Development of a C++/SPIR-V Shader-Runtime

FABIAN WAHLSTER, Technische Univerität München, Germany

Shader development with GLSL or HLSL can be uncomfortable compared to conventional software development with advanced language features and sophisticated tools like Visual Studio. The standard shader programming pipeline is detached from the conventional C++ workflow because different compilers are needed for each graphics API, and each API requiring different kinds of meta information to configure the device state. This work tries to facilitate modern C++ features and design patterns in a new SPIR-V based GPU programming language that offers direct access to hardware intrinsics and enables development of generic shading algorithms with templating and polymorphism from C++. Shader programs written in *SPEAR* are assembled and linked during host runtime into bigger shader libraries (DLLs) that can be modified and reloaded during rendering, shortening shader development iterations drastically. This approach allows *SPEAR* code to be directly executed and debugged on the host processor using existing tools for C++ development and profiling, as well as running the code in a Vulkan graphics engine on any compatible GPU.

CCS Concepts: • **Computing methodologies** → *Graphics systems and interfaces*;

Additional Key Words and Phrases: shading languages, real-time rendering, code generation

ACM Reference Format:
Fabian Wahlster. 2018. Development of a C++/SPIR-V Shader-Runtime. 1, 1 (April 2018), 13 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Current graphics development is still dominated by programming concepts originating from GPU hardware characteristics such as static control flow and the shader-per-stage programmable pipeline. Current engines like the Unreal Engine 4 [10] [9] tackle this with monolithic Ubershaders, using pre-processor defines to switch between code parts that influence graphic effects and material properties, avoiding dynamic control flow at cost of pre compiling all variants and then exchanging the whole shader program at runtime. On the other side, only very basic code fragments (from Ubershaders) can be reused between shaders of different stages, making it harder to write modular and composable shaders. This work promotes a new SPIR-V based shading language *SPEAR* that can be generated from modern C++ without the need for a separate compiler toolchain, offering modern programming principles and reusable shader libraries.

SPIR-V is a binary intermediate language designed by the Khronos Group [12] and forms the foundation of this framework. It can be used an a broad spectrum of compatible devices like CPUs, FPGAs and most importantly GPUs, where it offers low-level functionality to program graphics shaders and compute kernels for use in the Vulkan API.

The general idea of this work is to record SPIR-V instructions (in form of intermediate operations) whenever overloaded operators on specialized variable placeholders (similar to registers) are executed in a derived C++ context class. After all operations implemented in the context class are recorded, the assembler reorders and translates these operations to SPIR-V instructions with valid IDs. Control flow altering syntax like **if for** and **while** need to execute all possible control flow variants to be able to assemble a valid program.

Another primary goal of this runtime framework is to execute those C++ operators also on the host processor instead of just assembling SPIR-V code from it. User defined lambda expressions are used alter the state of variable placeholders when overloaded operators like the add operator+= are invoked. By default, variable placeholders are unwrapped to expose underlying GLM (OpenGL Mathematics) type variables which are fed to the specific lambda function, which in turn calls GLM functions. Directly executing SPIR-V programs on host processors also allows debugging within the same debugger / IDE environment, further simplifying shader development compared to decoupled approaches.

Modern shading languages like HLSL [3] and GLSL 4.6 [13] still lack common C++ features like generics via templating, dynamic polymorphism and auto type deduction. The DirectX 11.2 Function Linking Graph [2] adds polymorphism for reusable shader libraries without changing any HLSL language features, but instead extends the compilation pipeline with tools to link shader function libraries into complete shaders at runtime. Efforts made in [18] (McCool) and [8](ShaderComponents, iDSL) try to alleviate this by enhancing and wrapping existing languages with new compiler and language features but tend to be rendering specific. This work tries to pave the way for higher level generic shader programming with STL like containers and algorithms, closely integrated into the normal C++ workflow.

To improve shader development and iteration speed SPIR-V programs (which when executed either record SPIR-V instructions or directly compute the result) can be linked into dynamic libraries (DLLs) which can be reloaded during engine runtime, allowing the developer to directly inspect results of changed shader logic. The majority of shader compile time is now shifted to compiling those DLLs since SPIR-V assembling takes a considerably less time than parsing and compiling conventional shader programs.

During the recording and assembling phase, meta information associated to variables is collected and stored together with the final SPIR-V module. The Vulkan API can make use of this information by accumulating variables into descriptor sets and pipeline objects

Author's address: Fabian Wahlster, Technische Univerität München, Garching, Germany, f.wahlster@tum.de.

```
complex z(0.f, 0.f);
While(z.Conjugate() < 4.f && i < max)
{
  z = z * z + c;
  ++i;
});
```

```
%58 = OpFOrdLessThan %bool %56 %57
%60 = OpLoad %v2float %53
%61 = OpDot %float %60 %60
%62 = OpLoad %float %59
%63 = OpFOrdLessThan %bool %61 %62
%64 = OpLogicalAnd %bool %63 %58
      OpBranchConditional %64 %65 %83
%65 = OpLabel
```

Fig. 1. Translating *SPEAR* integrated domain specific language code (left) to SPIR-V assembly (middle) and rendering the shader module in Vulkan (right).

to configure the device for rendering objects. This process can be automated and optimized for different update frequencies, reducing the implementation workload for graphics engines even further.

The main contributes of this work are:

- A open source C++ domain specific language shading language that transfers modern language features to the graphics programming domain, enabling workflows closer to C++ while still allowing close integration of the target hardware features.
- Vulkan interoperability functions to use meta information gathered while translating *SPEAR* to SPIR-V.
- A plugin interface for modular and interchangeable shader library development based on boost.DLL.
- An algorithm to reflect C++ structures and types to SPIR-V with only standard C++17 code.

2 RELATED WORK

Shader metaprogramming [18] has been introduced at a time when only parts of GPUs were programmable (DirectX 9 was not yet specified) and fixed function APIs were broadly used. To overcome the lack of unified programmable shaders (introduced with DirectX 10) McCool et al. implemented a low-level API called SMASH that emulates hardware features not available at the time (i.e. branching). In contrast to *SPEAR*, SMASH generates parse-tree nodes (in overloaded operators) which are processed and translated by a recursive descent parser in a later step. *SPEAR* however does not use a higher level representation of a syntax tree and instead directly puts intermediate operations on a instruction stream in the order C++ operations are evaluated and only a simple ID resolve pass is needed at the end. SMASH's underlying assembly language operates on explicitly allocated register objects while SPIR-V is further abstracted from the specific architecture of the target hardware.

