Supporting Unified Shader Specialization by Co-opting C++ Features

Kerry A. Seitz, Jr.,*‡ Theresa Foley,†
Serban D. Porumbescu,* and John D. Owens*

*University of California, Davis
†NVIDIA
‡Now at Wētā Digital x Unity Technologies

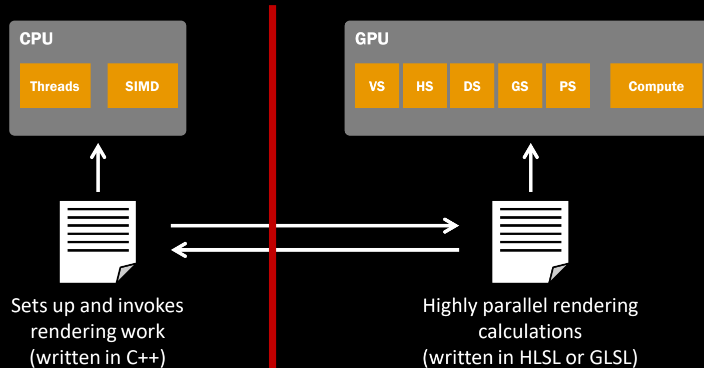
HPG 2022

Hi, my name is Kerry Seitz, and I'll be presenting our work, Supporting Unified Shader Specialization by Co-opting C++ Features, on behalf of my coauthors Theresa Foley, Serban Porumbescu, and John Owens.

The overarching motivation of this work is to think about the problem of the strict divide between host (CPU) code and GPU code that we have in real-time graphics programming.

Strict divide between host (CPU) and GPU code

HPG 2022



2

In real-time rendering, we generally have some code running on GPUs performs highly parallel rendering calculations written in a special-purpose shading language like HLSL or GLSL.

And there's corresponding host code sets up and invokes the rendering work, including choosing which bits of GPU code to invoke and passing parameters to the GPU, and other communication and coordination between host and GPU code. This is written in a general-purpose systems language like C++.

And in the real-time graphics APIs, there's a really strict divide between these two portions of code, which results in a lot of pain points, like subtle bugs if the two halves of code are not kept in sync with each other.

Instead, what I think we really want is a "unified" programming model for real-time graphics.

Unified Programming Environments

HPG 2022

- Host code and GPU code are in the:
  - Same programming language
  - Same file
  - Same lexical scope
- Benefits of Unified Programming
  - Host/GPU can share types and functions
  - Guaranteed compatibility between host/GPU
    - No magic number "bindings points," just proper variables
    - Type layouts are the same
- Unified Systems for GPU Compute
  - CUDA and SYCL

3

In these so call "unified" environments, host and GPU code can be in the same programming language, same file, and same lexical scope.

Unified systems have inherent benefits, such as sharing types and functions between host and GPU code and compatibility guarantees. Host and GPU code can communicate using proper variables, not magic number binding points, and type layouts are guaranteed to be the same.

Unfortunately, none of the real-time graphics APIs provide unified environments, but in the GPU compute space, unified system are the standard, like CUDA and SYCL.

So you may be wondering … Can't real-time graphics just use CUDA or SYCL?

Can’t graphics just use CUDA or SYLC?

Yes.

HPG 2022

4

Yes.

Thank you!

Kerry A. Seitz, Jr., Theresa Foley, Serban D. Porumbescu, and John D. Owens
kaseitz@ucdavis.edu

Source Code:
https://github.com/owensgroup/UnifiedShaderSpecialization

HPG 2022

Thank you for coming to my talk, you've been a great audience.

Can't graphics just use CUDA or SYLC?

No.

Existing popular unified systems lack adequate support for GPU shader code specialization

6

Ok, not really.

There are multiple reasons why graphics can't just use one of these existing systems.

In this work, we focused one in particular: the lack of adequate support for GPU shader code specialization.

So what is shader specialization. Let's look at an example.

Shader Specialization

HPG 2022

- Three materials
  - Standard Material
  - Skin Material
  - Cloth Material
- Lots of other complex code to run along with material-specific code
  - Need to slot in the material-specific code

7

Let’s say that we have a scene with three different materials that need to use different pieces of code to calculate their final rendered color. We have a standard material, that we use for most object. Special materials for skin and cloth.

Along with the material-specific shader code, we also have a bunch of other complex code that we need to run in the shader in order to calculate the pixel’s color. (Lots of handwaving here about what that is.) So we need to be able to slot in the material-specific code, at the appropriate place in the shader.

Shader Specialization

HPG 2022

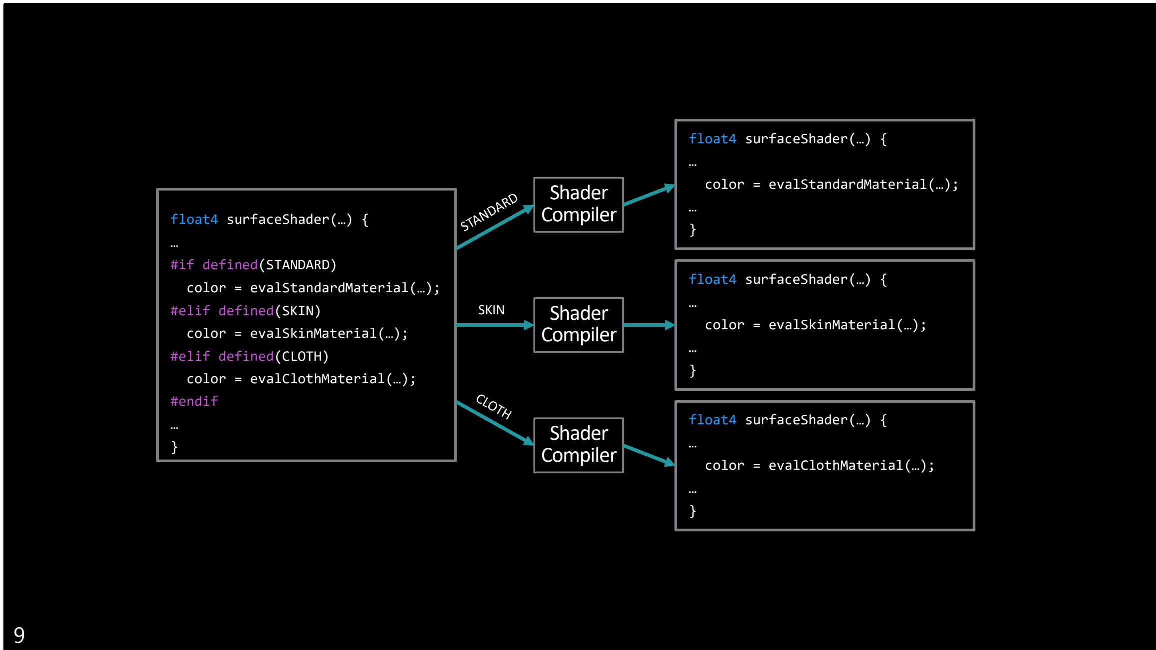
Specialized Material Shader

Statically specialized variants generated at shader compile time

```
float4 surfaceShader(…) {
…
#if defined(STANDARD)
  color = evalStandardMaterial(…);
#elif defined(SKIN)
  color = evalSkinMaterial(…);
#elif defined(CLOTH)
  color = evalClothMaterial(…);
#endif
…
}
```

8

To get the best performance, what shader programmers typically do is express these specialization options as static, compile-time branches. These will be evaluated when we compile the shader, depending on which material type we've defined during compilation.



We can generate a specialized shader variant just for the Standard material resulting in a compiled shader that just contains the Standard material code, stripping out the code for the other two.

We can do the same for the Skin and Cloth material, resulting in three total compiled shader variants, each specialized for exactly one material.

So when we're rendering an object that uses the standard material, we don't pay any of the costs associated other materials, such as shared local memory usage and higher register pressures.

