TensorFlow XLA コード解析：JIT編

r1.3版

@Vengineer

2017/03/18 (オリジナル)
2017/07/01 追記 (plugin)
2017/07/30, 8/4, 8/6追記 (r1.3版)

自己紹介

Twitter：@Vengineer

ブログ：Vengineerの戯言

http://blogs.yahoo.co.jp/verification_engineer

CQ出版社：インターフェース 8月号、9月号に
TensorFlow XLAのAOTについての記事を書きました。

8月号：
衝撃的な性能UPの可能性を秘めた注目テクノロジ速報
AIをサクサク動かすGoogle新機能TensorFlow「XLA」を探る
9月号：
最新テクノロジ・マニアの挑戦 …AIサクサク用TensorFlow XLA AOTコンパイラ探訪
初めてのGoogleソースコード！ AI用コンパイラの可能性を探る

TensorFlow XLAとは

https://www.tensorflow.org/performance/xla/

XLA(Accelerated Linear Algebra)は、TensorFlow計算を最適化する線形代数のドメイン固有のコンパイラです。結果として、サーバーおよびモバイルプラットフォームでの速度、メモリ使用率、移植性が向上します。当初、ほとんどのユーザーはXLAの大きなメリットは見られませんが、JIT(Just-In-Time)コンパイルやAOT(Ahead-Of-Time)コンパイルを使用してXLAを使用することで実験を開始できます。新しいハードウェアアクセラレータをターゲットとする開発者は、XLAを試すことを特にお勧めします。

原文(英語)をそのまま、Google翻訳にお願いしました。

ブログにも書きました

TensorFlow XLAの衝撃

2017年2月20日

http://blogs.yahoo.co.jp/verification_engineer/71016304.html

TensorFlow XLAって何？

Recap of TensorFlow DEV SUMMIT 2017で
発表された「XLAコンパイラ」
足立昌彦さん（株式会社カブク）

資料と解説を見てちょうだい

詳しくは、「TensorFlow XLAの情報と発表」
http://blogs.yahoo.co.jp/verification_engineer/71068236.html

簡単にまとめると

TensorFlow XLAでは、次の2つをサポートした

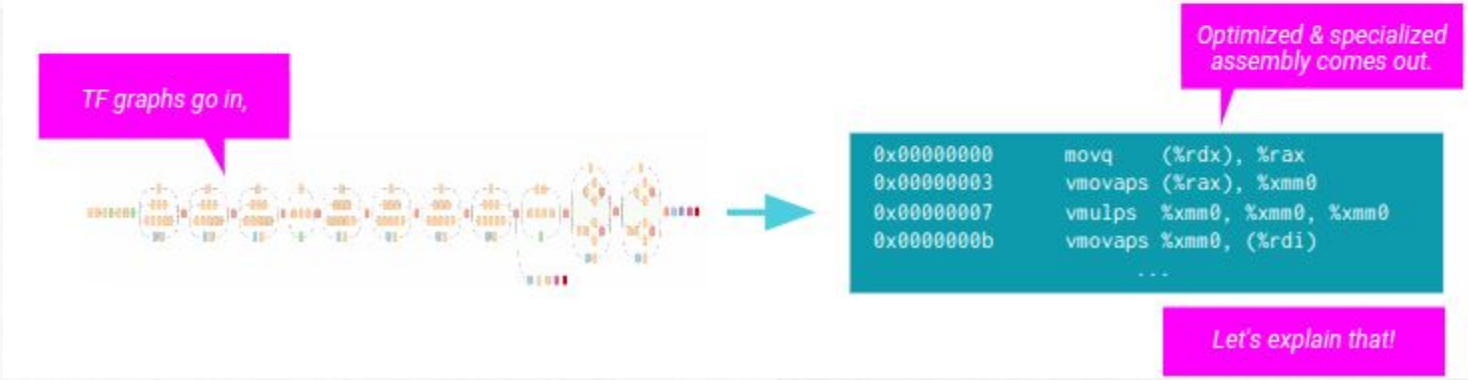
1)、JIT　(Just-In-Time) コンパイル
　　ただし、単一マシンのみで、GPUは1つ

2)、AOT　(Ahead-Of-Time) コンパイル
　　CPUのみ
　　x86/x86-64/ARM/AARCH64/PowerPC

この資料は、
TensorFlow XLAの
JITに関するコードを解析したものをまとめたです
TensorFlow r1.3対応
r1.2では、XLA Backend導入により、
r1.2からコードが変わっています！
ご利用は、自己責任でお願いします

Just-In-Time Compilation

via XLA, "Accelerated Linear Algebra" compiler



TensorFlow w/XLA: TensorFlow, Compiled! Expressiveness with performance
https://autodiff-workshop.github.io/slides/JeffDean.pdf

TensorFlow XLAに入る前に、

Pythonの式から生成される
グラフがどう変形されるのか？
見てみよう

TensorFlow XLAは、
まだ、
単一マシンでしか使えないので

DirectSessionの場合で

Session.runの動き

python/client/session.py

SessionInterface => BaseSession => Session

```
def run( self, fetches, feed_dict=None,
          options=None, run_metadata=None );
    _run
      _do_run
        tf_session.TF_Run
          ここからC++の世界
            c/c_api.ccのTF_Run関数
            c/c_api.ccのTF_Run_Helper関数
            Session::run (core/public/session.h)
                DirectSession::Run
```

C++のDirectSession::Run

DirectSession::Run (core/common_runtime/direct_session.cc)

Executorを生成する

```
GetOrCreateExecutors(pool, input_tensor_names,
                     output_names, target_nodes,
                     &executors_and_keys,
                     &run_state_args));
```

Executorは複数あり、各Executorが独立して実行し、
各Executor間の通信は非同期に行われる

C++のDirectSession::Runの続き

DirectSession::Run (core/common_runtime/direct_session.cc)

実行部分のところ

```
for (const auto& item : executors_and_keys->items) {
  item.executor->RunAsync(args, barrier->Get());
}
```

Executorが非同期に実行される

すべてExecutorの実行が終了するまで待つ

```
WaitForNotification(&run_state, &step_cancellation_manager,
                    run_options.timeout_in_ms() > 0
                        ? run_options.timeout_in_ms()
                        : operation_timeout_in_ms_);
```

executor->RunAsync

Executor::RunAync　(core/common_runtime/executor.h)
 ExecuteImple::RunAsync
  ExecuteState::RunAsync
   ExecuteState::ScheduleReady
    ExecuteState::Process (core/common_runtime/executor.cc)

・device->ComputeAsync 非同期の場合
・device->Compute　　同期の場合

え、
どこでグラフが
生成されるんだよ！

はい、ここです。

DirectSession::GetOrCreateExecutors

この CreateGraphs 関数内でグラフを生成し、分割する

```
CreateGraphs( options, &graphs, &ek->flib_def,
                        run_state_args));
```

その後に、分割されたグラフ単位で Executor にて実行される

グラフは次のステップで作られる

1)、Feed/Fetchノードの追加

subgraph::RewriteGraphForExecution
(core/graph/subgraph.cc)

2)、Placement

SimplePlacer::Run
(core/common_runtime/simple_placer.cc)

3)、グラフの分割 (同じデバイス＆実行単位)

Partition
(core/graph/graph_partition.cc)

RewriteGraphForExecution

core/graph/subgraph.cc

Feedノードを追加　(_Recv : .Attr("client_terminated", true))

```
if (!fed_outputs.empty()) {
 FeedInputs( g, device_info, fed_outputs, &name_index );
}
```

Fetchノードを追加　(_Send : .Attr("client_terminated", true))

```
std::vector<Node*> fetch_nodes;
if (!fetch_outputs.empty()) {
  FetchOutputs( g, device_info, fetch_outputs,
                    &name_index, &fetch_nodes );
}
```

SimplePlacer::Run

core/common_runtime/simple_placer.cc

1. First add all of the nodes.

2. Enumerate the constraint edges,
   and use them to update the disjoint node set.

3. For each node, assign a device based on the constraints in the disjoint node set.

4. Perform a second pass assignment for those nodes explicitly skipped during the first pass.

Partition

core/graph/graph_partition.cc

1)、各デバイスで実行できる単位に分割する

デバイス：cpu / gpu / XLA_CPU / XLA_GPU

2)、各デバイス間に、_Send / _Recv ノードを追加する

例えば、cpu => gpu の部分に、

cpu側には _Send ノードを

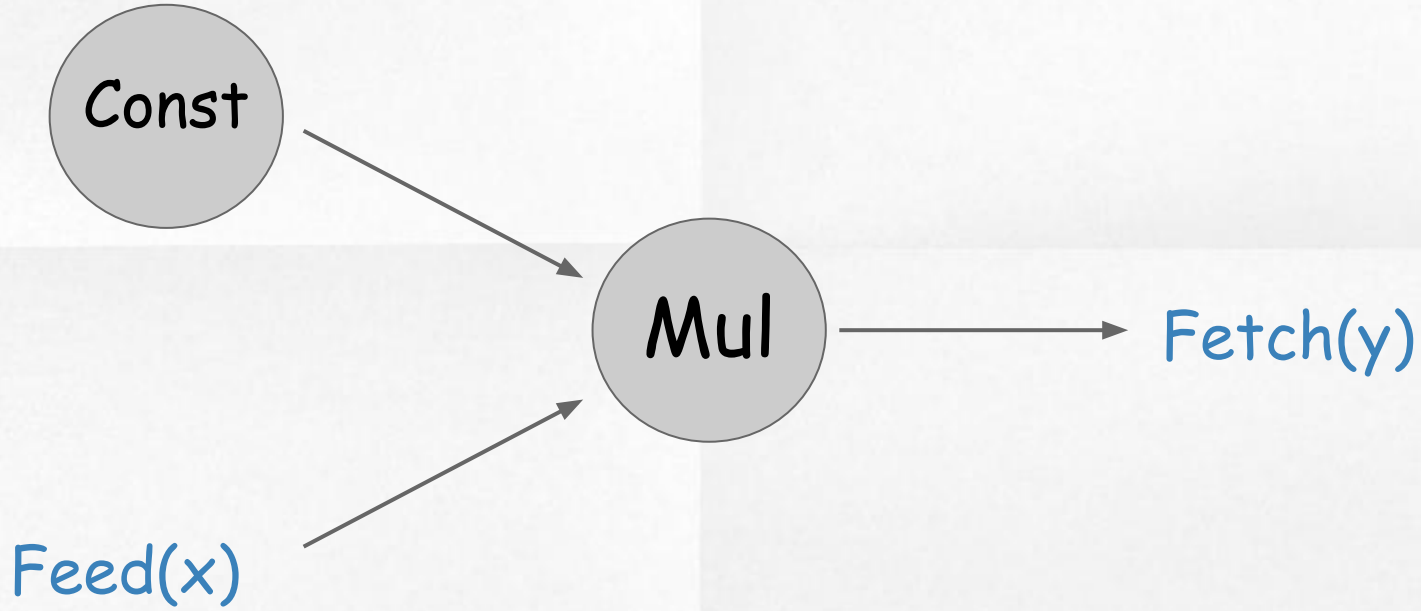
gpu側には _Recv ノードを追加する

サンプルコードで
確認してみよう

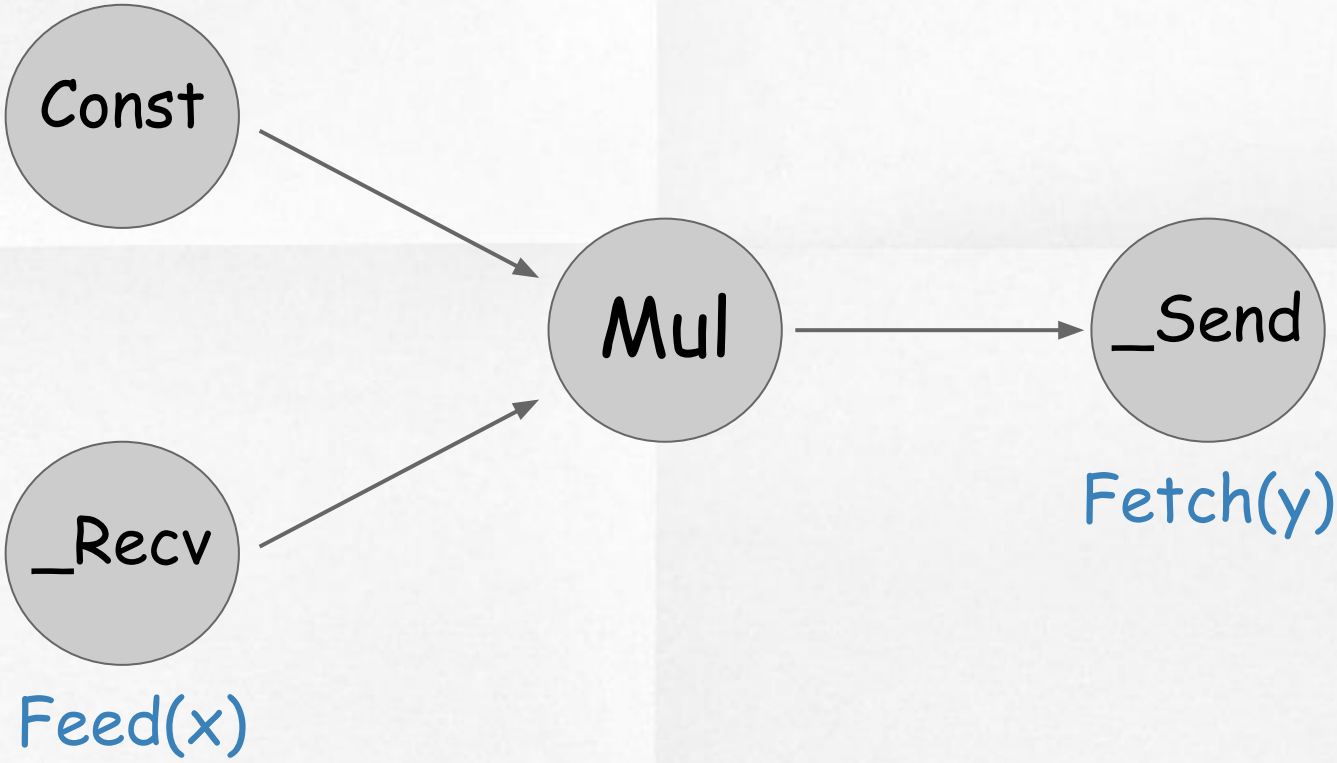
まずは、cpu で確認してみると

```
def test_cpu(self):

  with tf.Session() as sess:

    x = tf.placeholder(tf.float32, [2], name="x")
    with tf.device("cpu"):
      y = x * 2
    result = sess.run(y, {x: [1.5, 0.5]})
```

