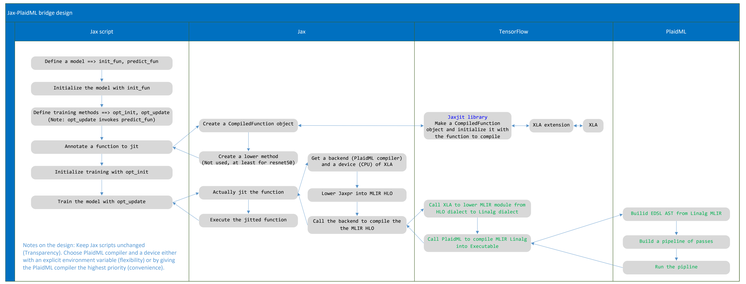

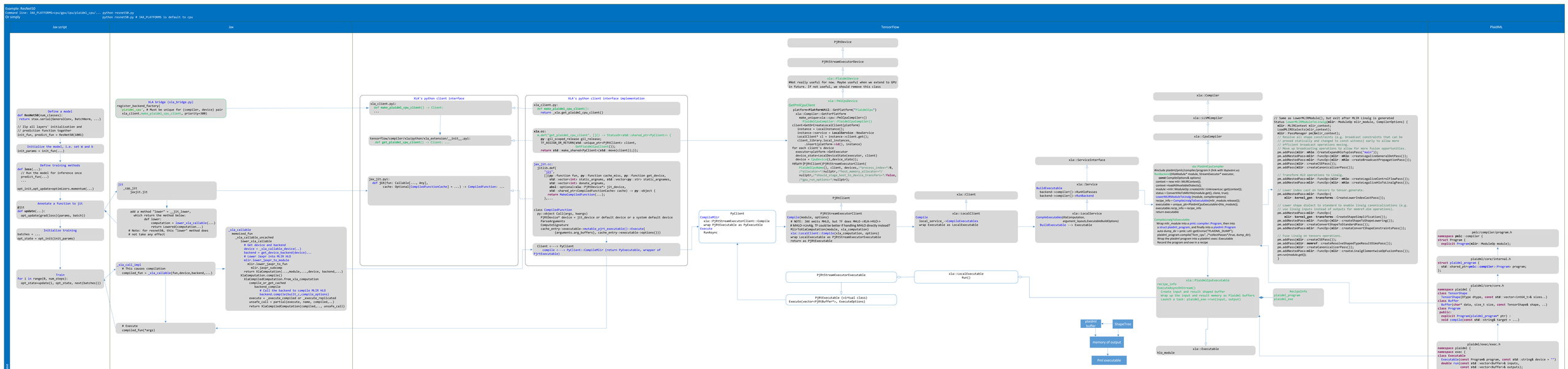
Example: ResNet50
Command line: JAX_PLATFORMS=cpu/gpu/ipu/plaidml_cpu/... python resnet50.py
Or simply python resnet50.py # JAX_PLATFORMS is default to cpu
Jax script
Jax
Tensorflow
PlaidML
Define a model
def ResNet50(num_classes):
return stax.serial(GeneralConv, BatchNorm, ...)
// Zip all layers' initialization and
// prediction function together
init_fun, predict_fun = ResNet50(10001)
Initialize the model, i.e. set W and b
init_params = init_fun(...)
Define training methods
def loss(...):
// Run the model for inference once
predict_fun(...)
...
opt_init,opt_update=optimizers.momentum(...)
Annotate a function to jit
@jit
def update(...):
opt_update(grad(loss)(params, batch))
Initialize training
batches = ...
opt_state = opt_init(init_params)
Train
for i in range(0, num_steps):
opt_state=update(i, opt_state, next(batches)))
XLA bridge (xla_bridge.py)
register_backend_factory(
'plaidml_cpu', # Must be unique for (compiler, device) pair
xla_client.make_plaidml_cpu_client, priority=300)
jit
_cpp_jit
jaxjit.jit
add a method "lower" = _jit_lower,
which return the method below:
def lower:
computation = lower_xla_callable(...)
return lowered(computation,...)
Note: for resnet50, this "lower" method does
not take any effect
_xla_call_impl
# This causes compilation
compiled_fun = _xla_callable(fun,device,backend,...)
_xla_callable
memoized_fun
_xla_callable_uncached
lower_xla_callable
# Get device and backend
device = _xla_callable_device(...)
backend = get_device_backend(device)...
