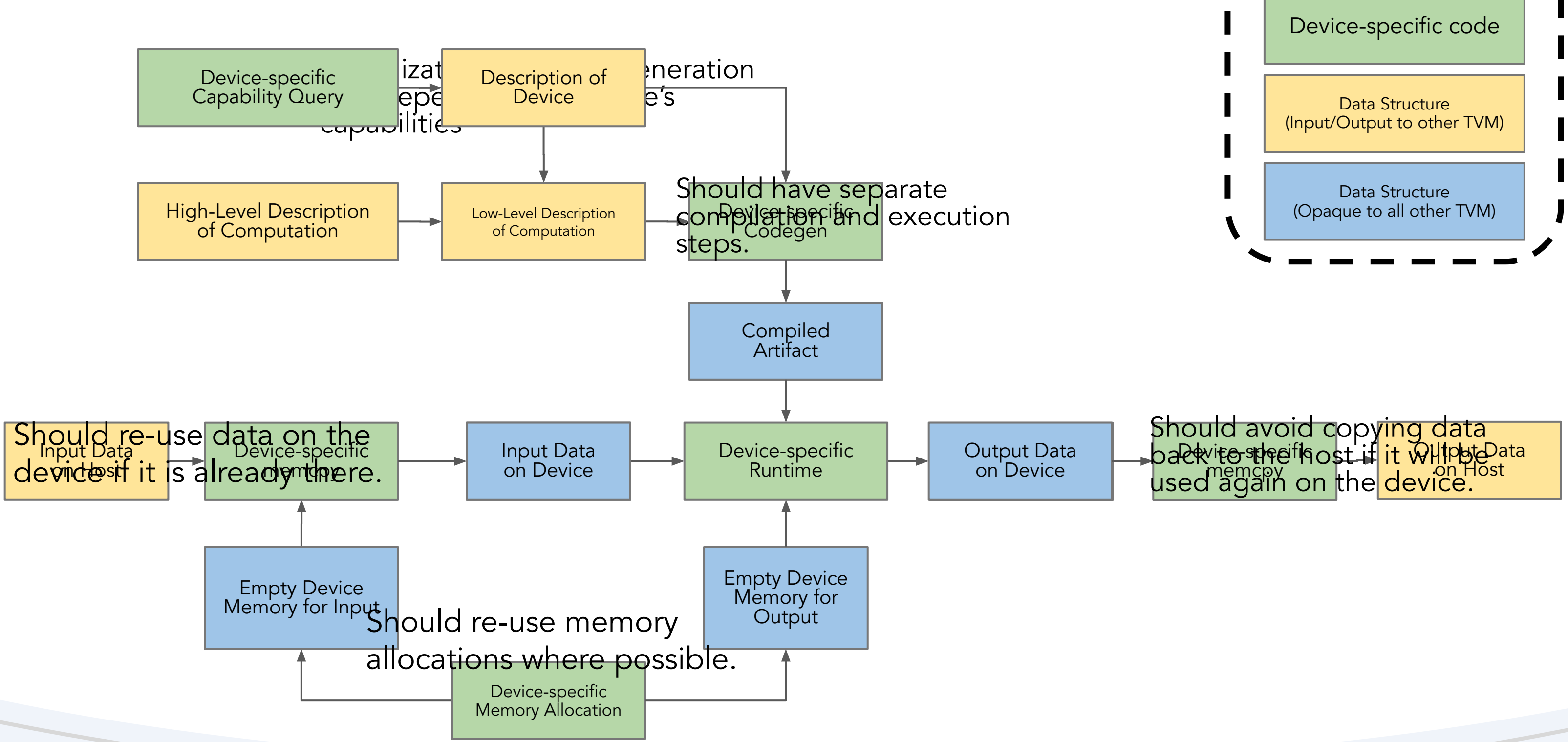
TVM Bootcamp - Backends

Eric Lunderberg

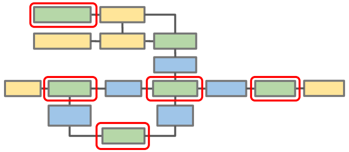
Overview

- TVM supports several backend frameworks (e.g. CUDA, Vulkan, ROCm)
- Each backend must conform to a standard interface when interacting with the TVM remainder of the framework.
- This walkthrough is aimed at developers who want to implement/extend support for additional devices, and aren't necessarily going to be adding compiler features/optimizations.
    - Everything outside of the device-specific code will be treated as opaque.
- All descriptions and links refer to commit 6c8ed60ec, the most recent commit on main as of 2021-12-03.