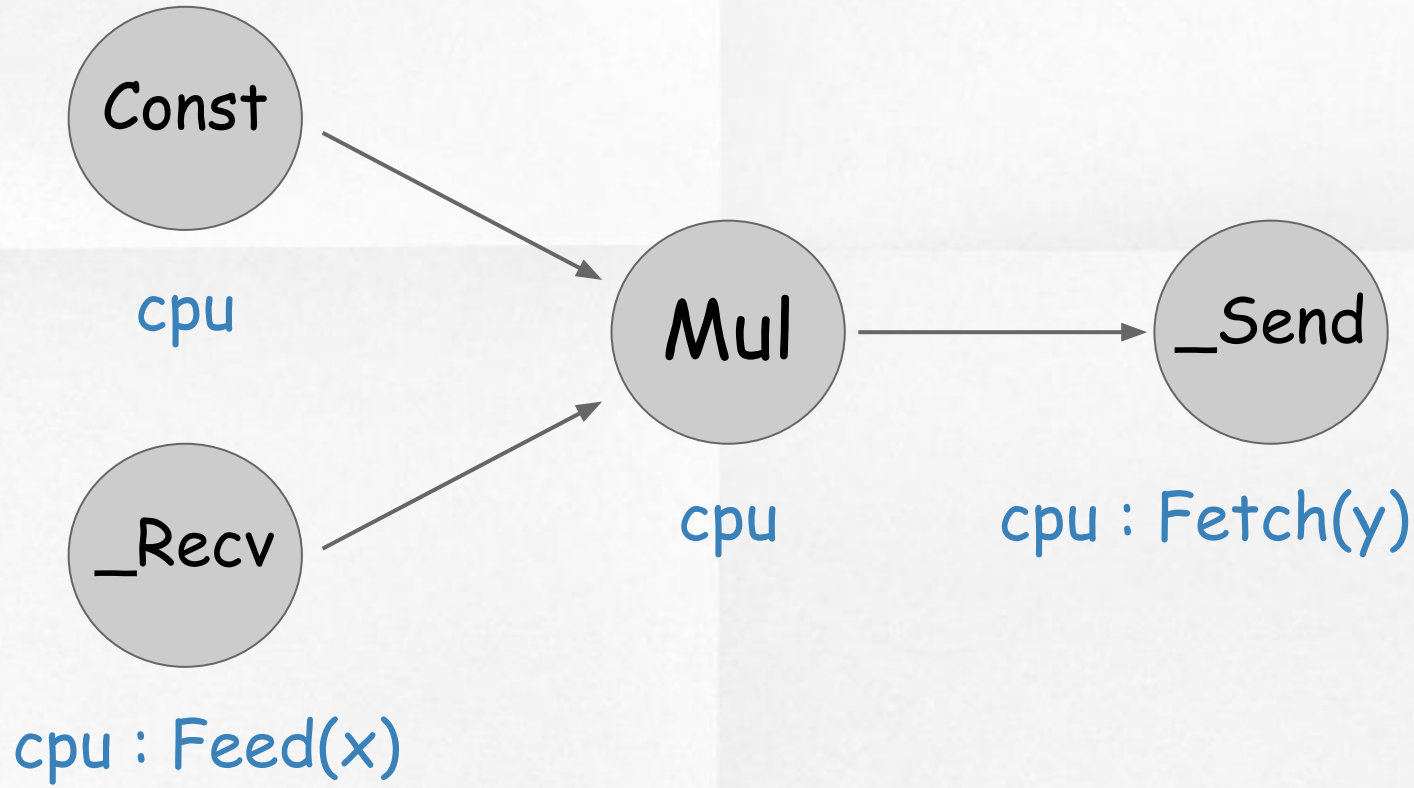
0)、最初



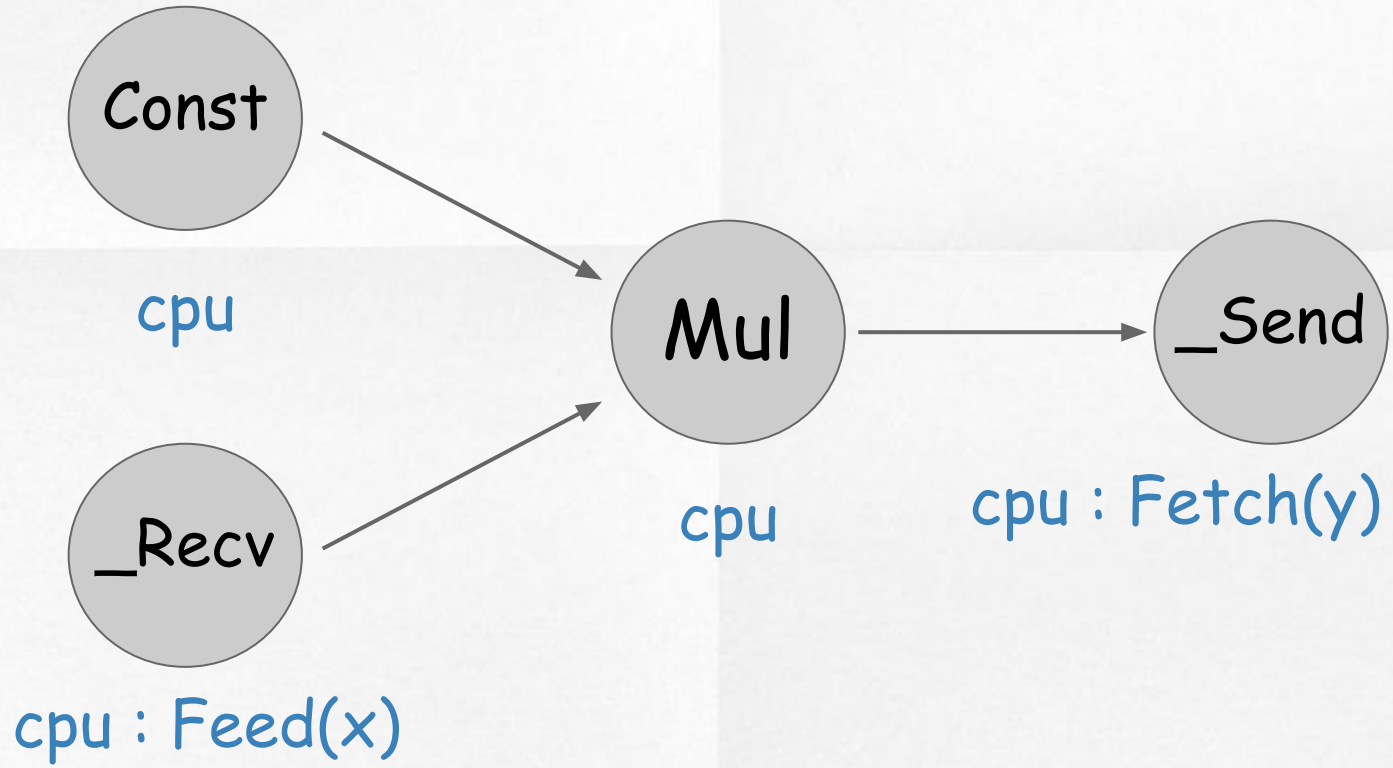
1)、Feed/Fetchノードの追加



2)、Placement



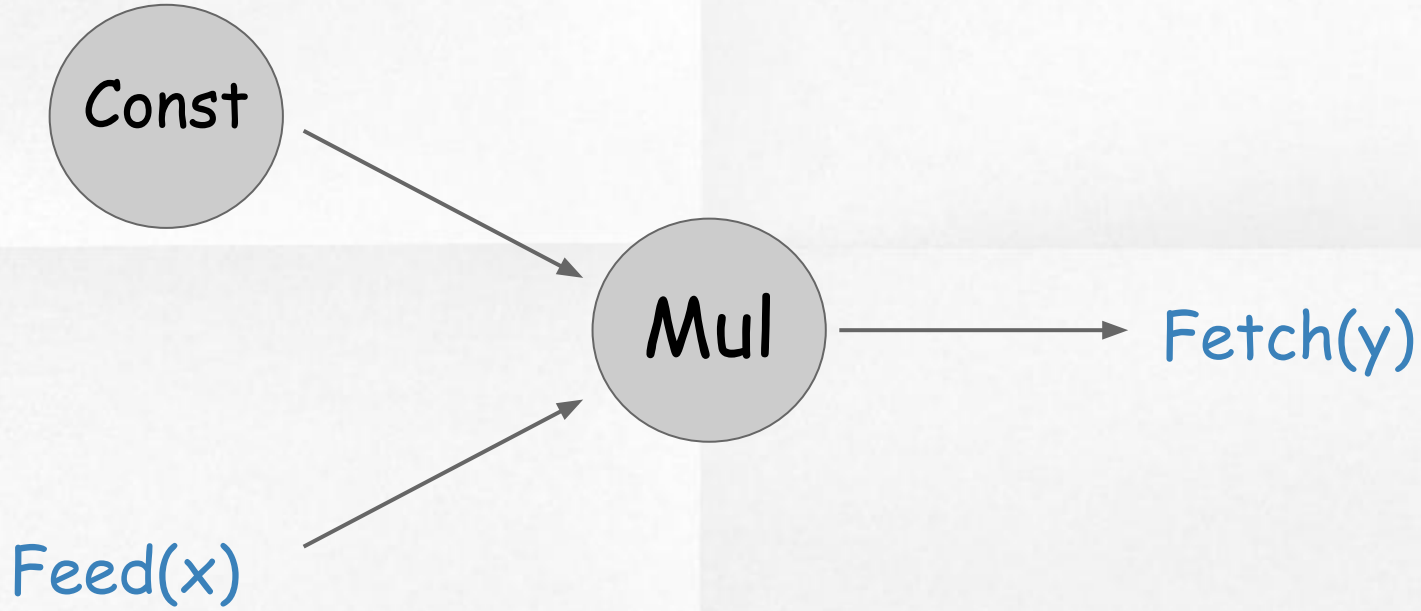
3)、グラフの分割



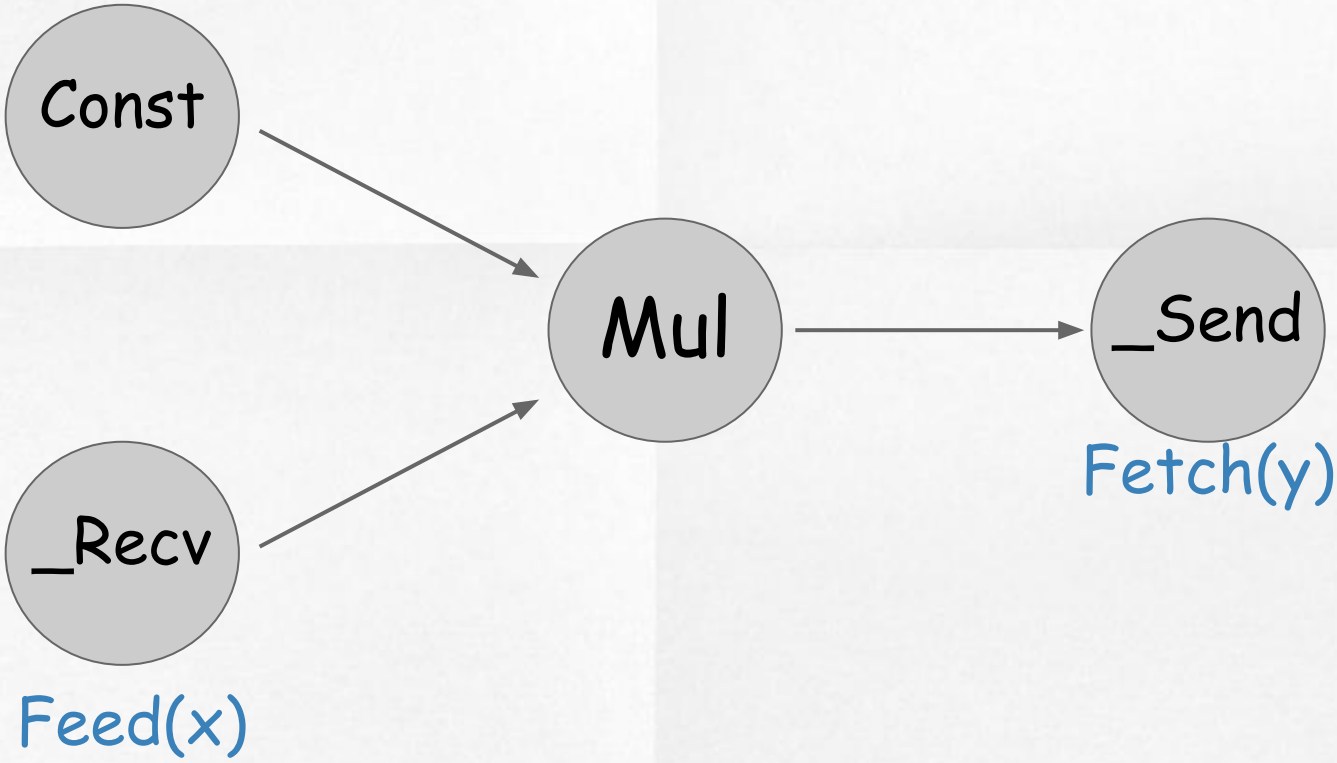
次に、gpu に変更してみると

```
def test_gpu(self):

  with tf.Session() as sess:

    x = tf.placeholder(tf.float32, [2], name="x")
    with tf.device("gpu"):
      y = x * 2
    result = sess.run(y, {x: [1.5, 0.5]})
```

