S., KARRAS T., AILA T.: Megakernels considered harmful: Wavefront path tracing on GPUs. In *Proceedings of the 5th High-Performance Graphics Conference* (New York, NY, USA, 2013), HPG '13, ACM, pp. 137–143. URL: http://doi.acm.org/10.1145/2492045.2492060, doi:10.1145/2492045.2492060. 3, 5

[McG11] MCGUIRE M.: Computer graphics archive, August 2011. URL: http://graphics.cs.williams.edu/data. 8

[MS00] MCNAMARA B., SMARAGDAKIS Y.: Static interfaces in C++. In *In First Workshop on C++ Template Programming* (2000). 2

[MT97] MÖLLER T., TRUMBORE B.: Fast, minimum storage ray-triangle intersection. *J. Graph. Tools 2*, 1 (Oct. 1997), 21–28. URL: http://dx.doi.org/10.1080/10867651.1997.10487468. 4

[Nau16] NAUMANN A.: *P0088R2, ISO/IEC JTC1 SC22 WG21, Variant: a type-safe union for C++17 (v7)*. Working paper, Mar. 2016. 3

[NBGS08] NICKOLLS J., BUCK I., GARLAND M., SKADRON K.: Scalable parallel programming with CUDA. *Queue 6*, 2 (Mar. 2008), 40–53. URL: http://doi.acm.org/10.1145/1365490.1365500. 2

[NVI17] NVIDIA C.: Tuning CUDA Applications for Kepler. http://docs.nvidia.com/cuda/kepler-tuning-guide/, June 2017. Accessed: 2017-06-12. 5

[PBD*10] PARKER S. G., BIGLER J., DIETRICH A., FRIEDRICH H., HOBEROCK J., LUEBKE D., MCALLISTER D., MCGUIRE M., MORLEY K., ROBISON A., STICH M.: OptiX: A general purpose ray tracing engine. *ACM Trans. Graph. 29*, 4 (July 2010), 66:1–66:13. URL: http://doi.acm.org/10.1145/1778765.1778803. 2

[PBMH02] PURCELL T. J., BUCK I., MARK W. R., HANRAHAN P.: Ray tracing on programmable graphics hardware. *ACM Transactions on Graphics 21*, 3 (July 2002), 703–712. ISSN 0730-0301 (Proceedings of ACM SIGGRAPH 2002). 2

[PJH16] PHARR M., JAKOB W., HUMPHREYS G.: *Physically Based Rendering: From Theory To Implementation*, 3rd ed. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2016. 2

[PM12] PHARR M., MARK W. R.: ispc: A SPMD compiler for high-performance CPU programming. In *2012 Innovative Parallel Computing (InPar)* (May 2012), pp. 1–13. doi:10.1109/InPar.2012.6339601. 3

[SFD09] STICH M., FRIEDRICH H., DIETRICH A.: Spatial splits in bounding volume hierarchies. In *Proceedings of the Conference on High Performance Graphics 2009* (New York, NY, USA, 2009), HPG '09, ACM, pp. 7–13. URL: http://doi.acm.org/10.1145/1572769.1572771, doi:10.1145/1572769.1572771. 5

[SG08] SLUSALLEK P., GEORGIEV I.: RTfact: Generic concepts for flexible and high performance ray tracing. In *Proceedings of the IEEE / EG Symposium on Interactive Ray Tracing 2008* (2008), Trew R. J., (Ed.), IEEE Computer Society, Eurographics Association, IEEE, pp. 115–122. 2

[Wal04] WALD I.: *Realtime Ray Tracing and Interactive Global Illumination*. PhD thesis, Computer Graphics Group, Saarland University, 2004. 2

[Wal11] WALD I.: Active Thread Compaction for GPU Path Tracing. In *Eurographics/ ACM SIGGRAPH Symposium on High Performance Graphics* (2011), Dachsbacher C., Mark W., Pantaleoni J., (Eds.), ACM. doi:10.1145/2018323.2018331. 3

[Wil93] WILT N. P.: *Object-Oriented Ray Tracing in C++*. John Wiley & Sons, Inc., New York, NY, USA, 1993. 2

[WWB*14] WALD I., WOOP S., BENTHIN C., JOHNSON G. S., ERNST M.: Embree: A kernel framework for efficient CPU ray tracing. *ACM Trans. Graph. 33*, 4 (July 2014), 143:1–143:8. URL: http://doi.acm.org/10.1145/2601097.2601199, doi:10.1145/2601097.2601199. 2

[ZWL17] ZELLMANN S., WICKEROTH D., LANG U.: Visionaray: A cross-platform ray tracing template library. In *Proceedings of the 10th Workshop on Software Engineering and Architectures for Realtime Interactive Systems (IEEE SEARIS 2017)* (in press, 2017), IEEE. 2, 4