With the introduction of a standardized assembly level shader language (SPIR-V) that is supported by the majority of current gen GPUs, it is easier than ever to build advanced domain specific shading languages and it makes sense to re-evaluate those concepts with more modern C++ metaprogramming features.

ShaderComponents [8] offer shading language extensions to better match current generation graphics APIs and improve CPU sided draw call performance. Shaders logic and resources are therefore organized into components of different purpose and update frequencies (e.g. ViewParams, ObjectTransform, MaterialPattern). Shader logic variations are implemented using static polymorphism while resources are grouped into descriptor sets that match certain time points within a fixed rendering loop (View →Materials →Objects). Besides deriving from these component interfaces, there is no way to influence descriptor sets and bindings manually, forcing the developer to adapt the fixed rendering loop in the engine using SPIRELang [7].

The Spark shading language [4] aims to enable good software engineering practices for pipeline stage (shader-per-stage) based programming models (found in HLSL and GLSL) by factoring logically distinct shader parts into independent modules that can be reused through out the complete graphics pipeline.

Sony's Open Shading Language (OSL) [5] is highly focused on offline rendering and comes with a rich standard library with primitives for ray-tracing and volumetric rendering. In comparison to other shading languages it's C like syntax also allows for operator overloading, but lacks most other modern C++ features (templating, polymorphism etc.).

Following May [17], *SPEAR* can be seen as a internal domain specific language (iDSL), embedding shading functionality (graphics domain) into a host language (C++). The choice of the host language has great influence on the implementation and features of the iDSL. May reasons that C-Sharp offers a feature rich standard library, reflection capabilities and a well established toolset where a new shading language would require a new compiler infrastructure. In contrast to this framework, HLSL is generated from a parse tree and then processed by the DirectX Compiler, whereas *SPEAR* does not use any higher level hierarchical representation to produce SPIR-V code. This alleviates the need for a second parsing process run by the DirectX toolchain. C-Sharp reflection could solve issues such as variable name resolution and struct initialization more elegantly, this work however is focused on close integration with high performance graphics engines based on the Vulkan API. Using C-Sharp as a host language for *SPEAR* would inherit the need for a .Net runtime and also require additional conversion between C-Sharp and C++ Vulkan types. As of 2018, C++ does not yet feature standardized reflection mechanisms but some compilers offer implementations. For compatibility reasons *SPEAR* implements all features in standard C++17 and mimics reflection functionality with other available features such as structured bindings. Inferring variable names directly from the code might also not be desired because mapping buffer inputs from the engine requires the developer to name those

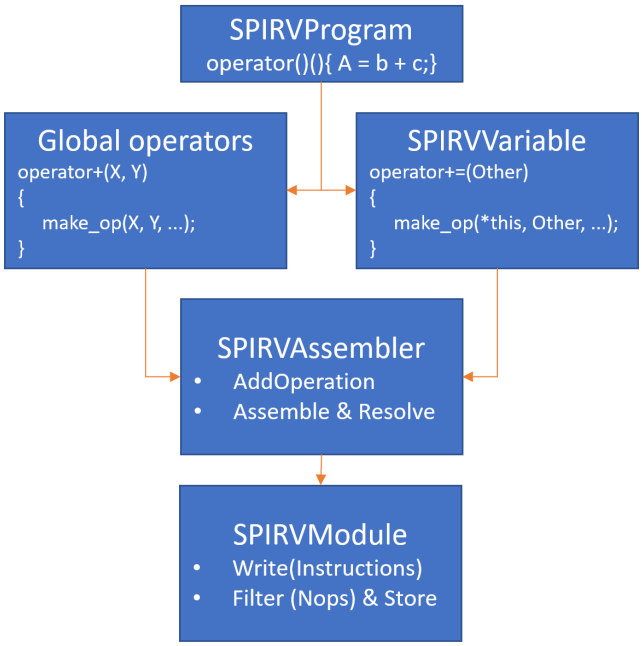

Fig. 2. The *SPEAR* translation process starts with the instantiation of a *SPIRVProgram* class where the function operator() is the main entry point to the shader. Any code using the overloaded operators of the *SPIRVVariable* invoke a variant of the make_op function when executed, resulting in respective calls to the global *SPIRVAssembler* instance and its *AddOperation* function. At the end of the *Assemble* process, the *SPIRVProgram* instance is destroyed and the intermediate instruction IDs are *Resolved* to the final result IDs used in *SPIRVInstructions*. The *SPIRVModule* is the final output of the *SPIRVAssembler* and its binary SPIR-V representation can be further processed by an external optimizer for example.

variables with descriptive and unique names in all shaders that use the same inputs. With the rise of the Rust language and ongoing efforts to support the Vulkan API [16] [14] in its ecosystem, the questions if Rust could be used as a host language (for *SPEAR*) arises. Klein [15] argues that "many concepts in Rust are not useful for shading languages".

3 SPIR-V CODE GENERATION AND EXECUTION

3.1 The SPIR-V Assembler

Instructions in SPIR-V [12] are encoded in a variable number of 32bit words where the first word includes the 16bit opcode and the word count for the following operands.

Instructions can produce unique IDs (starting from 1) which are used to reference types or results of operations and are consumed as operands by other instructions. The order of IDs is arbitrary with respect to the order of instructions in the stream, but one specific ID can only be the result of one particular instruction, forcing the module IDs to be in static single assignment (SSA) form. Operands can either be expressed as immediates used for constants (32bit word literals) or references in form of IDs.

In this framework the actual ID assignment is hidden by the *SPIRVOperation* class that abstracts the binary instruction format and the *SPIRVAssembler* that returns placeholder IDs (also called intermediates later on) when a operation is added.

In SPIR-V, modules must follow a specific instruction order starting with a preamble containing general information (Capabilities, Extensions, MemoryModel, Execution Mode, EntryPoint) followed by debug (Source, Name, MemberName, String) and annotation instructions. Types and constants must precede any instruction consuming them, therefore global variables and function declarations (and definitions) follow at the end of the module.

Since those special instructions can be created in any order, even before any other instruction is recorded (for example when a global variable is instantiated upon object construction) the assembler must reorder them to the front of the module before translating them to the final instruction stream.

Types and constants often form hierarchies where the definition of a float must precede the definition of a vec4 and array types require integer constants specifying the dimensionality. To avoid redefinitions, the underlying types and constants are stored in hashmaps which are queried whenever they are added to the instruction stream. This pattern enforces unique definitions and minimizes the number of generated instructions.

Depending on the optimization level of the host compiler, dead code might be or might not be removed before recording. In addition to this behaviour the *SPEAR* assembler allows to iteratively remove operations which have unreferenced result IDs and therefore are never consumed by any other operation, further reducing the size of the final SPIR-V module.