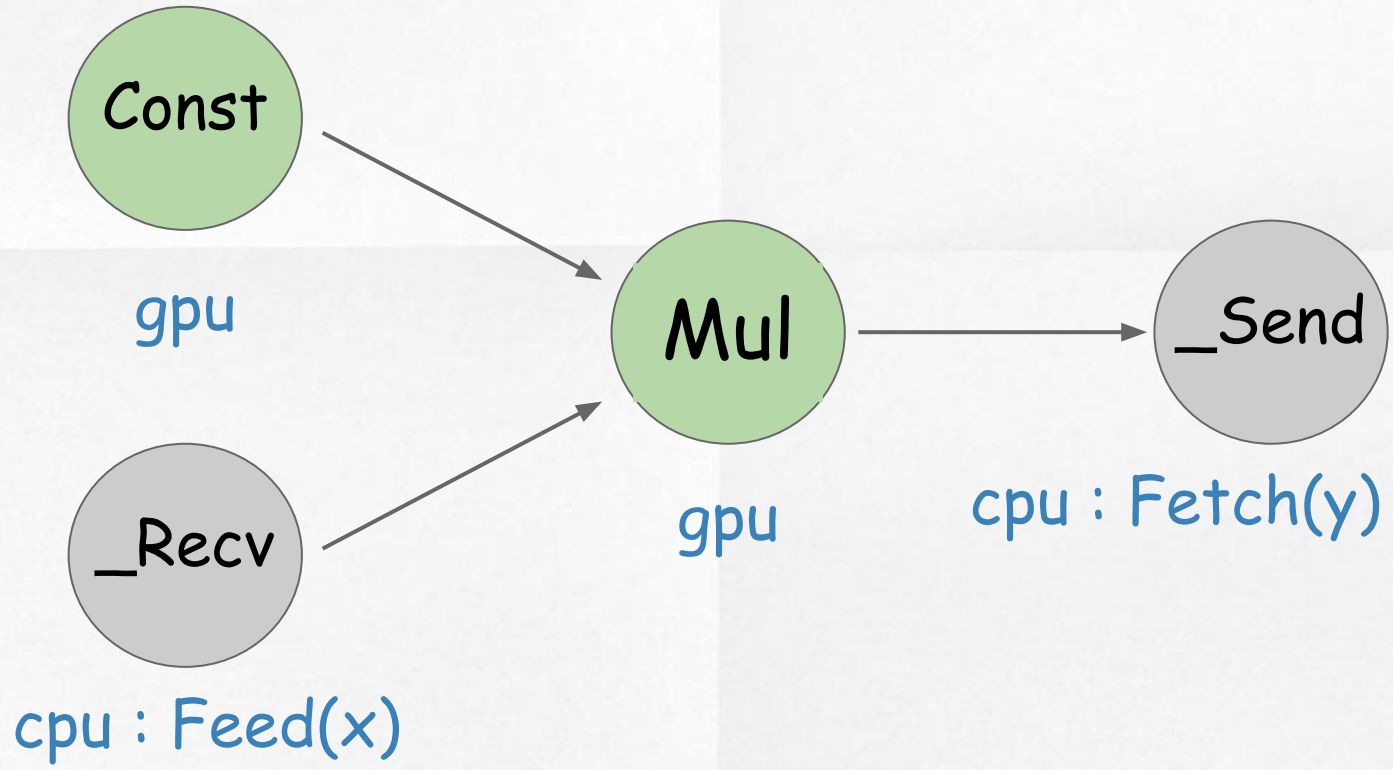
0)、最初



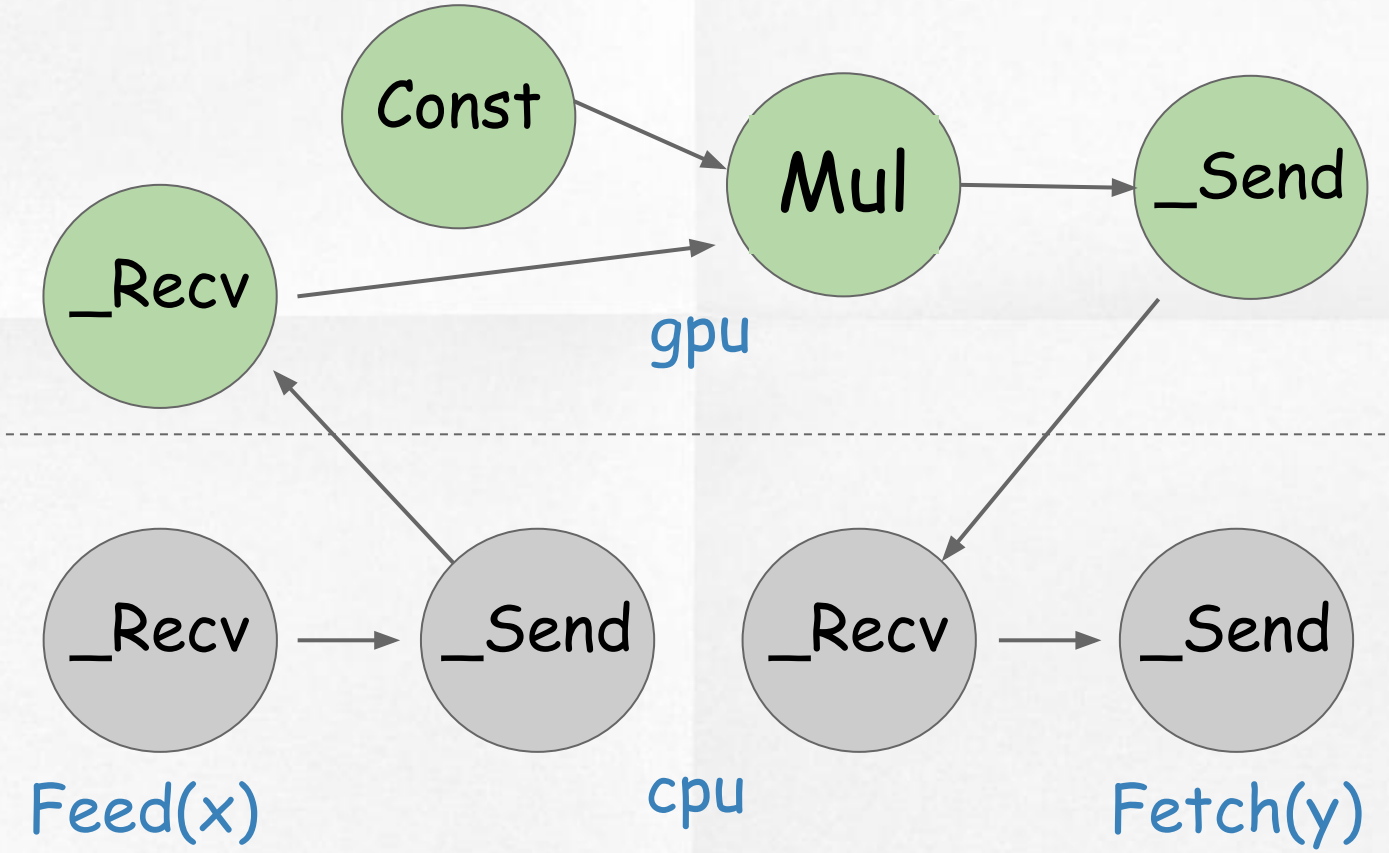
1)、Feed/Fetchノードの追加



2)、Placement



3)、グラフの分割



さあ、
グラフの分割については
わかったので、

Using JIT Compilation

https://www.tensorflow.org/performance/xla/jit

TensorFlow/XLA JITコンパイラは、XLAを使用してTensorFlowグラフの一部をコンパイルして実行します。
この標準的なTensorFlow実装の利点は、XLAが複数の演算子(カーネル融合)を少数のコンパイル済みカーネルに融合できることです。
TensorFlow Executorsが実行するように、演算子を融合させることで、メモリ帯域幅の要件を減らし、演算子を1つずつ実行するよりもパフォーマンスを向上させることができます。

原文(英語)をそのまま、Google翻訳にお願いしました。

TensorFlow XLA : JIT

バイナリでは提供されていないので、ソースコードからビルドする必要がある

JITが出来るようにビルドする

TensorFlowでXLAを使えるようにする
by @adamrocker
http://blog.adamrocker.com/2017/03/build-tensorflow-xla-compiler.html

の
「A: TensorFlowのビルド」
に詳しく書いてあります。

ディレクトリ構成

compilerディレクトリがTensorFlow XLA

- aot
- jit
- plugin (r1.3から)
- tests
- tf2xla
- xla

JIT関連は、主にjitディレクトリ内にある

TensorFlowは、
Bazelを使ってビルドしているので、

Bazel : https://bazel.build/

まずは、BUILDファイル

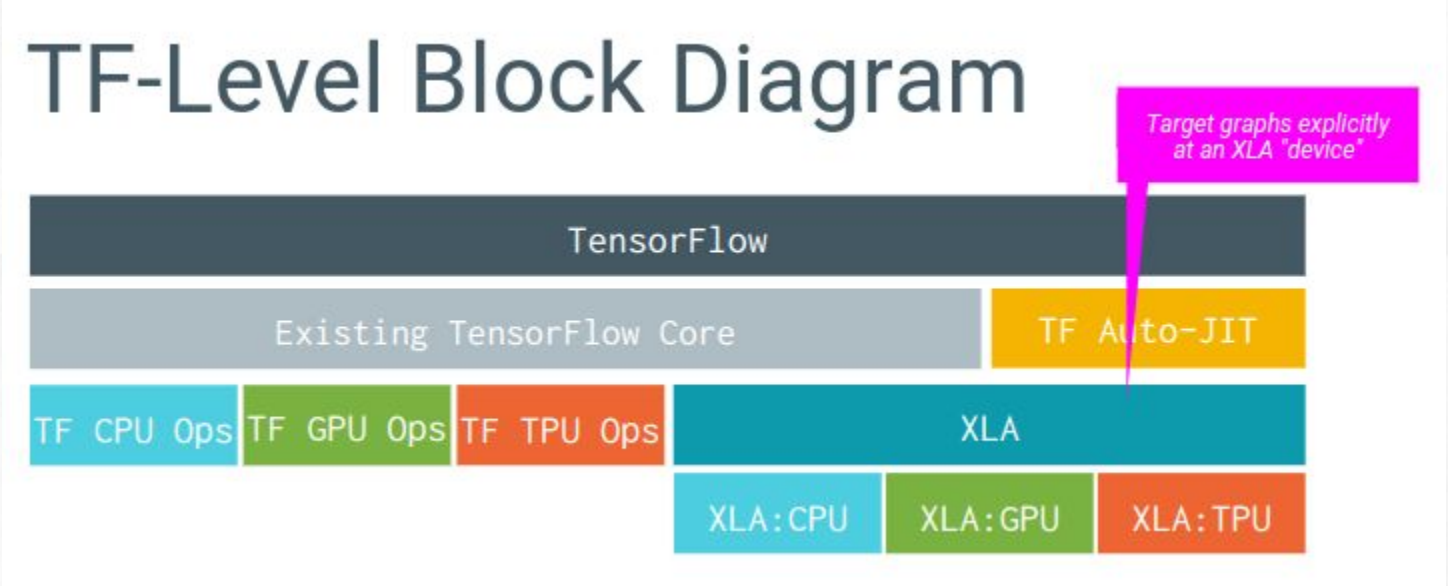
jit/BUILD

```
cc_library(
    name = "jit",
    visibility = [":friends"],
    deps = [
        ":xla_cpu_device",
        ":xla_cpu_jit",
        ":xla_gpu_device",
        ":xla_gpu_jit",
        :xla_gpu_jit",
        "//tensorflow/compiler/plugin",
    ],
```

新しいデバイスを追加
JITできるデバイスは、
- CPU
- GPU

r1.3でpluginを導入

XLA対応のデバイス



TensorFlow w/XLA: TensorFlow, Compiled! Expressiveness with performance
https://autodiff-workshop.github.io/slides/JeffDean.pdf

xla_cpu_jit/xla_gpu_jit

```
cc_library(
    name = "xla_cpu_jit",
    visibility = [":friends"],
    deps = [
        ":jit_compilation_passes",
        ":xla_local_launch_op",
        "//tensorflow/compiler/tf2xla/kernels:xla_ops",
        "//tensorflow/compiler/xla/service:cpu_plugin",
    ],
    alwayslink = 1,
)
```

<= xla_gpu_jit も同じ

デバイスの登録

core/common_runtime/device_factory.{h,c}

```
// The default priority values for built-in devices is:
// GPU: 210
// SYCL: 200
// GPUCompatibleCPU: 70
// ThreadPoolDevice: 60
// Default: 50
```