Shader Specialization: Host vs. GPU

Host code to invoke GPU code

```
void (auto shape : allShapes) {
  if(shape.materialType == mat::STANDARD)
    invokeStandardVariant();
  else if(shape.materialType == mat::SKIN)
    invokeSkinVariant();
  else if(shape.materialType == mat::CLOTH)
    invokeClothVariant();
}
```

Compile GPU Shader Variants

Standard Variant

```
float4 surfaceShader(…) {
…
  color = evalStandardMaterial(…);
…
}
```

Skin Variant

```
float4 surfaceShader(…) {
…
  color = evalSkinMaterial(…);
…
}
```

Cloth Variant

```
float4 surfaceShader(…) {
…
  color = evalClothMaterial(…);
…
}
```

HPG 2022

10

Then, when shading an object, we can use the shader variant specific to the material of that object.

On the left the corresponding host code that chooses which variant to invoke based on the runtime information in the scene being rendered.

Shader Specialization: Host vs. GPU

HPG 2022

Host code to invoke GPU code

```
void (auto shape : allShapes) {
  if(shape.materialType == mat::STANDARD)
    invokeStandar Variant();
  else if(shape.ma   ialType == mat::SKIN)
    invokeSkinVarian
  else if(shape.
    invokeClothV
}
```

Runtime Parameters
(based on shapes in the scene)

Specialized Material Shader

```
float4 surfaceShader(…) {
…
#if defined(STANDARD)
  color = eval  andardMaterial(…);
#elif defi      IN)
                  erial(…);

  color = evalClothMaterial(…);
#endif
…
}
```

Compile-time Parameters
(based on #define when compiled)

11

And if we compare the host code to the GPU shader code from a few slides ago, notice that the GPU code is using compile-time parameters to generate different variants.

But the host really needs to use runtime versions of those same parameters, since it doesn't know what material types are used in the scene until runtime.

Key Challenge of Unified Shader Specialization

- Specialization parameters are
  - Compile-time parameters for GPU code
  - Runtime parameters for host code
- Workarounds in non-unified systems
  - #define in GPU code
  - Runtime parameter in host code
- Fundamental problem in unified systems
  - Same parameter must serve both a compile-time and runtime role

HPG 2022

12

And that’s a key challenge with supporting unified shader specialization.

Specialization parameters are compile-time parameters in GPU code, but runtime parameters in host code.

In a non-unified world, we can work around this as show in the previous example.

But in a unified system, where we want a singular definition for these parameters, that same parameter now need to serve both a compile-time and a runtime purpose.

HPG 2022

Related Work

- Unified Shader Programming
  - BraidGL [Sampson et al. 2017]
    - Uses BraidGL's *static staging* feature to express shader code and specializations
  - Selos [Seitz et al. 2019]
    - Relies on *staged metaprogramming*, a key set of language features available in Lua-Terra [DeVito et al. 2013]
- Neither BraidGL nor Lua-Terra are widely used in real-time graphics
- Static staging and staged metaprogramming aren't available in the widely used languages (e.g., C++)

13

Some prior works have explored unified shader programming, including shader specialization.

BraidGL utilizes a programming language feature called "static staging" to express shader code and specializations.

Also, our prior work, Selos, uses a key set of language features called "staged metaprogramming" that are available in the Lua-Terra language to enable a unified shader system.

However, neither BraidGL nor Lua-Terra are widely used languages in real-time graphics.

And more importantly, these key features that they rely on aren't available in the widely used language either.

So what we wanted to ask is: Can we bring the benefits of unified programming to today's shader programming systems in the near term? Can we meet these existing systems where they are instead of relying on features they don't have access to?

HPG 2022

Goals and Constraints

- Use a language widely used in real-time graphics
  - And allow programmers to write code that looks and feels familiar in that language
  - Minimize changes to the language as much as possible
- Equivalent performance compared to current implementations
- Ease of adoption / integration into real-time graphics applications
- First-class support for composition and specialization of (GPU) shader code

14

So to that end, we have some goals and constraints that guide our efforts.

We of course want to use a language that's widely used in real-time graphics. But more than that, we want to allow programmers to write code that really looks and feels familiar in that language. And so, we need to minimize any changes as much as possible.

Performance is obviously important in real-time graphics.

In contrast to many prior works, we want to make sure that our approach and implementation strategies are ones that modern applications could adopt in the near.

And finally, as mentioned earlier, we think these systems need to have first-class support for shader composition and specialization in order to handle the compile-time vs. runtime dichotomy of specialization.

Key Insight

Unified shader programming can be integrated into an existing, widely used programming language in real-time graphics by co-opting existing language features and implementing them with alternate semantics that are better suited to the domain of real-time graphics and its corresponding hardware.

15

Which bring me to the key insight of our work, which is that we can create such a unified system in an existing, widely used language by co-opting existing features of the language and implementing them with alternate semantics that are better suited to the domain of real-time graphics.

Co-opting existing features allows us to repurpose language features that are unsuitable for GPU code to instead express GPU- and graphics-specific semantics and optimizations. Also, by reusing existing features, we don't have to add new language features and then worry about how those features interact with the rest of the language.

To demonstrate this, we created a unified shader system integrated into an existing game engine, specifically Unreal Engine 4.

Shader Programming in Unreal Engine 4

HPG 2022

Unreal Engine 4 (UE4)

[Epic Games, Inc. 2015]

HPG 2022

- Popular, widely used game engine
- GPU shader code in HLSL
- Corresponding C++ host class for each shader entry point

17

UE4 is a popular, widely used game engine.

In UE4, users can write GPU shader code in HLSL. And for each HLSL entry point, users must also write a corresponding C++ host class to manage the host-side aspects of shader set up and invocation.

Let’s look at an example.

Example Shader in Unreal Engine 4

HPG 2022

HLSL shader code (GPU)

```
int MyStruct_TileSizeX;
int MyStruct_TileSizeY;
int MyStruct_NumTilesX;
int MyStruct_NumTilesY;

Texture2D ColorTexture;
SamplerState ColorSampler;

uint GetIterationCount() {
#if QUALITY == LOW
  return 2;
#elif QUALITY == MEDIUM
  return 4;
#elif QUALITY == HIGH
  return 8;
#else
  return 16;
}

[numthreads(THREADGROUP_SIZEX, THREADGROUP_SIZEY, 1)]
void MainCS(uint2 DispatchThreadID : SV_DispatchThreadID) {
  const uint IterationCount = GetIterationCount();
  ...
}
```

C++ shader code (host)

```
class MyShaderCS : public FGlobalShader {
public:
  DECLARE_SHADER_TYPE(MyShaderCS, Global);

  BEGIN_SHADER_PARAMETER_STRUCT(FParameters, )
    SHADER_PARAMETER_STRUCT(MyStructType, MyStruct)
    SHADER_PARAMETER_RDG_TEXTURE(Texture2D, ColorTexture)
    SHADER_PARAMETER_SAMPLER(SamplerState, ColorSampler)
  END_SHADER_PARAMETER_STRUCT()

  class QualityDimension :
    SHADER_PERMUTATION_ENUM_CLASS("QUALITY", QualityEnumType);
  class ThreadgroupSizeXDimension :
    SHADER_PERMUTATION_SPARSE_INT("THREADGROUP_SIZEX", 8, 16);
  class ThreadgroupSizeYDimension :
    SHADER_PERMUTATION_SPARSE_INT("THREADGROUP_SIZEY", 8, 16);

  using FPermutationDomain = TShaderPermutationDomain<
    QualityDimension,
    ThreadgroupSizeXDimension,
    ThreadgroupSizeYDimension>;
};
IMPLEMENT_GLOBAL_SHADER(MyShaderCS, "/path/to/HLSL/file.usf",
  "MainCS", SF_Compute);
```

18

On the left, we have GPU shader code, written HLSL.

And on the right, we have the corresponding host shader code, written in C++.

I’m not going to talk about the code in detail, but I’ll just highlight a few key points.

