LuisaRender

A High-Performance Rendering System

Shaokun Zheng[1], Zhiqian Zhou[1], Xin Chen[1], Difei Yan[1], Chuyan Zhang[1], Yuefeng Geng[2], Yan Gu[3], and Kun Xu[1]

[1] Tsinghua University [2] Recreate Games [3] University of California, Riverside

luisa-render.com

LuisaRender

A High-Performance Rendering System

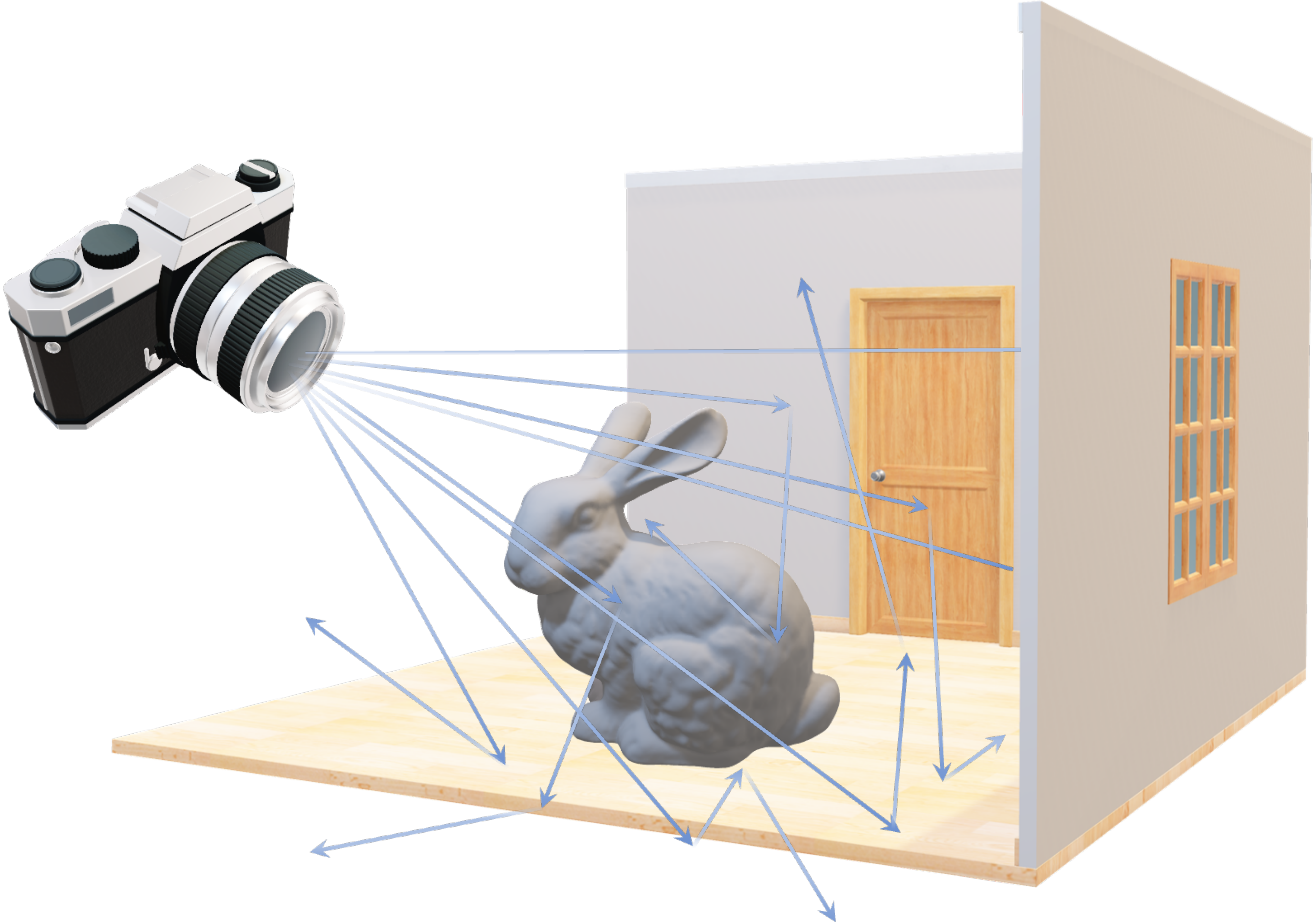
LuisaRender

A High-Performance Rendering System with Layered and Unified Interfaces on Stream Architectures

LuisaCompute

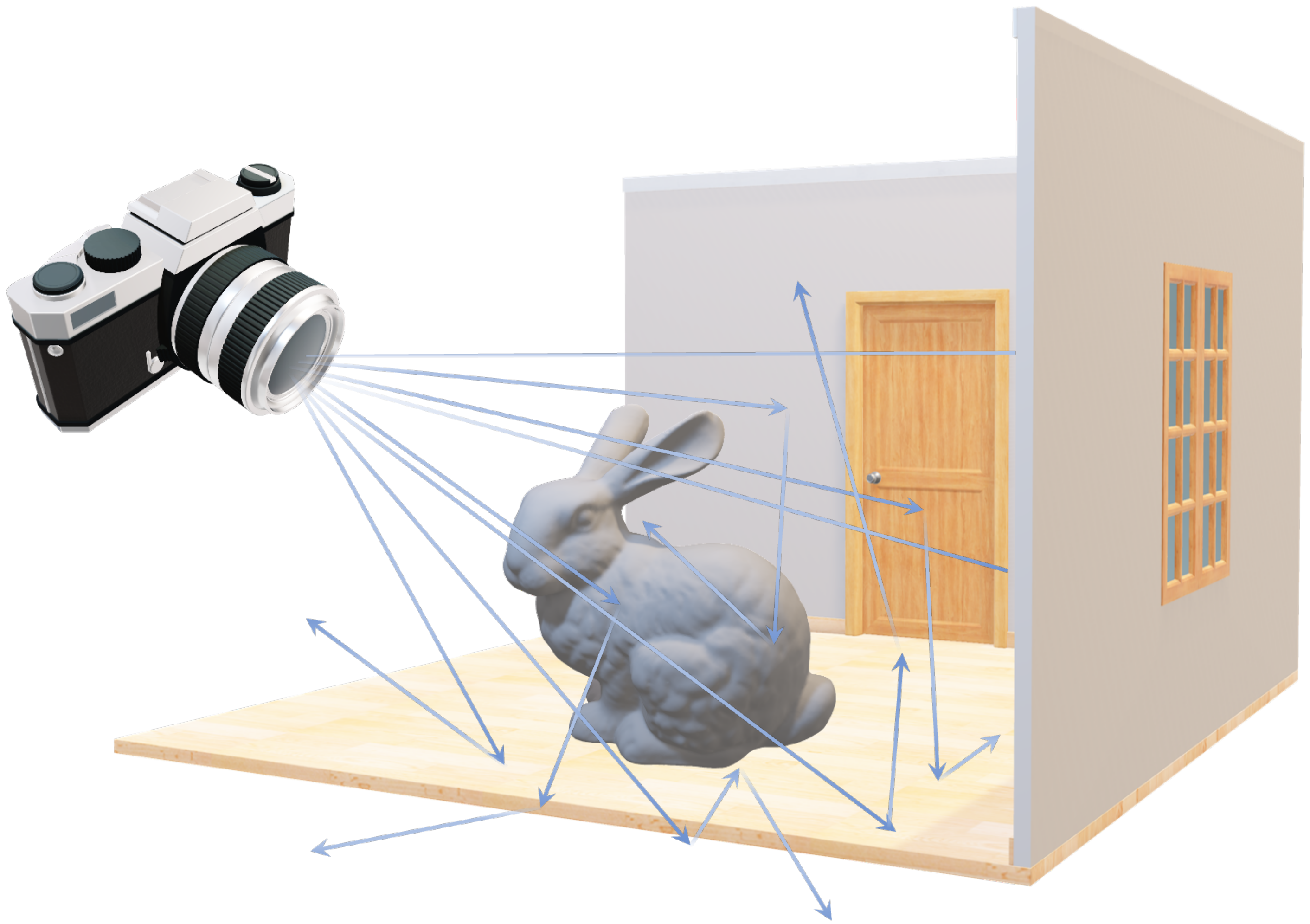
A Computing Framework on Stream Architectures for Rendering and beyond

我们为什么要做一个
新的渲染计算框架？



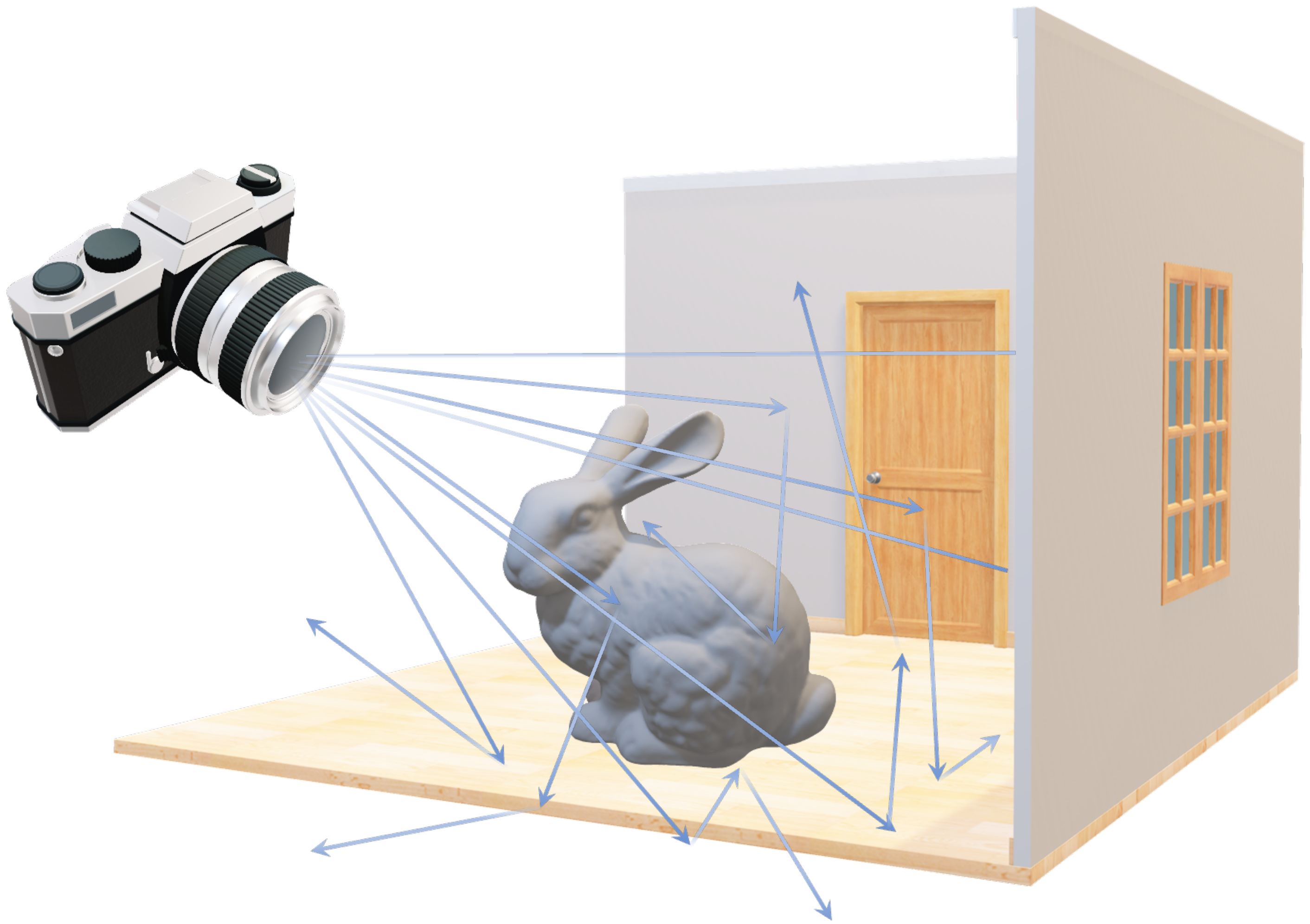
Step 1: CPU 上的玩具渲染器

```
for pixel in image:
  Li = [0, 0, 0]
  ray = camera.generate_ray(pixel)
  for depth in range(bounces):
    hit = scene.trace_closest(ray)
    Li += hit.material.shade(ray, hit)
    ray = hit.material.sample(ray, hit)
  film[pixel] += Li
```