REGISTER_LOCAL_DEVICE_FACTORYマクロで設定する

xla_cpu_deviceの登録

```
const char* const DEVICE_XLA_CPU = "XLA_CPU";

class XlaCpuDeviceFactory : public DeviceFactory {
 public:
  Status CreateDevices(const SessionOptions& options, const
string& name_prefix,
                       std::vector<Device*>* devices) override;
};
```

xla_cpu_deviceの登録

```
REGISTER_LOCAL_DEVICE_FACTORY(
    DEVICE_XLA_CPU, XlaCpuDeviceFactory);

constexpr std::array<DataType, 5> kAllXlaCpuTypes = {{
    DT_INT32, DT_INT64, DT_FLOAT,
    DT_DOUBLE, DT_BOOL}};

REGISTER_XLA_LAUNCH_KERNEL(
    DEVICE_XLA_CPU, XlaDeviceLaunchOp, kAllXlaCpuTypes);
REGISTER_XLA_DEVICE_KERNELS(
    DEVICE_XLA_CPU, kAllXlaCpuTypes);
```

xla_gpu_deviceの登録

```
const char* const DEVICE_XLA_GPU = "XLA_GPU";

class XlaGpuDeviceFactory : public DeviceFactory {
 public:
  Status CreateDevices(const SessionOptions& options, const
string& name_prefix,
                       std::vector<Device*>* devices) override;
};
```

xla_gpu_deviceの登録

```
REGISTER_LOCAL_DEVICE_FACTORY(
    DEVICE_XLA_GPU, XlaGpuDeviceFactory);

constexpr std::array<DataType, 5> kAllXlaGpuTypes = {{
    DT_INT32, DT_INT64, DT_FLOAT,
    DT_DOUBLE, DT_BOOL}};

REGISTER_XLA_LAUNCH_KERNEL(
    DEVICE_XLA_GPU, XlaDeviceLaunchOp, kAllXlaGpuTypes);
REGISTER_XLA_DEVICE_KERNELS(
    DEVICE_XLA_GPU, kAllXlaGpuTypes);
```

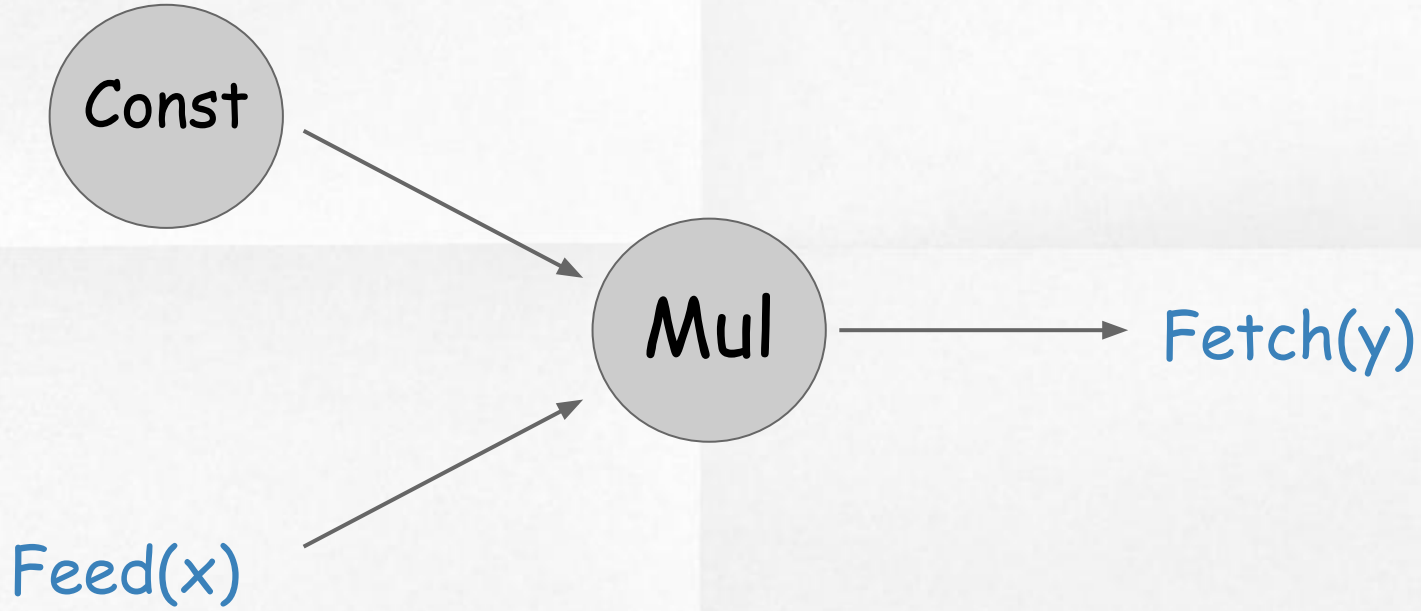
先ずは、
TensorFlow XLAのJITでは

グラフがどのように変更されるか、
確認してみよう

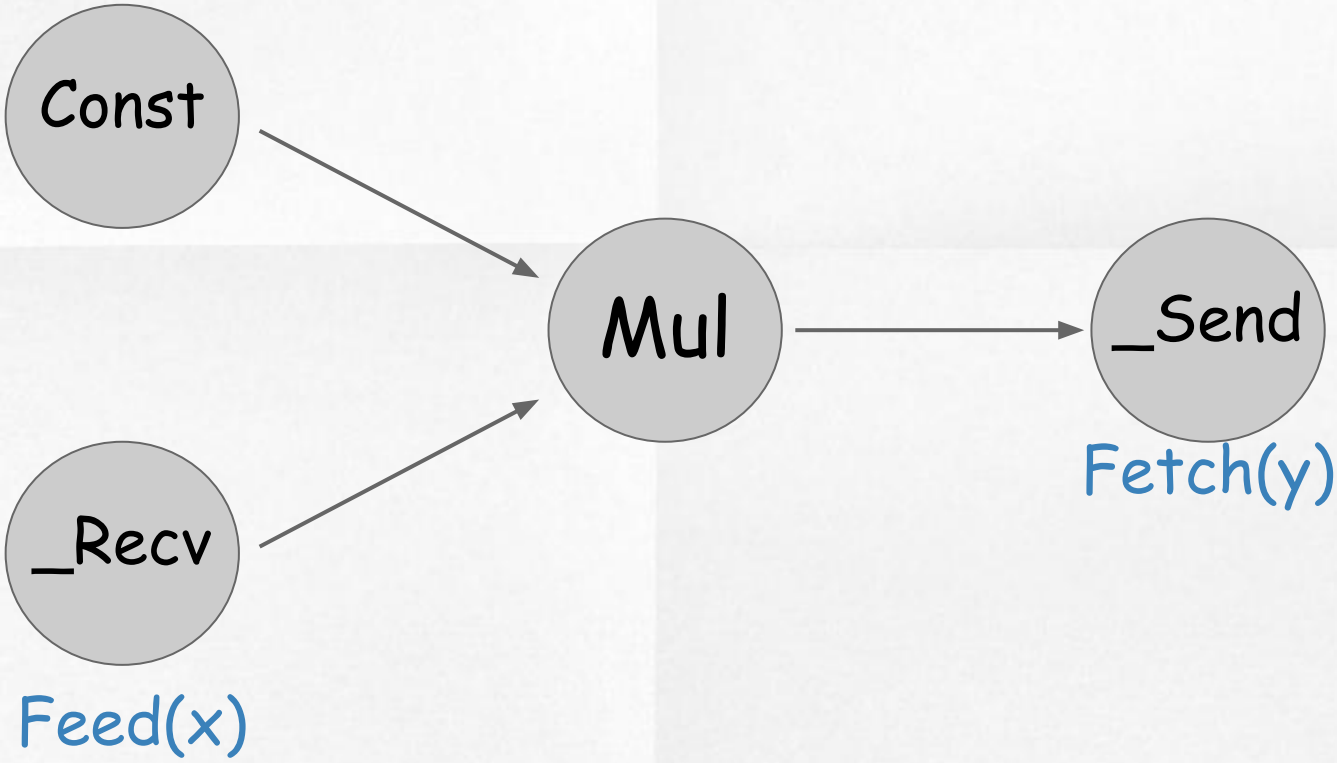
cpu or gpu を XLA_CPU に変更

```
def testXLA_JIT(self):

  with tf.Session() as sess:

    x = tf.placeholder(tf.float32, [2], name="x")
    with tf.device("device:XLA_CPU:0"):
      y = x * 2
    result = sess.run(y, {x: [1.5, 0.5]})
```