For the purpose of shader development with *SPEAR*, some simplifications with respect to module generation can be made: Kernel capabilities (OpCapability) are omitted because OpenCL 2.0 and CUDA 9 already offer modern C++ features and are well established tools for compute tasks. Currently only *Shader* capabilities are required to build a functional *SPEAR* module, additional capabilities can be specified on a per-program level. SPIR-V supports multiple entry points, *SPEAR* however only uses one main entry point for one specific stage (execution model) to simplify translation of the preamble. This is possible because translation by operator overloading implicitly inlines functions into the operation stream, resulting in one monolithic entry point function for the entire *SPIRVProgram*, removing the need for SPIR-V function concept translation.

3.2 Translation via operator overloading

Generating SPIR-V instructions by overloading operators does not work for primitive C++ types, therefore a new variable type is introduced that encapsulates the specific data type and implements needed operators. As a consequence, all operations in a *SPIRVProgram* must use those variable wrappers (*SPIRVVariable*, var_t<T, Assemble, StorageClass> class) to be translated to *SPIRVOperations*. Normal variables can still be used to change the instruction recording behaviour and allow developers to write meta-programs within

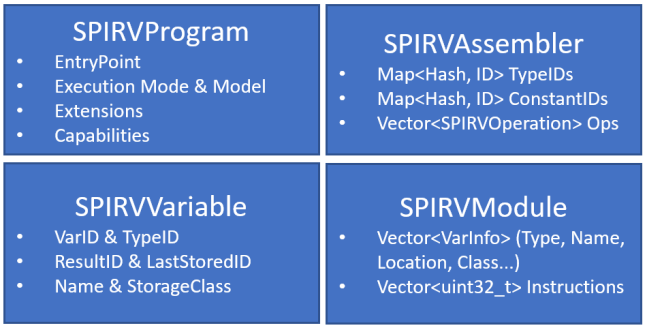

Fig. 3. Data associated to the main classes of this framework: *SPIRVProgram* holds information on which stage and capabilities the shader targets and whether extension instructions are used. *SPIRVVariable* contains the current result IDs of operations manipulating this variable, as well as general information on the variable storage class, its type and name. The *SPIRVAssembler* class accumulates result IDs of type and constant operations to ensure uniqueness of these IDs. Operations are recorded to the internal vector of *SPIRVOperations* and translated to *SPIRVInstructions* which are stored to the *SPIRVModule* along with meta information of the used variables.

shaders. Meta-programs can be used to influence code translation (changing *SPIRVAssembler* behaviour, i.e. forcing loads/stores etc) and generation by altering the shader control flow.

SPIRVVariables can be seen as registers as they hold the current result ID of a operation that produced the current value. This result ID is updated by operations applied to this variable within overloaded operators. The initial state of a variable can be set using *SPIRVConstants* passed during variable construction, or is implied by the SPIR-V Input *StorageClass*. To access variable memory, a variable pointer/ID must be generated from an *OpVariable* or *OpAccessChain* instruction. The *StorageClass* of a variable signifies the storage type and its visibility to different invocation scopes, for example *StorageClass* Input and Output refer to pipeline stage inputs (e.g. vertex attributes for vertex stage) and outputs (render targets for the pixel stage). *StorageClass* Uniform and UniformConstant are used to declare shader resources which are shared across all invocations and work groups while Function variables are local to the current invocation.

The process of generating a *SPIRVOperation* for a function that takes two operands can be easily shown by the example of the **make_op** (Listing 8) helper function: The **Assemble** template argument controls whether the host code (C++ Functor) is executed or SPIR-V code is generated for the current operation. STL's invoke_result_t is used to infer the variable type from a user defined functor that consumes two operands, for example vectors that should be multiplied using GLM's multiplication operator. Because SPIR-V exposes different instruction variants for the different operand types, the exact opcode must be decided either on the operand or result type. The division instruction for example comes in three different variants (OpFDiv, OpSDiv, OpUDiv) which are all passed to the *OpTypeDecider* to select the opcode matching the template argument type. A new intermediate variable is constructed to hold the result ID (or value) of the generated operation. To access the operands result IDs, the variables must first be loaded (and thus creating the OpLoad or OpAccessChain instruction if needed) in order to feed them to the consuming operation.

Together with the resulting variable type ID and the intermediate operands, the new operation is added to the operation stream through the *SPIRVAssembler* AddOperation routine, and finally the division operator can be overloaded using the **make_op** helper as seen in listing 1.

3.3 Creating SPIR-V variables from C++

SPIRVVariable creation is probably the most complex task in this framework as it involves type hierarchy decomposition (which requires reflection information) to create memory accessor instructions (OpVariable and OpAccesschain), C++ to SPIR-V type conversion and initialization from variable arguments and constants. Arrays, structures, textures and samplers all need special care because of their fundamental hierarchical layout differences to simple primitives. Structures are especially hard, they can be nested and contain other complex variables as well as non-*SPIRVVariable* typed members or user defined structures that should not be decomposed. To discriminate between user structs and those that should be reflected into SPIR-V, a type tag is introduced to mark them for further decomposition.

This work introduces a recursive type traversal algorithm based on the implementation of Anatoliy Tomilov [21], mimicking missing C++ 17 class reflection features. A key part of this algorithm is the *aggregate arity* counter mechanisms which determines the number of member components in the current nesting level. It is used to steer the recursive traversal and is fed into the second part, the *GetStructMember* function which interprets the current structure level as a tuple in a structured binding of length N (*aggregate arity*) and index I is used to select the member element from this tuple. I is incremented for each member variable in the current level, and the recursion is advanced when ever the *aggregate arity* is greater 1. A type trait based on SFINAE is used to check if the current member is marked with either struct or variable tag, if not the member is ignored and the algorithm advances to the next member until the complete hierarchy is consumed and the structure is decomposed into primitives which can be easily initialized and added to the SPIR-V reflection meta information.

Nested types in SPIR-V require a linear representation of the member position relative to the nesting level of the structure to address them when loading subcomponents. This representation is called AccessChain and is used with the *OpAccessChain* instruction to generate a ID pointing to the member. In practice, the iteration index I can be appended to a vector (linear representation) whenever the algorithm advances into the next nesting level and stored into the current member variable. It is important to note that the *OpVariable* instruction refers to the top level of the structure while *OpAccessChain* is used to load a member instead of the regular *OpLoad* instruction. After the top level variable ID has been created, member variables also store this ID as the base ID for later use with

Listing 1. Division operator overloading

```
template <class U, class V, bool Assemble, spv::StorageClass C1, spv::StorageClass C2>
inline auto operator/(const var_t<U, Assemble, C1>& l, const var_t<V, Assemble, C2>& r)
{
  return make_op(l, r, [](const U& v1, const V& v2) {return v1 / v2; }, kOpTypeBase_Result, spv::OpFDiv, spv::
    OpSDiv, spv::OpUDiv);
}
```

the *OpAccessChain*. Care must be taken when members are not 16 byte aligned: the Vulkan specification [11] declares alignment rules which must be obeyed in SPIR-V's *OpMemberOffset* decoration. Booleans for example are implicitly converted to 4 byte words when they are used inside a structure. Alignment rules and AccessChains also apply to arrays, but can be easily addressed with a stride size and a linear index.