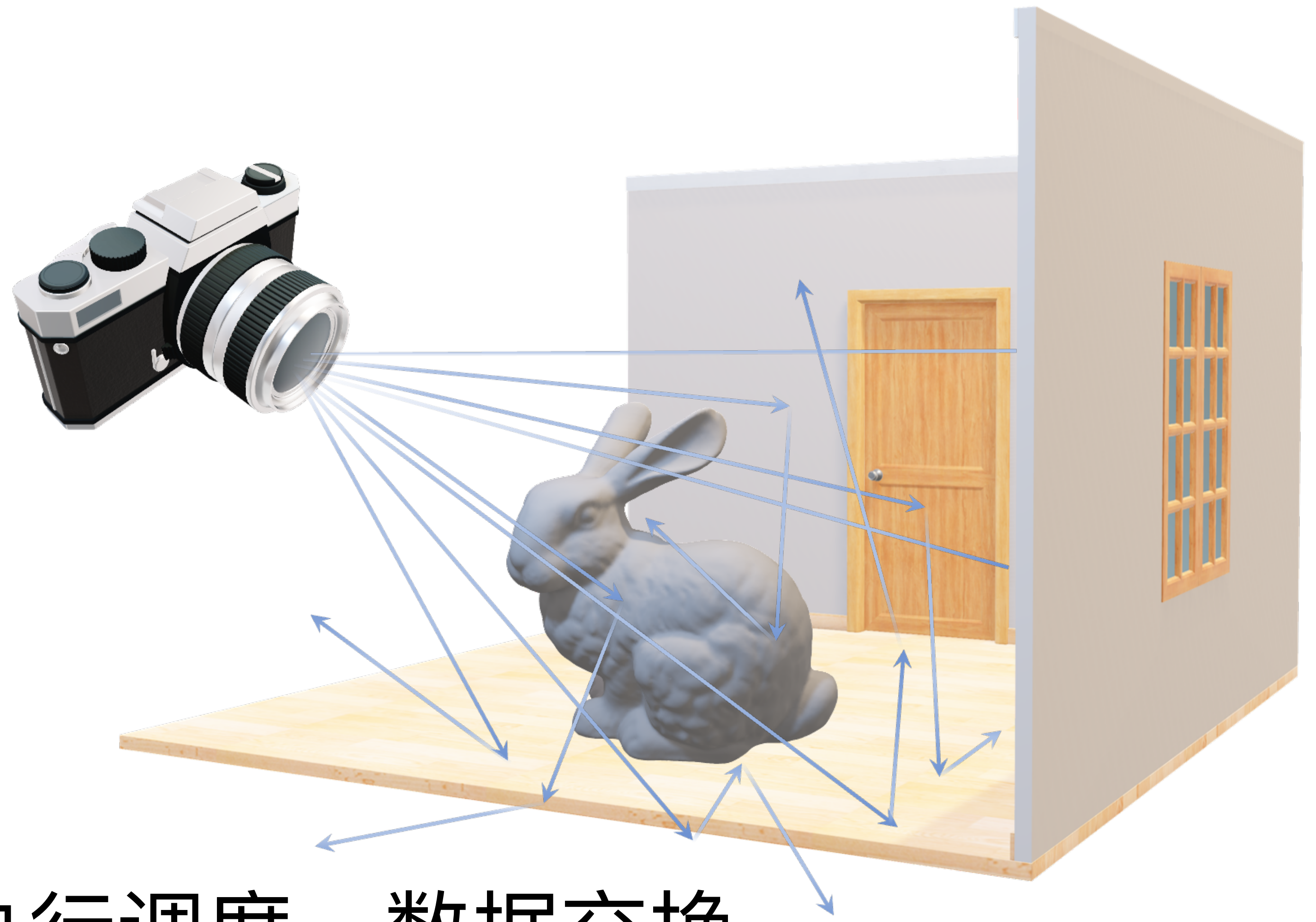
Step 1: CPU 上的玩具渲染器

```
for pixel in image:
  Li = [0, 0, 0]
  ray = camera.generate_ray(pixel)
  for depth in range(bounces):
    hit = scene.trace_closest(ray)
    Li += hit.material.shade(ray, hit)
    ray = hit.material.sample(ray, hit)
  film[pixel] += Li
```



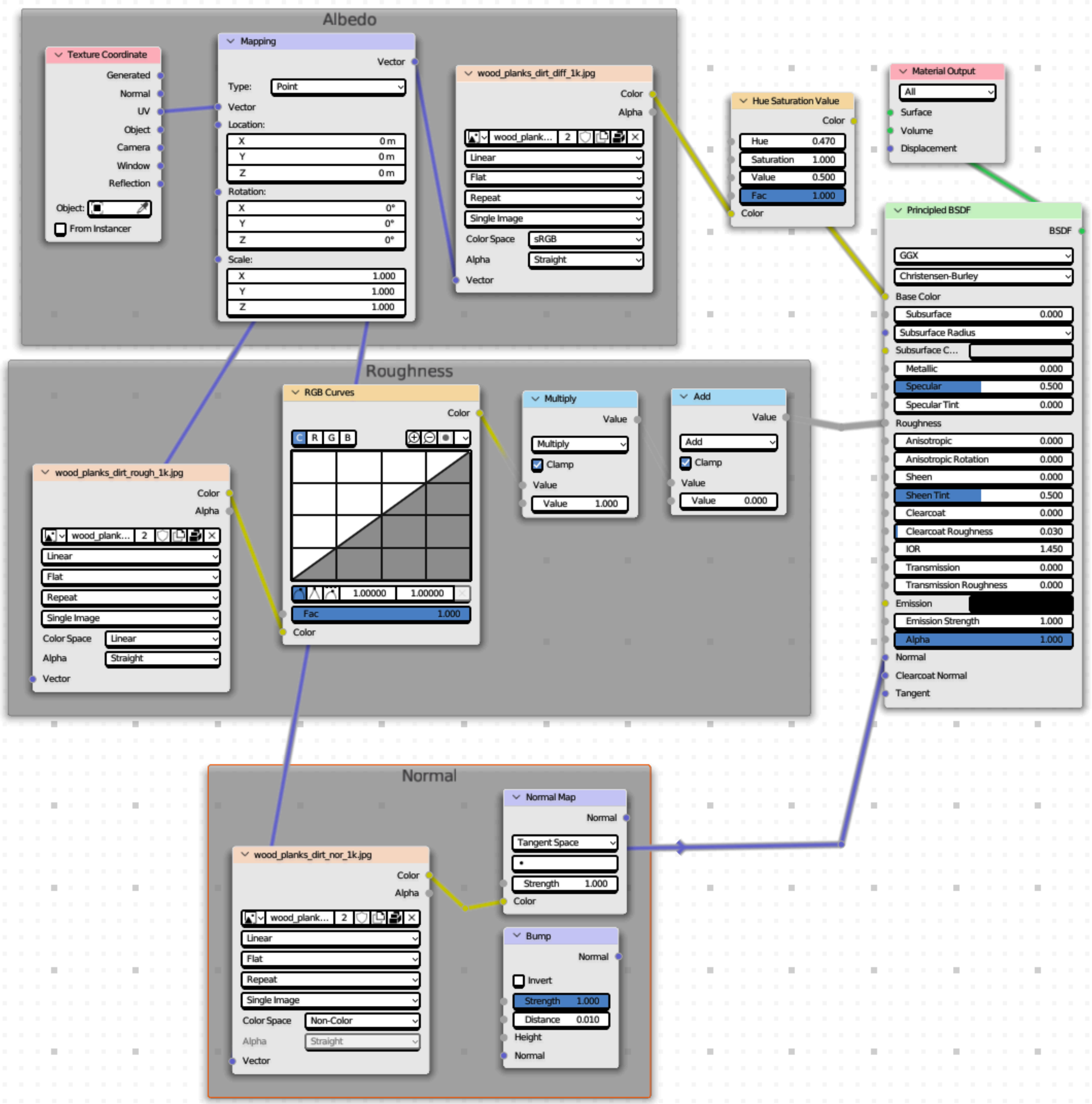
Step 2: GPU 上的玩具渲染器

```
kernel render(scene, camera, film):
  Li = [0, 0, 0]
  ray = camera.generate_ray(pixel)
  for depth in range(bounces):
    hit = scene.trace_closest(ray)
    Li += hit.material.shade(ray, hit)
    ray = hit.material.sample(ray, hit)
  film[pixel] += Li

gpu_device.dispatch(render, pixels)
```



复杂度提升：着色器编写、资源管理、执行调度、数据交换...

Step 3.1: 丰富 GPU 渲染器的材质/光照/...



❌ 着色语言表达力不佳

- 大量的运行时分支/着色器变体?
- 如何动态扩展新插件?
- 如何支持 Shader Graph?

Step 3.2: 支持其他硬件/平台...

```
kernel void add_arrays(device const float* inA,
                       device const float* inB,
                       device float* result,
                       uint index [[thread_position_in_grid]])
{
    // the for-loop is replaced with a collection of threads, each of which
    // calls this function.
    result[index] = inA[index] + inB[index];
}
```

```
__global__ void vecAdd(int *A,int *B,int *C,int N)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    C[i] = A[i] + B[i];
}
```

❌ 着色语言表达力不佳

❌ 分裂的 API 和着色语言

- 相同的逻辑编写 N 遍?
- 使用大量宏统一语法?
- 复杂的着色器交叉编译?

DirectX XII ULTIMATE

NVIDIA CUDA

Vulkan™

LLVM

oneAPI

Step 4: 维护和性能优化...

❌ 着色语言表达力不佳

❌ 分裂的 API 和着色语言

❌ 维护复杂，优化困难，极易出错

Step 4: 维护和性能优化...

❌ 着色语言表达力不佳

❌ 分裂的 API 和着色语言

❌ 维护复杂，优化困难，极易出错

- 扩展能力低下
- 大量的着色器变体
- 多平台的大量重复代码
- 繁杂的编译/配置流程
- 大量的运行时分支
- 性能低下的 Shader Virtual Machine
- 混乱的宏开关

可编程性

跨平台

高性能

❌ 着色语言表达力不佳

❌ 分裂的 API 和着色语言

❌ 维护复杂，优化困难，极易出错

可编程性

跨平台

高性能

相关工作

- Slang [He et al. 2017]
  - 扩展 HLSL，提供泛型和接口语法，用于多态编程与着色器特化
  - 引入“参数块”功能以提升着色器参数绑定效率
- Rodent [Pérard-Gayot et al. 2019]
  - 利用 AnyDSL [Leißa et al. 2018] 的 Partial Evaluation 能力为每个场景特化渲染器
- Dr.JIT [Jakob et al. 2022]
  - 嵌入 C++ 和 Python 的追踪式的可微分领域特定语言（DSL）
- 渲染之外：Halide [Ragan-Kelley et al. 2012]、Taichi [Hu et al. 2019]

相关工作

领域特定语言

- Slang [He et al. 2017]
  - 扩展 HLSL，提供泛型和接口语法，用于多态编程与着色器特化
  - 引入“参数块”功能以提升着色器参数绑定效率
- Rodent [Pérard-Gayot et al. 2019]
  - 利用 AnyDSL [Leißa et al. 2018] 的 Partial Evaluation 能力为每个场景特化渲染器
- Dr.JIT [Wenzel et al. 2022]
  - 嵌入 C++ 和 Python 的追踪式的可微分领域特定语言（DSL）
- 渲染之外：Halide [Ragan-Kelley et al. 2012]、Taichi [Hu et al. 2019]

相关工作

领域特定语言

- Slang [He et al. 2017]
  - 扩展 HLSL，提供泛型和接口语法，用于多态编程与着色器特化
  - 引入“参数块”功能以提升着色器参数绑定效率
- Rodent [Pérard-Gayot et al. 2019]
  - 利用 AnyDSL [Leißa et al. 2018] 的 Partial Evaluation 能力为每个场景特化渲染器
- Dr.JIT [Wenzel et al. 2022]
  - 嵌入 C++ 和 Python 的追踪式的可微分领域特定语言（DSL）
- 渲染之外：Halide [Ragan-Kelley et al. 2012]、Taichi [Hu et al. 2019]

即时编译

相关工作

- Slang [He et al. 2017]
  - 扩展 HLSL，提供泛型和接口语法，用于多态编程与着色器特化
  - 引入“参数块”功能以提升着色器参数绑定效率
- Rodent [Pérard-Gayot et al. 2019]
  - 利用 AnyDSL [Leißa et al. 2018] 的 Partial Evaluation 能力为每个场景特化渲染器
- Dr.JIT [Wenzel et al. 2022]
  - 嵌入 C++ 和 Python 的追踪式的可微分领域特定语言（DSL）
- 渲染之外：Halide [Ragan-Kelley et al. 2012]、Taichi [Hu et al. 2019]

领域特定语言

即时编译

特化

相关工作

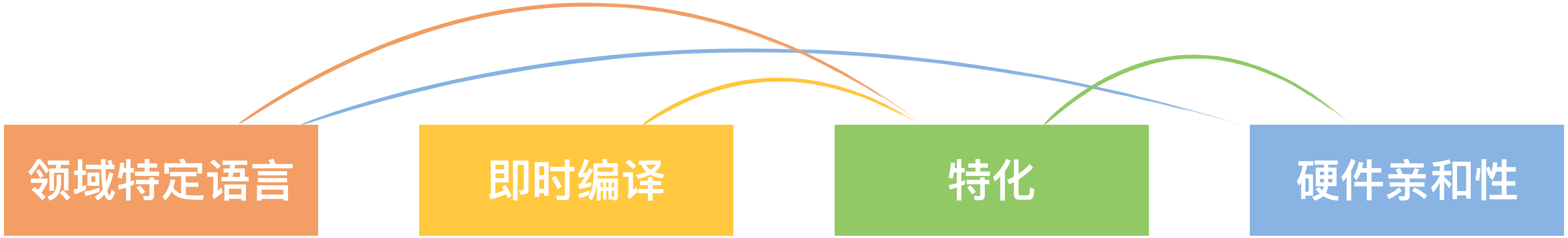
- Slang [He et al. 2017]
  - 扩展 HLSL，提供泛型和接口语法，用于多态编程与着色器特化
  - 引入“参数块”功能以提升着色器参数绑定效率
- Rodent [Pérard-Gayot et al. 2019]
  - 利用 AnyDSL [Leißa et al. 2018] 的 Partial Evaluation 能力为每个场景特化渲染器
- Dr.JIT [Wenzel et al. 2022]
  - 嵌入 C++ 和 Python 的追踪式的可微分领域特定语言（DSL）
- 渲染之外：Halide [Ragan-Kelley et al. 2012]、Taichi [Hu et al. 2019]