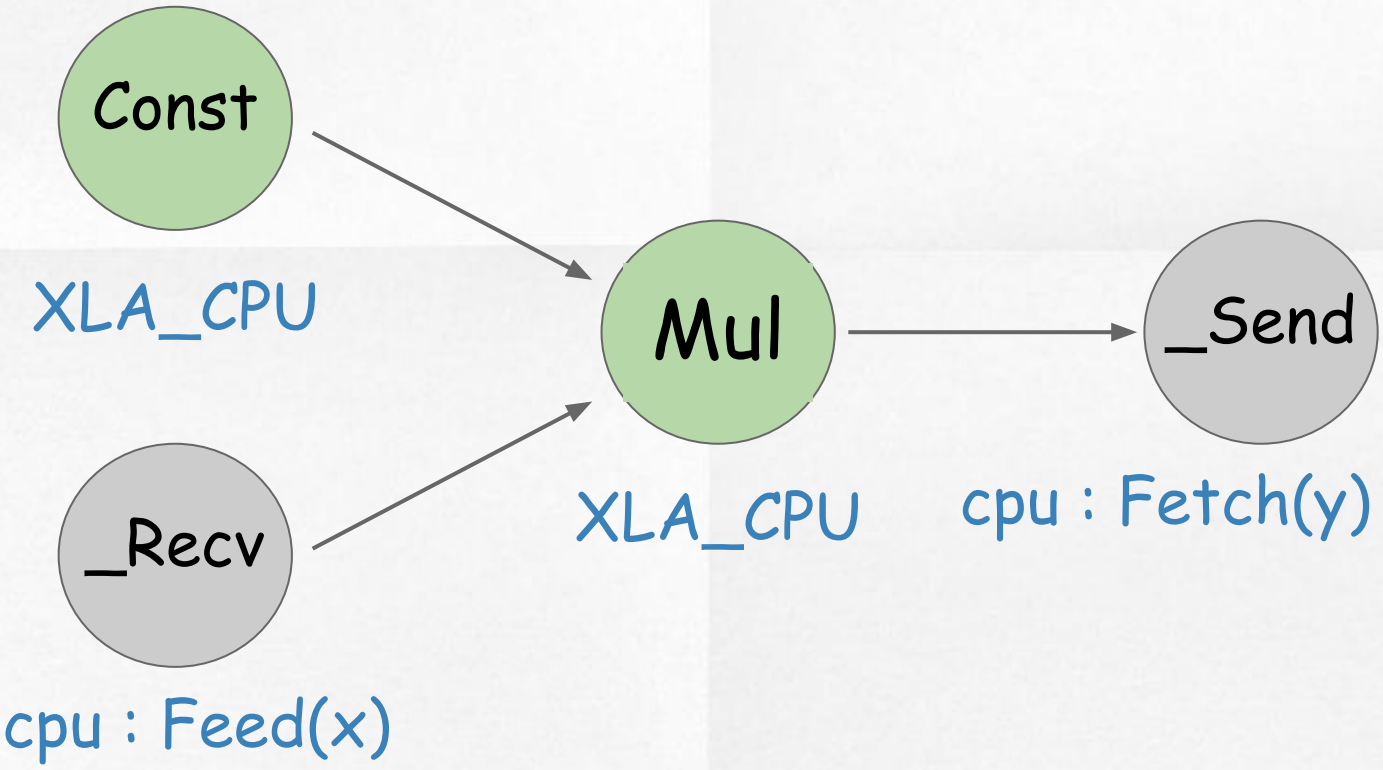
0)、最初



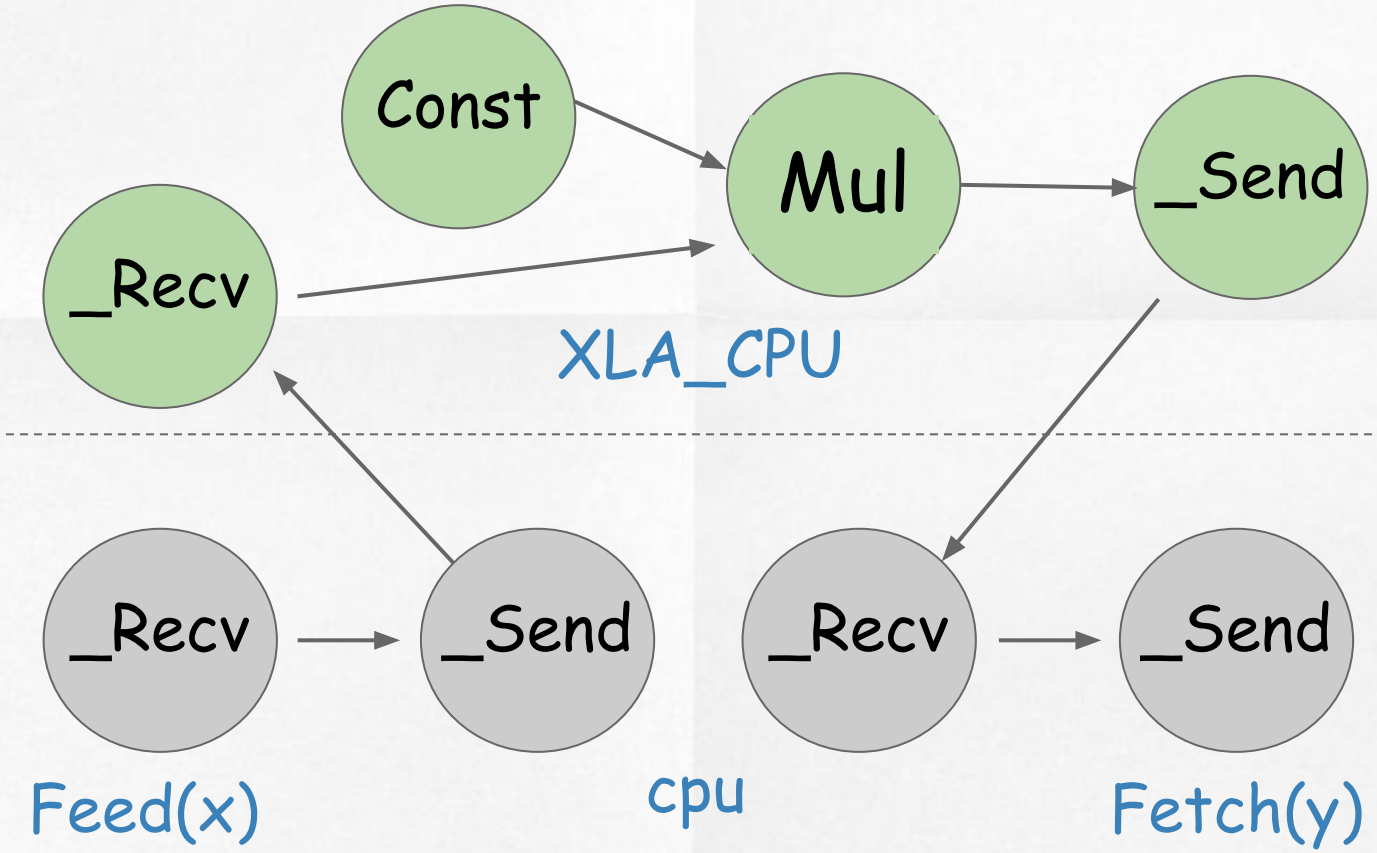
1)、Feed/Fetchノードの追加



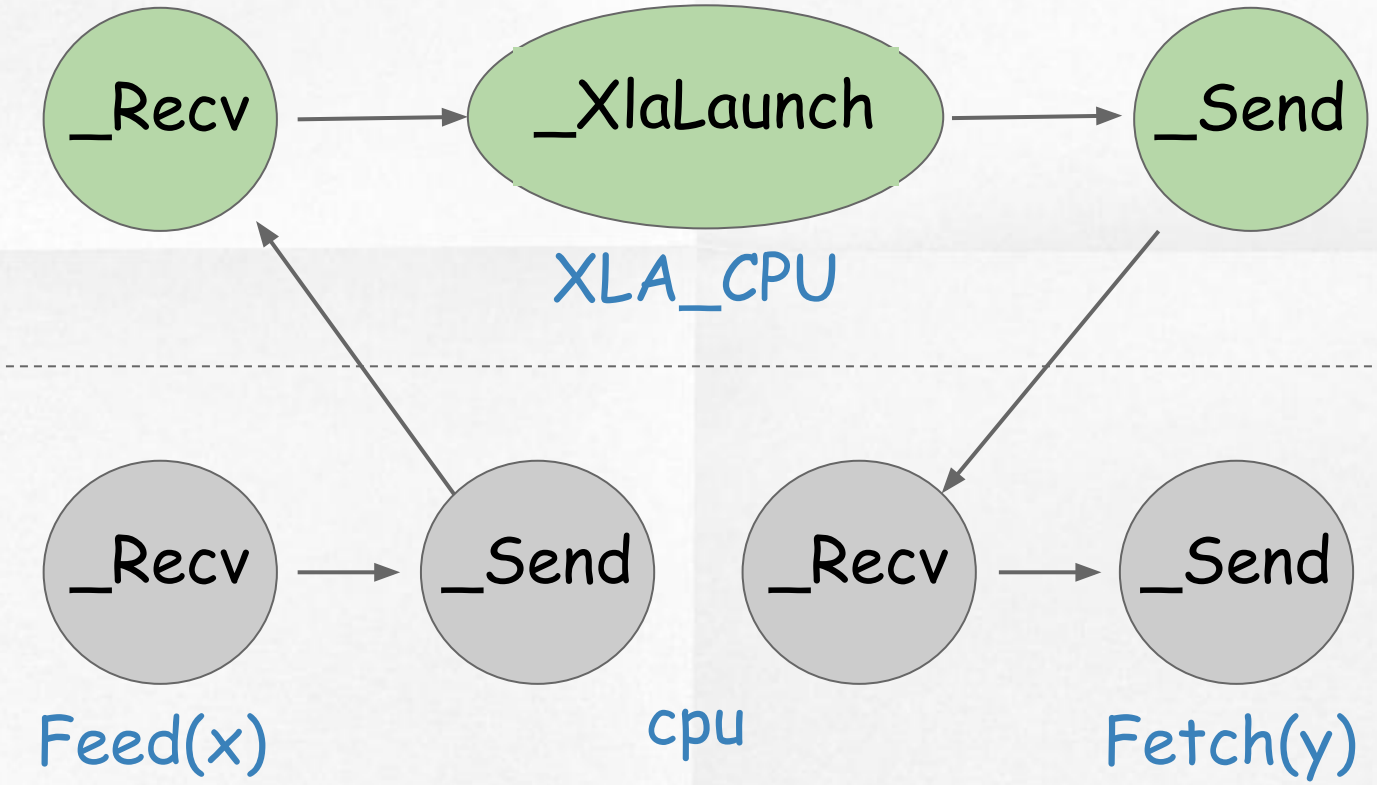
2)、Placement



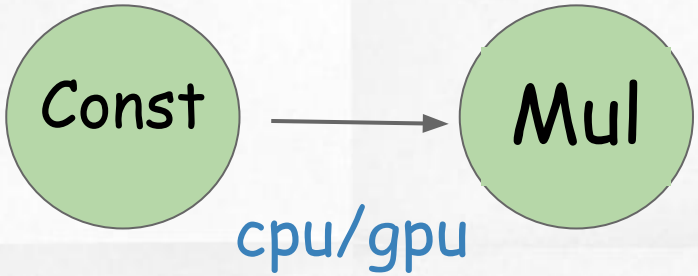
3)、グラフの分割

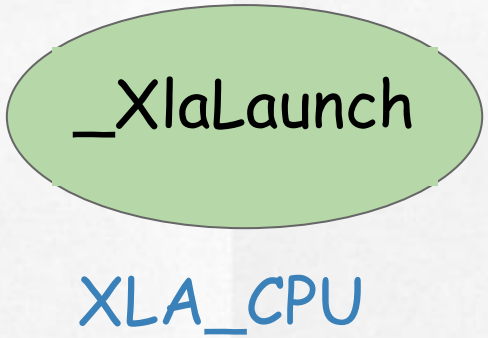

Const
Mul
_Send
_Recv
XLA_CPU
_Recv
_Send
_Recv
_Send
Feed(x)
cpu
Fetch(y)

3)、グラフの分割


_Recv
_XlaLaunch
_Send
XLA_CPU
_Recv
_Send
_Recv
_Send
Feed(x)
cpu
Fetch(y)

複数Opsを_XlaLaunch Opに変換





ええええ、
なんで、_XlaLaunch
になっちゃうの？

どうして？

xla_cpu_jit/xla_gpu_jit

```
cc_library(
    name = "xla_cpu_jit",
    visibility = [":friends"],
    deps = [
        ":jit_compilation_passes",
        ":xla_local_launch_op",
        "//tensorflow/compiler/tf2xla/kernels:xla_ops",
        "//tensorflow/compiler/xla/service:cpu_plugin",
    ],
    alwayslink = 1,
)
```

<= xla_gpu_jit も同じ

jit_compilation_passes

```
cc_library(
    name = "jit_compilation_passes",
    srcs = ["jit_compilation_pass_registration.cc"],
    deps = [
        ":compilation_passes",
        "//tensorflow/core:core_cpu_internal",
    ],
    alwayslink = 1,
)
```

compilation_passes

```
cc_library(
    name = "compilation_passes",
    srcs = [
        "build_xla_launch_ops_pass.cc",
        "encapsulate_subgraphs_pass.cc"
        "mark_for_compilation_pass.cc",
    ],
    hdrs = [
        "build_xla_launch_ops_pass.h",
        "encapsulate_subgraphs_pass.h",
        "mark_for_compilation_pass.h",
    ],
```

この3つのファイルにて
JIT用のグラフに変換してる。

Passを使ってグラフを変形してるよ

compiler/jit/jit_compilation_pass_registration.cc

REGISTER_OPTIMIZATIONマクロを使って、
OptimizationPassRegistry::POST_REWRITE_FOR_EXEC
Passを追加

- MarkForCompilationPass　// コンパイル可能なものにマーク
  mark_for_compilation_pass.{h,cc}
- EncapsulateSubgraphsPass // サブグラフを関数ノード
  Encapsulate_subgraphs_pass.{h,cc}
- BuildXlaLaunchOpsPass // 関数ノードを_XlaLaunchに置換
  build_xla_launch_ops_pass.{h,cc}

上から順番に実行される

これらのPassはいつ実行される？

1)、Feed/Fetchノードの追加
　subgraph::RewriteGraphForExecution

　ここで、PRE_PLACEMENTパス を実行
2)、Placement
　ここで、POST_PLACEMENTパス を実行

SimpleGraphExecutionState::BuildGraph関数で
POST_REWRITE_FOR_EXEC を実行
3)、グラフの分割
　Partition
　ここで、POST_PARTITIONINGパス を実行

MarkForCompilationPass

compiler/jit/mark_for_compilation_pass.cc

```
Status MarkForCompilationPass::Run(
    const GraphOptimizationPassOptions& options) {
 .....
  各ノードが下記の条件のとき、コンパイルが必要だとマークする
  1)、If this device requires a JIT, we must say yes.

  2)、If there is a _XlaCompile annotation, use its value.

  3)、Otherwise use the value of global_jit_level.
 .....
}
```

EncapsulateSubgraphsPass

compiler/jit/encapsulate_subgraphs_pass.cc

```
Status EncapsulateSubgraphsPass::Run(
   const GraphOptimizationPassOptions& options) {
 .....
 サブグラフを関数ノードにする
 EncapsulateSubgraphsInFunctions(
      kXlaClusterAttr, **options.graph, rewrite_subgraph,
      flags->tf_xla_parallel_checking, &graph_out, library));
 }
 .....
 AddNodeAttr(kXlaCompiledKernelAttr, true, node);
 .....
}
```

EncapsulateSubgraphsInFunctions

compiler/jit/encapsulate_subgraphs_pass.cc

```
Encapsulator encapsulator(std::move(group_attribute), &graph_in);
s = encapsulator.SplitIntoSubgraphs();

s = encapsulator.BuildFunctionDefs(rewrite_subgraph_fn, library);

std::unique_ptr<Graph> out(new Graph(library));
out->set_versions(graph_in.versions());
s = encapsulator.BuildOutputGraph(parallel_checking, out.get());

*graph_out = std::move(out);
```

EncapsulateSubgraphsInFunctions

1)、MarkForCompilationPassマークされたノードを
　サブグラフに分割

```
s = encapsulator.SplitIntoSubgraphs();
```

2)、サブグラフにFunctionDefを作成

```
s = encapsulator.BuildFunctionDefs(rewrite_subgraph_fn, library);
```

3)、サブグラフにFunction callノードを追加

```
s = encapsulator.BuildOutputGraph(parallel_checking, out.get());
```

BuildXlaLaunchOpsPass

```
Status BuildXlaLaunchOpsPass::Run(
  const GraphOptimizationPassOptions& options) {
  Graph* graph = options.graph->get();

    // Only compile nodes that are marked for compilation by the
    // compilation-marking pass (via 'attr_name').
    if (IsXlaCompiledKernel(*n)) {
      ReplaceNodeWithXlaLaunch(graph, n);
    }
  }
  return Status::OK();
}
```

IsXlaCompiledKernel

compiler/jit/encapsulate_subgraphs_pass.cc

ノードにkXlaCompiledKernelAttrアトリビュートがある？

```
bool IsXlaCompiledKernel(const Node& node) {
  bool is_compiled = false;
  bool has_compilation_attr =
      GetNodeAttr(node.def(),
                  kXlaCompiledKernelAttr,
            // EncapsulateSubgraphsPass::Runで追加した
                  &is_compiled).ok()
      && is_compiled;
  return has_compilation_attr ? is_compiled : false;
}
```

ReplaceNodeWithXlaLaunch

compiler/jit/build_xla_launch_ops_pass.cc

```
static Status ReplaceNodeWithXlaLaunch(
                Graph* graph, Node* node) {
  ....
  Node* launch_node;  // ノードを生成
  BuildLaunchNode(graph->NewName(node->name()),
                  node->type_string(), node->def().attr(), node->def().device(),
                  const_dtypes, arg_dtypes, node->output_types(), graph,
                  &launch_node));
  ......
  ノードを launch_node (_XlaLaunch) に置き換える
```

BuildLaunchNode

compiler/jit/build_xla_launch_ops_pass.cc

引数のnodeを_XlaLaunch Opのノードに置き換える

```
static Status BuildLaunchNode(
    const string& nodename, const string& function_name,
    const AttrValueMap& function_attr,
    const string& device_name,
    const DataTypeVector& constant_dtypes,
    const DataTypeVector& arg_dtypes,
    const DataTypeVector& result_dtypes,
    Graph* graph, Node** node) {
```

BuildLaunchNode

ノードの定義

```
NodeDef def;
def.set_name(graph->NewName(nodename));
def.set_op("_XlaLaunch");  // _XlaLaunch Op
def.set_device(device_name);
```

