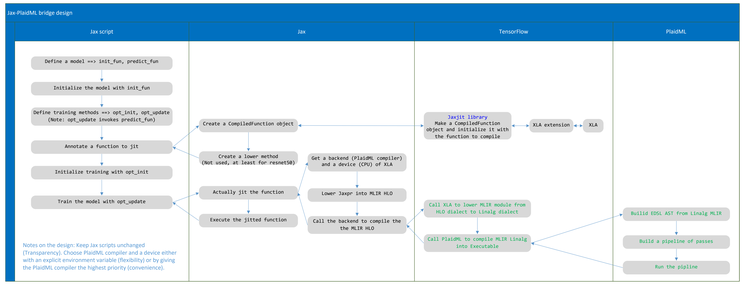
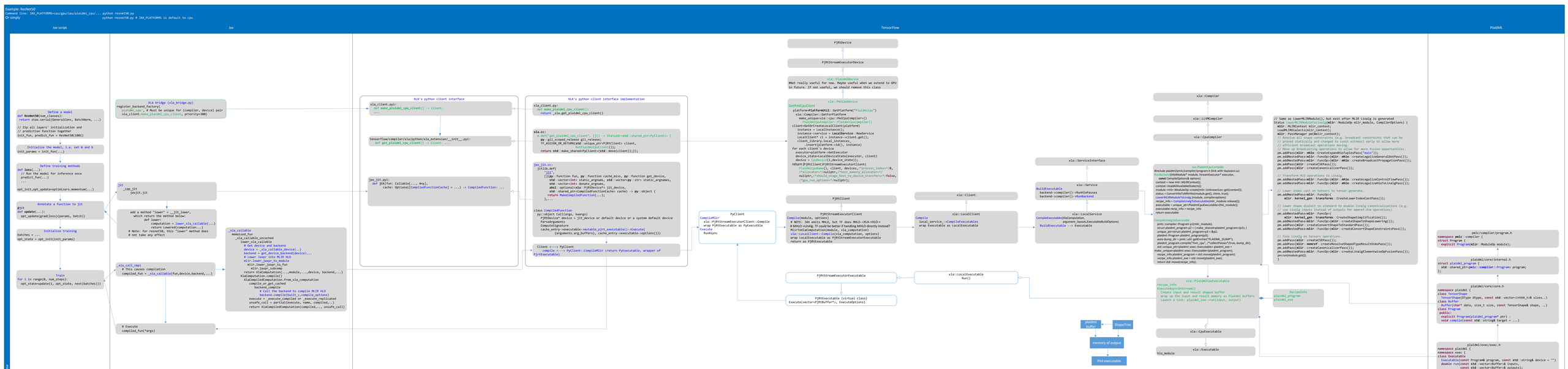

Jax-PlaidML bridge design
Jax script
Jax
TensorFlow
PlaidML
Define a model ==> init_fun, predict_fun
Initialize the model with init_fun
Define training methods ==> opt_init, opt_update (Note: opt_update invokes predict_fun)
Annotate a function to jit
Initialize training with opt_init
Train the model with opt_update
Create a CompiledFunction object
Create a lower method (Not used, at least for resnet50)
Get a backend (PlaidML compiler) and a device (CPU) of XLA
Actually jit the function
Lower Jaxpr into MLIR HLO
Execute the jitted function
Call the backend to compile the the MLIR HLO
Jaxjit library
Make a CompiledFunction object and initialize it with the function to compile
XLA extension
XLA
Call XLA to lower MLIR module from HLO dialect to linalg dialect
Call PlaidML to compile MLIR linalg into Executable
Build DSSL AST from linalg MLIR
Build a pipeline of passes
Run the pipline
Notes on the design: Keep Jax scripts unchanged (Transparency). Choose PlaidML compiler and a device either with an explicit environment variable (flexibility) or by giving the PlaidML compiler the highest priority (convenience)


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
ir.UnitAttr.get()
return XlaComputation(,,,,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));});
PyClient
CompileMlir
xla::PjRtStreamExecutorClient::Compile
wrap PjRtExecutable as PyExecutible
Execute
RunAsync
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
GetPmlCpuClient
platform=Platformutil::GetPlatform("PlaidmlCpu")
xla::Compiler::GetForPlatform
make_unique<xla::cpu::PmlCpuCompiler()
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:
executor=platform->GetExecutor
device_state=LocalDeviceState(executor, client)
device = CpuDevice(i,device_state));