TVM Compilation/Runtime Flow



DeviceAPI

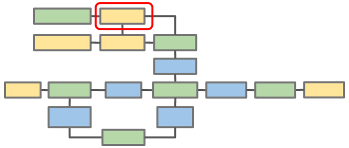


- Main interaction point with the physical device. (Additional details are on later slides.)
- Each DeviceAPI subclass must have a function to return the global object Implemented as a singleton for each device type.
  - e.g. VulkanDeviceAPI::Global(), CUDADeviceAPI::Global()
- Register the global function as `"device_api.$NAME"`
  - e.g. `TVM_REGISTER_GLOBAL("device_api.cuda")`
  - Global function is called from DeviceAPIManager::GetAPI to interact with a device.
- Add an entry to the TVMDeviceExtType enum representing the new DeviceAPI.
  - The value should be an unused value greater than DLDeviceType::kDLExtDev, but less than DeviceAPIManager::kMaxDeviceAPI.
  - This value is used to represent the device type both internally to TVM and when interacting with types defined in dlpack.
- Add the conversion from enum value to string in tvm::runtime::DeviceName. This string representation should match the name used earlier in TVM_REGISTER_GLOBAL.
- Add the conversion from enum value to string in tvm.runtime.Device.MASK2STR.
- Add the conversion from string to enum value in tvm.runtime.Device.STR2MASK.

Device-specific Capability Query	DeviceAPI::GetAttr DeviceAPI::GetTargetProperty
Device-specific Memory Allocation	DeviceAPI::AllocDataSpace DeviceAPI::AllocWorkspace
Device-specific memcpy	DeviceAPI::CopyDataFromTo
Device-specific Runtime	runtime::PackedFunc

OctoML

tvm::Target Object

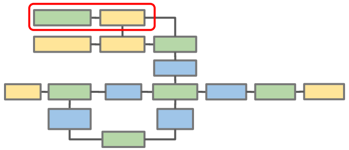


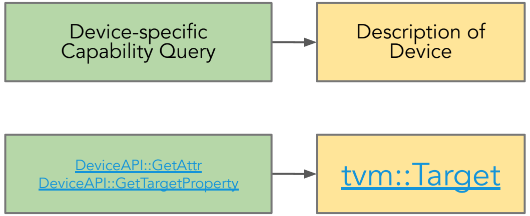
- Contains properties about the device (e.g. warp size), and the code generator to be used (e.g. llvm/nvptx/cuda).
- Can be serialized/deserialized for remote
- New targets should be registered with the TVM_REGISTER_TARGET_KIND macro.
  - Defines the mapping from compile-time code generator to runtime device. (e.g. TVM_REGISTER_TARGET_KIND("cuda", kDLCUDA) defines the "cuda" code generator, which should run on a device of type kDLCUDA.)
  - Defines attributes for the target. (e.g. .add_attr_option<Integer>("thread_warp_size", Integer(32)) defines the "thread_warp_size" option, with default value of 32.)
- Multiple code generators may be associated with the same runtime device.
  - e.g. The "llvm" and "c" code generators both run on the CPU.

Description of Device

tvm::Target

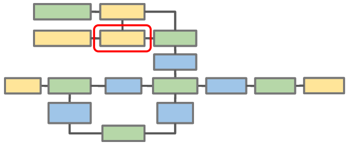
Device Capability Query





- Typically stored into a tvm::Target object.
- DeviceAPI::GetAttr
    - Earlier API, looks up a device attribute listed in the enum DeviceAttrKind.
    - Most cases that require device information should look up the cached values in the tvm::Target object.
- DeviceAPI::GetTargetProperty
    - More recent API, looks up a device attribute by string.
    - Called during construction of a Target object, if the `"-from_device=$DEVICE_NUM"` property is given.
        - e.g. `target = tvm.target.Target("vulkan -from_device=0")`