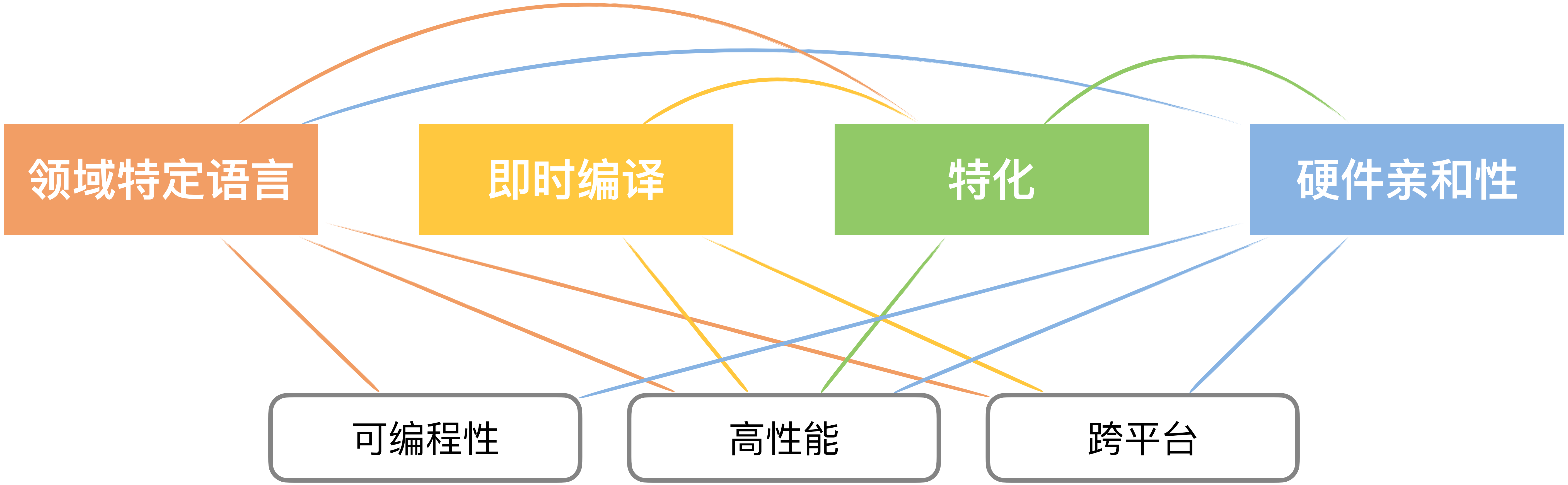
领域特定语言

硬件亲和性

即时编译

特化


领域特定语言
即时编译
特化
硬件亲和性



领域特定语言
即时编译
特化
硬件亲和性
可编程性
高性能
跨平台

可编程性

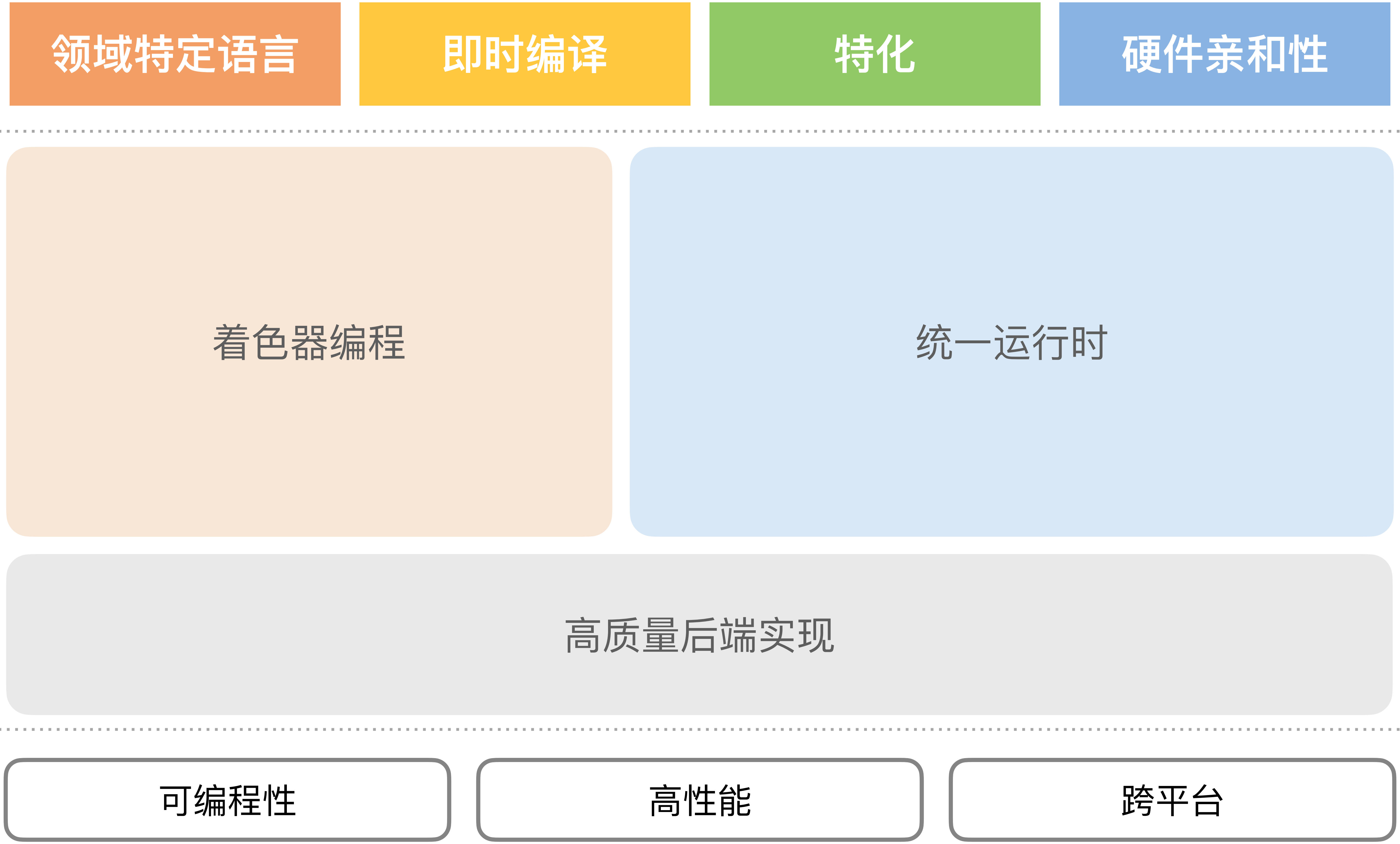
- **嵌入式领域特定语言**
  - 设备端与宿主端代码交互便利：参数绑定、布局兼容...
  - 支持高级抽象模式的语法元素，如动态多态、泛型、模板...
  - 复用宿主语言的类型检查与推断、提供丰富的内置函数...

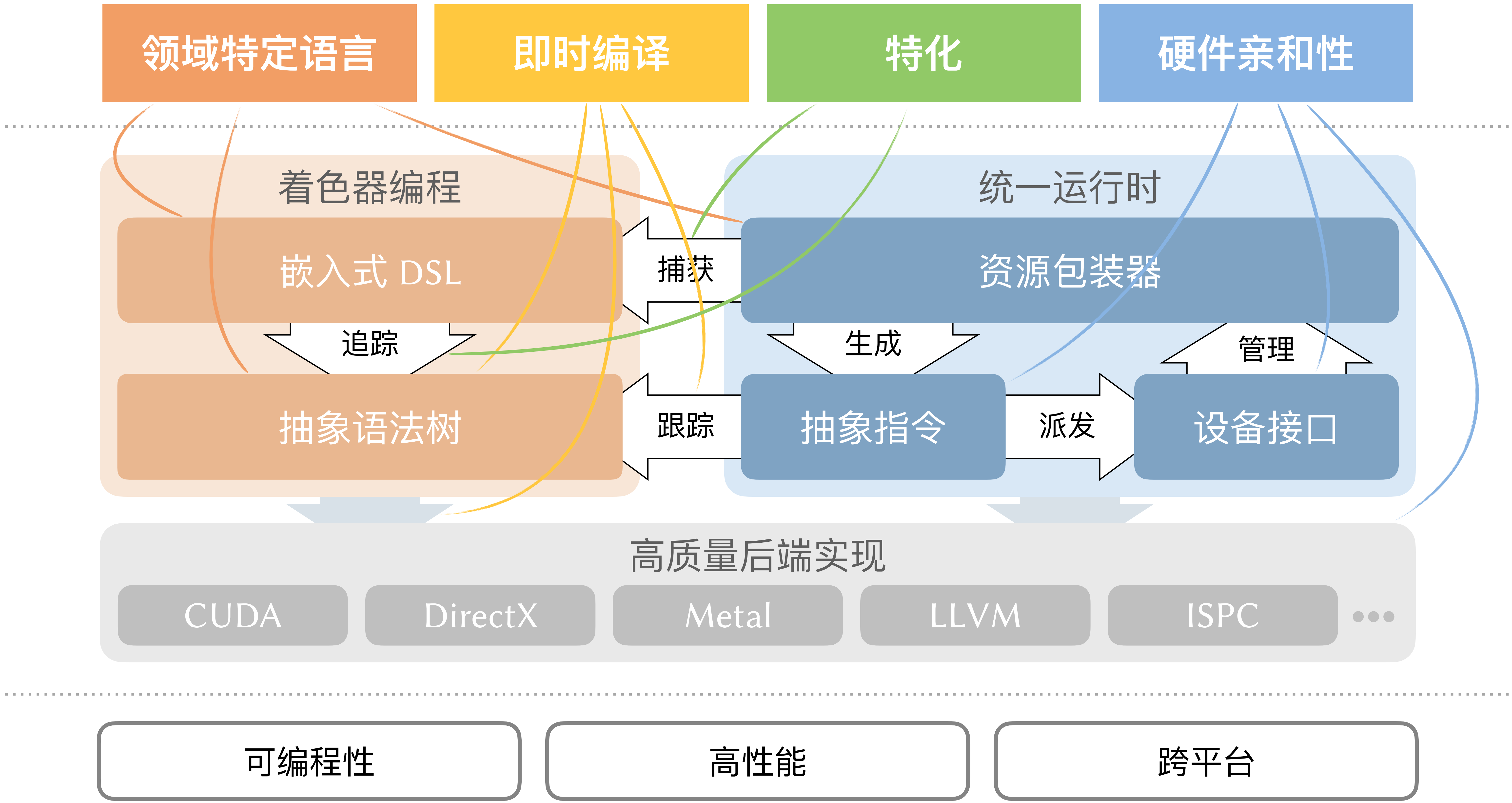
跨平台

- 利用**抽象层**实现“一次编写，各处运行”
  - 跨平台统一的**运行时编程接口**
  - 跨平台统一的**着色器编程 DSL**
- **适配各平台特性**，无需逐平台性能调优
  - 不同后端根据硬件平台特性，充分利用相应资源

高性能

- **动态代码生成**与**编译优化**
- 良好的**分层设计**，粒度合适的**低开销抽象**
- **高硬件亲和性**的后端实现
  - **单指令多线程**（SIMT）的编程模式
  - 根据硬件特性利用**专有硬件**和**编译器指令**


领域特定语言
即时编译
特化
硬件亲和性
着色器编程
统一运行时
高质量后端实现
可编程性
高性能
跨平台



领域特定语言
即时编译
特化
硬件亲和性
着色器编程
统一运行时
嵌入式 DSL
捕获
资源包装器
追踪
生成
管理
抽象语法树
跟踪
抽象指令
派发
设备接口
高质量后端实现
CUDA
DirectX
Metal
LLVM
ISPC
可编程性
高性能
跨平台

样例

```cpp
// initialize the device
auto device = context.create_device("cuda");

// define the rendering kernel
Kernel2D kernel = [&](ImageFloat image) {
  auto pos = dispatch_id().xy();
  auto color = sin(make_float2(p)) * .5f + .5f;
  image.write(pos, make_float4(color, 1.f, 1.f));
};

// create resources on the device and on the host
auto size = make_uint2(1024u);
auto render = device.compile(kernel);
auto image = device.create_image<float>(BYTE4, size);
auto host_image = std::vector<float4>(size.x * size.y);

// create a stream for submitting tasks
auto stream = device.create_stream();

// dispatch tasks and wait for completion
stream << render(image).dispatch(size)
       << image.copy_to(host_image.data())
       << synchronize();
```

技术细节

编程模型

- Single instruction, multiple threads (SIMT)