return (PjRtClient)PjRtStreamExecutorClient(
PlaidMlCpuName(), 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->
# XlaHloCatalog TF could be better if handling MHLO directly instead?
mlir::LocalClient::CompileAndInfer(xla_computation,
build_options, xla_computation, options)
wrap LocalExecutable as PjRtStreamExecutorExecutable
return as PjRtExecutable
PjRtStreamExecutorExecutable
PjRtExecutable (virtual class)
Execute(vector<PjRtBuffer*>, ExecuteOptions)
xla::Client
xla::LocalClient
Compile
local_service_->CompileExecutables
wrap Executable as LocalExecutable
xla::LocalExecutable
Run()
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
#include plaidml/pmlc/compiler/program.h (link with libplaidml.so)
RunHloPasses(std::unique_ptr<HloModule> module,
const CompileOptions& options)
context = new mlir::MLIRContext();
context->loadAllAvailableDialects();
module.emplace(HloModuleOp::create(mlir::UnknownLoc::get(context));
status = ConvertHloToMlirHlo(module.get(), clone, true);
LowerMLIRModuleToLinalg(module, compileropts)
recipe_info = CompileLinalgToExecutable(mlir_module.release());
executable = unique_ptr<PlaidmlCpuExecutable>(hlo_module);
executable.recip_info = recipe_info
return executable
CompileLinalgToExecutable
pmlc::compiler::Program p(mlir_module);
strut plaidml_program p2 = (make_shared<plaidml_program>(p));
unique_ptr<strut plaidml_program>9 = &p2;
plaidml::Program plaidml_program(p);
auto dump_dir = pmlc::util::getEnvVar("PLAIDML_DUMP");
std::string dump_dir_str(dump_dir ? dump_dir : "/tmp/plaidml_dump_dir);
std::unique_ptr<plaidml::exec::Executable> plaidml_exe =
make_unique<plaidml::exec::Executable>(plaidml_program);
recipe_info.plaidml_program = std::move(plaidml_program);
recipe_info.plaidml_exe = std::move(plaidml_exe);
return std::move(recipe_info);
xla::PlaidMlCpuExecutable
recipe_info_
ExecuteAsyncOnStream()
Create input and result shaped buffer
Wrap up the input and result memory as Plaidml buffers
Launch a task! plaidml_exe->run(input, output)
plaidml buffer
ShapeTree
memory of output
Pml executable
xla::CpuExecutable
xla::Executable
hlo_module
// Same as LowerMLIRModule(), but exit after MLIR Linalg is generated
Status LowerMLIRModuleToLinalg(mlir::ModuleOp mlir_module, CompilerOptions) {
mlir::MLIRContext mlir_context;
LoadMLIRDialects(mlir_context);
mlir::PassManager pm(&mlir_context);
// Resolve all shape constraints (e.g. broadcast constraints that can be
// proved statically and changed to const witness) early to allow more
// efficient broadcast operations moving
// inside each function within the module.
pm.addPass(mlir::mhlo::CreateEarlyBroadcastPass("main"));
pm.addNestedPass<mlir::FuncOp>(mlir::mhlo::createLegalizeGeneralDotPass());
pm.addNestedPass<mlir::FuncOp>(mlir::mhlo::createBroadcastPropagationPass());
pm.addPass(mlir::createCSEPass());
pm.addPass(mlir::createCanonicalizerPass());
// Transform HLO operations to Linalg.
pm.addNestedPass<mlir::FuncOp>(mlir::mhlo::createLegalizeControlFlowPass());
pm.addNestedPass<mlir::FuncOp>(mlir::mhlo::createLegalizeToLinalgPass());
// Lower index cast on tensors to tensor.generate.
pm.addNestedPass<mlir::FuncOp>(
mlir::kernel_gen::transforms::CreateLowerIndexCastPass());
// Lower shape dialect to standard to enable linalg canonicalizations (e.g.
// use linalg inputs instead of outputs for memref.dim operations).
pm.addNestedPass<mlir::FuncOp>(
mlir::kernel_gen::transforms::CreateShapeSimplification());
pm.addNestedPass<mlir::FuncOp>(mlir::mlir::createShapeToShapeLowering());
pm.addPass(mlir::createConvertShapeToStandardPass());
pm.addNestedPass<mlir::FuncOp>(mlir::createConvertShapeConstraintsPass());
// Fuse linalg on tensors operations.
pm.addPass(mlir::createCSEPass());
pm.addPass(mlir::memref::createResolveShapedTypeResultDimsPass());
pm.addPass(mlir::createCanonicalizerPass());
pm.addNestedPass<mlir::FuncOp>(mlir::createLinalgElementwiseOpFusionPass());
pm.run(module.get());
}
RecipeInfo
plaidml_program
plaidml_exe
pmlc/compiler/program.h
namespace pmlc::compiler {
struct Program {
explicit Program(mlir::ModuleOp module);
plaidml/core/internal.h
struct plaidml_program {
std::shared_ptr<pmlc::compiler::Program> program;
};
plaidml/core/core.h
namespace plaidml {
class TensorShape
TensorShape(DType dtype, const std::vector<int64_t>& sizes...)
class Buffer
Buffer(char* data, size_t size, const TensorShape& shape, ...)
class Program
public:
explicit Program(plaidml_program* ptr) :
void compile(const std::string& target = ...)
plaidml/exec/exec.h
namespace plaidml {
namespace exec {
class Executable
Executable(const Program& program, const std::string& device = "")
double run(const std::vector<Buffer>& inputs,
const std::vector<Buffer>& outputs);
Proof