アトリビュート

```
AddNodeAttr("Tconstants", constant_dtypes, &def);
AddNodeAttr("Targs", arg_dtypes, &def);
AddNodeAttr("Tresults", result_dtypes, &def);
```

BuildLaunchNode

```
  アトリビュート function を追加
  NameAttrList function;
  function.set_name(function_name);
  *function.mutable_attr() = function_attr;
  AddNodeAttr("function", function, &def);

  Status status;
  ノードをグラフに追加
  *node = graph->AddNode(def, &status);
  return status;
}
```

_XlaLaunch Op って？

TensorFlow XLA：JITでは！

同じデバイス内で実行されるSubgraph単位の
ノードをギュギュッと1つにまとめて、

_XlaLaunch Op

内で実行する

_XlaLaunchは、
TensorFlow XLA専用のOpとして実装されている

Adding a New Op

https://www.tensorflow.org/versions/master/how_tos/adding_an_op/

必要なものは、

- Register the new Op in a C++ file
- Implement the Op in C++

- Optionally, create a Python wrapper
- Optionally, write a function to compute gradients for the Op
- Test the Op, typically in Python

_XlaLaunch Opで実装は？

- Register the new Op in a C++ file
- Implement the Op in C++

compiler/jit/kernels/xla_local_launch_op.h
compiler/jit/kernels/xla_local_launch_op.cc

_XlaLaunch Op の登録

```
REGISTER_OP("_XlaLaunch")
    .Input("constants: Tconstants")
    .Attr("Tconstants: list(type) >= 0")
    .Input("args: Targs")
    .Attr("Targs: list(type) >= 0")
    .Output("results: Tresults")
    .Attr("Tresults: list(type) >= 0")
    .Attr("function: func")
    .Doc("XLA Launch Op. For use by the XLA JIT only.");
```

_XlaLaunch Op の実装

```
class XlaLocalLaunchOp : public OpKernel {
 public:
  explicit XlaLocalLaunchOp(OpKernelConstruction* ctx);
  ~XlaLocalLaunchOp() override;
  void Compute(OpKernelContext* ctx) override;
      // 実行時は、この Compute メソッドが呼ばれる
private:
 ....
 TF_DISALLOW_COPY_AND_ASSIGN(XlaLocallLaunchOp);
};
```

XlaLocalLaunchOp::Compute

・XlaCompilationCacheクラスのインスタンス(compiler)を生成

・_XlaLaunch Op内で実行する一連の関数群をコンパイル

```
cache->Compile( .... );
```

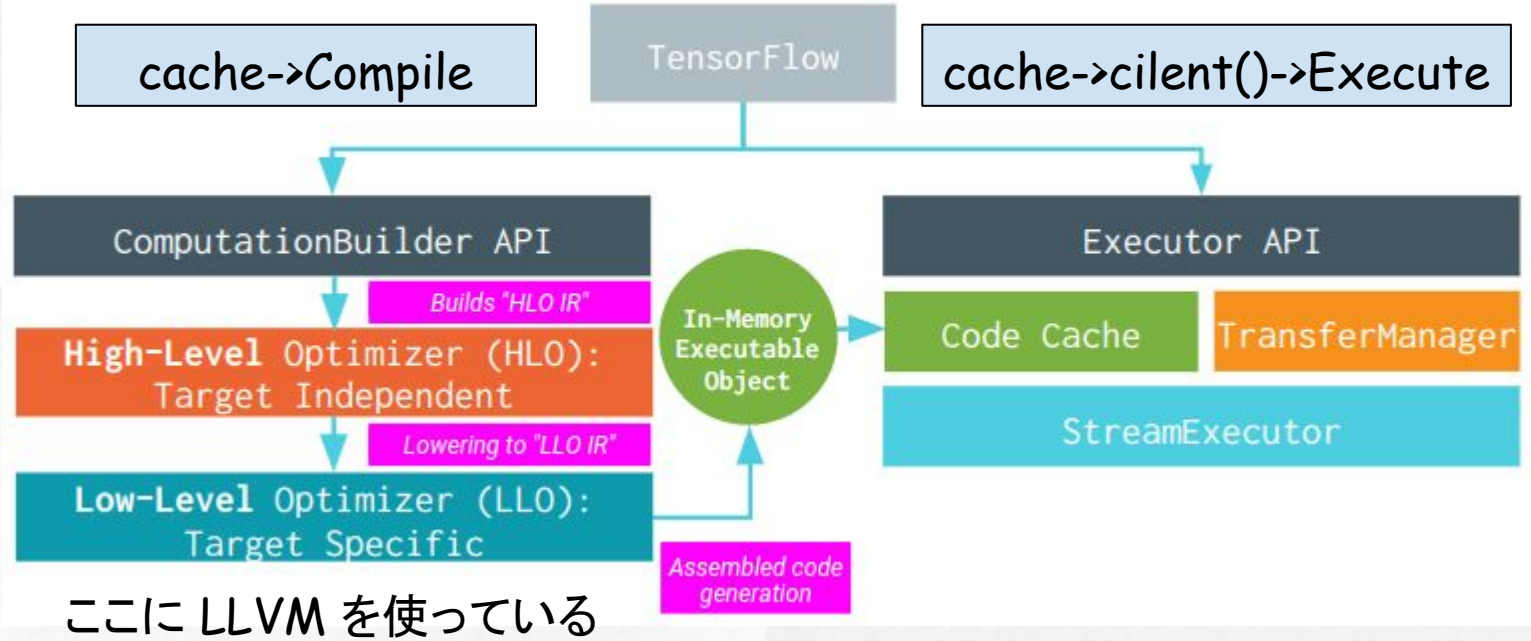
・各種パラメータ＆入力リストをXLA用データに変換

・キャッシュの生成＆実行

```
cache->client()->Execute(.....);
```

・XLA用データを出力リストに変換

Computeの処理



ここに LLVM を使っている

TensorFlow w/XLA: TensorFlow, Compiled! Expressiveness with performance
https://autodiff-workshop.github.io/slides/JeffDean.pdf

XlaLocalLaunchOp::Compute

jit/kernels/xla_local_launch_op.cc

```
const XlaCompiler::CompilationResult* kernel;

// キャッシュをコンパイル
ctx = cache->Compile(options, function_, num_constant_args_,
                     variables, ctx, &kernel, nullptr);

VLOG(1) << "Executing XLA Computation...";

// キャッシュ(XLA Computation) の実行
auto result = cache->client()->Execute(
                                       *kernel->computation,
                                       arg_ptrs,
                                       &execution_options,
                                       &profile);
```

コンパイル後、キャッシュする

TensorFlowグラフから
実行コードへの変換

XlaCompilationCache::Compile

jit/xla_compilation_cache.cc

メンバー compiler_ は、XlaCompiler

・TensorFlowグラフを XLA Computation にコンパイルする

```
entry->compiled = true;
entry->compilation_status = compiler_.CompileFunction(
    flr.get(), function, args, &entry->compilation_result);
```

・XLA Computation からでExecutableを生成する

```
entry->compilation_status = compiler_.BuildExecutable(
        entry->compilation_result, &entry->executable);
*executable = entry->executable.get();
```

XlaCompiler::CompileFuntion

xf2xla/xla_compiler.cc

・CompileFunction　　関数内のグラフからマシン語まで生成

1)、グラフの最適化　　(OptimizeGraph)
TensorFlowの標準関数

2)、グラフのコンパイル　　(CompileGraph)
TensorFlowグラフからXLA Compution へ

ここで、XLA Computation になっている

XlaCompiler::BuildExecutable

xf2xla/xla_compiler.cc

```
xla::LocalClient* local_client = static_cast<xla::LocalClient*>(client());
xla::ExecutableBuildOptions build_options;
build_options.set_device_ordinal(local_client->default_device_ordinal());
build_options.set_platform(local_client->platform());
build_options.set_result_layout(result.xla_output_shape);
build_options.set_has_hybrid_result(local_executable_has_hybrid_result_);
```

XLA Computation をデバイスコードに変換

```
auto compile_result = local_client->Compile(result.computation,
                                            argument_layouts, build_options);
*executable = std::move(compile_result.ValueOrDie());
```

LocalClient::Compile

xla/client/local_client.cc

2-3)、XLA Computation から LocalExecutable 生成

左側：XLA Computation からデバイスコード に生成

```
std::unique_ptr<Executable> executable =
local_service_->CompileExecutable(computation.handle(),
        argument_layouts,
                options.result_layout(), device_ordinal,
                options.has_hybrid_result());
```

右側：LocalExecutable生成

```
return WrapUnique(new LocalExecutable(std::move(executable),
                local_service_->mutable_backend(),
                device_ordinal, options));
```

LocalService::CompileExecutable

xla/client/local_service.cc

```
se::StreamExecutor * executor =
  execute_backend_->stream_executor(device_ordinal));

return BuildExecutable(versioned_handle, std::move(module_config),
                       /*executable_for_compute_constant=*/false,
                       argument_buffers, execute_backend_.get(), executor);
```

Service::BuildExecutable

xla/service/service.cc

XLA Computation から HLO へ変換

```
for (const VersionedComputationHandle& versioned_handle : versioned_handles) {
  auto module = computation_tracker_.BuildHloModule(
                                Versioned_handle, true));
  modules.push_back(std::move(module));
}
....
```

HLO から Executableに変換

```
std::unique_ptr<Executable> executable =
   backend->compiler()->Compile( std::move(modules), executor));
```

BuildHloModule

xla/service/computation_tracker.cc

```
.....
std::unique_ptr<HloComputation> hlo_computation =
    computation->BuildHloComputation(
                                    versioned_handle.version, resolver,
                                    include_unused_parameters);
.....
```

BuildHloComputation

xla/service/user_computation.cc

.....

```
std::unique_ptr<HloComputation> hlo_computation =
    ComputationLowerer::Lower( ここで演算の最適化を行う
        tensorflow::strings::StrCat(name(), ".v", version), session_computation_,
        version, std::move(hlo_resolver), include_unused_parameters);
```

.....

HLOからデバイスコードに

xla/service/compiler.h

```
// Compiles the HLO module for execution on a device given by the executor,
// and returns an executable object or an error status. Takes ownership of the
// HLO module and is free to transform it.
virtual StatusOr<std::unique_ptr<Executable>> Compile(
    std::unique_ptr<HloModule> module,
    perftools::gputools::StreamExecutor* executor) = 0;
```

CPU

xla/service/cpu_compiler.{h,c}

RunHloPasses：HLOの最適化

LLVMでCPU命令コードの生成

CpuExecutable

GPU

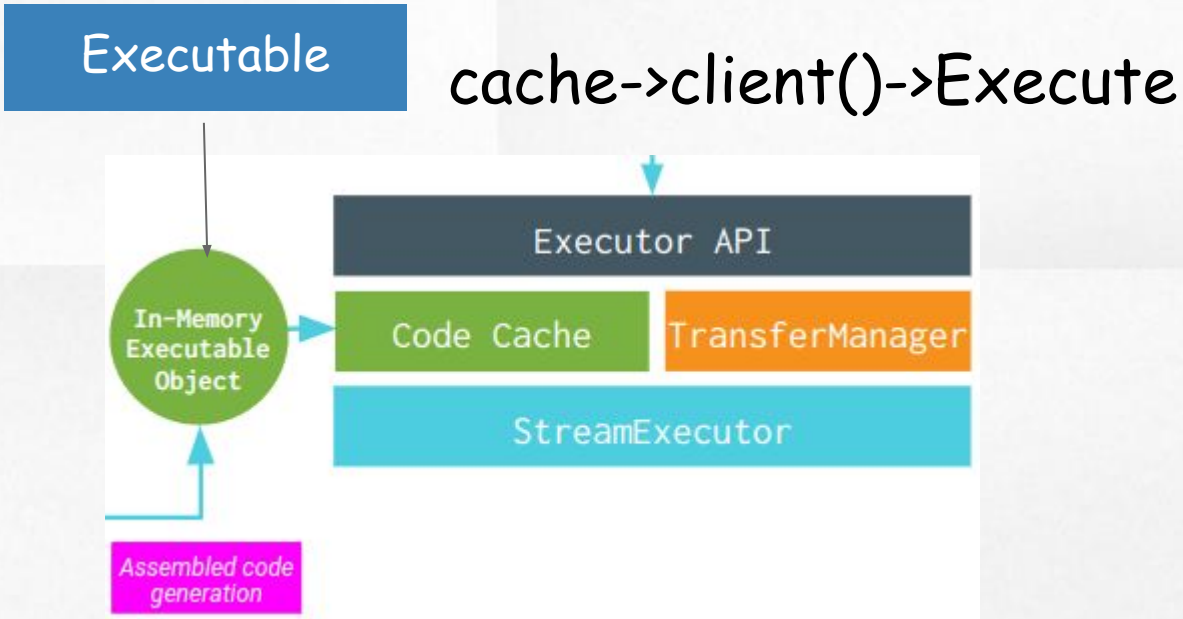
xla/service/gpu_compiler.{h,c}

OptimizeHloModule : HLOの最適化

LLVMでGPU命令コードの生成

GpuExecutable

cacheの生成＆実行



TensorFlow w/XLA: TensorFlow, Compiled! Expressiveness with performance
https://autodiff-workshop.github.io/slides/JeffDean.pdf

Client::Execute

xla/client/client.cc

```
StatusOr<std::unique_ptr<GlobalData>> Client::Execute(
    const Computation& computation,
    tensorflow::gtl::ArraySlice<GlobalData*> arguments,
    const ExecutionOptions* execution_options,
    ExecutionProfile* execution_profile) {
  ExecuteRequest request;
  ExecuteResponse response;

  VLOG(1) << "making execute request: " << request.ShortDebugString();
  Status s = stub_->Execute(&request, &response);
  VLOG(1) << "done with request";
```