Low-Level TIR



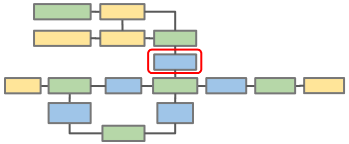
- Some construct's in TVM's IR are high-level structures that are removed during the lowering process. Device-specific codegen only needs to handle the low-level structures that remain after the lowering has completed.
- Allowed in low-level TIR
  - tir::PrimFunc function definitions.
  - Control flow (e.g. ForNode, IfThenElseNode)
  - Variable definition and access (e.g. LetNode, VarNode)
  - Computations (e.g. AddNode, SubNode)
  - Memory allocation and access (e.g. AllocateNode, StoreNode[1], LoadNode[1])
- Not allowed in low-level TIR
  - relay::Function function definitions.
  - TE-specific nodes such as ProducerLoad/ProducerStore.
  - Attributes that indicate transformations to be performed. (e.g. AttrStmtNode::attr_key set to attr::double_buffer_scope)
  - Calls to TVM-specific built-in functions. (e.g. CallNode::op set to builtin::tvm_call_packed)

Low-Level Description of Computation

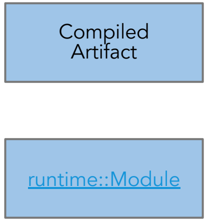
IRModule containing only Low-Level TIR

1. LoadNode and StoreNode may be deprecated in the near future, see RFC#0039 for details.

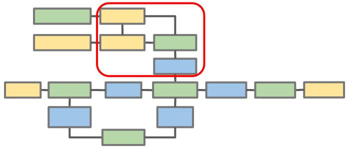
runtime::Module



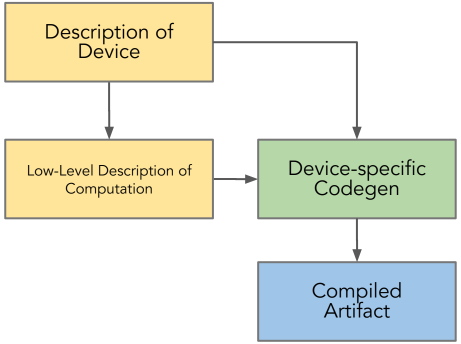
- A container with a GetFunction method, mapping from function name to runtime::PackedFunc.
- Exact contents vary for each type of device, depending on what will be most useful during runtime.
  - e.g. LLVM codegen contain the compiled llvm::Module.
  - e.g. Vulkan generates a binary SPIR-V shader.
- For cross-compiling or distribution of models, should be able to read/write to disk.
  - Override virtual function ModuleNode::SaveToFile.
  - Register a function named “runtime.module.loadfile_$EXTENSION” (e.g. runtime.module.loadfile_so). This is called from runtime::Module::LoadFromFile, after using the file extension to determine which loader to run.

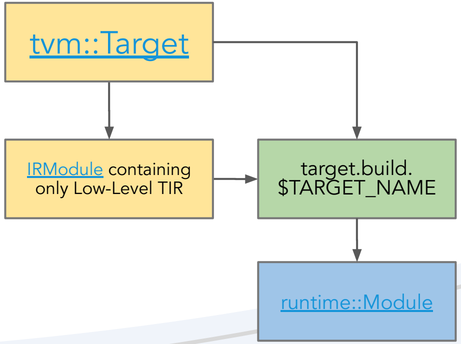


Code Generation

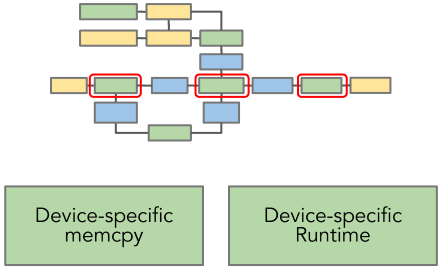


- From low-level TIR, build a runtime::Module that will be used at runtime.
  - Function signature
    ```
    runtime::Module BuildCUDA(IRModule mod, Target target)
    ```
  - Must be registered as "target.build.$TARGET_NAME"
    - e.g. TVM_REGISTER_GLOBAL("target.build.nvptx").set_body_typed(BuildNVPTX);
    - Note: $TARGET_NAME must be the name of the target code generator, previously used in the TVM_REGISTER_TARGET_KIND macro, not the name of the device. In some cases these are identical (e.g. “cuda” target uses “target.build.cuda” codegen, then runs on “cuda” device), but it is not always the case (e.g. “llvm” target uses “target.build.llvm” codegen, then runs on “cpu” device).
  - Called from codegen::Build
- Typical flow for a code generator.
  1. Loop over all functions in IRModuleNode::functions
  2. Generate a function signature from PrimFuncNode::params and PrimFuncNode::ret_type.
  3. Generate a function body from PrimFuncNode::body. Iteration over the function body can be done by writing a class that inherits from both ExprFunctor<void(const PrimExpr&)> and StmtFunctor<void(const Stmt&)>, then implementing handlers for the overloaded ExprFunctor::VisitExpr_ and StmtFunctor::VisitStmt_ methods.
  4. If the output code generated requires it, perform any additional compilation steps.
     - e.g. target.build.cuda produces CUDA source code, which is then compiled.
     - e.g. target.build.vulkan produces binary SPIR-V bytecode, which does not require additional compilation.
  5. Wrap the build artifacts in a subclass of runtime::ModuleNode