编程模型

- Single instruction, multiple threads (SIMT)

```
Kernel2D fill = [&](ImageFloat image) {
  auto coord = dispatch_id().xy();
  auto size = make_float2(dispatch_size().xy());
  auto rg = make_float2(coord) / size;
  // invoke the callable
  auto srgb = to_srgb(make_float3(rg, 1.f));
  image.write(coord, make_float4(srgb, 1.f));
};
```

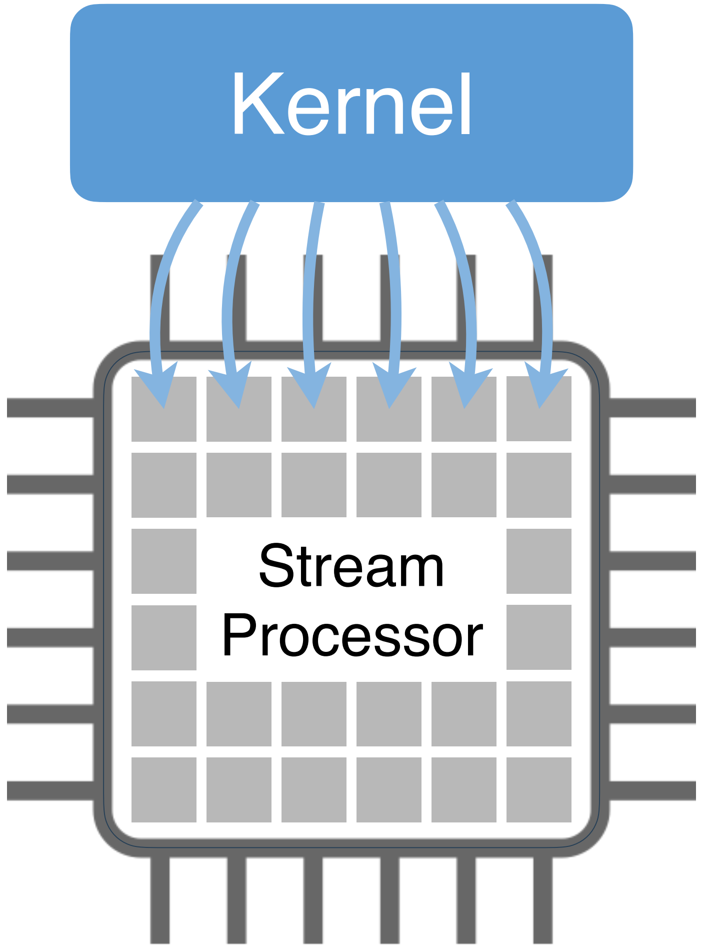
编程模型

- Single instruction, multiple threads (SIMT)

Kernel

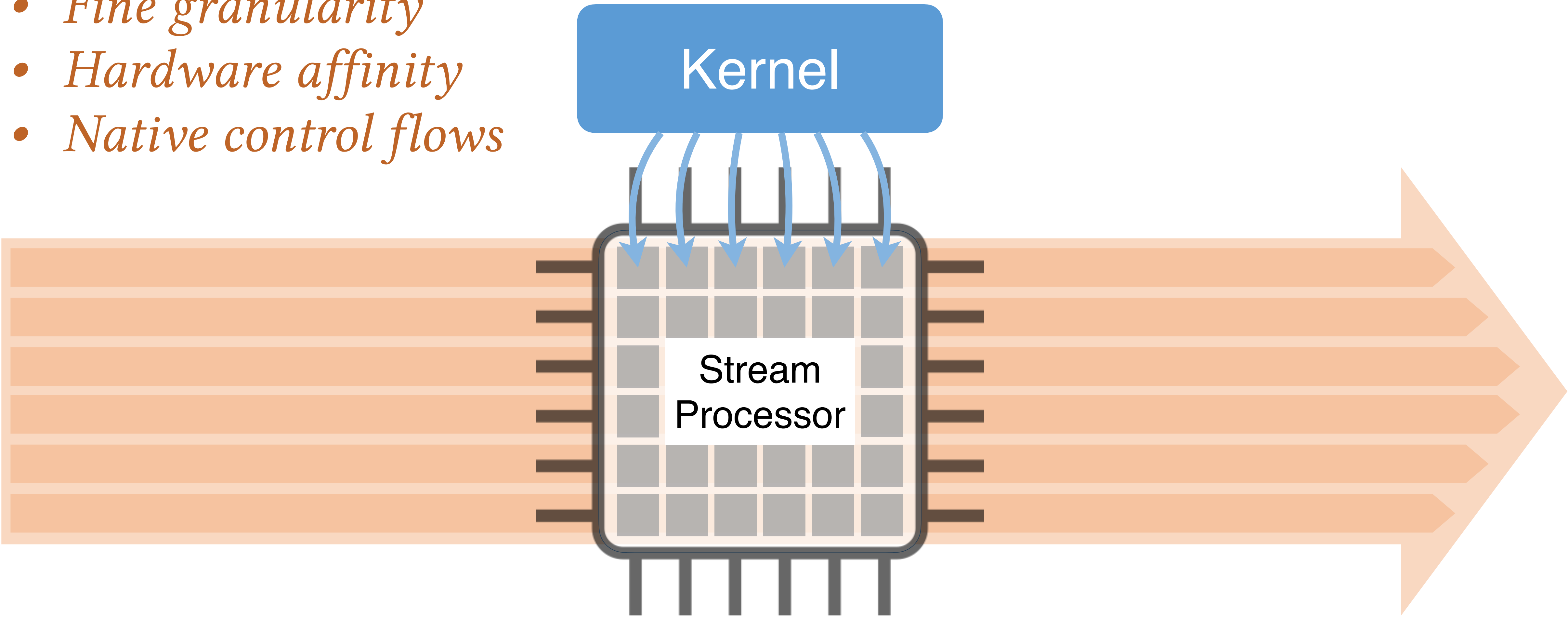
编程模型

- Single instruction, multiple threads (SIMT)



编程模型

- Single instruction, multiple threads (SIMT)
  - *Fine granularity*
  - *Hardware affinity*
  - *Native control flows*



DSL 语法

- Types

```
/* aliases for commonly used instantiations */
using Int  = Var<int>;
using Int2 = Var<int2>;
using Int3 = Var<int3>;
using Int4 = Var<int4>;
/* ... */

/* aliases for runtime resources */
using BufferInt = Var<Buffer<int>>;
using ImageInt  = Var<Image<int>>;
/* ... */
```

- Expressions and statements

```
auto a = def(0u);            // Var<uint> defined in DSL
auto b = def(1u);            // Var<uint> defined in DSL

/* operators, assignments, and type inference */
auto c = a + b;       // operator+: (uint, uint) -> uint
auto d = a < b;       // operator<: (uint, uint) -> bool
b = a - c * 3u;       // operator- and *, and assignment

/* static type check and conversion */
auto u = 1 + c;      // literal int(1) converted to uint
// float3(1.f) + u => compile-time error: float3 + uint
```

- Control flows

```
$if (cond) { /*...*/ } $else { /*...*/ };
$if (cond) { /*...*/ } $elif (cond2) { /*...*/ };
$while (cond) { /*...*/ };
$for (variable, n) { /*...*/ };
$for (variable, begin, end) { /*...*/ };
$for (variable, begin, end, step) { /*...*/ };
$loop { /*...*/ }; // infinite loop, unless $break'ed
$switch (variable) {
  $case (value) { /*...*/ };
  $default { /*...*/ };
};
$break; $continue;
```

- Built-in and custom functions

```
Callable to_srgb = [](Float3 x) {
  $if (x <= 0.00031308f) {
    x = 12.92f * x;
  } $else {
    x = 1.055f * pow(x, 1.f / 2.4f) - .055f;
  };
  return x;
};
```

  - Including ray tracing and texture sampling

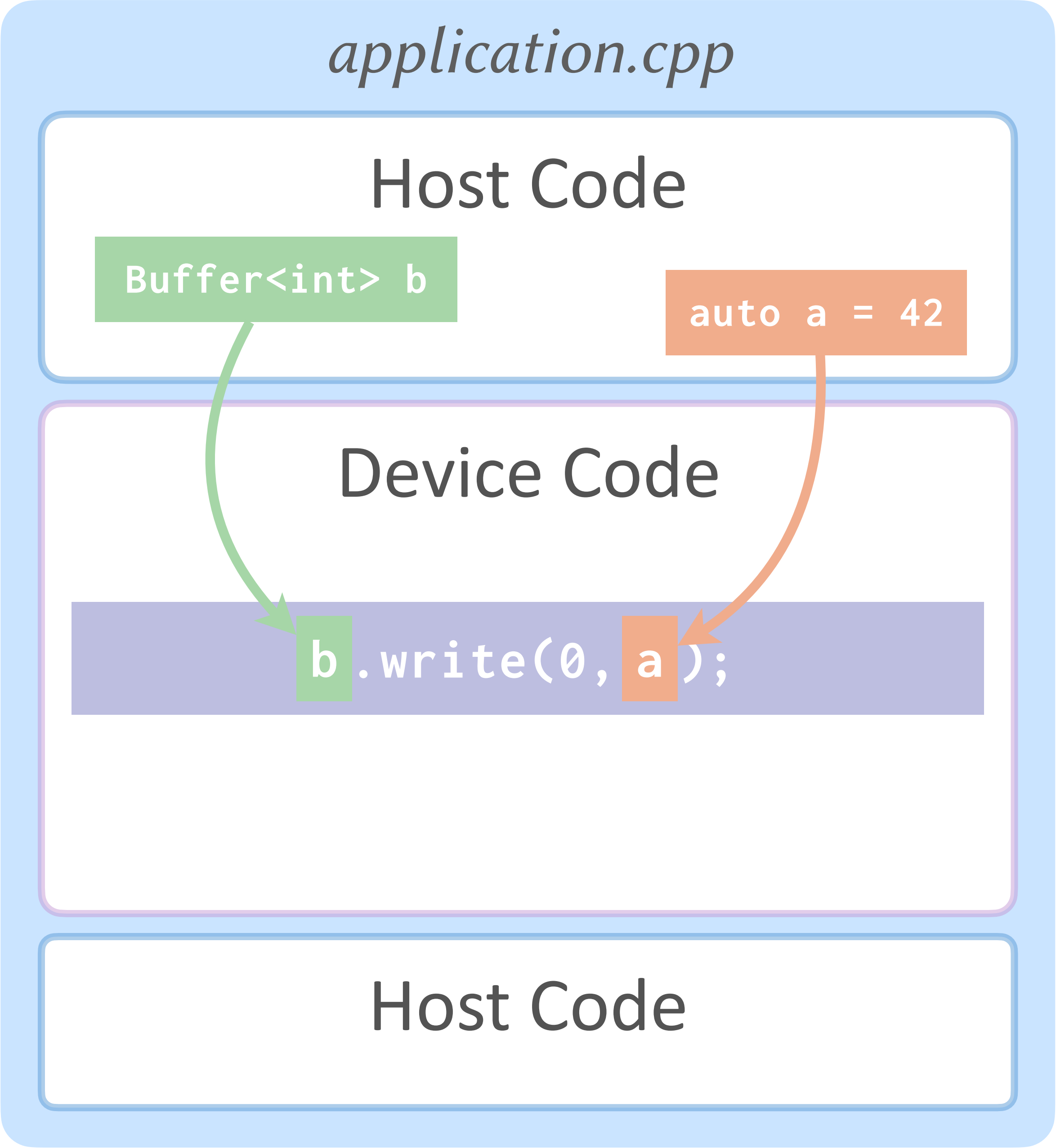
DSL 语法

- Embedded in pure C++20
  - No need for custom preprocessors or compilers
  - Static type checking and inference, templates, classes, etc.

```cpp
template<typename T1, typename T2>
inline auto cosine_sample_hemisphere(T1 u1, T2 u2) {
  auto r = sqrt(u1);
  auto phi = 2.f * pi * u2;
  auto x = r * cos(phi);
  auto y = r * sin(phi);
  auto z = sqrt(1.f - u1);
  return make_float3(x, y, z);
};
```

DSL 语法

- Embedded in pure C++20
  - No need for custom preprocessors or compilers
  - Static type checking and inference, templates…
- Easy interactions between the host and the device
  - Mix host and device code in a single file
  - Inlining host variables as DSL literals
  - Capturing resources without explicit binding



DSL 语法

- Embedded in pure C++20
  - No need for custom preprocessors or compilers
  - Static type checking and inference, templates, classes, etc.
- Easy interactions between the host and the device

```cpp
auto image = device.create_image<float>(/* ... */);
Kernel2D fill = [&] {
  auto coord = dispatch_id().xy();
  auto rg = make_float2(coord) /
            make_float2(dispatch_size().xy());
  auto srgb = to_srgb(make_float3(rg, 1.f));
  image.write(coord, make_float4(srgb, 1.f));
};
```

AST 追踪

- Abstract syntax trees are the intermediate representation
  - Traced through proxy objects

```
auto l = def<float3>();
l = dot(lumi, beta);
```

抽象语法树

- Abstract syntax trees are the intermediate representation
  - Traced through proxy objects

Proxy object

```
auto l = def<float3>();
l = dot(lumi, beta);
```

```
LOCAL l
```

抽象语法树

- Abstract syntax trees are the intermediate representation
  - Traced through proxy objects

```
auto l = def<float3>();
l = dot(lumi, beta);
```