# Lower Jaxpr into MLIR HLO
mlir.lower_jaxpr_to_module
mlir.lower_jaxpr_to_fun
lower_jaxpr_subcomp
return XlaComputation(_ir.get_module,...,device, backend,...)
XlaComputation.compile()
XlaCompiledComputation.from_xla_computation
compile_or_get_cached
backend_compile
# Call the backend to compile MLIR HLO
backend.compile(built_c,compile_options)
execute = _execute_compiled or _execute_replicated
unsafe_call = partial(execute, name, compiled,...)
return XlaCompiledComputation(compiled,..., unsafe_call)
# Execute
compiled_fun(*args)
XLA's python client interface
xla_client.py:
def make_plaidml_cpu_client() -> Client:
...
tensorflow/compiler/xla/python/xla_extension/__init__.pyi:
def get_plaidml_cpu_client() -> Client: ...
jax_jit.cc:
jitlib.def(
"jit",
[](py::function fun, py::function cache_miss, py::function get_device,
std::vector<int> static_argnums, std::vector<py::str> static_argnames,
std::vector<int> donate_argnums,
absl::optional<xla::PjRtDevice*> jit_device,
std::shared_ptr<CompiledFunctionCache> cache) -> py::object {
return MakeCompiledFunction(...);
},...
class CompiledFunction
py::object Call(args, kwargs)
PjRtDevice* device = jit_device or default device or a system default device
ParseArguments
ComputeSignature
cache_entry->executable->mutable_pjrt_executable().->Execute(
(arguments.arg_buffers), cache_entry->executable->options())))
Client <---> PyClient
compile <---> PyClient::CompileMlir (return PyExecutable, wrapper of PjRtExecutable)
XLA's python client interface implementation
xla_client.py:
def make_plaidml_cpu_client():
return _xla.get_plaidml_cpu_client()
xla.cc:
m.def("get_plaidml_cpu_client", []() -> StatusOr<std::shared_ptr<PyClient>> {
py::gil_scoped_release gil_release;
TF_ASSIGN_OR_RETURN(std::unique_ptr<PjRtClient> client,
GetPlaidMlCpuClient());
return std::make_shared<PyClient>(std::move(client));});
GetPlaidMlCpuClient():
device_state=LocalDeviceState(executor, client)
device = CpuDevice(id,device_state);
return [PjRtClient]PjRtStreamExecutorClient(
"PlaidMlCpuHost", client, devices,/*process_index=*/0,
/*allocator=*/nullptr,/*host_memory_allocator=*/
nullptr,/*should_stage_host_to_device_transfers=*/false,
/*gpu_run_options=*/nullptr);
PjRtDevice
PjRtStreamExecutorDevice
xla::PlaidMlDevice
#Not really useful for now. Maybe useful when we extend to GPU in future. If not useful, we should remove this class
xla::PlaidMlCpuDevice
GetPlaidMlCpuClient
platform=PlatformUtil::GetPlatform("PlaidMlCpu")
xla::Compiler::GetForPlatform
make_unique<xla::cpu::PlaidMlCpuCompiler()
PlaidMlCpuCompiler::PlaidMlCpuCompiler()
client=GetOrCreateLocalClient(platform)
instance = LocalInstance();
instance->service = LocalService::NewService
LocalClient* cl = instance->client.get();
client_library.local_instances.
insert(platform->id(), instance)
for each local device ordinal
executor=platform->GetExecutor
device_state=LocalDeviceState(executor, client)
device = CpuDevice(id,device_state);
return [PjRtClient]PjRtStreamExecutorClient(
"PlaidMlCpuHost", client, devices,/*process_index=*/0,
/*allocator=*/nullptr,/*host_memory_allocator=*/
nullptr,/*should_stage_host_to_device_transfers=*/false,
/*gpu_run_options=*/nullptr);
PjRtClient
PjRtStreamExecutorClient
Compile(module, options)