SPIRVVariables can be initialized either from constants or other *SPIRVVariables* passed to the constructor. If the parameter list only contains constants, the process becomes fairly easy because the result ID from the constant composition (combination of existing and newly introduced values) can be directly passed to *OpVariable* instruction for initialization. If however, the parameter list contains at least one *SPIRVVariable*, the new variable must be created using the *OpCompositeConstruct* instruction which combines components of other variables to a new composite variable. Each parameter passed to the constructor is checked for the variable type tag using the *is_var<T>* trait, if the type is not tagged, it is treated as a constant and the appropriate ID is appended to the list of *OpCompositeConstruct* operands, otherwise the parameter is decomposed into primitive components and the ID is generated using the *OpCompositeExtract* instruction.

To minimize store operations, each variable keeps track of the last result ID that has been saved to memory and variables are only loaded if no result ID has been set by either a load or store operation. For nested types the creation of the *OpAccessChain* instruction (which uses the base ID and member index I) can be delayed to the first load of the variable, creating the final variable ID consumed by the *OpLoad* instruction and *SPIRVDecorations*. *OpName* instructions are materialized on the first load because the final variable and base IDs might not be known when the *SPIRVVariable* is constructed.

Load operations are invoked whenever a operation consumes a variable while store operations are triggered when copy assignments are executed or the destructor is called when the variable goes out of scope.

3.4 Implementing language features

In contrast to C++, shading languages offer component wise vector access and swizzle operations to mix and replicate the element values to a new variable that by itself can be a scalar or vector type. MSVC's declspec property is used to mimic the same syntax, hiding away the underlying functions which insert or extract components (as seen in listing 2). Both HLSL and GLSL's vector types have up to 4 dimensions, where each component is denoted by **x y z w** or **r g b a** properties. A separate generator was implemented to create a C++ header with all combinations because writing and modifying this many functions would be impractical. Selecting the right components is realized with template arguments that index into the source vector and copy the elements needed for (swizzle) access. In the case that the swizzle indices are monotonic and match the dimensionality of the source vector, the identity result ID can be returned instead of generating *OpCompositeExtract* instructions. The return type of a extract operation can be deduced from the component type of the vector and the number of valid swizzle indices. To avoid unnecessary generation of extract instructions when the same elements are accessed in a later operation, result IDs are cached in a hash-map where the key is derived from the swizzle indices.

The process of adding new operations to the instruction stream is serial, subsequent operations can not be directly referenced as operands since the intermediate instruction ID is only known after it has been added to the stream (the assembler assigns the instruction ID relative to its position in the stream). To overcome this limitation of the serial assembler concept, a pointer to the newly added operation is handed out to allow the developer to modify and add operands to previously inserted operations with already fixed stream locations.

This feature is mostly used for the definition of the shader EntryPoint and the implementation of branching keywords such as **if while** and **for**, the implementation of which is discussed in detail at the example of **if** (Listing 9): The if-else construct is based on a structure called *BranchNode* which stores pointers to *OpSelectionMerge* and *OpBranchConditional* operations or (in execution mode) the Boolean value of the condition variable to be used in the else branch. The content of the respective branches is passed to the *IfNode*-function using a lambda expression, which, when executed assembles its contents based on the Assemble/Execution mode: In assemble mode, the branch is always translated, ignoring the evaluation of the branch condition. First the condition variable is loaded to create the result ID if it has not been accessed yet, then *OpSelectionMerge* (with a replacement ID for the false-branch label ID) and a *OpBranchConditional* (with the condition variables result ID as Operand) are generated. *OpLabels* for both *true* and *false* branches are inserted before and after the lambda function with the branch content is executed. *OpBranch* takes the result ID of the false-label as operand which is reused for the else-branch. Finally the false-label ID is added to the operands of *OpBranchConditional* and the previous *OpSelectionMerge* and the *BranchNode* structure is returned from the *IfNode*-function. The else-branch is implemented

Listing 2. Vector component access

```
1 TExtractType<3> XYZ() const { return ExtractComponent<3, 0, 1, 2>(); }
2 template <spv::StorageClass C1> void XYZ(const var_t<vec_type_t<base_type_t<T>, 3>, Assemble, C1>& _var) const {
      InsertComponent<3, 0, 1, 2>(_var);}
3 void XYZ(const vec_type_t<base_type_t<T>, 3>& _var) const { InsertComponent<3, 0, 1, 2>(var_t<vec_type_t<
      base_type_t<T>, 3>, Assemble, spv::StorageClassFunction>(_var));}
4 __declspec(property(get = XYZ, put = XYZ)) TExtractType<3> xyz;
5 __declspec(property(get = XYZ, put = XYZ)) TExtractType<3> rgb;
```

as a member function of the *BranchNode* (which stores all needed information) and works in the same way. In execution mode the lambda function is called based on the Boolean value of the condition variable.

Because the host hardware configuration is known during assemble-time, vendor specific extensions (like AMD's shader trinary minmax) can be emitted when feasible, automatically detecting such situations however can be rather complex. Therefore a set of functions and operations implementing the extension on a C++ language level is exposed in a distinct namespace to avoid unintentionally mixing of plain SPIR-V and vendor extensions. The language facilities offered by this framework enable the developer to easily add extensions not yet supported by other shading languages, further improving adoption of current generation hardware features.

4 C++ RUNTIME

4.1 Dynamic Shader Libraries

Modern rendering engines often utilize huge shader libraries [9] to render a broad variety of effects. To reduce the space needed to store all variants of shaders, this framework compiles *SPIRVPrograms* (and the *SPIRVAssembler* to produce the final module) into a dynamic link library (DLL). This has several advantages: compilation time (HLSL parsing, include file loading, AST generation, optimization) is reduced to assemble time (assembled SPIR-V modules are cached in host memory), and dynamic libraries can be reloaded during runtime, allowing the developer to change shader code while the engine is running. For platform independence, library loading is implemented using the boost.DLL project that abstracts operation system specific API calls such as *LoadLibrary*. A simple interface is introduced to validate compatibility of the shader library and the game engine that loads it.