Repeated declarations for uniform/varying parameters

HPG 2022

HLSL shader code (GPU)

```
int MyStruct_TileSizeX;
int MyStruct_TileSizeY;
int MyStruct_NumTilesX;
int MyStruct_NumTilesY;

Texture2D ColorTexture;
SamplerState ColorSampler;

uint GetIterationCount() {
#if QUALITY == LOW
  return 2;
#elif QUALITY == MEDIUM
  return 4;
#elif QUALITY == HIGH
  return 8;
#else
  return 16;
}

[numthreads(THREADGROUP_SIZEX, THREADGROUP_SIZEY, 1)]
void MainCS(uint2 DispatchThreadID : SV_DispatchThreadID) {
  const uint IterationCount = GetIterationCount();
  ...
}
```

Uniform Parameters

Varying Parameters

C++ shader code (host)

```
class MyShaderCS : public FGlobalShader {
public:
  DECLARE_SHADER_TYPE(MyShaderCS, Global);

  BEGIN_SHADER_PARAMETER_STRUCT(FParameters, )
    SHADER_PARAMETER_STRUCT(MyStructType, MyStruct)
    SHADER_PARAMETER_RDG_TEXTURE(Texture2D, ColorTexture)
    SHADER_PARAMETER_SAMPLER(SamplerState, ColorSampler)
  END_SHADER_PARAMETER_STRUCT()

  class QualityDimension :
    SHADER_PERMUTATION_ENUM_CLASS("QUALITY", QualityEnumType);
  class ThreadgroupSizeXDimension :
    SHADER_PERMUTATION_SPARSE_INT("THREADGROUP_SIZEX", 8, 16);
  class ThreadgroupSizeYDimension :
    SHADER_PERMUTATION_SPARSE_INT("THREADGROUP_SIZEY", 8, 16);

  using FPermutationDomain = TShaderPermutationDomain<
    QualityDimension,
    ThreadgroupSizeXDimension,
    ThreadgroupSizeYDimension>;
};
IMPLEMENT_GLOBAL_SHADER(MyShaderCS, "/path/to/HLSL/file.usf",
  "MainCS", SF_Compute);
```

19

We can see uniform parameters declared in both the GPU and host code. Users are responsible for keeping these declarations in sync.

I'll also briefly mention varying parameters. In this compute shader case, only the GPU code needs to declare them, but other shader types require both host and GPU declarations for varying parameters.

Looking at a specific uniform parameter for a second…

Host and GPU do not share types

HLSL shader code (GPU)

```
int MyStruct_TileSizeX;
int MyStruct_TileSizeY;
int MyStruct_NumTilesX;
int MyStruct_NumTilesY;
```

Redeclared (and flattened) in GPU code

C++ shader code (host)

```
struct MyStructType {
  int TileSizeX;
  int TileSizeY;
  int NumTilesX;
  int NumTilesY;
};
```

Struct in host code

HPG 2022

20

I want to point out that host and GPU code don't share types. We've declared a struct in host code that must be redeclared in GPU code.

SKIP:

(You could use a shared header between host and GPU code, but still programmers are responsible for handling memory layouts manually, since the GPU compiler will insert padding for certain types. Ideally, programmers should be able to use the same types in host and GPU code so that they don't need to worry about this issue.)

(Also, a quirk of UE4 is that these struct uniform parameters are flattened in GPU code. This again is something that users have to deal with manually, but even without this quirk, using different types for host and GPU code still causes issues.)

Specialization parameters are implicit in GPU code

HPG 2022

HLSL shader code (GPU)

```
int MyStruct_TileSizeX;
int MyStruct_TileSizeY;
int MyStruct_NumTilesX;
int MyStruct_NumTilesY;

Texture2D ColorTexture;
SamplerState ColorSampler;

uint GetIterationCount() {
#if QUALITY == LOW
  return 2;
#elif QUALITY == MEDIUM
  return 4;
#elif QUALITY == HIGH
  return 8;
#else
  return 16;
}

[numthreads(THREADGROUP_SIZEX, THREADGROUP_SIZEY, 1)]
void MainCS(uint2 DispatchThreadID : SV_DispatchThreadID) {
  const uint IterationCount = GetIterationCount();
  ...
}
```

Specialization Parameters (Usage)

C++ shader code (host)

```
class MyShaderCS : public FGlobalShader {
public:
  DECLARE_SHADER_TYPE(MyShaderCS, Global);

                 _STRUCT(FParameters, )
                 UCT(MyStructType, MyStruct)
                 _TEXTURE(Texture2D, ColorTexture)
                 PLER(SamplerState, ColorSampler)
  EN    ADER_PARAMETER_STRUCT()

  class QualityDimension :
    SHADER_PERMUTATION_ENUM_CLASS("QUALITY", QualityEnumType);
  class ThreadgroupSizeXDimension :
    SHADER_PERMUTATION_SPARSE_INT("THREADGROUP_SIZEX", 8, 16);
  class ThreadgroupSizeYDimension :
    SHADER_PERMUTATION_SPARSE_INT("THREADGROUP_SIZEY", 8, 16);

  using FPermutationDomain = TShaderPermutationDomain<
    QualityDimension,
    ThreadgroupSizeXDimension,
    ThreadgroupSizeYDimension>;
};
IMPLEMENT_GLOBAL_SHADER(MyShaderCS, "/path/to/HLSL/file.usf",
  "MainCS", SF_Compute);
```

Specialization Parameters (Declaration)

21

Going back to the example shader, we have these specialization parameters. But they are only declared in the host code.

However, they are then used in GPU code. I'm going to omit a discussion of how this all works in UE4, but the point I want to make is that it's really easy to mess this up.

If I have a typo in the GPU code that uses a specialization parameter, I won't get any sort of warning or error, and similarly so if there's a typo in the string name on the host side.

Also, forgetting to declare a parameter in host code doesn't result in any diagnostic messages either. So future readers might wonder whether it was intentionally omitted or a bug, which actually happened to me.

Specialization: Compile time for GPU; Runtime for host

HPG 2022

HLSL shader code (GPU)

```
#define LOW 0
#define MEDIUM 1
#define HIGH 2
#define VERYHIGH 3

...

uint GetIterationCount() {
#if QUALITY == LOW
  return 2;
#elif QUALITY == MEDIUM
  return 4;
#elif QUALITY == HIGH
  return 8;
#else
  return 16;
}
```

← Compile-time constant set by Runtime value →

C++ shader code (host)

```
enum class QualityEnumType : uint32 {
  Low,
  Medium,
  High,
  VeryHigh,
  MAX,
};

...

MyShaderCS::PermutationDomain PermutationVector;
PermutationVector.Set<MyShaderCS::Quality>(quality);
PermutationVector.Set<MyShaderCS::ThreadgroupSizeX>(sizeX);
PermutationVector.Set<MyShaderCS::ThreadgroupSizeY>(sizeY);
```

22

An very critical---but very subtle---property of specialization parameters is that parameters that must be compile-time constant in GPU code are set by runtime values in host code. As mentioned before, this is ok in a system where host and GPU code are separated, and the UE4 toolchain handles compilation and dispatch appropriately. But this is a real issue in unified systems that we must deal with.

The Host-GPU Interface

- Need to express:
  - Uniform Parameters
  - Varying Parameters
  - Specialization Parameters
  - Entry Point Functions
    - And other GPU-compatible functions

HPG 2022

23

We've identified the required elements of interface between host code and GPU code.

The interface needs to express Uniform, Varying, and Specialization Parameters, as well an entry point function. I'm also going to claim that it's useful to be able to denote which other functions are GPU-compatible. This allows host-only functions to use host-only language features, and vice versa.

Now, let's look at our unified shader design, both to see how we express these various element and to see some of the benefits a unified system provides.