Service::Execute

xla/service/service.cc

```
TF_ASSIGN_OR_RETURN(
  std::shared_ptr<Executable> executable,
   BuildAndCacheExecutable(versioned_handle, std::move(module_config),
                           arguments, execute_backend_.get(),
                           execute_backend_->default_stream_executor(),
                           result->mutable_profile()));


TF_ASSIGN_OR_RETURN(
   *result->mutable_output(),
   ExecuteAndRegisterResult(
      executable.get(), arguments, execute_backend_.get(),
      execute_backend_->default_stream_executor(),
      "result of " + user_computation->name(), result->mutable_profile()));
```

Service::BuildAndCacheExecutable

xla/service/service.cc

```
std::shared_ptr<Executable> executable =
    compilation_cache_.LookUp(versioned_handle, *module_config);

if (executable != nullptr) {
  // Executable found in the computation cache.  キャッシュされていた！
  if (profile != nullptr) {
    profile->set_compilation_cache_hit(true);
  }
  return executable;
}
```

Service::BuildAndCacheExecutable

xla/service/service.cc

```
HloModuleConfig original_module_config = *module_config;

// キャッシュされていないときは、キャッシュを生成する

TF_ASSIGN_OR_RETURN(
    std::unique_ptr<Executable> executable_unique_ptr,
    BuildExecutable(versioned_handle, std::move(module_config),
                    /*executable_for_compute_constant=*/false, arguments,
                    backend, executor));
```

Service::ExecuteAndRegisterResult

xla/service/service..cc

```
result = executable->ExecuteOnStreamWrapper<se::DeviceMemoryBase>(
            &run_options[0], profile, arguments));
```

Executable::ExecuteOnStreamWrapper

xla/service/exacutable.h

```
VLOG(1) << "enqueueing executable on stream...";
 // If the profiling flag isn't enabled, we pass nullptr as the profile to
 // indicate profiling is not requested.
 HloExecutionProfile hlo_execution_profile;
 legacy_flags::ServiceFlags* flags = legacy_flags::GetServiceFlags();
 HloExecutionProfile* profile_ptr =
     flags->xla_hlo_profile && hlo_profiling_enabled() ? &hlo_execution_profile
                                                       : nullptr;

 auto return_value = ExecuteOnStream(run_options, arguments, profile_ptr);
```

ExecuteOnStream

xla/service/cpu/cpu_executable.cc

```
se::Stream* stream = run_options->stream();
```

メモリの割当て

```
DeviceMemoryAllocator* memory_allocator = run_options->allocator();
std::vector<se::DeviceMemoryBase> buffers(assignment_->Allocations().size());
AllocateBuffers(
    memory_allocator, stream->parent()->device_ordinal(), &buffers);
```

関数の実行

```
ExecuteComputeFunction(run_options, arguments, buffers,
                                        hlo_execution_profile));
```

ExecuteComputeFunction

xla/service/cpu/cpu_executable.cc

デバイスコードに変換された関数 (compute_function_) を実行

```
compute_function_(result_buffer, run_options, args_array.data(),
                  buffer_pointers.data(), profile_counters.data());
```

CpuExecutableのコンストラクタで compute_function_ は設定

```
CpuExecutable::CpuExecutable( ….,
                              Const string& entry_function_name, …) {

 llvm::JITSymbol sym = jit_->FindSymbol(entry_function_name);
compute_function_ =reinterpret_cast<ComputeFunctionType>(sym.getAddress());
}
```

ちょっと確認してみよう！

環境変数:TF_CPP_MIN_VLOG_LEVEL

C++コード内にはデバックのために下記のようなマクロ(VLOG_IS_ON)を使っている。

下記の条件を満足させるには、
TF_CPP_MIN_VLOG_LEVELに1以上の値を設定すればいい。

```
# export TP_CPP_MIN_VLOG_LEVEL=1
```

XLA_CPU でグラフをダンプ

```
def testXLA_JIT(self):

  with tf.Session() as sess:
    options  = tf.RunOptions(output_partition_graphs=True)
    metadata = tf.RunMetadata()

    x = tf.placeholder(tf.float32, [2], name="x")
    with tf.device("device:XLA_CPU:0"):
      y = x * 2
    result = sess.run(y, {x: [1.5, 0.5]}, options=options,
                         run_metadata=metadata)
```

EncapsulateSubgraphsInFunctions

jit/encapsulate_subgraphs_pass.cc

```
if (VLOG_IS_ON(1)) {
    dump_graph::DumpGraphToFile(
                    "before_encapsulate_subgraphs",
                    **options.graph, options.flib_def);
}
.....
if (VLOG_IS_ON(1)) {
    dump_graph::DumpGraphToFile(
                    "after_encapsulate_subgraphs",
                    *graph_out, options.flib_def);
}
```

生成されたグラフ

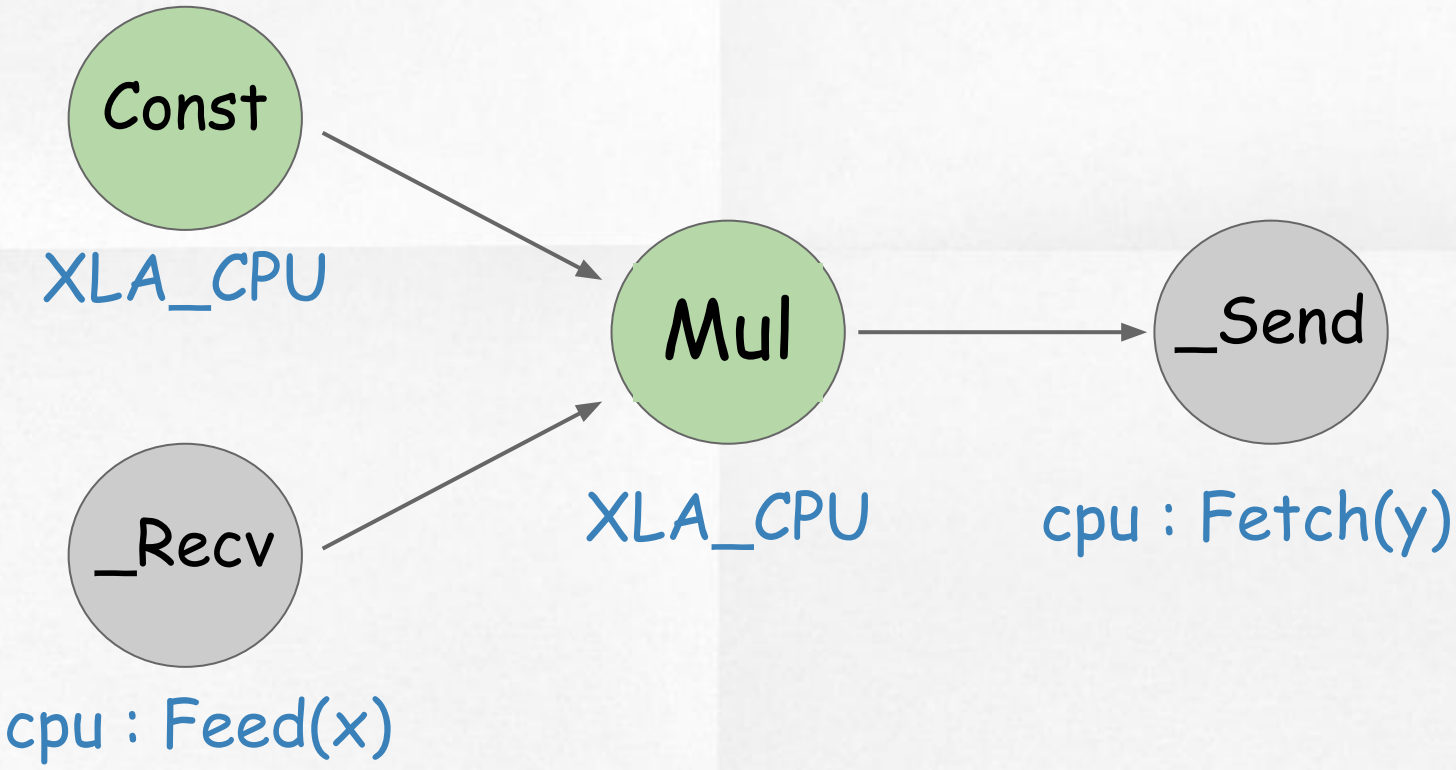
EncapsulateSubgraphsPass前に生成されたグラフ

before_encapsulate_subgraphs.pbtxt

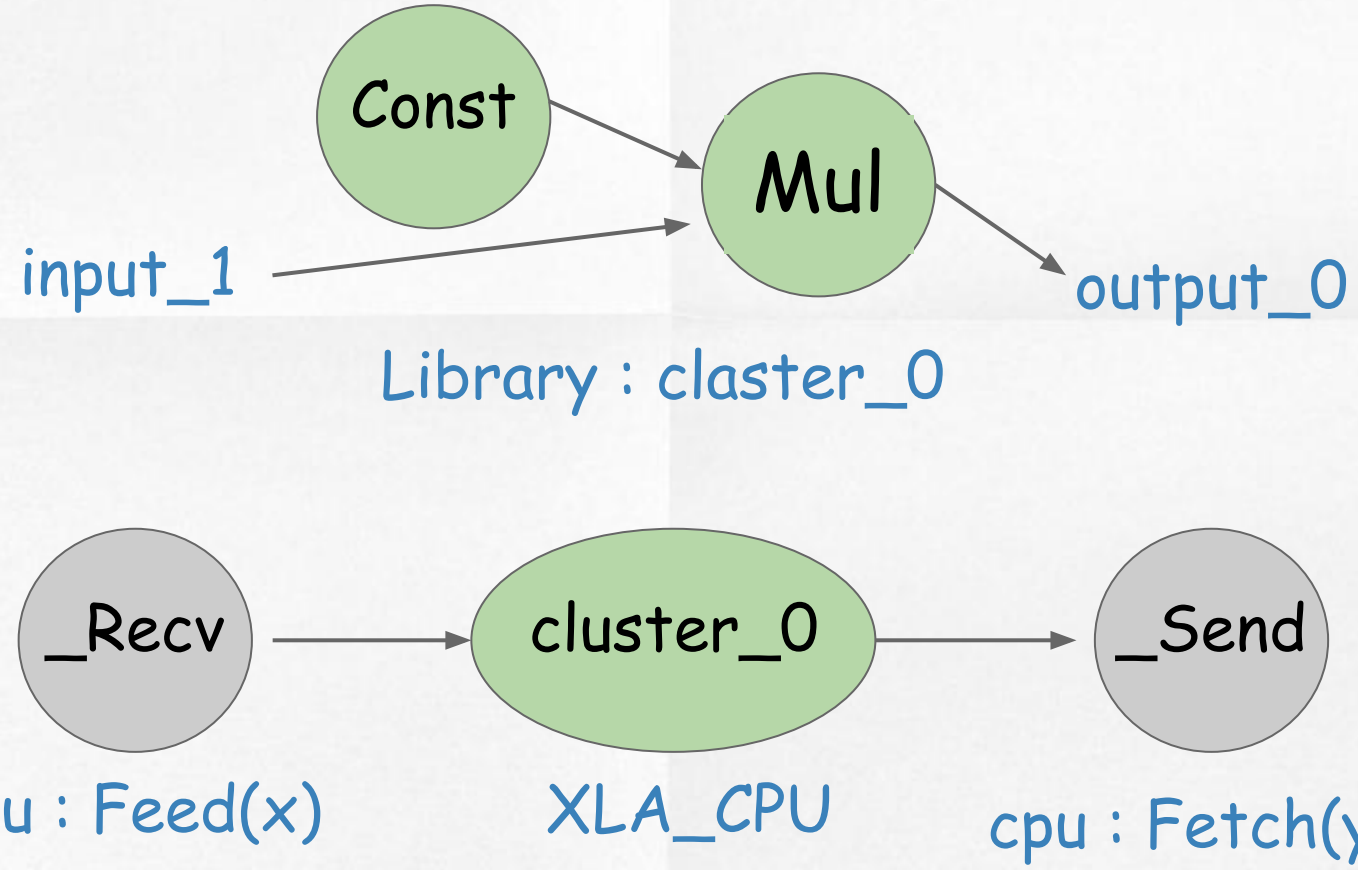
EncapsulateSubgraphsPass後に生成されたグラフ

after_encapsulate_subgraphs.pbtxt

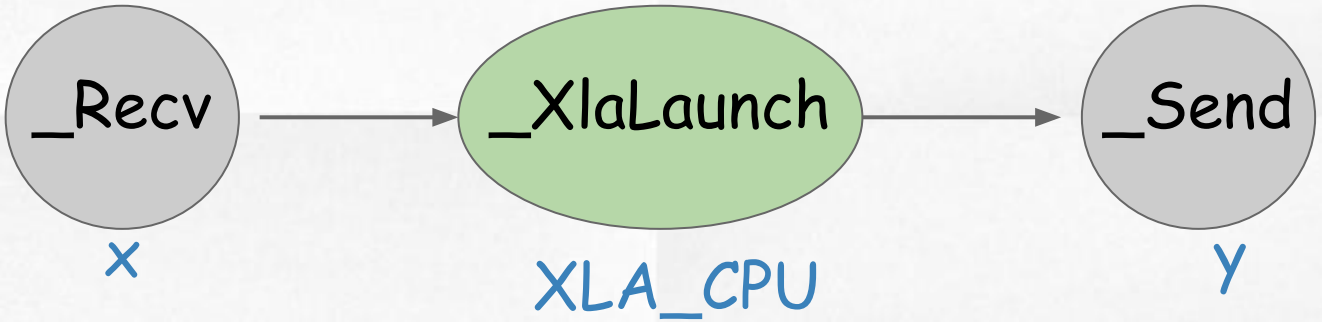
EncapsulateSubgraphsPass前



EncapsulateSubgraphsPass後



BuildXlaLaunchOpsPass後



JITで実行する正攻法は？

Using JIT Compilation

https://www.tensorflow.org/versions/master/experimental/xla/jit

Sessionでは、

```
# Config to turn on JIT compilation

config = tf.ConfigProto()

config.graph_options.optimizer_options.global_jit_level =
tf.OptimizerOptions.ON_1


sess = tf.Session(config=config)
```