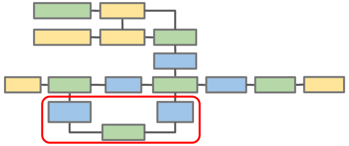


Execution Streams, Synchronization

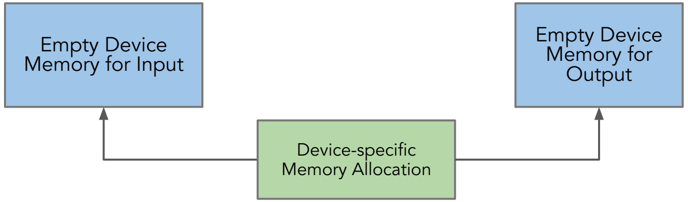


- Streams represent separate queues of work, which execute independently.
  - In TVM, streams can contain either data transfers or computations.
- Operations
  - DeviceAPI::CreateStream - Allocate a new stream of execution. TVM treats the return value as an opaque pointer, and passes it unmodified into other stream-related functions.
  - DeviceAPI::StreamSync - Synchronize between host and device. Host should wait until all operations queued to that device/stream execution have completed.
  - DeviceAPI::SyncStreamFromTo - Add a synchronization point between two streams.
  - DeviceAPI::FreeStream - Deallocate the stream.
- For runtimes that are synchronous (e.g. CPU), CreateStream should return nullptr, and stream manipulation functions should be no-ops.
- For runtimes that support only a single stream of execution (e.g. TVM's Vulkan runtime, as of end-of-year 2021), CreateStream should return nullptr, StreamSync should synchronize that single stream with the host, and all other stream manipulation functions should be no-ops.

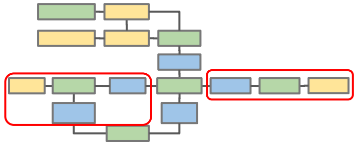
Memory Allocation



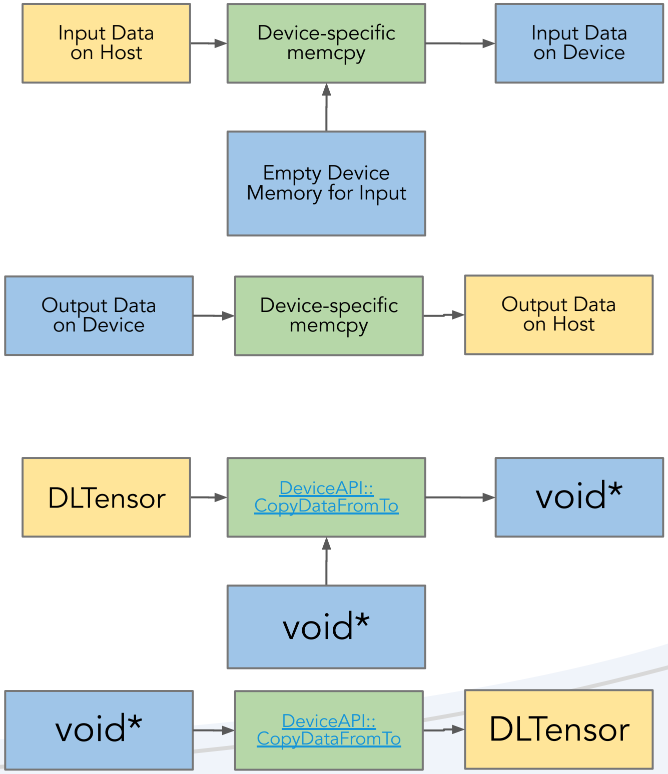
- Data space
  - Large, infrequently allocated
  - Allocated with DeviceAPI::AllocDataSpace.
  - Free with DeviceAPI::FreeDataSpace
  - May be triggered by an end user
    - C API, TVMDeviceAllocDataSpace
    - Python, tvm.nd.array
  - May be allocations internal to a Relay graph.
- Work space
  - Like data space, but with usage patterns typical of a stack. Devices may implement special handling for improved performance (e.g. OpenCL's AllocWorkspace, which uses runtime::WorkspacePool), but it isn't required.
  - If not implemented by a subclass, will fall back to using data space.
  - Allocate with DeviceAPI::AllocWorkspace
  - Free with DeviceAPI::FreeWorkspace
- The return value is a void*, but is treated as an opaque pointer by everything outside of the device-specific code.
  - e.g. TVM's Vulkan runtime must track use of both VkBuffer and VkDeviceMemory, so VulkanDeviceAPI::AllocDataSpace returns a structure that contains both handles.
  - e.g. CUDADeviceAPI::AllocDataSpace does not require any additional bookkeeping, and returns the result of cudaMalloc.