Unified Shaders in C++

HPG 2022

Unified Shader Design

HPG 2022

C++ shader code (host and GPU)

```
class [[ShaderClass]] MyShader {
public:
  [[uniform]] MyStructType MyStruct;
  [[uniform]] Texture2D ColorTexture;
  [[uniform]] SamplerState ColorSampler;

  [[specialization]]
  QualityEnumType Quality;

  [[specialization-sparseInt(8,16)]]
  int ThreadgroupSizeX;

  [[specialization-sparseInt(8,16)]]
  int ThreadgroupSizeY;

  // Continued on the right
```

```
  // Continued from the left

  [[gpu]]
  uint GetIterationCount() const {
    if (Quality == QualityEnumType::Low)
      return 2;
    else if (Quality == QualityEnumType::Medium)
      return 4;
    else if (Qualiiy == QualityEnumType::High)
      return 8;
    else
      return 16;
  }

  [[entry-computeShader(ThreadgroupSizeX, ThreadgroupSizeY, 1)]]
  void MainCS([[SV_DispatchID]] uint2 DispatchThreadID) const {
    const uint IterationCount = GetIterationCount();
    ...
  }
};
```

25

Here’s the example shader in our unified system. Note that the left and right parts of the slide are both part of the same C++ class, all contained within one file.

Like UE4, shaders are written as C++ classes. But in contrast, this class contains both host and GPU code.

The various elements of the host-GPU interface are…

Use C++ attributes for elements of the host-GPU interface

HPG 2022

C++ shader code (host and GPU)

```
class [[ShaderClass]] MyShader {
public:
  [[uniform]] MyStructType MyStruct;
  [[uniform]] Texture2D ColorTexture;
  [[uniform]] SamplerState ColorSampler;

  [[specialization]]
  QualityEnumType Quality;

  [[specialization-sparseInt(8,16)]]
  int ThreadgroupSizeX;

  [[specialization-sparseInt(8,16)]]
  int ThreadgroupSizeY;

  // Continued on the right
```

```
  // Continued from the left

  [[gpu]]
  uint GetIterationCount() const {
    if (Quality == QualityEnumType::Low)
      return 2;
    else if (Quality == QualityEnumType::Medium)
      return 4;
    else if (Qualiiy == QualityEnumType::High)
      return 8;
    else
      return 16;
  }

  [[entry-computeShader(ThreadgroupSizeX, ThreadgroupSizeY, 1)]]
  void MainCS([[SV_DispatchID]] uint2 DispatchThreadID) const {
    const uint IterationCount = GetIterationCount();
    ...
  }
};
```

26

(The various elements of the host-GPU interface are…)

expressed using C++ attributes, including uniform and specialization parameters, as well as marking GPU-compatible code and the GPU code entry point.

Because both the host and GPU code are unified into the same C++ class…

Use C++ attributes for elements of the host-GPU interface

HPG 2022

C++ shader code (host and GPU)

```
class [[ShaderClass]] MyShader {
public:
  [[uniform]] MyStructType MyStruct;
  [[uniform]] Texture2D ColorTexture;
  [[uniform]] SamplerState ColorSampler;

  [[specialization]]
  QualityEnumType Quality;

  [[specialization-sparseInt(8,16)]]
  int ThreadgroupSizeX;

  [[specialization-sparseInt(8,16)]]
  int ThreadgroupSizeY;

  // Continued on the right
```

Single declaration of parameters

```
  // Continued from the left

  [[gpu]]
  uint GetIterationCount() const {
    if (Quality == QualityEnumType::Low)
      eturn 2;
       if (Quality == QualityEnumType::Medium)
      turn 4;
       if (Qualiiy == QualityEnumType::High)
      turn 8;
    else
      return 16;
  }

  [[entry-computeShader(ThreadgroupSizeX, ThreadgroupSizeY, 1)]]
  void MainCS([[SV_DispatchID]] uint2 DispatchThreadID) const {
    const uint IterationCount = GetIterationCount();
    ...
  }
};
```

27

(Because both the host and GPU code are unified into the same C++ class…)

The programmer only needs to declare parameters once, rather than needing separate declarations for host and GPU code.

Related to this…

Explicit declaration of specialization parameters (for GPU)

HPG 2022

C++ shader code (host and GPU)

```
class [[ShaderClass]] MyShader {
public:
  [[uniform]] MyStructType MyStruct;
  [[uniform]] Texture2D ColorTexture;
  [[uniform]] SamplerState ColorSampler;

  [[specialization]]
  QualityEnumType Quality;

  [[specialization-sparseInt(8,16)]]
  int ThreadgroupSizeX;

  [[specialization-sparseInt(8,16)]]
  int ThreadgroupSizeY;

  // Continued on the right
```

```
  // Continued from the left

  [[gpu]]
  uint GetIterationCount() const {
    if (Quality == QualityEn
      return 2;
    else if (Quality == Qua
      return 4;
    else if (Qualiiy == QualityEnumType::High)
      return 8;
    else
      return 16;
  }

  [[entry-computeShader(ThreadgroupSizeX, ThreadgroupSizeY, 1)]]
  void MainCS([[SV_DispatchID]] uint2 DispatchThreadID) const {
    const uint IterationCount = GetIterationCount();
    ...
  }
};
```

Compile-time error (instead of a bug)

28

Specialization parameters are now explicitly declared not only for host code, but also for GPU code.

This enables the compiler to provide extra validation, such as catching this typo as a compile-time error, rather than it causing a bug if it were only implicitly declared.

Also thanks to the unified design…

Types shared between host and GPU

HPG 2022

C++ shader code (host and GPU)

```
class [[ShaderClass]] MyShader {
public:
  [[uniform]] MyStructType MyStruct;
  [[uniform]] Texture2D ColorTexture;
  [[uniform]] SamplerState ColorSampler;

  [[specialization]]
  QualityEnumType Quality;

  [[specialization-sparseInt(8,16)]]
  int ThreadgroupSizeX;

  [[specialization-sparseInt(8,16)]]
  int ThreadgroupSizeY;

  // Continued on the right
```

```
  // Continued from the left

  [[gpu]]
  uint GetIterationCount() const {
    if (Quality == QualityEnumType::Low)
      return 2;
    else if (Quality == QualityEnumType::Medium)
      return 4;
    else if (Qualiiy == QualityEnumType::High)
      return 8;
    else
      return 16;
  }

  [[entry-computeShader(ThreadgroupSizeX, ThreadgroupSizeY, 1)]]
  void MainCS([[SV_DispatchID]] uint2 DispatchThreadID) const {
    const uint IterationCount = GetIterationCount();
    ...
  }
};
```

29

Types are shared between host and GPU code. So the GPU code that uses the Quality enum parameter can reference the enum type options directly.

Specialization: Handling Compile-time vs. Rutnime

HPG 2022

C++ shader code (host and GPU)

```
class [[ShaderClass]] MyShader {
public:
  [[uniform]] MyStructType MyStruct;
  [[uniform]] Texture2D ColorTexture;
  [[uniform]] SamplerState ColorSampler;

  [[specialization]]
  QualityEnumType Quality;

  [[specialization-sparseInt(8,16)]]
  int ThreadgroupSizeX;

  [[specialization-sparseInt(8,16)]]
  int ThreadgroupSizeY;
```

Options listed at compile time

```
// Elsewhere in the code

MyShader shader;
shader.Quality = quality;
shader.ThreadgroupSizeX = sizeX;
shader.ThreadgroupSizeY = sizeY;
```

Values set at runtime

30

And lastly, remember that specialization parameters must be compile-time constant in GPU code but must then be set by runtime values in host code.

We handle this difference by requiring that specialization parameter declarations include a list of all possible options for those parameters as compile-time constants. Then, in host code, the programmer is free to set those parameters using runtime values.

SKIP:

(As a side note, our system generates asserts to validate that the runtime values match one of the compile-time options (and, to be fair, UE4 already provides similar validation for host code.))