Proxy objects

LOCAL **l**

REF **lumi**

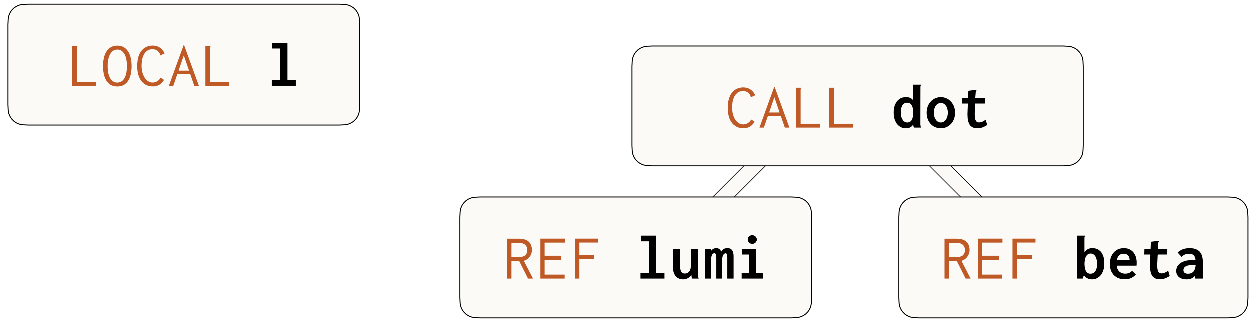
REF **beta**

抽象语法树

- Abstract syntax trees are the intermediate representation
  - Traced through proxy objects

```
auto l = def<float3>();
l = dot(lumi, beta);
```

Overloaded functions for proxy objects

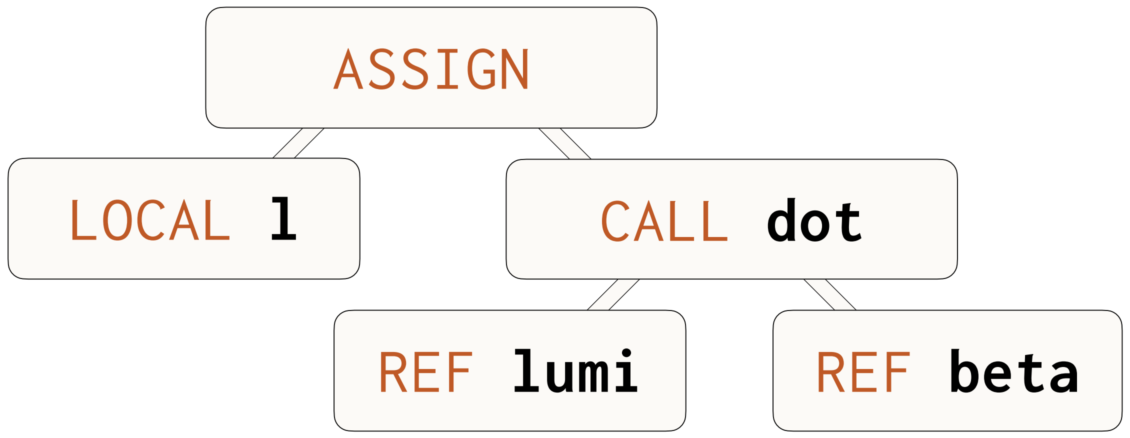


抽象语法树

- Abstract syntax trees are the intermediate representation
  - Traced through proxy objects

```
auto l = def<float3>();
l = dot(lumi, beta);
```

Overloaded assignment operator

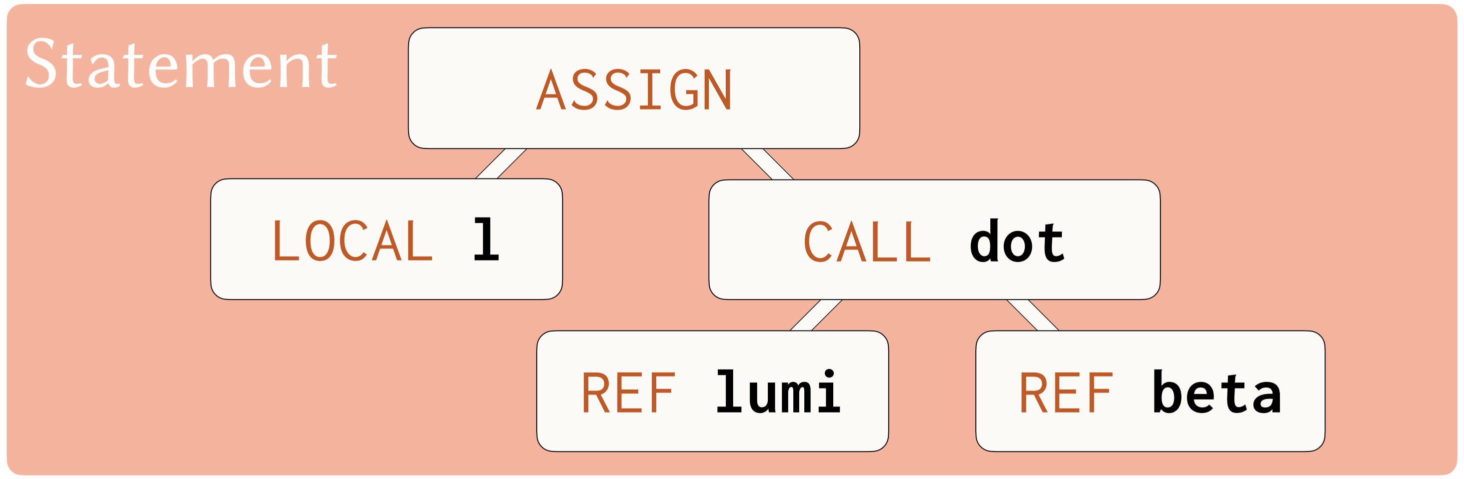


抽象语法树

- Abstract syntax trees are the intermediate representation
  - Traced through proxy objects

```
auto l = def<float3>();
l = dot(lumi, beta);
```

Overloaded assignment operator



即时代码生成与编译

即时代码生成与编译

Kernel AST

Backends

Shader Code Generation

Dynamic Compilation

即时代码生成与编译

- Dynamic multi-stage programming
  - User-controlled loop unrolling, function inlining, constant folding, dead code elimination, de-virtualization, etc.

```
inline auto tea(UInt s, UInt v0, UInt v1) {
  for (auto i = 0; i < 4; i++) { // 该循环会被展开 4 次
    s += 0x9e3779b9u;
    v0 += ((v1 << 4) + 0xa341316cu) ^ (v1 + s) ^
          ((v1 >> 5u) + 0xc8013ea4u);
    v1 += ((v0 << 4) + 0xad90777du) ^ (v0 + s) ^
          ((v0 >> 5u) + 0x7e95761eu);
  }
  return v0;
};
```

即时代码生成与编译

- Dynamic multi-stage programming
  - User-controlled loop unrolling, function inlining, constant folding, dead code elimination, de-virtualization, etc.
  - Dynamically load parts of device code from plug-ins

```
// defined in plug-in: tonemapping_aces.dll
Float3 apply(Float3 in) { /*...*/ }
```

```
// defined in plug-in: tonemapping_uncharted2.dll
Float3 apply(Float3 in) { /*...*/ }
```

```
// defined in plug-in: tonmapping_filmic.dll
Float3 apply(Float3 in) { /*...*/ }
```

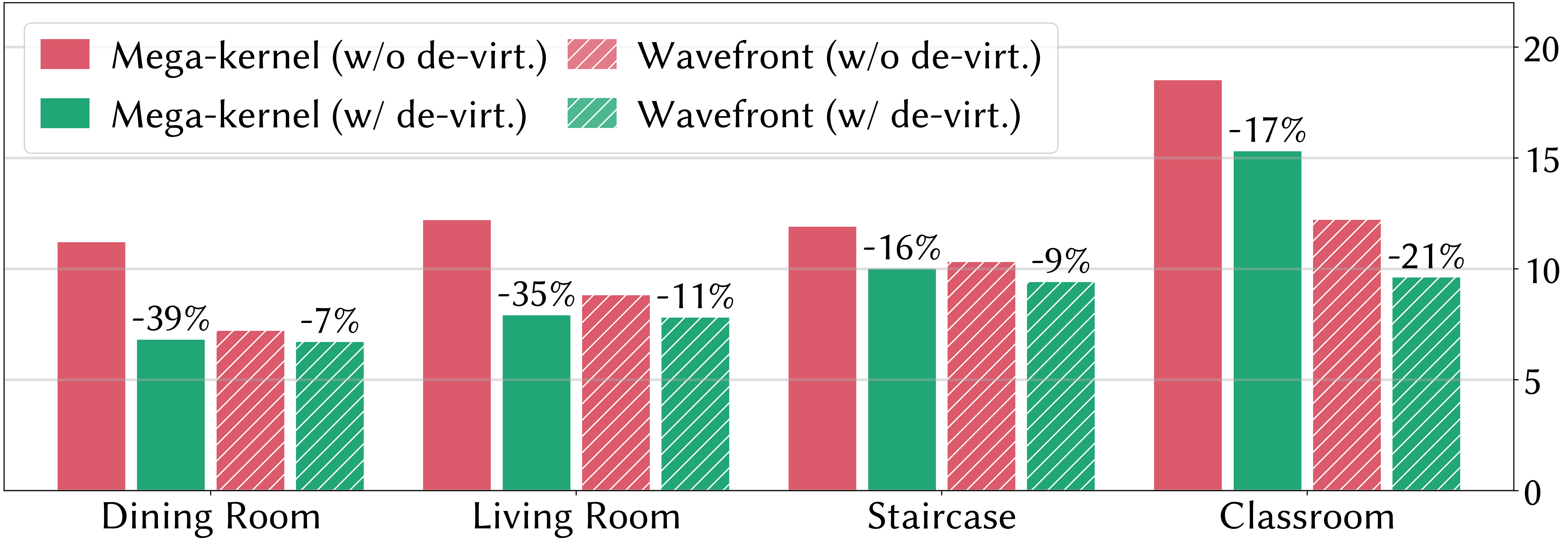
```
auto create_kernel(function<Float3(Float3)> op) {
  Kernel2D kernel = [&op](ImageFloat image) {
    auto p = dispatch_id().xy();
    auto color = image.read(p);
    // op(): a host-side dynamic call, expanding the
    // polymorphic logic into the shader, which is
    // effectively de-virtualized on the device side
    auto mapped = op(color.xyz());
    image.write(p, make_float4(mapped, color.w));
  };
  return kernel;
}

// at runtime, load a tonemapping plug-in dynamically
auto tm_plugin = load_module(/*...*/);
// now use the dynamically loaded op to create kernels
auto tm_kernel = create_kernel(tm_plugin.get("apply"));
```

```cpp
class BRDF {
  virtual Eval eval(Float3 wo, Float3 wi) = 0;
};
class Lambertian : public BRDF {
  Eval eval(Float3 wo, Float3 wi) override { /*...*/ }
};
class Microfacet : public BRDF {
  Eval eval(Float3 wo, Float3 wi) override { /*...*/ }
};
```

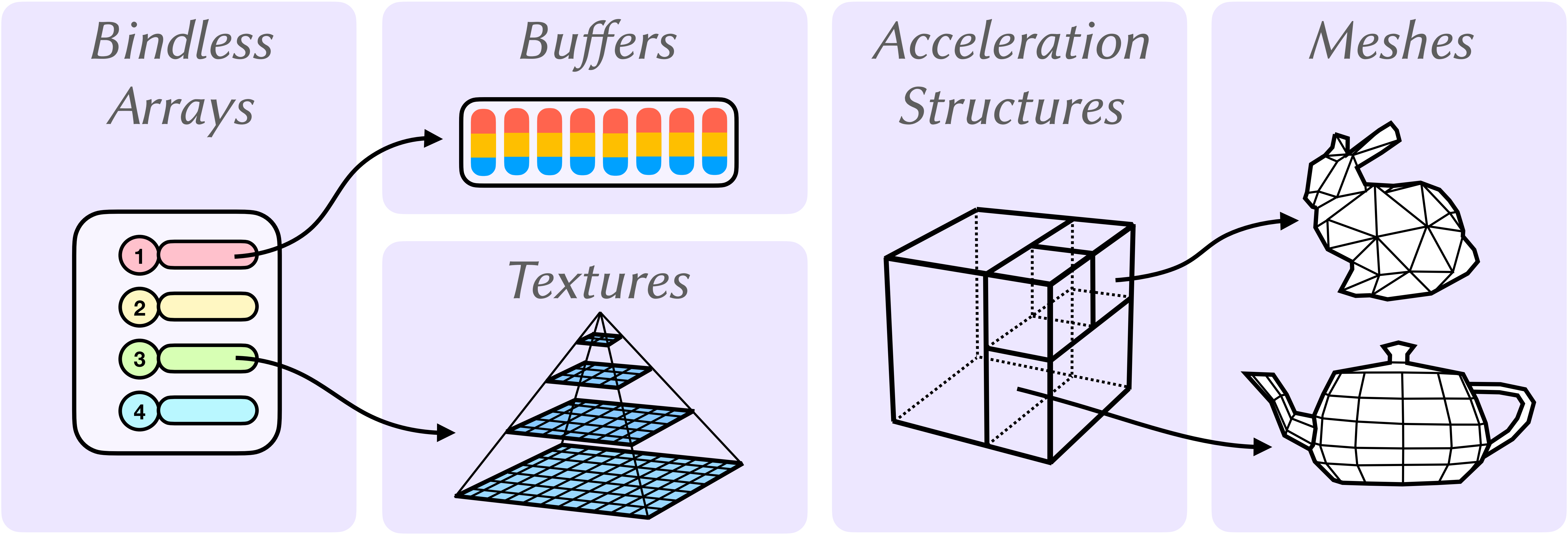
```cpp
class BRDFEvaluator {
private:
  Polymorphic<BRDF> _f;
public:
  void do_registration() {
    auto tag1 = _f.create<Lambertian>(); // tag1 == 0
    auto tag2 = _f.create<Microfacet>(); // tag2 == 1
    // register other BRDFs...
  }
  auto evaluate(Hit hit, Float3 wo, Float3 wi) {
    Eval eval;
    _f->dispatch(hit->brdf_tag(), [&](auto f) {
      eval = f->eval(wo, wi);
    });
    // equivalently expands to
    // $switch (hit->brdf_tag()) {
    //   /* calling Lambertian::eval() */
    //   $case(0) { eval = _f[0]->eval(wo, wi); };
    //   /* calling Microfacet::eval() */
    //   $case(1) { eval = _f[1]->eval(wo, wi); };
    //   ...
    // };
    return eval;
  }
};
```