Memory Transfer

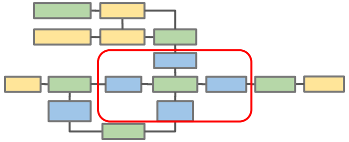


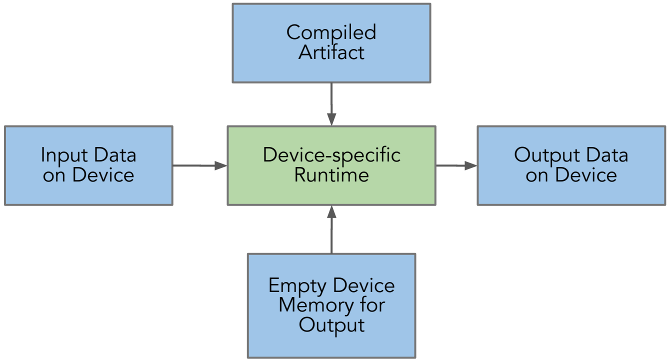
- DeviceAPI::CopyDataFromTo copies data in either direction between the host and the device, or between multiple devices.
  - public DeviceAPI::CopyDataFromTo, whose implementation is a wrapper around the device-specific overload and takes DLTensor objects as arguments.
  - protected DeviceAPI::CopyDataFromTo, to be implemented by subclasses, takes arguments as described below.
- Arguments “dev_from” and “dev_to” are DLDevice objects, and indicate which the device holds the source and destination locations.
- Arguments “from” and “to” give the source and destination memory locations.
  - These will be C-style memory pointers if the Device::device_type for the corresponding device argument is kDLCPU.
  - Otherwise, these will be pointers generated by AllocDataSpace or AllocWorkspace.
- If the TVMStreamHandle passed in is non-null, the copy should be queued onto that execution stream and performed asynchronously. Otherwise, the copy should be performed synchronously.