Design Decisions

HPG 2022

- Use C++
  - One of (if not the) most-used languages in real-time graphics
  - (And because it's what UE4 uses)
- Express a "shader" as a C++ class
  - C++'s mechanism for encapsulating code and data
  - Both host and GPU code in the same class

31

Now that we know what a shader looks like in this unified system, let's dive in to some of the specific design decisions for our system. We've already covered some of them.

For starters, we use C++ as our programming language of choice. It is one of (if not the) most-used languages in real-time graphics. And it's a natural choice for us since it's what UE4 uses.

Similarly, UE4 already expresses host shader code using C++ classes, so we do the same in our system.

However, unlike UE4, our unified shader classes contain both host code and GPU code.

Design Decisions (cont.)

- Co-opt C++ attributes to express elements of the host-GPU interface
  - [[uniform]] float3 myColor;
- Co-opt the C++ inheritance and virtual functions model to implement composition and specialization

32

The last two design decisions I have are very much related to the "minimize changes" idea.

The first we've already seen: co-opt C++ attributes to express the elements of the host-GPU interface. This attribute syntax is standard since C++11, but C++ doesn't provide support for user-controlled attributes. Also, C++ compilers are free to ignore attributes they don't understand. Therefore, attributes should not change the semantics of a program. However, in our case, ignoring these attributes will result in an incorrect program, so we're very much violating the standard semantics of C++ attributes and instead repurposing them for the needs of graphics programming.

But even more so than C++ attributes, where our design really departs from standard C++ is our use of inheritance and virtual functions to implement shader composition and specialization.

Why do we need to co-opt anything for compositing and specialization? Well…

C++ is insufficient for composition + specialization

HPG 2022

- Inheritance and virtual functions
  - Dynamic dispatch of functions
  - Appropriate for host shader code

Dynamic dispatch is bad on GPUs

- Templates
  - Static dispatch of functions
  - Appropriate for GPU shader code

Templates lead to unwieldy host code

33

C++ really doesn't have what we need for shader specialization. It has two main ways to compose together different classes.

We could use standard inheritance and virtual functions. This leads to dynamic dispatch of function implementations through the virtual function table.

In contrast, could use C++ template. All template parameters must be fully specified at compile time, so this leads to static dispatch of functions.

And what I'll claim is that dynamic dispatch is better for host shader code, since we want to invoke GPU rendering work based on runtime parameter values.

But in GPU shader code, specialization is really important to achieve the best performance. So we really want static dispatch of all functions on the GPU. But this leads to problems for the host code.

So we have this problem where neither of these options are suitable for unified shader code, because the host code and the GPU code have these differing requirements.

I hope it’s clear why virtual functions aren’t a good solution, but in case you don’t believe me about templates, I have an extra slide we can talk about.

Solution: Co-opt virtual functions for specialization

- Author shader code using inheritance and virtual functions
- Codegen different GPU specializations based on subtypes
- Host sets parameters using regular C++ syntax
- System invokes different (specialized) GPU code based on runtime types

HPG 2022

34

The solution we came up with is to co-opt virtual functions to implement specialization.

Programmers author shader code using inheritance and virtual functions, but then the system will codegen different GPU specializations based on the relevant subtypes.

Host code can then set parameter just as it normally would in C++, but under the hood, the system will invoke different specialized GPU code based on the runtime types. So in effect, we get the best of both worlds, all while allowing programmers to express composition in a very C++ style.

To look at this visually…

Write code using virtual functions

```
class [[ShaderClass]] StandardMaterial : public Material
{
public:
  [[gpu]] virtual float4 eval(...) {
    /* GPU code to evaluate the standard material */
  }
};

class [[ShaderClass]] ClothMaterial : public Material {
public:
  [[gpu]] virtual float4 eval(...) {
    /* GPU code to evaluate the cloth material */
  }
};

class [[ShaderClass]] MyShader {
public:
  Material* material;

  [[gpu]] void entryPoint(...) {
    float4 color = material->eval(...);
  }
};
```

HPG 2022

35

The code programmers write looks similar to regular virtual function usage.

Generate de-virtualized functions for GPU code

```
class [[ShaderClass]] StandardMaterial : public Material
{
public:
  [[gpu]] virtual float4 eval(...) {
    /* GPU code to evaluate the standard material */
  }
};

class [[ShaderClass]] ClothMaterial : public Material {
public:
  [[gpu]] virtual float4 eval(...) {
    /* GPU code to evaluate the cloth material */
  }
};

class [[ShaderClass]] MyShader {
public:
  Material* material;

  [[gpu]] void entryPoint(...) {
    float4 color = material->eval(...);
  }
};
```

```
[[gpu]] float4 evalStandardMaterial(...) {
  /* GPU code to evaluate the standard material */
}

[[gpu]] float4 evalClothMaterial(...) {
  /* GPU code to evaluate the cloth material */
}
```

HPG 2022

36

But then the system will generate de-virtualized versions of the virtual functions for GPU code.

Generate specialized GPU shader variants for subtypes

HPG 2022

```
class [[ShaderClass]] StandardMaterial : public Material
{
public:
  [[gpu]] virtual float4 eval(...) {
    /* GPU code to evaluate the standard material */
  }
};

class [[ShaderClass]] ClothMaterial : public Material {
public:
  [[gpu]] virtual float4 eval(...) {
    /* GPU code to evaluate the cloth material */
  }
};

class [[ShaderClass]] MyShader {
public:
  Material* material;

  [[gpu]] void entryPoint(...) {
    float4 color = material->eval(...);
  }
};
```

```
[[gpu]] float4 evalStandardMaterial(...) {
  /* GPU code to evaluate the standard material */
}

[[gpu]] float4 evalClothMaterial(...) {
  /* GPU code to evaluate the cloth material */
}
```

```
[[gpu]] void entryPoint_Standard(...) {
  evalStandardMaterial(...);
}

[[gpu]] void entryPoint_Cloth(...) {
  evalClothMaterial(...);
}
```

37

And it will also generate specialized GPU shader variants for shaders that use these virtual functions.

This allows programmers to write regular looking C++ code for their shaders, but then that code gets transformed into an implementation that is better suited to GPU hardware.

Implementation

HPG 2022

- Source-to-source translator using Clang's LibTooling
  - Code live external to Clang
- Use Clang AST to retrieve info from user-written shader source code
- Generate HLSL for GPU code
- Generate C++ for host code
  - Use UE4's existing shader system under-the-hood
  - Could support other engines via additional backends

38

I'll briefly mention the implementation.

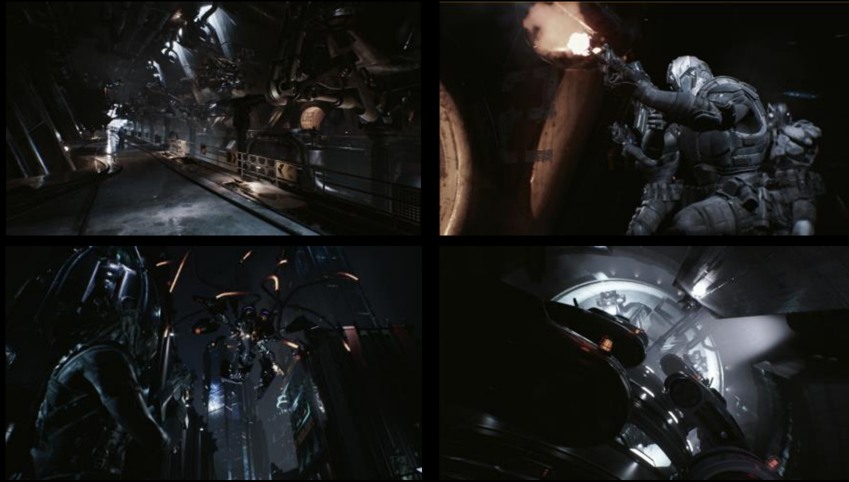
We use a source-to-source translator built on top of Clang, but in a way that doesn't require modifying the Clang codebase (which helps constrain implementation and maintenance costs).