即时代码生成与编译

Example: de-virtualized texture evaluation reduces rendering time up to 39%



设备资源



Bindless Arrays
1
2
3
4
Buffers
Textures
Acceleration Structures
Meshes

资源包装

- Texture copy in CUDA

```
CUDA_MEMCPY3D copy{};
copy.srcMemoryType = CU_MEMORYTYPE_DEVICE;
copy.srcDevice = /* buffer address */;
copy.srcPitch = /* texture pitch (in bytes) */;
copy.srcHeight = /* texture height (in texels) */;
copy.dstMemoryType = CU_MEMORYTYPE_ARRAY;
copy.dstArray = /* texture handle */;
copy.WidthInBytes = /* texture pitch (in bytes) */;
copy.Height = /* texture height (in texels) */;
copy.Depth = /* texture depth (in texels) */;
cuMemcpy3DAsync(&copy, /* stream handle */);
```

- Texture copy in LuisaRender

```
stream << texture.copy_from(buffer);
```

资源包装

- Texture copy in DirectX

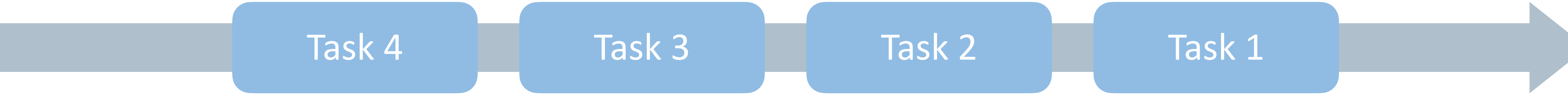
```
// with native DirectX
uint32_t width = texture->Width();
uint32_t height = texture->Height();
uint32_t depth = texture->Depth();
auto c = cb->cmdList.Get();
D3D12_TEXTURE_COPY_LOCATION src;
src.pResource = buffer.buffer->GetResource();
src.Type =
  D3D12_TEXTURE_COPY_TYPE_PLACED_FOOTPRINT;
src.PlacedFootprint.Offset = buffer.offset;
src.PlacedFootprint.Footprint = {
  static_cast<DXGI_FORMAT>(texture->Format()),
  width,
  height,
  depth,
  ((width / texture->PixelSize()) +
  (D3D12_CONSTANT_BUFFER_DATA_PLACEMENT_ALIGNMENT-1)) &
  ~(D3D12_CONSTANT_BUFFER_DATA_PLACEMENT_ALIGNMENT-1),
};
D3D12_TEXTURE_COPY_LOCATION dst;
dst.Type =
  D3D12_TEXTURE_COPY_TYPE_SUBRESOURCE_INDEX;
dst.SubresourceIndex = 0;
dst.pResource = texture->GetResource();
c->CopyTextureRegion(&dst, 0, 0, 0,
                     &src, nullptr);
```

- Texture copy in LuisaRender

```
stream << texture.copy_from(buffer);
```

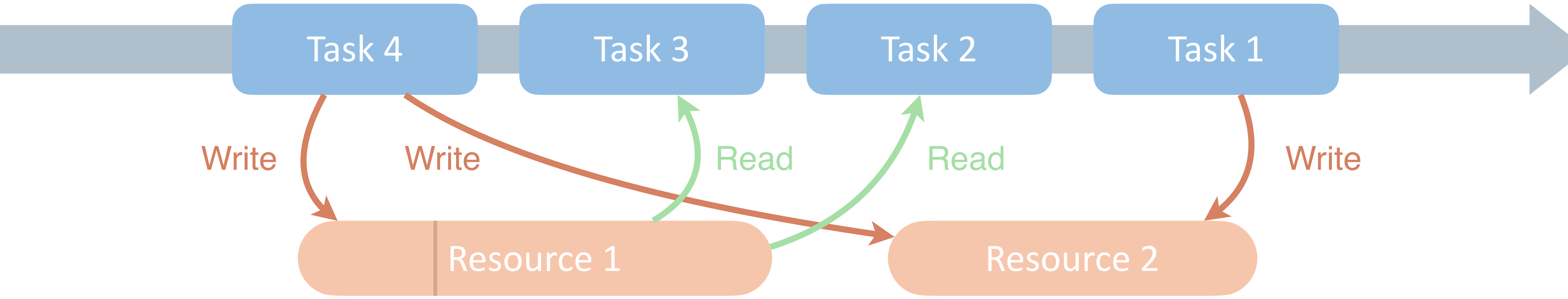
命令调度

- Usage information in ASTs helps re-schedule commands



命令调度

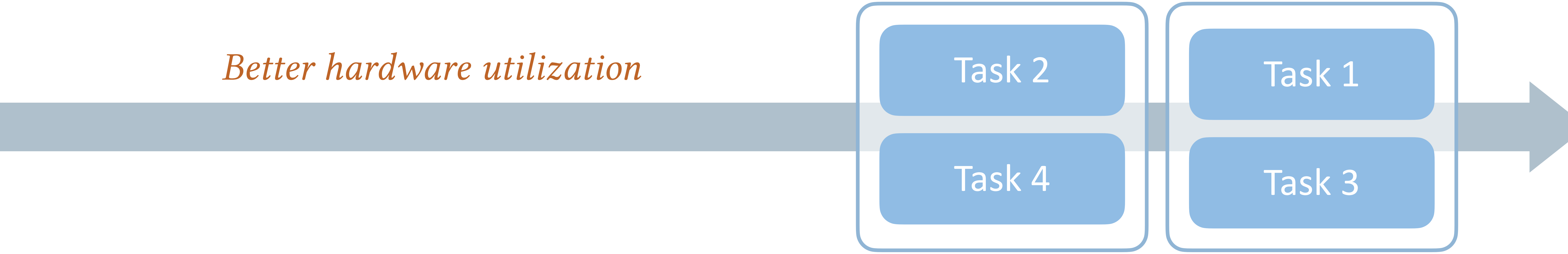
- Usage information in ASTs helps re-schedule commands



命令调度

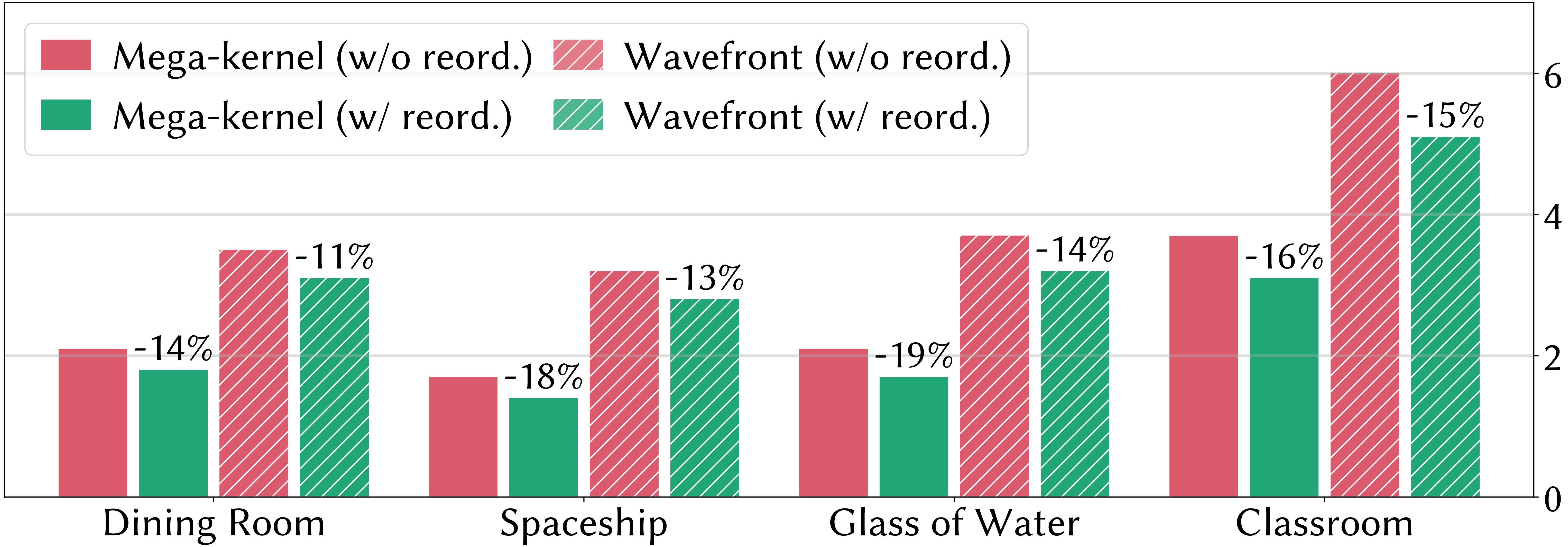
- Usage information in ASTs helps re-schedule commands

Better hardware utilization



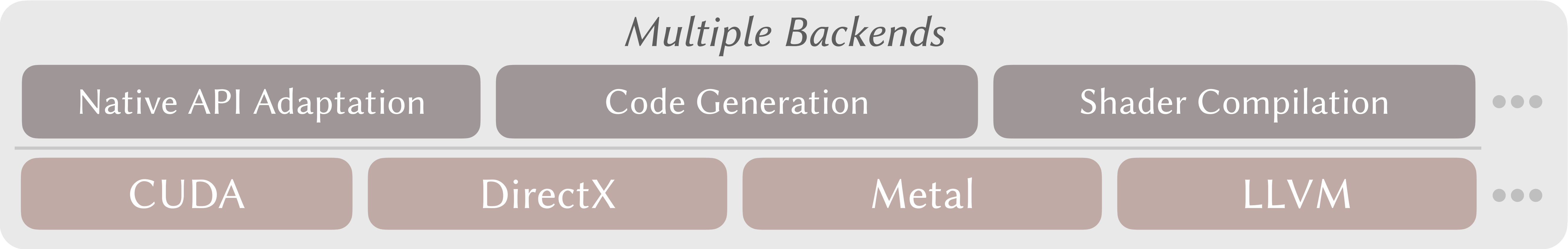
命令调度

Example: reordering wavefront path tracing commands reduces rendering time up to 19%



后端实现

- Various parallel computing backends are supported
  - CPU: LLVM (scalar)
  - GPU: DirectX, CUDA, Metal
  - Optimized and tuned for the underlying platform APIs
- Abstraction layers ease addition of new backend
  - Vulkan & Remote (WIP)