The main compilation speedup comes from the fact that for a *SPIRVProgram* with many variants (generated by divergent control flow) only one C++ compilation run is needed, while for a normal HLSL shader with permutations controlled by macros, a compilation run (again with text parsing) is required for each variant. The total number of compiler runs is reduced to $Compiletime_k + (K! * Assembletime_k)$ from $K! * Compiletime_k$ offline runs for K variants. To verify this behaviour, a shader with 64 permutations is compiled using *SPEAR* and a permutation compiler based on the DirectX Compiler [22]. The DirectX permutation compiler already pre-processes

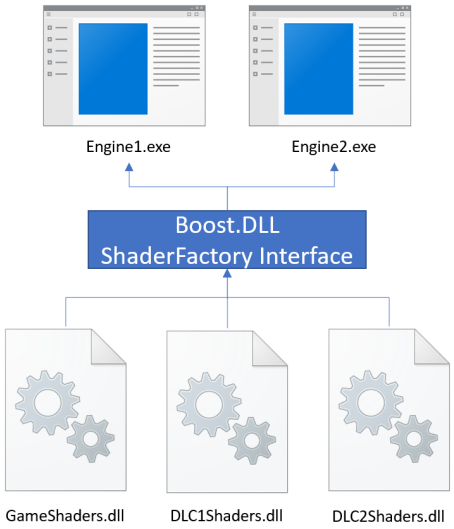

Fig. 4. Shaders written in *SPEAR* can be compiled into larger interchangeable libraries and loaded at runtime. This framework offers a simple interface to handle shading libraries as generic C++ plugins, allowing shaders to be reused in different applications and engines.

all includes to avoid any filesystem interaction during compilation (among other optimizations) but is still outperformed by the naïve *SPEAR* assembler as seen in table 1.

Whenever a shader library is reloaded, all consuming renderpasses are notified to update and recreate their pipeline state objects if needed.

4.2 Vulkan Integration

To avoid micro stutters caused by assembling the *SPEAR* shader at runtime, it makes sense to preassemble common (regularly used) shaders together with information on the associated pipeline configuration when the engine is starting up. Changing shaders and pipelines is extremely costly [20] and should be avoided during rendering, but for a renderer to be dynamic and flexible enough, it is a important feature to switch between materials and shader permutations on a per object basis. Therefore the pipeline cache from the last application run together with derivative pipelines is used to speedup the worst case scenario: changing pipeline mid recording draw calls. Sorting objects by material/shader identifiers further reduces the cost of unnecessary switching if a simple caching mechanism is implemented. For this mechanism to work it

is required that materials IDs can not change during the recording phase of a commandbuffer. Shader variants that render to different render targets are not supported in this scenario because this would require the framebuffer and renderpass objects to be rebuilt, forcing the commandbuffer to be split for maintaining the possibility to be recorded in parallel.

4.3 Executing Shaders on the CPU

When executing code on the CPU, the memory residency of *SPIRV-Variables* must be reflected to the host counterparts to ensure the same behaviour as executed on a GPU. Uniforms and constants can be shared across shader instances and therefore it makes sense to invoke a group of threads on the same *SPIRVProgram* instance. Optionally the programmer could provide an instance with data assigned to the uniform variables to the dispatcher instead of automatically creating the instances in the dispatcher. StorageClass function and private variables are local to the function if they are declared in the scope of the function operator (otherwise they exist in the scope of the class which is not desired). Some built-in variables like *GlobalInvocationID* need to be assigned on a per invocation basis, requiring the memory to be local to the thread (instead of the *SPIRVProgram* instance), hence C++ thread_local qualifier should be used to make sure each invocation gets its distinct ID.

However, this is only feasible if there is a 1-to-1 relation between threads and invocations. Some thread/task-scheduling implementations use techniques like thread stealing to assign a new task to an idling thread to redistribute the remaining work to the available resources. This technique is important as hardware threads (SMT) are a scarce resource on consumer CPUs (in comparison to GPUs) and idling threads should be avoided.

The *SPEAR* implementation currently lacks any thread scheduling implementation and users directly have to invoke *SPIRVProgram* instances to debug a certain shader configuration (with corresponding variable assignment). With the recent introduction of parallel algorithms in the C++ standard library [19], a modern and standardized implementation without any third-party library (such as Intel TBB, Microsoft's PPL or OpenMP) is possible. Both SPIR-V and C++ synchronization primitives (atomic operations, fences and semaphores/mutexes) must be implemented such that the same semantic results in the same program behaviour on the different architectures of CPUs and GPUs. This however only accounts for the execution of the programmable shader stages, fragment generation (interpolation, rasterization, depth and stencil operations) for instance would still be required to create a functional software renderer, as well as proper CPU side support for image sampling and various compression formats.

5 RESULTS

The *SPEAR* implementation comes with primitive types for complex (Listing 4) and quaternion numbers which can be used in a straight forward approach to render the Mandelbrot fractal (Listing 5) in just a couple of lines of shader code as shown in figure 1. Besides additional arithmetic types, C++ features can be used to make some functions like Min(a,b) (Listing 3) more flexible and take any number of arguments when implementing it with a variadic template argument list. Listings 6 and 7 demonstrate a possible application of inheritance in shader programming where the *SphereSDF* class derives from *SDFObject* to implement the *Distance(float3 point)* interface. The abstract *SDFObject* class can be used to implement constructive solid geometry (CSG) algorithms without knowing the concrete object type at compile-time.

To compare the compile and assemble performance of *SPEAR*, a deferred PBR lighting pixel-shader was translated from HLSL, resulting in 400 lines of equivalent C++ code. The shader has two permutations for environment and shadow mapping, and a unrolled loop for each type of dynamic light source (point and spot lights) to compute the respective illumination contribution. Light source data is stored in two uniform constant arrays where the *SPEAR* type requires the array dimension to be a template parameter, therefore two template class arguments are passed to the *SPIRVProgram* that implements the shader and effectively unrolls the light computation according to the parameters. The array sizes increase in steps of 16 light sources, up to a total of 48 resulting in 64 variants of the shader (16 combinations of array lengths, 4 combinations of shadow and environment mapping). The total compile-time of a shader is the sum of C++ project compile-time and assemble-time of each variant. Visual Studio's precompiled headers features is disabled because the DirectX Compiler has no comparable functionality and binary output (saving the shader to disk) is not included in the measurements. All measurements are averaged over 10 samples with one excluded warmup execution running on a Intel Core i7-6700K @ 4.0GHz and 16 GB DDR4 memory system.