The translator walks the Clang AST to retrieve relevant info from user-written shader source code.

It then generates HLSL for GPU code, including multiple shader variants to turn dynamic dispatch via virtual functions into static dispatch.

For the host code, it generates C++ and uses UE4's existing shader system under-the-hood. This helps to facilitate ease of integration and adoption, by making these unified shaders compatible with existing UE4 code.

We could add support for other shading language and engines by writing additional backends.



To evaluate our system, we rewrote some of UE4’s shaders to use the unified system, and we ran the Infiltrator Demo using both the original shaders and our unified shaders.

Evaluation – GPU Performance

HPG 2022

Similar Performance

Shader	Motion Blur Filter (time in ms)			Temporal AA (time in ms)		
	Min	Avg	Max	Min	Avg	Max
Original UE4 Code	0.06	0.18	0.70	0.23	0.28	0.74
Unified Code	0.06	0.18	0.70	0.24	0.28	0.75

40

As shown, the performance of our unified shaders is similar to the original UE4 versions.

We also wanted to make sure our abstraction didn't lead to excessive code bloat.

Evaluation – Lines of Code

HPG 2022

Similar Code Sizes

Shader	Motion Blur Filter	Temporal AA
Original UE4 Code	902	2,138
Unified Code	920	2,251

41

And similarly, the Lines of Code counts are comparable as well. Some of the additional lines are due to stylistic differences, some due to some minor code duplication from HLSL header files, and yes, some are due to our abstractions, but we think the trade-offs of more robust code is worth it.

Limitations

HPG 2022

- Only support compute shaders
  - An increasingly large portion of a modern game's shader code
  - Pixel shaders can be supported similarly
- Only support equivalent of UE4's *Global Shaders*
  - Do not interface with the material or mesh systems
  - Future work to explore *Material* and *MeshMaterial* shaders

42

As for limitations, our current implementation only supports compute shaders. These are an increasingly large portion of a modern game's shader code, and they are sufficient to demonstrate the challenges in unified shader programming.

Similarly, our translator only supports UE4's "Global Shaders," which don't interface with the material or mesh systems.

Wrapping up

HPG 2022

Summary

HPG 2022

- Support unified shader specialization in C++ by co-opting its existing features and implementing them with alternate semantics
  - Co-opt C++ attributes for the host-GPU interface
  - Co-opt inheritance and virtual functions for shader specialization
- Maintain compatibility as C++ evolves, since we aren't adding new features
- Similar performance and code size to original UE4 shaders

44

We added unified shader programming to C++ by co-opting its existing features and implementing them with alternate semantics.

We co-opt C++'s attributes to express elements of the host-GPU interface, and we co-opt inheritance and virtual functions for shader specialization.

Since we aren't adding new features to C++, just re-using existing ones, our system can maintain compatibility with future versions of C++.

Our unified shaders achieve similar performance and code sizes compared to the original UE4 shaders

Future Work

HPG 2022

- Co-opting features of another language for graphics
  - E.g., Rust's generics
- Optimizations across the host-GPU boundary
  - Unified view of the code → more opportunities to optimize
    - E.g., Memory allocations & transfers, synchronization between host/GPU & render passes
- Programming models beyond separate shaders per rendering pass

45

As future work, we think it'd be interesting to try co-opting features of another language for graphics programming.

Also, what kinds of optimizations can we perform by giving the compiler a unified view of the code.

And what will make those optimizations even more powerful is developing programming models beyond just separate shaders per rendering pass.

Future Work (cont.)

HPG 2022

- Programming models beyond separate shaders per rendering pass
  - Write rendering code as one integrated unit → (JIT) compiler splits it up into different shader kernels
  - Some benefits from compile-time analysis
  - Greater benefits from runtime optimizations too?
    - Utilizing render graphs to generate optimized kernels per frame

Unified shader programming and specialization support is a prerequisite for these types of future systems!

46

For example, maybe we could write rendering code as one integrated unit and let the compiler split it up into different shader kernels.

I think there could be some benefit from compile-time analysis, but there's a greater potential for incorporating runtime optimizations too, perhaps leveraging information available in a render graph system.

And I want to note that unified shader programming and specialization support is a prerequisite for these types of future systems!

And with that…

Acknowledgments

HPG 2022

- Guidance, feedback, and technical advice
  - Anjul Patney, Chuck Rozhon, Yong He, Brian Karis, Ola Olsson, Andrew Lauritzen, Yuriy O'Donnell, Angelo Pesce, Charlie Birtwistle, Michael Vance, Dave Shreiner, and the anonymous reviewers
- Hardware donations and financial support
  - Intel Corporation
  - NVIDIA Corporation

47

I'd like to thank the many people who helped us with this work.

And also thank you to Intel Corporation and NVIDIA Corporation for financial support.

Thank you!

Kerry A. Seitz, Jr., Theresa Foley, Serban D. Porumbescu, and John D. Owens
kaseitz@ucdavis.edu

Source Code:
https://github.com/owensgroup/UnifiedShaderSpecialization

HPG 2022

And thank you! The source code of our translator is available on GitHub.

And I'd be happy to answer any questions.

References

- Zachary DeVito, James Hegarty, Alex Aiken, Pat Hanrahan, and Jan Vitek. 2013. Terra: A Multi-Stage Language for High-Performance Computing. In Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '13). 105–116. https://doi.org/10.1145/2491956.2462166
- Epic Games, Inc. 2015. Infiltrator Demo. https://www.unrealengine.com/marketplace/en-US/product/infiltrator-demo
- Epic Games, Inc. 2019. Unreal Engine 4 Documentation. https://docs.unrealengine.com/en-us/
- Adrian Sampson, Kathryn S. McKinley, and Todd Mytkowicz. 2017. Static Stages for Heterogeneous Programming. Proceedings of the ACM on Programming Languages 1, OOPSLA, Article 71 (Oct. 2017), 27 pages. https://doi.org/10.1145/3133895
- Kerry A. Seitz, Jr., T. Foley, Serban D. Porumbescu, and John D. Owens. 2019. Staged Metaprogramming for Shader System Development. ACM Transactions on Graphics 38, 6 (Nov. 2019), 202:1–202:15. https://doi.org/10.1145/3355089.3356554

HPG 2022

Extra Slides

HPC 2022

References for Extra Slides

- NVIDIA Corporation. 2022. cuda-samples: reduction_kernel.cu. https://github.com/NVIDIA/cuda-samples/blob/b312abaa07ffdc1ba6e3d44a9bc1a8e89149c20b/Samples/2_Concepts_and_Techniques/reduction/reduction_kernel.cu#L633-L1025

- Kekoa Proudfoot, William R. Mark, Svetoslav Tzvetkov, and Pat Hanrahan. 2001. A Real-Time Procedural Shading System for Programmable Graphics Hardware. In Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH 2001). 159–170. https://doi.org/10.1145/383259.383275

HPG 2022

Overcoming the Limitations

HPG 2022

- Global Shader Pixel Shaders
  - Trivial extension of the Compute Shader case
- Mesh and MeshMaterial Shaders
  - Challenge: Coordinating varying parameters between shader types
    - E.g., Vertex Shader outputs → Pixel Shader inputs
  - Possible Solution: *Pipeline Shader* design [Proudfoot et al. 2001]
  - Challenge: Different amounts and type of data that need to be stored / accessed
    - E.g., Eye Material might have more data than Standard Material
  - Possible Solution: Compiler could specialize datatypes if only one material is used
    Compiler could union the types if multiple are used
    (e.g., auto-generated Gbuffer)

52

Why Templates are Insufficient

HPG 2022

reduction_kernel.cu (CUDA Sample)

```
...

template <typename T, unsigned int blockSize, bool nIsPow2>
__global__ void reduce6(T *g_idata, T *g_odata, unsigned int n) { ... }

template <typename T, unsigned int blockSize, bool nIsPow2>
__global__ void reduce7(T *g_idata, T *g_odata, unsigned int n) { ... }

...
```