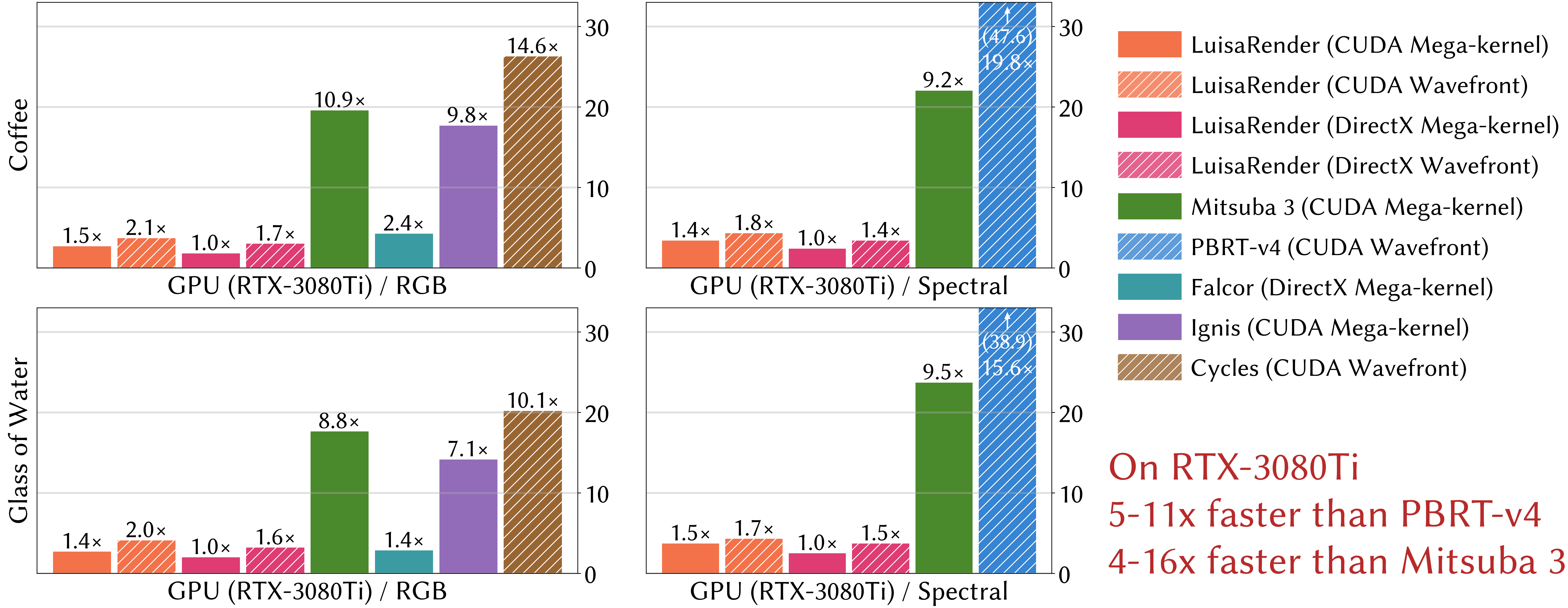


应用

蒙特卡洛渲染系统 LuisaRender

[Rendering Resources, Bitterli 2016]

蒙特卡洛渲染系统 LuisaRender



On RTX-3080Ti
5-11x faster than PBRT-v4
4-16x faster than Mitsuba 3

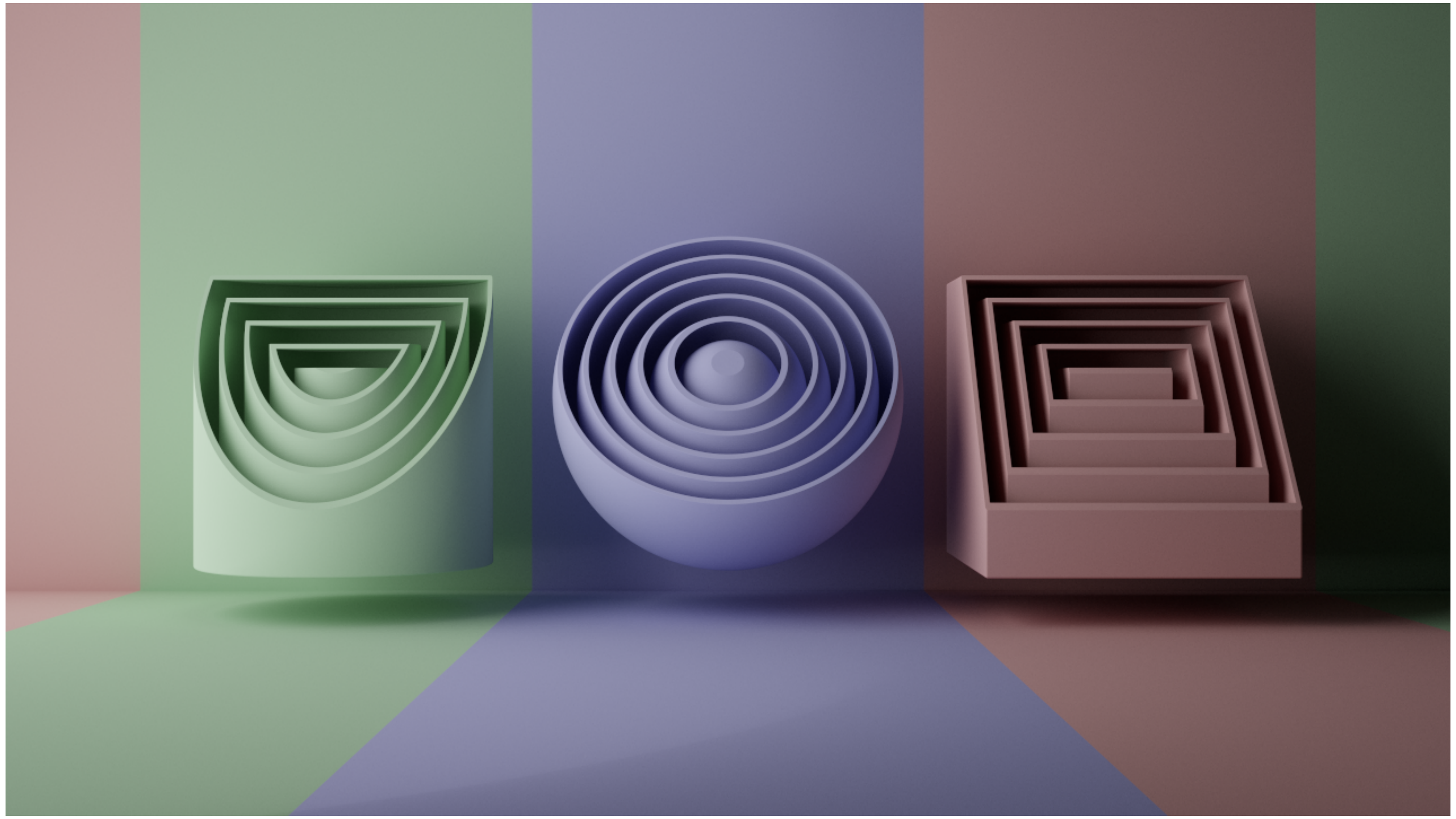
可微分渲染

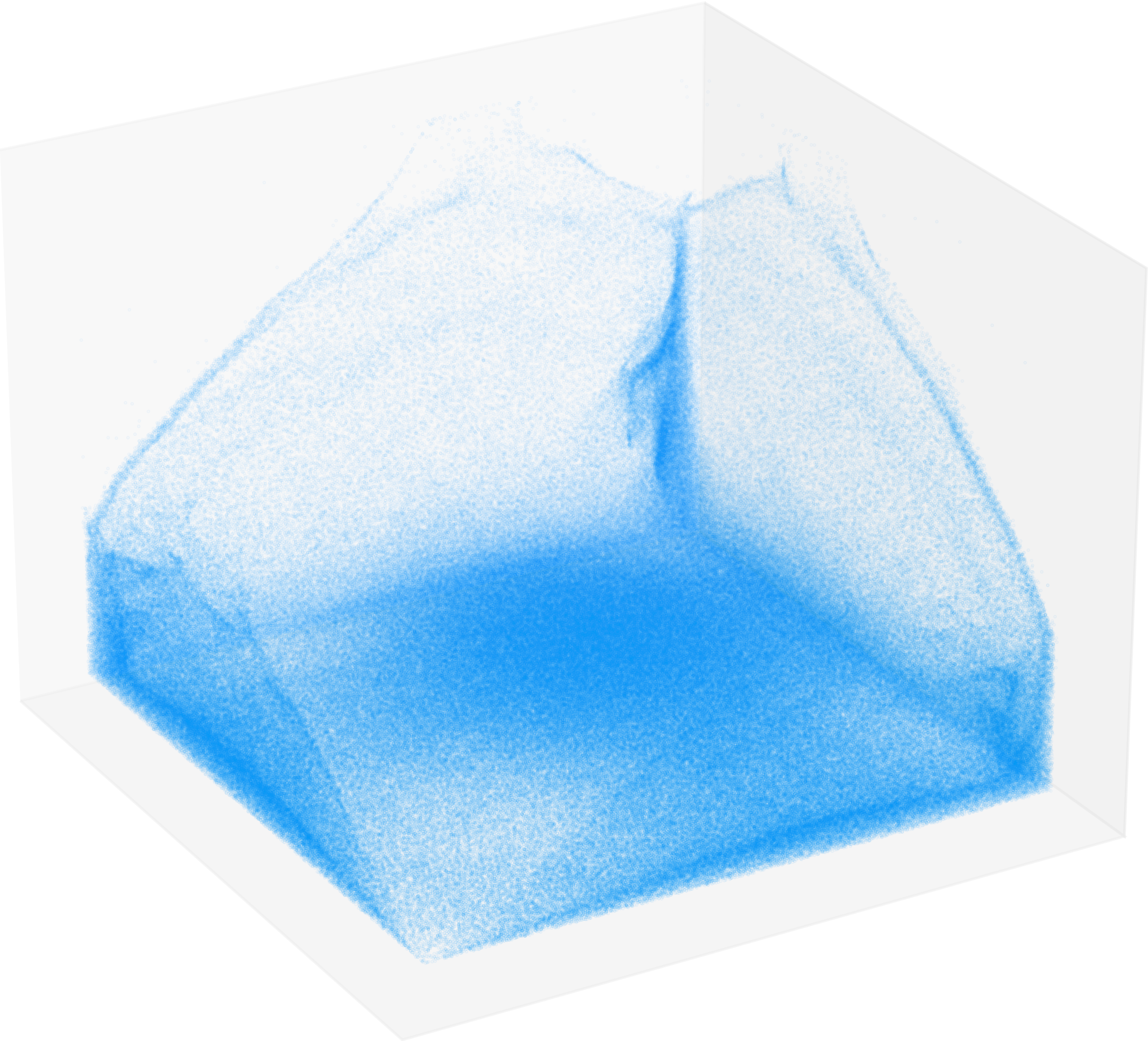
- Path Replay Backpropagation [Vicini et al. 2021]

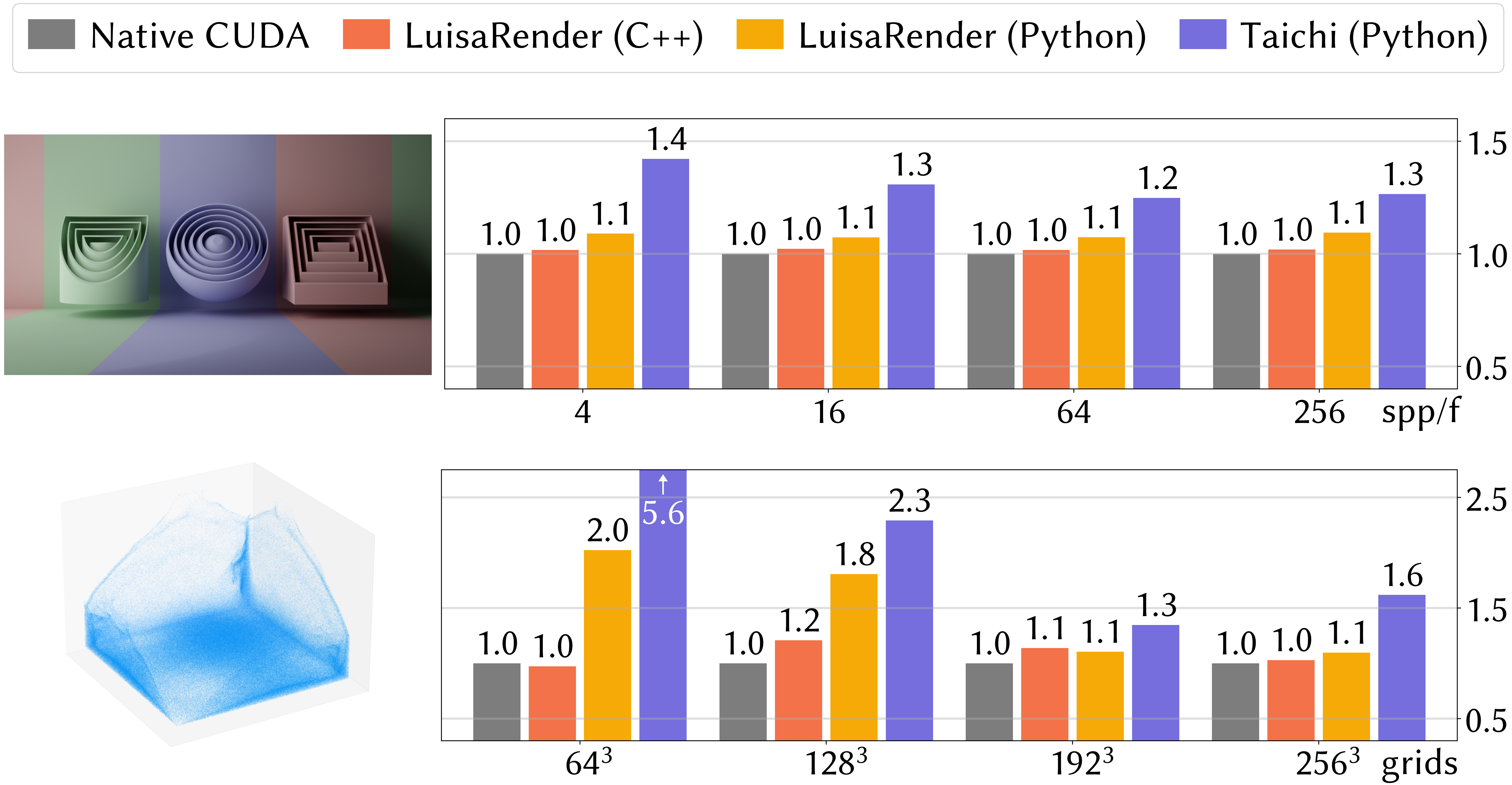
LUISA

luisa-render.com

通用计算（来自 Taichi Examples）






Native CUDA
LuisaRender (C++)
LuisaRender (Python)
Taichi (Python)
1.0 1.0 1.1 1.4
1.0 1.0 1.1 1.3
1.0 1.0 1.1 1.2
1.0 1.0 1.1 1.3
1.5
1.0
0.5
4
16
64
256
spp/f
1.0 1.0 2.0 5.6
1.0 1.2 1.8 2.3
1.0 1.1 1.1 1.3
1.0 1.0 1.1 1.6
2.5
1.5
0.5
64^3
128^3
192^3
256^3
grids