Compiling the *SPEAR* code with the latest MSVC version (coming with Visual Studio 15.6.4) took 12.295 seconds on average when using 16 template instances of the derived DeferredLighting class (one instance for each combination of light source array lengths 1, 16, 32, 48), while compiling one instance with fixed point and spot-light count took approximately more than a third of the time: 4.325 seconds. The rather simple Mandelbrot shader took 3.092 seconds on average to compile. The C++ compiliation time varies depending on the complexity of the shader and is influence by the amount of inlining and template instantiation used. For smaller shader libraries it is possible to compile DLLs in less then 10 seconds, together with the assemble time for all shader variants, the *SPEAR* framework outperforms traditional shader compiler toolchains (Table 2) by an order a magnitude.

The assemble time measurements (Table 1) indicate that it is possible to assemble small to medium large shaders in a fraction of a millisecond, which is fast enough to use this framework in an on-demand manner when an object or material with the needed shader becomes visible. Large shaders with long unrolled loops for example should still be preloaded when the base pipeline object is generated before entering the rendering loop to avoid stalls when waiting for the shader to be completely assembled. With further optimization of the *SPEAR* framework and its on-the-fly recording and ID resolve technique it might be possible to utilize the Khronos

Listing 3. Enhancing existing functions with C++ templating features

```
template <class T, bool Assemble, spv::StorageClass C1, spv::StorageClass C2, class ...Ts>
inline var_t<T, Assemble, spv::StorageClassFunction> Min(const var_t<T, Assemble, C1>& X, const var_t<T, Assemble
    , C2>& Y, const Ts& ..._Args)
{
  return Min(Min(X, Y), _Args...);
}
```

Tabelle 1. *SPEAR* Assemble Timings

Permutations	Optimization	Lights	Sec
64	none	1-48 PL 1-48 SL	0.32123
64	AllPerf	1-48 PL 1-48 SL	2.44190
64	RemoveUnRef	48 PL 48 SL	5.80169
4	none	1 PL 1 SL	0.00263
1	none	1 PL 1 SL	0.00077
1	AllPerf	1 PL 1 SL	0.00382
1	AllPerf	48 PL 48 SL	0.01027

Measuring time needed to translate *SPEAR* to SPIR-V at runtime using different configurations: # PL: PointLight array dimension # SL: SpotLight array dimension. *RemoveUnRef*: remove operations with unreferenced instruction IDs using naïve $O(n^2)$ implementation. *AllPerf*: All performance optimization passes of spirv-opt.

Tabelle 2. HLSL Compile Timings

Permutations	Execution	Lights	Sec
64	serial	1-48 PL 1-48 SL	68.506
64	4 cores 8 threads	1-48 PL 1-48 SL	25.662
4	serial	1 PL 1 SL	0.2545
4	serial	48 PL 48 SL	11.430

Compile-time measurements of a shader permutation compiler based on the DirectX 11 HLSL Compiler. Compiling and assembling equivalent *SPEAR* code is 5.43 times faster on average (Compile 12.295s + Assemble 0.321s) compared to the common HLSL workflow (12.616s vs. 68.506s).

spirv-opt library to run performance and size optimization passes on-demand while the engine is rendering.

Due to the rather naïve implementation of the unreferenced-operation-removal algorithm, its runtime and effectiveness (as seen in tables 1 and 3) can not compete with *spirv-opt*'s size optimization passes but might be useful in combination, when the module size is the main optimization goal. The verbosity of *SPEAR*'s emitted instructions (compared to spirv-opt all performance pass) comes from the fact that function-class *SPIRVVariables* create *OpVariable* instructions when a shared *OpConstant* would suffice, i.e. the variable is never stored and therefore does not need to be backed memory, just as for variables that should only propagate the result ID of a preceding operation instead of allocation its own *OpVariable* instruction. At the time when the *SPIRVVariable* is created and recorded, there is no information available if the variable will generate a *OpStore* based on other operations (such as assignment). A possible solution

Tabelle 3. Module Sizes

Method	Word Count	Reduction	Factor
Default	1777068	100.00%	1.0
RemoveUnRef	1638604	92.21%	1.085
AllPerf	147680	8.31%	12.033
RemoveUnRef + AllPerf	140576	7.91%	12.641

Instruction word count is accumulated across all 64 shader variants (each SPIR-V instruction has a multiple of 4 byte long words). The average size of a unoptimized SPIR-Vmodule generated by this framework is about 12 times lager than a module optimized by spirv-opt.

could be to emit a *OpNop* instead of OpVariable and store the pointer to the *SPIRVOperation* to replace it when the variable actually needs to be materialized and later remove all *OpNop* instructions when translating the final module.

6 RESTRICTIONS OF *SPEAR*

This work is a proof of concept, many features of GLSL and SPIR-V are not implemented but might be added in the future. Due to the nature of this integrated domain specific language some restrictions apply to syntax and semantics of *SPEAR*.

Variable types in *SPEAR* can be rather long based on their template argument list and abbreviations are only accessible within a *SPIRVProgram* class where the **Assemble** parameter can be inferred from the context. Macro definitions can be used to shorten common types outside the *SPIRVProgram* context class but some template arguments (like *binding*, *set* and *location*) have to be defaulted to specific values.

Conditional *if*, *for* and *while* keywords have to be replaced by macros **If For** and **While** to be translated to SPIR-V. The *switch* statement has not been implemented, *else if* blocks are currently not supported as well.

Caution is advised when mixing regular C++ variables and *SPEAR* variables in computations, *SPEAR* assumes C++ operands are constants and causes runtime valued data to be included in the module, possibly resulting in different modules for each run.

Vector components can not be extracted by reference, making it impossible to assign a value to part of a vector like it is in valid C++.

Type casts on input arguments to *SPIRVVariables* will cause the assembler to misinterpret the data and select the wrong instruction variant. *SPEAR* detects type mismatches and creates *OpConvertXtoY* instructions for primitive types.

In comparison to HLSL, attributes that change execution behaviour (e.g. [numthreads(X, Y, Z)]) have to be expressed using decorates and built-in variables, which is not as elegant.

Recursion is only partially supported: each function call records its instructions to the global stream and because branches on *SPIRVVariable* conditions are always evaluated, the recursion won't be terminated (and assembly will never finish, generating instructions until the stack overflows or the app runs out of memory). However, it is possible to stop the recursion based on regular C++ control flow decisions.

Return statements in any function to be translated change the C++ control-flow but have no other effect than skipping translation of operations following the statement. The user might try to return from a *SPEAR* function with a *return* statement within a conditional block (which is always executed in assemble mode), resulting in dead untranslated code after the statement.