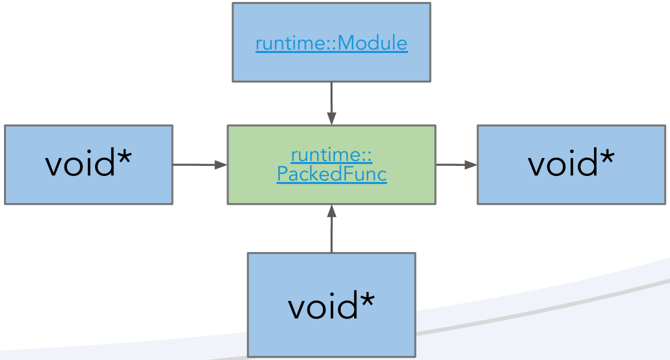


Kernel Execution

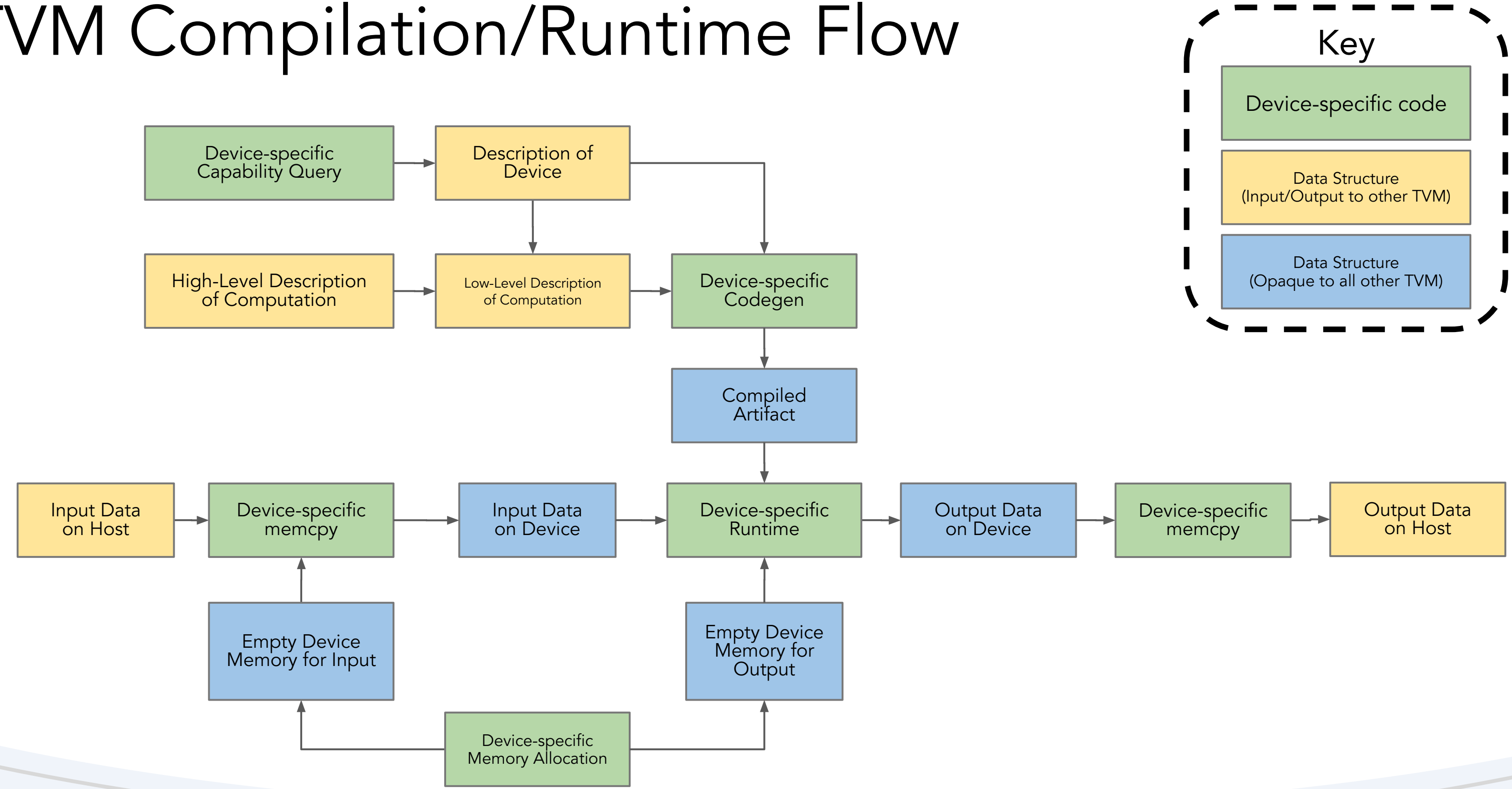
- Uses runtime::PackedFunc, a wrapper around std::function and TVM's primary FFI structure.
- A PackedFunc generated from the runtime::ModuleNode::GetFunction is called.
  - Function arguments give the array inputs and any scalar parameters.
  - DeviceAPI::SetDevice and DeviceAPI::SetStream determine the execution stream for the computation.
- Things the device-specific code should do
  - Load the stored data if necessary.
    - e.g. cuModuleLoadData for CUDA
    - e.g. vkCreateShaderModule for Vulkan
  - Launch the shader using the buffers given.
    - e.g. execute a function pointer for LLVM modules, cuLaunchKernel for CUDA
    - e.g. vkQueueSubmit for Vulkan
- Things the device-specific code doesn't need to do
  - Argument/array type checks.
    - Already inserted as part of tir::transform::MakePackedAPI, part of the compiled code.
  - Transfer data to/from the host.
    - Already handled by DeviceAPI::CopyDataFromTo.
  - Wait for execution to complete, unless strictly necessary.
    - Synchronization handled by DeviceAPI::StreamSync.
    - Rare exception: Device-specific resource conflict, such as a Vulkan descriptor set already being in use.