Our version

```
// Boilerplate
template <typename T>
class [[ShaderClass]] ReduceBase {
  [[specialization_SparseInt(1024, 512, ...)]] int numThreads;
  [[specialization_Bool]] bool nIsPow2;

  [[gpu]] virtual void reduceKernel(...) = 0;
};

// Boilerplate
template <typename T>
class [[ShaderClass]] ReduceRunner {
  [[specialization_ShaderClass]] ReduceBase* reducer;

  [[entry_ComputeShader(reducer->numThreads, 1, 1)]]
  void runReducer(...) { reducer->reduceKernel(); }
};

// Our version of reduce6 and reduce7
template <typename T>
class [[ShaderClass]] ShaderReduce6 : public ReduceBase {
  [[gpu]] virtual void reduceKernel(...) override { ... }
};

template <typename T>
class [[ShaderClass]] ShaderReduce7 : public ReduceBase {
  [[gpu]] virtual void reduceKernel(...) override { ... }
};
```

53 [NVIDIA Corporation 2022]

On the left is an example from the NVIDIA CUDA samples. There are multiple reduction kernel implementations that are optimized for different situations. These kernels use template parameters to generate specialized implementations for better performance. The code on the left shows the GPU kernel declarations from the CUDA sample.

On the right is a version written using our abstractions. Ours has a little bit of extra boilerplate to declare the base ShaderClass, and also to declare the Runner ShaderClass (the latter is necessary with our current translator implementation, but we could eliminate the need for it with minor modifications in the future).

Then, we can declare the reduction implementation similarly, with just a little extra boilerplate. So at first, yes ours looks longer. But…

Why Templates are Insufficient

HPG 2022

reduction_kernel.cu (CUDA Sample)

```
// Running one of the reduce kernels

void reduce(...) {
  ...

  switch (whichKernel) {
    case 0:
      reduce0<T><<<dimGrid, dimBlock, smemSize>>>(d_idata, d_odata, size);
      break;

    case 1:
      reduce1<T><<<dimGrid, dimBlock, smemSize>>>(d_idata, d_odata, size);
      break;

    case 2:
      reduce2<T><<<dimGrid, dimBlock, smemSize>>>(d_idata, d_odata, size);
      break;

    case 3:
      reduce3<T><<<dimGrid, dimBlock, smemSize>>>(d_idata, d_odata, size);
      break;
  ...
```

Our version

```
// Running one of the reduce shaders
template <typename T>
void reduce(...) {
  ...
  ReduceBase* reduceShader;

  switch (whichKernel) {
    case 0: reduceShader = new ReduceShader0<T>(); break;
    case 1: reduceShader = new ReduceShader1<T>(); break;
    case 2: reduceShader = new ReduceShader2<T>(); break;
    case 3: reduceShader = new ReduceShader3<T>(); break;
    case 4: reduceShader = new ReduceShader4<T>(); break;
    case 5: reduceShader = new ReduceShader5<T>(); break;
    case 6: reduceShader = new ReduceShader6<T>(); break;
    case 7: reduceShader = new ReduceShader7<T>(); break;
    case 8: reduceShader = new ReduceShader8<T>(); break;
    case 9: reduceShader = new ReduceShader9<T>(); break;
  }

  reduceShader->numThreads = threads;
  reduceShader->nIsPow2 = isPowTwo(size);

  reduceRunner runner;
  runner.reducer = reduceShader;
  runner.addComputePass(...);
}
```

54 [NVIDIA Corporation 2022]

Where we get huge wins is when we look at the code that sets up and invokes these reduce kernels. On the right is our version, where we simply choose which class to use, set the numThreads and nIsPow2 parameters, and run the Runner ShaderClass. And we’re done.

The CUDA version look similar at first, where it’s selecting between different kernel implementations.

HPG 2022

Why Templates are Insufficient

reduction_kernel.cu (CUDA Sample)

```
case 4:
  switch (threads) {
    case 512:
      reduce4<T, 512>
          <<<dimGrid, dimBlock, smemSize>>>(d_idata, d_odata, size);
      break;

    case 256:
      reduce4<T, 256>
          <<<dimGrid, dimBlock, smemSize>>>(d_idata, d_odata, size);
      break;

    case 128:
      reduce4<T, 128>
          <<<dimGrid, dimBlock, smemSize>>>(d_idata, d_odata, size);
      break;

    case 64:
      reduce4<T, 64>
          <<<dimGrid, dimBlock, smemSize>>>(d_idata, d_odata, size);
      break;

    case 32:
      reduce4<T, 32>
          <<<dimGrid, dimBlock, smemSize>>>(d_idata, d_odata, size);
      break;
```

Our version

```
// Nothing :)
```

55 [NVIDIA Corporation 2022]

But then, we see the real problem. There's a bunch of extra code to map the runtime threads variable to template parameter, since these template parameters must be all specified at compile time. So there are cases for 512 to 32…

Why Templates are Insufficient

HPG 2022

reduction_kernel.cu (CUDA Sample)

```
  case 16:
    reduce4<T, 16>
        <<<dimGrid, dimBlock, smemSize>>>(d_idata, d_odata, size);
    break;

  case 8:
    reduce4<T, 8>
        <<<dimGrid, dimBlock, smemSize>>>(d_idata, d_odata, size);
    break;

  case 4:
    reduce4<T, 4>
        <<<dimGrid, dimBlock, smemSize>>>(d_idata, d_odata, size);
    break;

  case 2:
    reduce4<T, 2>
        <<<dimGrid, dimBlock, smemSize>>>(d_idata, d_odata, size);
    break;

  case 1:
    reduce4<T, 1>
        <<<dimGrid, dimBlock, smemSize>>>(d_idata, d_odata, size);
    break;
}

break;
```

Our version

```
// Nothing :)
```

56

[NVIDIA Corporation 2022]

And then for 16 to 1.

HPG 2022

Why Templates are Insufficient

reduction_kernel.cu (CUDA Sample)

```
case 5:
  switch (threads) {
    case 512:
      reduce5<T, 512>
          <<<dimGrid, dimBlock, smemSize>>>(d_idata, d_odata, size);
      break;

    case 256:
      reduce5<T, 256>
          <<<dimGrid, dimBlock, smemSize>>>(d_idata, d_odata, size);
      break;

    case 128:
      reduce5<T, 128>
          <<<dimGrid, dimBlock, smemSize>>>(d_idata, d_odata, size);
      break;

    case 64:
      reduce5<T, 64>
          <<<dimGrid, dimBlock, smemSize>>>(d_idata, d_odata, size);
      break;

    case 32:
      reduce5<T, 32>
          <<<dimGrid, dimBlock, smemSize>>>(d_idata, d_odata, size);
      break;
```

Our version

```
// Nothing :)
```

57 [NVIDIA Corporation 2022]

And then we move on to the next kernel implementation, where we have to have a switch over the runtime variable again.

Why Templates are Insufficient

HPG 2022

reduction_kernel.cu (CUDA Sample)

```
case 16:
  reduce5<T, 16>
      <<<dimGrid, dimBlock, smemSize>>>(d_idata, d_odata, size);
  break;

case 8:
  reduce5<T, 8>
      <<<dimGrid, dimBlock, smemSize>>>(d_idata, d_odata, size);
  break;

case 4:
  reduce5<T, 4>
      <<<dimGrid, dimBlock, smemSize>>>(d_idata, d_odata, size);
  break;

case 2:
  reduce5<T, 2>
      <<<dimGrid, dimBlock, smemSize>>>(d_idata, d_odata, size);
  break;

case 1:
  reduce5<T, 1>
      <<<dimGrid, dimBlock, smemSize>>>(d_idata, d_odata, size);
  break;
}

break;
```

Our version

```
// Nothing :)
```

58 [NVIDIA Corporation 2022]

And again.