# NOTE: JAX emits MHLO, but TF does MHLO->XLA-HLO->
# StablehloToXla() TF could be better if handling MHLO directly instead?
mlir::LocalClient::CompileAndInfer(xla_computation,
mlir::LocalClient::Compile(xla_computation, options)
wrap LocalExecutable as PjRtStreamExecutorExecutable
return as PjRtExecutable
PyClient
CompileMlir
xla::PjRtStreamExecutorClient::Compile
wrap PjRtExecutable as PyExecutible
Execute
RunAsync
xla::Client
xla::LocalClient
Compile
local_service_->CompileExecutables
wrap Executable as LocalExecutable
xla::Compiler
xla::LLVMCompiler
xla::CpuCompiler
xla::ServiceInterface
xla::Service
BuildExecutable
backend->compiler()->RunHloPasses
backend->compiler()->RunBackend
xla::LocalService
CompileExecutables(XlaComputation,
argument_layouts,ExecutableBuildOptions)
BuildExecutable --> Executable
xla::PlaidMlCpuCompiler
xla::PlaidMlCpuExecutable
#include plaidml/exec/exec.h plaidml/exec/exec.pb.h
#include "tensorflow/compiler/xla/service/llvm_ir/llvm_util.h"
#include "tensorflow/core/platform/logging.h"
#include "tensorflow/core/platform/threadpool.h"
#include "tensorflow/core/platform/cpu_info.h"
namespace {
namespace xla {
namespace cpu {
namespace {
namespace {
// Sanity check.
CHECK_EQ(1, num_partitions());
CHECK_GE(num_replicas(), 1);
CHECK_LE(num_replicas(), hlo_module->config().replica_count());
// Create a PlaidMlExecutable.
int64 const* const* unused = nullptr;
bool const* const* unused2 = nullptr;
int64 replicas = num_replicas();
int64 devices = 1;
int64 partitions = 1;
int64 partition = 0;
int8 broadcast_one = 1;
int8 unique_indices = 0;
int64 num_args = parameter_count();
int64 num_outputs = 1;
int64 ninsts = 0;
int64 noutputs = 0;
int64 ninstr_ref = 0;
int64 noutputs_ref = 0;
int64 nargs = 0;
int64 ninstr_fusion = 0;
int64 noutputs_fusion = 0;
int64 nargs_fusion = 0;
int64 ninstr_nested_fusion = 0;
int64 noutputs_nested_fusion = 0;
int64 nargs_nested_fusion = 0;
int64 ninstr_while = 0;
int64 noutputs_while = 0;
int64 nargs_while = 0;
int64 ninstr_conditional = 0;
int64 noutputs_conditional = 0;
int64 nargs_conditional = 0;
int64 ninstr_call = 0;
int64 noutputs_call = 0;
int64 nargs_call = 0;
int64 ninstr_custom_call = 0;
int64 noutputs_custom_call = 0;
int64 nargs_custom_call = 0;
int64 ninstr_infeed = 0;
int64 noutputs_infeed = 0;
int64 nargs_infeed = 0;
int64 ninstr_outfeed = 0;
int64 noutputs_outfeed = 0;
int64 nargs_outfeed = 0;
int64 ninstr_send = 0;
int64 noutputs_send = 0;
int64 nargs_send = 0;
int64 ninstr_recv = 0;
int64 noutputs_recv = 0;
int64 nargs_recv = 0;
int64 ninstr_send_done = 0;
int64 noutputs_send_done = 0;
int64 nargs_send_done = 0;
int64 ninstr_recv_done = 0;
int64 noutputs_recv_done = 0;
int64 nargs_recv_done = 0;
int64 ninstr_all_reduce = 0;
int64 noutputs_all_reduce = 0;
int64 nargs_all_reduce = 0;
int64 ninstr_all_gather = 0;
int64 noutputs_all_gather = 0;
int64 nargs_all_gather = 0;
int64 ninstr_all_to_all = 0;
int64 noutputs_all_to_all = 0;
int64 nargs_all_to_all = 0;
int64 ninstr_collective_permute = 0;
int64 noutputs_collective_permute = 0;
int64 nargs_collective_permute = 0;
int64 ninstr_collective_broadcast = 0;
int64 noutputs_collective_broadcast = 0;
int64 nargs_collective_broadcast = 0;
int64 ninstr_replica_id = 0;
int64 noutputs_replica_id = 0;
int64 nargs_replica_id = 0;
int64 ninstr_partition_id = 0;
int64 noutputs_partition_id = 0;
int64 nargs_partition_id = 0;
int64 ninstr_rng = 0;
int64 noutputs_rng = 0;
int64 nargs_rng = 0;
int64 ninstr_rng_get_and_update_state = 0;
int64 noutputs_rng_get_and_update_state = 0;
int64 nargs_rng_get_and_update_state