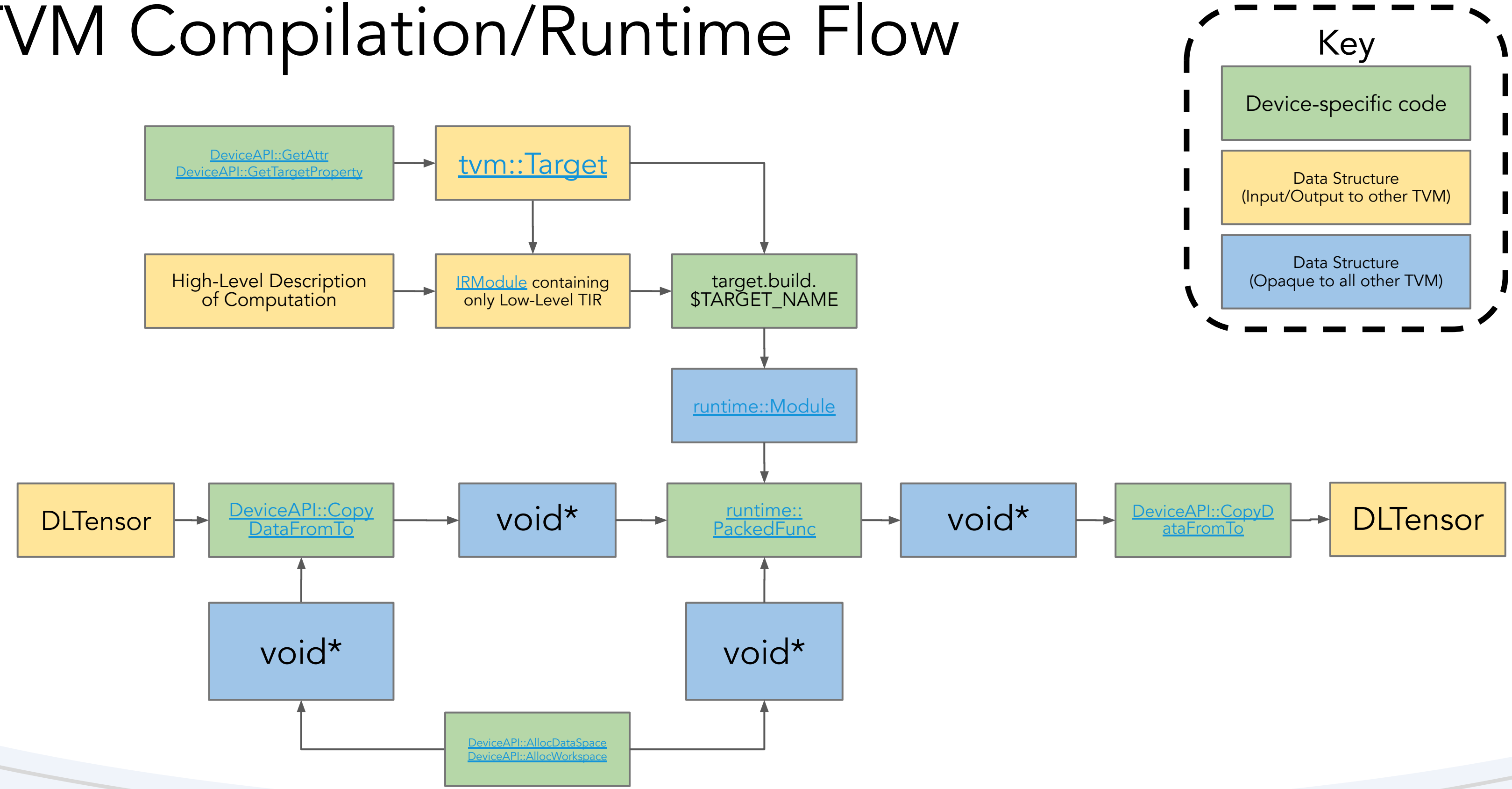




TVM Compilation/Runtime Flow



TVM Compilation/Runtime Flow



Potential Future Improvements

- Low-level runtime-specific test suite
  - Quantify support of features across backends.
  - Easier debugging after a test failure. High-level end-to-end tests of models can have backend-specific failures, which require a non-trivial amount of effort to track down to a root cause.
  - Provide a roadmap for additional runtimes to be implemented, by defining which low-level features are required.
- Low-level TIR validation
  - Would be an explicit check on the TIR being low-level, rather than relying on error checks within the code generation.