Since the ternary conditional operator ? can not be overloaded in C++, *SPEAR* programmers have to resort to the *Select* function (based on *OpSelect*) to replicate the same behaviour. This is a common problem for integrated domain specific languages where the host language does not allow overloading certain operators.

7 FUTURE WORK

The GLSL 4.50 SPIR-V standard extension exposes a Fused-Multiply-Add (FMA) operation computing expressions of form $a * b + c$ in one GLSLstd450Fma instruction with fewer rounding errors, increasing the overall throughput and precision. The current standard GLSL compiler does not to emit FMA instructions because its rounding behaviour differs from the separate add and mul instructions, the *SPIRVAssembler* could optionally fuse $a * b + c$ expressions: Each Add instruction where one operand is a result of a multiplication which is not consumed by any other instruction can be replaced by a FMA instruction either in a post process or while recording the individual Add instruction. In contrast do GLSL *SPEAR* directly exposes the FMA functionality, giving more control and optimization options to the shader programmer.

The value of these *SPEAR* sided (instruction level) optimizations highly depends on the optimizations that the GPU driver performs while translating SPIR-V to the GPU specific ISA, possibly removing code transformation options by altering the instruction stream in a suboptimal way.

Besides additions and optimizations to the *SPEAR* language itself, algorithms and libraries can be built on top of framework to enrich graphics development with new utilities and features. The D* Symbolic Differentiation Algorithm [1] is a reasonable candidate for further investigation as it is able to compute derivatives at shader compile-time rather than doing it by hand or using third-party software (Mathematica or Maple). Symbolic derivatives can be useful when working with surface normals and curvature (but also Jacobians and velocities) but numeric approximations (forward/central differences) and hardware implementations (ddx/ddy instructions only in fragment shaders) are not feasible. The onePass Differentiation Algorithm [6] (a descendant of D*) is a good candidate for future versions of *SPEAR*, an implementation of which can be found in the DirectX 11 framework.

Bringing more high-level features to shader programming is one of the main goals of *SPEAR* and STL-like iterators are an important step towards this goal. A possible SPIR-V implementation might require to define the end() function as begin() + 1 to force range based for loops to translate the loop body in assemble mode. As a result the regular C++ keyword can be used instead of *For*-macros.

Listing 4. Type polymorphism example

```
// z = a + bi, a = real, b = imag
template <bool Assemble = true, spv::StorageClass Class = spv::StorageClassFunction>
struct SPIRVComplex : public var_t<float2_t, Assemble, Class>
{
  using var_t::var_t;

  template <spv::StorageClass C1>
  SPIRVComplex& operator*=(const SPIRVComplex<Assemble, C1>& _Other)
  {
    // z1z2 = (a1 + b1i)(a2 + b2i) = (a1a2 - b1b2)(a1b2 + b1a2)i
    x = x * _Other.x - y * _Other.y;
    y = x * _Other.y + y * _Other.x;

    return *this;
  }
}
```

Listing 5. Mandelbrot

```
class Mandelbrot : public FragmentProgram
{
public:
  Mandelbrot() : FragmentProgram("Mandelbrot"){};
  RenderTarget OutputColor;

  inline void operator()()
  {
    f32 i = 0.f, max = 100.f;
    complex c(Lerp(-1.f, 1.f, kFragCoord.x / 1600.f), kFragCoord.y / 900.f);
    complex z(0.f, 0.f);
    While(z.Conjugate() < 4.f && i < max)
    {
      z = z * z + c;
      ++i;
    });
    f32 scale = i / max;
    OutputColor = float4(scale, scale, scale, 0.f);
  }
};
```

LITERATUR

[1] 2007. *The D* Symbolic Differentiation Algorithm.* ACM. https://www.microsoft.com/en-us/research/publication/the-d-symbolic-differentiation-algorithm/

[2] Microsoft Corp. [n. d.]. Function linking graph. ([n. d.]). Retrieved January 09, 2018 from https://msdn.microsoft.com/en-us/library/windows/desktop/dn312084%28v=vs.85%29.aspx

[3] Microsoft Corp. [n. d.]. Reference for HLSL. ([n. d.]). Retrieved March 21, 2018 from https://msdn.microsoft.com/en-us/library/windows/desktop/bb509638(v=vs.85).aspx

[4] Tim Foley and Pat Hanrahan. 2011. Spark: Modular, Composable Shaders for Graphics Hardware. *ACM Trans. Graph.* 30, 4, Article 107 (July 2011), 12 pages. https://doi.org/10.1145/2010324.1965002

[5] Larry Gritz. 2009-2017. *The Open Shading Language 1.9 Language Specification.* Technical Report. https://github.com/imageworks/OpenShadingLanguage/blob/master/src/doc/osl-languagespec.pdf

[6] Brian Guenter. 2007. Efficient Symbolic Differentiation for Graphics Applications. *ACM Trans. Graph.* 26, 3, Article 108 (July 2007). https://doi.org/10.1145/1276377.1276512

[7] Yong He, Tim Foley, and Kayvon Fatahalian. 2016. A System for Rapid Exploration of Shader Optimization Choices. *ACM Trans. Graph.* 35, 4, Article 112 (July 2016), 12 pages. https://doi.org/10.1145/2897824.2925923

[8] Yong He, Tim Foley, Teguh Hofstee, Haomin Long, and Kayvon Fatahalian. 2017. Shader Components: Modular and High Performance Shader Development. *ACM Trans. Graph.* 36, 4, Article 100 (July 2017), 11 pages. https://doi.org/10.1145/3072959.3073648

[9] Matt Hoffman. [n. d.]. Unreal Engine 4 Rendering Part 5: Shader Permutations. ([n. d.]). Retrieved January 06, 2018 from https://goo.gl/R2dREi

[10] Epic Games Inc. [n. d.]. Unreal Engine 4 Documentation. ([n. d.]). Retrieved January 06, 2018 from https://docs.unrealengine.com/latest/INT/Programming/Rendering/ShaderDevelopment/

[11] The Khronos Group Inc. [n. d.]. Vulkan® 1.0.71 - A Specification. ([n. d.]). Retrieved March 21, 2018 from https://www.khronos.org/registry/vulkan/specs/1.