未来工作

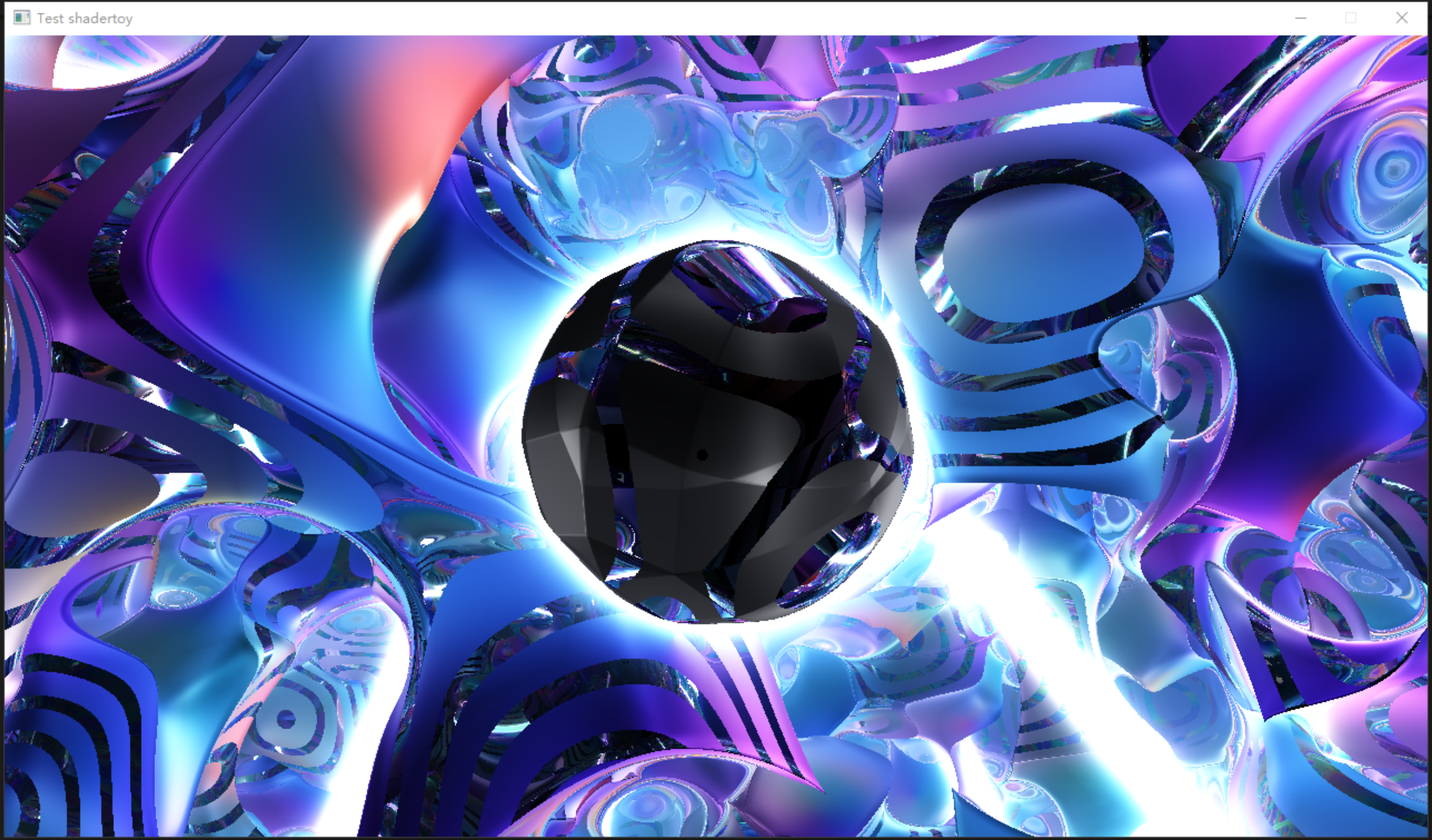
Python 前端

- 强调易用性与快速原型设计
  - 动态语言，无需编译
  - 完整支持各种设备资源：Buffers, Textures, Acceleration Structures, …
  - 重计算任务上性能与 C++ 一致
- 与 C++ 一致的 SIMT 编程模型和流式执行模型，可控性高
- 翻译 Python 原生 AST 至 LC AST
  - 可作为脚本为核心模块扩展功能

test-shader-visuals-present.py

src > tests > python > test-shader-visuals-present.py > render_kernel

```python
    fact = length(sin(r * (ite(dl, 4., 3.)))
                  * .5 + .5) / sqrt(3.) * .7 + .3
    matcol = lerp(float3(.9, .4, .3), float3(.3, .4, .8),
                  smoothstep(-1., 1., sin(data.d1 * 5. + time * 2.)))
    matcol = lerp(matcol, float3(.5, .4, 1.), smoothstep(
        0., 1., sin(data.d2 * 5. + time * 2.)))
    matcol = ite(dl, lerp(1., matcol, .1) * .2 + .1, matcol)
    col = matcol * fact * ss + pow(fact, 10.)
    col = ite(lz, float3(4.), col)
    fragColor = col * atten + glo * glo + fogcol * glo
    fragColor = lerp(fragColor, fogcol, fog)
    fragColor = ite(dl, fragColor, abs(
        erot(fragColor, normalize(sin(p * 2.)), .2 * (1. - fog))))
    fragColor = ite(trg or dl, fragColor, fragColor +
                    dlglo * dlglo * .1 * float3(.4, .6, .9))
    fragColor = sqrt(fragColor)
    color = smoothstep(0., 1.2, fragColor)
    image.write(dispatch_id().xy, float4(pow(color, 2.2), 1.))


@func
def clear_kernel(image):
    coord = dispatch_id().xy
    image.write(coord, float4(0.3, 0.4, 0.5, 1.))


res = 1280, 720
image = Texture2D(*res, 4, float, storage="BYTE")
gui = GUI("Test shadertoy", res)
clear_kernel(image, dispatch_size=(*res, 1))
time = 0.0
while gui.running():
    gui.set_image(image)
    render_kernel(image, time, dispatch_size=(*res, 1))
    # use seconds
    time += gui.show() / 1000.0
synchronize()
```



Rust 前端

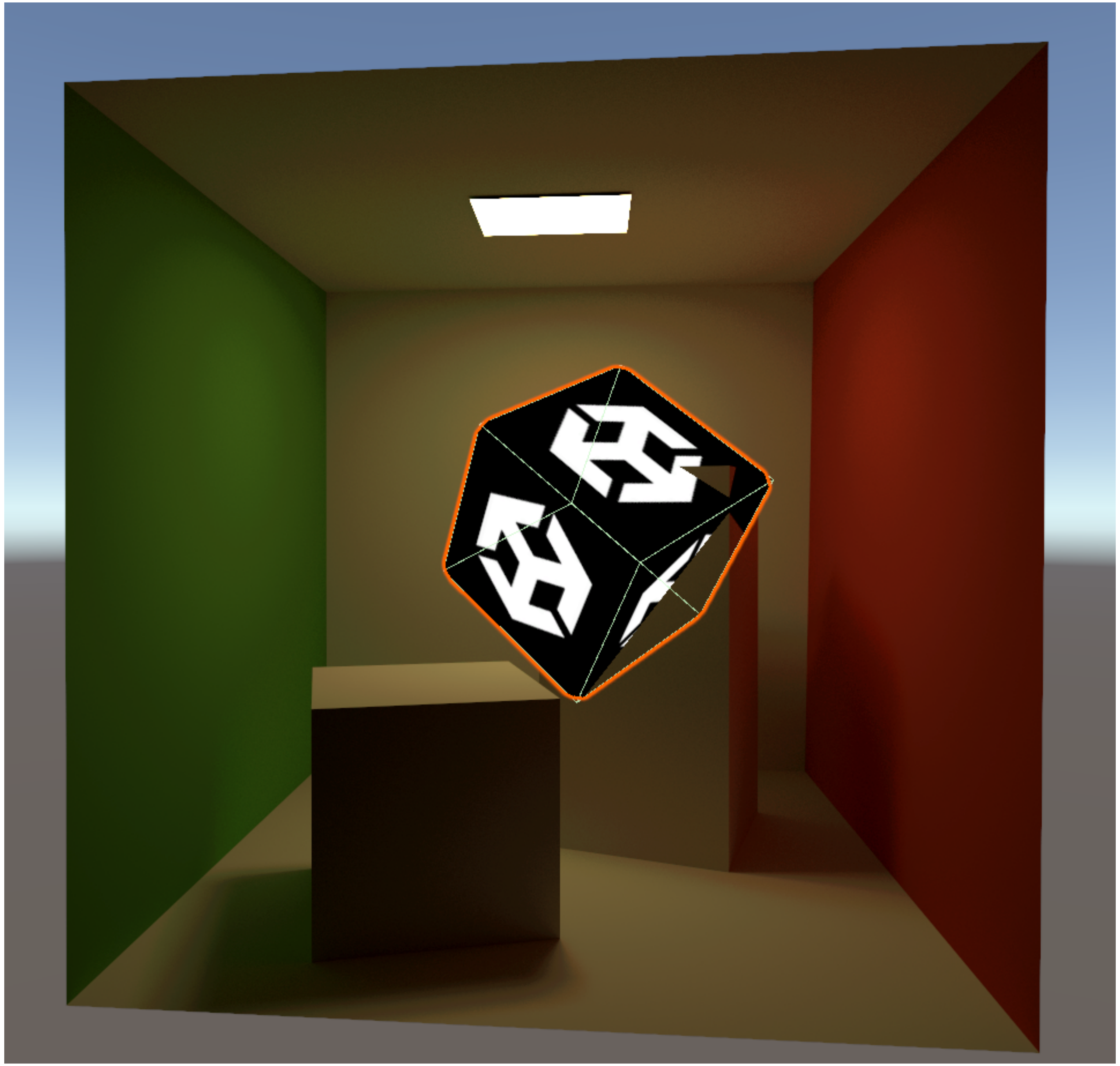
- 使用**过程宏**实现语法
  - if/while/switch 等控制流
  - 动态多态、资源捕获等
- （几乎）**安全**的 Rust 风格 API
- IR 与**自动微分**

```rust
if_!(cond, { /* then */});
if_!(cond, { /* then */}, { /* else */});
if_!(cond, { value_a }, { value_b })
while_!(cond, { /* body */});
break_();
continue_();
let (x,y) = switch::<(Expr<i32>, Expr<f32>)>(value)
    .case(1, || { ... })
    .case(2, || { ... })
    .default(|| { ... })
    .finish();
```

```rust
use luisa::prelude::*;
use luisa_compute as luisa;

fn main() {
    init();
    let device = create_cpu_device().unwrap();
    let x = device.create_buffer::<f32>(1024).unwrap();
    let y = device.create_buffer::<f32>(1024).unwrap();
    let z = device.create_buffer::<f32>(1024).unwrap();
    x.view(..).fill_fn(|i| i as f32);
    y.view(..).fill_fn(|i| 1000.0 * i as f32);
    let kernel = device
        .create_kernel::<(Buffer<f32>,)>(&|buf_z| {
            // z is pass by arg
            let buf_x = x.var(); // x and y are captured
            let buf_y = y.var();
            let tid = dispatch_id().x();
            let x = buf_x.read(tid);
            let y = buf_y.read(tid);
            buf_z.write(tid, x + y);
        })
        .unwrap();
    kernel.dispatch([1024, 1, 1], &z).unwrap();
    let z_data = z.view(..).copy_to_vec();
    println!("{:?}", &z_data[0..16]);
}
```

扩展与集成

- 提供 DeviceExtension 接口
  - 方便对框架做自定义扩展，如贴图压缩、毛发求交等
  - 支持资源导入/导出，实现与其他系统的交互



总结

总结

- 我们实现了一个适用于渲染等场景的并行计算框架 LuisaCompute
  - 提供嵌入于 C++ 和 Python 等前端的高表达力 DSL
  - 可利用最新光线追踪硬件的统一运行时抽象层
  - 在此框架基础上实现了高性能渲染系统 LuisaRender
- 在未来…
  - 支持更多的前端（如 Rust 前端正在开发中）、后端
  - 更方便灵活的编程模式与更好的性能
  - 更多功能与应用场景（如自动微分、光栅化渲染功能正在开发中）

Thanks

luisa-render.com

github.com/LuisaGroup