Why Templates are Insufficient

HPG 2022

reduction_kernel.cu (CUDA Sample)

```
case 6:
  if (isPow2(size)) {
    switch (threads) {
      case 512:
        reduce6<T, 512, true>
            <<<dimGrid, dimBlock, smemSize>>>(d_idata, d_odata, size);
        break;

      case 256:
        reduce6<T, 256, true>
            <<<dimGrid, dimBlock, smemSize>>>(d_idata, d_odata, size);
        break;

      case 128:
        reduce6<T, 128, true>
            <<<dimGrid, dimBlock, smemSize>>>(d_idata, d_odata, size);
        break;

      case 64:
        reduce6<T, 64, true>
            <<<dimGrid, dimBlock, smemSize>>>(d_idata, d_odata, size);
        break;

      case 32:
        reduce6<T, 32, true>
            <<<dimGrid, dimBlock, smemSize>>>(d_idata, d_odata, size);
        break;
```

Our version

```
// Nothing :)
```

59 [NVIDIA Corporation 2022]

And again, except now there is a nested switch, because we also need to handle the nIsPow2 parameter in the same way.

Why Templates are Insufficient

HPG 2022

reduction_kernel.cu (CUDA Sample)

```
  case 16:
    reduce6<T, 16, true>
        <<<dimGrid, dimBlock, smemSize>>>(d_idata, d_odata, size);
    break;

  case 8:
    reduce6<T, 8, true>
        <<<dimGrid, dimBlock, smemSize>>>(d_idata, d_odata, size);
    break;

  case 4:
    reduce6<T, 4, true>
        <<<dimGrid, dimBlock, smemSize>>>(d_idata, d_odata, size);
    break;

  case 2:
    reduce6<T, 2, true>
        <<<dimGrid, dimBlock, smemSize>>>(d_idata, d_odata, size);
    break;

  case 1:
    reduce6<T, 1, true>
        <<<dimGrid, dimBlock, smemSize>>>(d_idata, d_odata, size);
    break;
}
```

Our version

```
// Nothing :)
```

60

[NVIDIA Corporation 2022]

And so on.

Why Templates are Insufficient

HPG 2022

reduction_kernel.cu (CUDA Sample)

```
} else {
  switch (threads) {
    case 512:
      reduce6<T, 512, false>
          <<<dimGrid, dimBlock, smemSize>>>(d_idata, d_odata, size);
      break;

    case 256:
      reduce6<T, 256, false>
          <<<dimGrid, dimBlock, smemSize>>>(d_idata, d_odata, size);
      break;

    case 128:
      reduce6<T, 128, false>
          <<<dimGrid, dimBlock, smemSize>>>(d_idata, d_odata, size);
      break;

    case 64:
      reduce6<T, 64, false>
          <<<dimGrid, dimBlock, smemSize>>>(d_idata, d_odata, size);
      break;

    case 32:
      reduce6<T, 32, false>
          <<<dimGrid, dimBlock, smemSize>>>(d_idata, d_odata, size);
      break;
```

Our version

```
// Nothing :)
```

61

[NVIDIA Corporation 2022]

And so on.

Why Templates are Insufficient

HPG 2022

reduction_kernel.cu (CUDA Sample)

```
    case 16:
      reduce6<T, 16, false>
          <<<dimGrid, dimBlock, smemSize>>>(d_idata, d_odata, size);
      break;

    case 8:
      reduce6<T, 8, false>
          <<<dimGrid, dimBlock, smemSize>>>(d_idata, d_odata, size);
      break;

    case 4:
      reduce6<T, 4, false>
          <<<dimGrid, dimBlock, smemSize>>>(d_idata, d_odata, size);
      break;

    case 2:
      reduce6<T, 2, false>
          <<<dimGrid, dimBlock, smemSize>>>(d_idata, d_odata, size);
      break;

    case 1:
      reduce6<T, 1, false>
          <<<dimGrid, dimBlock, smemSize>>>(d_idata, d_odata, size);
      break;
  }
}

break;
```

Our version

```
// Nothing :)
```

62

[NVIDIA Corporation 2022]

And so on.

Why Templates are Insufficient

HPG 2022

reduction_kernel.cu (CUDA Sample)

```
case 7:
  // For reduce7 kernel we require only blockSize/warpSize
  // number of elements in shared memory
  smemSize = ((threads / 32) + 1) * sizeof(T);
  if (isPow2(size)) {
    switch (threads) {
      case 1024:
        reduce7<T, 1024, true>
            <<<dimGrid, dimBlock, smemSize>>>(d_idata, d_odata, size);
        break;
      case 512:
        reduce7<T, 512, true>
            <<<dimGrid, dimBlock, smemSize>>>(d_idata, d_odata, size);
        break;

      case 256:
        reduce7<T, 256, true>
            <<<dimGrid, dimBlock, smemSize>>>(d_idata, d_odata, size);
        break;

      case 128:
        reduce7<T, 128, true>
            <<<dimGrid, dimBlock, smemSize>>>(d_idata, d_odata, size);
        break;

      case 64:
        reduce7<T, 64, true>
            <<<dimGrid, dimBlock, smemSize>>>(d_idata, d_odata, size);
        break;
```

Our version

```
// Nothing :)
```

63

[NVIDIA Corporation 2022]

And so on.

Why Templates are Insufficient

HPG 2022

reduction_kernel.cu (CUDA Sample)

```
// etc. etc. etc.
```

Our version

```
// Nothing :)
```

64 [NVIDIA Corporation 2022]

And… I think you see the point.

Why Templates are Insufficient

HPG 2022

reduction_kernel.cu (CUDA Sample)

~400 Lines of Code

```
// Running one of the reduce kernels

void reduce(...) {
  ...

  switch (whichKernel) {
    case 0:
      reduce0<T><<<dimGrid, dimBlock, smemSize>>>(d_idata, d_odata, size);
      break;

    case 1:
      reduce1<T><<<dimGrid, dimBlock, smemSize>>>(d_idata, d_odata, size);
      break;

    case 2:
      reduce2<T><<<dimGrid, dimBlock, smemSize>>>(d_idata, d_odata, size);
      break;

    case 3:
      reduce3<T><<<dimGrid, dimBlock, smemSize>>>(d_idata, d_odata, size);
      break;
```

Our version

~30 Lines of Code

```
// Running one of the reduce shaders
template <typename T>
void reduce(...) {
  ...
  ReduceBase* reduceShader;

  switch (whichKernel) {
    case 0: reduceShader = new ReduceShader0<T>(); break;
    case 1: reduceShader = new ReduceShader1<T>(); break;
    case 2: reduceShader = new ReduceShader2<T>(); break;
    case 3: reduceShader = new ReduceShader3<T>(); break;
    case 4: reduceShader = new ReduceShader4<T>(); break;
    case 5: reduceShader = new ReduceShader5<T>(); break;
    case 6: reduceShader = new ReduceShader6<T>(); break;
    case 7: reduceShader = new ReduceShader7<T>(); break;
    case 8: reduceShader = new ReduceShader8<T>(); break;
    case 9: reduceShader = new ReduceShader9<T>(); break;
  }

  reduceShader->numThreads = threads;
  reduceShader->nIsPow2 = isPowTwo(size);

  reduceRunner runner;
  runner.reducer = reduceShader;
  runner.addComputePass(...);
}
```

65 [NVIDIA Corporation 2022]

The original CUDA sample version uses about 400 lines of code to set up and invoke the reduce kernels.

In contrast, our version does the same in about 30 lines of code.

So I hope this helps to demonstrate why C++ templates are not a good solution for implementing shader specialization. Templates result in a bunch of extra boilerplate code needed to map runtime parameters to their compile-time counterparts. Imagine having to add another case to all of these switch statements!

In our system, we support shader specialization as a first-class operation, and we designed our system to elegantly handle the compile-time vs. runtime dichotomy of specialization parameters, resulting in a much more maintainable implementation. So adding another case is much simpler!