Listing 6. Inheritance example: Base class SDFObject

```
template <bool Assemble>
class SDFObject
{
public:
  SDFObject() {};
  virtual ~SDFObject() {};

  // user has to override this function:
  virtual var_t<float, Assemble, spv::StorageClassFunction> Distance(const var_t<float3_t, Assemble, spv::
    StorageClassFunction>& _Point) const = 0;

  template <spv::StorageClass C1>
  inline var_t<float, Assemble, spv::StorageClassFunction> Eval(const var_t<float3_t, Assemble, C1>& _Point)
    const { return Distance(make_intermediate(_Point)); };

  template <spv::StorageClass C1, spv::StorageClass C2>
  inline var_t<float3_t, Assemble, spv::StorageClassFunction> Normal(const var_t<float3_t, Assemble, C1>& _Point,
     const var_t<float, Assemble, C2>& _Epsilon) const
  {
    return ForwardDiffNormal(_Point, _Epsilon, [&](const auto& v) {return this->Eval(v); });
  }
};
```

Listing 7. Inheritance example: derived class SphereSDF

```
template <bool Assemble, spv::StorageClass Class = spv::StorageClassFunction>
class SphereSDF : public SDFObject<Assemble>
{
public:
  var_t<float, Assemble, Class> fRadius;

  template <spv::StorageClass C1>
  SphereSDF(const var_t<float, Assemble, C1> _fRadius = mcvar(1.f)) : fRadius(_fRadius) {}

  inline var_t<float, Assemble, spv::StorageClassFunction> Distance(const var_t<float3_t, Assemble, spv::
    StorageClassFunction>& _Point) const final
  {
    return Dot(_Point, _Point) - fRadius * fRadius;
  }
};
```


0/html/vkspec.html#interfaces-resources

[12] Raun Krisch John Kessenich, Boaz Ouriel. [n. d.]. SPIR-V Specification. ([n. d.]). Retrieved January 03, 2018 from https://www.khronos.org/registry/spir-v/specs/1.2/SPIRV.pdf

[13] Rost Kessenich, Baldwin. [n. d.]. The OpenGL Shading Language. ([n. d.]). Retrieved March 21, 2018 from https://www.khronos.org/registry/OpenGL/specs/gl/GLSLangSpec.4.60.pdf

[14] Maik Klein. [n. d.]. Ash: A very lightweight wrapper around Vulkan. ([n. d.]). Retrieved January 16, 2018 from https://github.com/MaikKlein/ash

[15] Maik Klein. [n. d.]. RLSL- A new shading language. ([n. d.]). Retrieved January 16, 2018 from https://maikklein.github.io/shading-language-part1/

[16] Pierre Krieger. [n. d.]. Vulkano: Safe and rich Rust wrapper around the Vulkan API. ([n. d.]). Retrieved January 16, 2018 from https://github.com/vulkano-rs/vulkano

[17] Michael May. 2015. Design and Implementation of a Shader Infrastructure and Abstraction Layer. (Sept. 2015).

[18] Michael D. McCool, Zheng Qin, and Tiberiu S. Popa. 2002. Shader Metaprogramming. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware (HWWS '02)*. Eurographics Association, Aire-la-Ville, Switzerland, Switzerland, 57–68. http://dl.acm.org/citation.cfm?id=569046.569055

[19] C++ reference. [n. d.]. Parallel execution policy. ([n. d.]). Retrieved March 14, 2018 from http://en.cppreference.com/w/cpp/algorithm/execution_policy_tag_t

[20] G. Sellers. 2016. *Vulkan Programming Guide*. Addison-Wesley Professional. https://books.google.de/books?id=SNKLnQAACAAJ

[21] Anatoliy V. Tomilov. [n. d.]. Aggregate arity. ([n. d.]). Retrieved February 05, 2018 from https://stackoverflow.com/questions/39768517/structured-bindings-width/39784019#39784019

[22] Fabian Wahlster. 2016. Development of a Framework for Shader-Permutation Evaluation. (Sept. 2016).

Listing 8. Creating an operation

```
template <class U, class V, class OpFunc, bool Assemble, spv::StorageClass C1, spv::StorageClass C2,  class T =
    std::invoke_result_t<OpFunc, const U&, const V&>, class ...Ops >
inline var_t<T, Assemble, spv::StorageClassFunction> make_op(const var_t<U, Assemble, C1>& l, const var_t<V,
    Assemble, C2>& r, const OpFunc& _OpFunc, const EOpTypeBase _kOpTypeBase, const Ops ..._Ops)
{
  auto var = var_t<T, Assemble, spv::StorageClassFunction>(TIntermediate());

  if constexpr(Assemble == false)
  {
    var.Value = _OpFunc(l.Value, r.Value); // evaluate on host
  }
  else // Assemble
  {
    LoadVariables(l, r); // generate accesschain, materialze decorates, opload result id

    spv::Op kOpCode = (spv::Op)OpTypeDeciderEx<T, U, V>(_Ops...);

    SPIRVOperation NewOp(kOpCode, var.uTypeId, // result type
    NewOp.AddIntermediate(l.uResultId); // operand1
    NewOp.AddIntermediate(r.uResultId); // operand2

    var.uResultId = GlobalAssembler.AddOperation(NewOp);
  }

  return var;
}
```

Listing 9. If translation

```
template<bool Assemble, class LambdaFunc, spv::StorageClass Class>
inline BranchNode<Assemble> IfNode(const var_t<bool, Assemble, Class>& _Cond, const LambdaFunc& _Func, const spv::SelectionControlMask _kMask = spv::SelectionControlMaskNone)
{
  BranchNode<Assemble> Node;

  if constexpr(Assemble)
  {
    _Cond.Load();

    GlobalAssembler.AddOperation(SPIRVOperation(spv::OpSelectionMerge,
      {
        SPIRVOperand(kOperandType_Intermediate, HUNDEFINED32), // merge id
        SPIRVOperand(kOperandType_Literal, (const uint32_t)_kMask) // selection class
      }), &Node.pSelectionMerge);

    GlobalAssembler.AddOperation(SPIRVOperation(spv::OpBranchConditional, SPIRVOperand(kOperandType_Intermediate, _Cond.uResultId)), &Node.pBranchConditional);
  }
  else
  {
    Node.bCondition = _Cond.Value;
  }

  if (_Cond.Value || Assemble)
  {
    if constexpr(Assemble)
    {
      // add true label
      Node.pBranchConditional->AddIntermediate(GlobalAssembler.AddOperation(SPIRVOperation(spv::OpLabel)));
    }

    _Func(); // evaluate on host

    if constexpr(Assemble)
    {
      // end of then block
      GlobalAssembler.AddOperation(SPIRVOperation(spv::OpBranch), &Node.pThenBranch);

      const uint32_t uFalseLableId = GlobalAssembler.AddOperation(SPIRVOperation(spv::OpLabel));
      Node.pThenBranch->AddIntermediate(uFalseLableId);

      // use false label as merge label
      Node.pSelectionMerge->GetOperands().front().uId = uFalseLableId;
      Node.pBranchConditional->AddIntermediate(uFalseLableId);
    }
  }

  return Node;
}
```