Using JIT Compilation

https://www.tensorflow.org/versions/master/experimental/xla/jit

マニュアルでは、

```
jit_scope = tf.contrib.compiler.jit.experimental_jit_scope

x = tf.placeholder(np.float32)
with jit_scope():
  y = tf.add(x, x)  # The "add" will be compiled with XLA.
```

Using JIT Compilation

https://www.tensorflow.org/versions/master/experimental/xla/jit

Placing operators on XLA devices

```
with
tf.device("/job:localhost/replica:0/task:0/device:XLA_GPU:0"):
  output = tf.add(input1, input2)
```

compiler/jit.py

```
def experimental_jit_scope(compile_ops=True):
  """Enable or disable JIT compilation of operators within the scope.
  NOTE: This is an experimental feature.
  The compilation is a hint and only supported on a best-effort basis.

  Example usage:
    with tf.contrib.compiler.experimental_jit_scope():
      c = tf.matmul(a, b)  # compiled
    with tf.contrib.compiler.experimental_jit_scope(compile_ops=False):
      d = tf.matmul(a, c)  # not compiled
    with tf.contrib.compiler.experimental_jit_scope(
        compile_ops=lambda node_def: 'matmul' in node_def.op.lower()):
      e = tf.matmul(a, b) + d  # matmul is compiled, the addition is not.
```

Plugin

Intel Nervana
Graphcore

もXLAをサポートするかも？

r1.2でXLA Backendを導入
r1.1に対して変更がされています

また、
r1.3でpluginが追加されました

XLA Backend導入で修正されたコード

r1.2

compiler/tf2xla/xla_op_registry.{h,cc}

r1.3 (追加)

compiler/plugin

backend (tf2xla/xla_op_registry.h)

```
// REGISTER_XLA_BACKEND() registers an XLA backend. Example usage:
// REGISTER_XLA_BACKEND(DEVICE_GPU_XLA_JIT, kGpuAllTypes, GpuOpFilter);

#define REGISTER_XLA_BACKEND(NAME, ...) \
  REGISTER_XLA_BACKEND_UNIQ_HELPER(__COUNTER__, NAME, __VA_ARGS__)

#define REGISTER_XLA_BACKEND_UNIQ_HELPER(COUNTER, NAME, ...) \
  REGISTER_XLA_BACKEND_UNIQ(COUNTER, NAME, __VA_ARGS__)

#define REGISTER_XLA_BACKEND_UNIQ(CTR, NAME, ...) \
  static ::tensorflow::XlaBackendRegistrar        \
      xla_backend_registrar__body__##CTR##__object(NAME, __VA_ARGS__);
```

backend (tf2xla/xla_op_registry.{h,cc})

```
class XlaBackendRegistrar {
 public:
  XlaBackendRegistrar(StringPiece name, gtl::ArraySlice<DataType> types,
                      XlaOpRegistry::BackendOpFilter op_filter = nullptr);
};

XlaBackendRegistrar::XlaBackendRegistrar(
    StringPiece name, gtl::ArraySlice<DataType> types,
    XlaOpRegistry::BackendOpFilter op_filter) {

  XlaOpRegistry& registry = XlaOpRegistry::Instance();
  registry.RegisterBackend(name.ToString(), types, op_filter);
}
```

backend (tf2xla/xla_op_registry.h)

```
// Registers an XLA backend. `compilation_device_name` is the name of the
// device used for symbolic execution during compilation. `supported_types`
// is the list of non-resource types supported by the device. Each operators
// will be registered for the intersection of the operator's supported types
// and the device's supported types. `backend_op_filter` is a function used
// to exclude or modify operator registrations on the device; it may be
// nullptr, in which case all ops are included.
// `backend_op_filter` should return true if the op should be registered on
// the device; it may optionally modify the KernelDef.

typedef bool (*BackendOpFilter)(KernelDef* kdef);
static void RegisterBackend(const string& compilation_device_name,
                            gtl::ArraySlice<DataType> supported_types,
                            BackendOpFilter op_filter);
```

backend (tf2xla/xla_op_registry.cc)

CPUもbackendに！

```
bool CpuOpFilter(KernelDef* kdef) {
  // TODO(b/34339814): implement inverse erf for double types and remove this
  // workaround.
  if (kdef->op() == "RandomStandardNormal") {
    kdef->clear_constraint();
    // Change the type constraint to permit only DTD_FLOAT.
    KernelDef::AttrConstraint* attr_constraint = kdef->add_constraint();
    attr_constraint->set_name("dtype");
    attr_constraint->mutable_allowed_values()->mutable_list()->add_type(
        DT_FLOAT);
    return true;
  }
  return true;
}
REGISTER_XLA_BACKEND(DEVICE_CPU_XLA_JIT, kCpuAllTypes, CpuOpFilter);
```

backend (tf2xla/xla_op_registry.cc)

GPUもbackendに！

```
bool GpuOpFilter(KernelDef* kdef) {
  // TODO(b/31361304): The GPU backend does not parallelize PRNG ops, leading to
  // slow code.
  // TODO(b/34969189) The implementation of TruncatedNormal generates illegal
  // code on GPU.
  if (kdef->op() == "RandomStandardNormal" || kdef->op() == "RandomUniform" ||
      kdef->op() == "RandomUniformInt" || kdef->op() == "TruncatedNormal") {
    return false;
  }
  return true;
}

REGISTER_XLA_BACKEND(DEVICE_GPU_XLA_JIT, kGpuAllTypes, GpuOpFilter);
```

Pluginのサンプルコード

compiler/plugin/BUILD

"""Configuration file for an XLA plugin.
- please don't check in changes to this file
- to prevent changes appearing in git status, use:
 git update-index --assume-unchanged
tensorflow/compiler/plugin/BUILD
To add additional devices to the XLA subsystem, add
targets to the
dependency list in the 'plugin' target. For instance:
  deps =
["//tensorflow/compiler/plugin/example:plugin_lib"],
"""

compiler/plugin/executor

pluginでは、executorを利用している

- BUILD
- device.cc
- compiler.{cc, h}
- executable.{cc, h}
- executor.{cc, h}
- platform.{cc, h}
- platform_id.h
- transfer_manager.{cc, h}

XLA_EXECの登録 (device.cc)

```
const char* const DEVICE_XLA_EXEC = "XLA_EXEC";
const char* const DEVICE_EXEC_XLA_JIT =
                                        "XLA_EXEC_JIT";
constexpr std::array<DataType, 5> kExecAllTypes = {
            {DT_INT32, DT_FLOAT, DT_BOOL, DT_DOUBLE, DT_INT64}};
class XlaExaDeviceFactory : public DeviceFactory {
 public:
  Status CreateDevices(const SessionOptions& options, const
string& name_prefix,
                       std::vector<Device*>* devices) override;
```

XLA_EXECの登録 (device.cc)

```
REGISTER_LOCAL_DEVICE_FACTORY(
    DEVICE_XLA_EXEC, XlaExaDeviceFactory, 40);

constexpr std::array<DataType, 5> kAllXlaCpuTypes = {{
    DT_INT32, DT_INT64, DT_FLOAT,
    DT_DOUBLE, DT_BOOL}};

REGISTER_XLA_LAUNCH_KERNEL(
    DEVICE_XLA_EXEC, XlaDeviceLaunchOp, kExecAllTypes);
REGISTER_XLA_DEVICE_KERNELS(
    DEVICE_XLA_EXEC, kExecAllTypes);
```

デバイスの登録

core/common_runtime/device_factory.{h,c}

```
// The default priority values for built-in devices is:
// GPU: 210
// SYCL: 200
// GPUCompatibleCPU: 70
// ThreadPoolDevice: 60
// Default: 50
```

REGISTER_LOCAL_DEVICE_FACTORYマクロで設定する

XLA_EXECの登録 (device.cc)

```
REGISTER_XLA_BACKEND(
        DEVICE_EXEC_XLA_JIT, kExecAllTypes, OpFilter);
```

tf2xla/xla_op_registry.h に r1.2で追加された

```
// REGISTER_XLA_BACKEND() registers an XLA backend. Example usage:
// REGISTER_XLA_BACKEND(DEVICE_GPU_XLA_JIT, kGpuAllTypes, GpuOpFilter);
#define REGISTER_XLA_BACKEND(NAME, ...) \
  REGISTER_XLA_BACKEND_UNIQ_HELPER(__COUNTER__, NAME, __VA_ARGS__)
```

Compile

plugin/executor/compiler.{h,c}

RunHloOptimization：HLOの最適化

```
// Typically you would visit the HLO graph, building up a compiled equivalent
// In this case we are using an Hlo evaluator at execution time, so we don't
 // need to compile anything
// ここでPluginに対応したコード生成を行う
```

ExecutorExecutableの生成

StreamExecutor Runtime Library

https://github.com/henline/streamexecutordoc

- executable.{cc, h}
  class ExecutorExecutable : public Executable

- executor.{cc, h}
  class ExecutorExecutor
  : public internal::StreamExecutorInterface

- platform.{cc, h}
  class ExecutorPlatform : public Platform

ExecuteOnStream

plugin/executor/executable.{h,cc}

```
virtual StatusOr<perftools::gputools::DeviceMemoryBase>
ExecuteOnStream(
    const ServiceExecutableRunOptions* run_options,
    tensorflow::gtl::ArraySlice<perftools::gputools::DeviceMemoryBase>
      arguments,
    HloExecutionProfile* hlo_execution_profile) = 0;
```

各デバイスに対応した実装をする必要がある

ExecuteOnStream

```
// ここで引数の処理をしている  => arg_literals_ptrs に処理結果がストアされる

// HloEvaluator を使って、グラフ( computation )を実行する
// Execute the graph using the evaluator
HloEvaluator evaluator;
TF_ASSIGN_OR_RETURN(std::unique_ptr<Literal> output,
                    evaluator.Evaluate(computation, arg_literals_ptrs));
```

ExecuteOnStream

```
// 引数のstream を使って、StreamExecutor (executor) を生成、
// executor の実装 (executor->implementation() ) から
// ExecutorExecutor (executorExecutor)を生成
// Copy the result into the return buffer
perftools::gputools::StreamExecutor* executor(stream->parent());
sep::ExecutorExecutor* executorExecutor(
    static_cast<sep::ExecutorExecutor*>(executor->implementation()));

// 出力バッファを割り当てる
se::DeviceMemoryBase ret =
    AllocateOutputBuffer(executorExecutor, *(output.get()));
```

けど、処理は何もしていないよ。

ExecutorExecutor

plugin/executor/executor.{h,cc}

StreamExecutorインターフェースを実装

・バッファ管理
・メモリ移動
・グラフの実行
・イベント処理
・タイマー管理

pluginは、ライブラリに

compiler/plugin/BUILD

```
cc_library(
    name = "plugin",
    deps = [

"//tensorflow/compiler/plugin/executor:plugin_lib",
    ],
)
```

libplugin.so というライブラリが生成される

XLA_EXECを確認してみよう

```
import tensorflow as tf

def test_xla():
  config = tf.ConfigProto()
  jit_level = tf.OptimizerOptions.ON_1
  config.graph_options.optimizer_options.global_jit_level = jit_level
  with tf.Session(config=config) as sess:
    x = tf.placeholder(tf.float32, [2], name="x")
    with tf.device("device:XXX:0"):
      y = x * 2
    result = sess.run(y, {x: [1.5, 0.5]})
    print('x * 2 = result : ', result)

if __name__ == '__main__':
  test_xla()
```

```
with tf.device("device:CPU:0"):

with tf.device("device:XLA_CPU:0"):

with tf.device("device:XLA_EXEC:0"):
```

では、どうなるのか？

```
$ python test_jit.py

with tf.device("device:CPU:0"):
('x * 2 = result : ', array([ 3.,  1.], dtype=float32))

with tf.device("device:XLA_CPU:0"):
platform Executor present with 1 visible devices
platform Host present with 1 visible devices
XLA service 0x25cf010 executing computations on platform Host. Devices:
StreamExecutor device (0): <undefined>, <undefined>
('x * 2 = result : ', array([ 3.,  1.], dtype=float32))
```

with tf.device("device:XLA_EXEC:0"):

F tensorflow/compiler/xla/statusor.cc:41] Attempting to fetch value instead of handling error Not found: could not find registered computation placer for platform Executor -- check target linkage
Aborted (core dumped)

エラーが出て、実行できず。。。

could not find registered computation placer for platform Executor

compiler/xla/service/computation_placer.cc

```
static bool InitModule() {
  xla::ComputationPlacer::RegisterComputationPlacer(
                              se::host::kHostPlatformId,
                              &CreateComputationPlacer);
  xla::ComputationPlacer::RegisterComputationPlacer(
                              se::cuda::kCudaPlatformId,
                              &CreateComputationPlacer);
  return true;
}

static bool module_initialized = InitModule();
```

```
namespace perftools { namespace gputools { namespace executorplugin {
   extern const Platform::Id kExecutorPlatformId;
} } }

static bool InitModule() {
  xla::ComputationPlacer::RegisterComputationPlacer(
                           se::host::kHostPlatformId,
                           &CreateComputationPlacer);
  xla::ComputationPlacer::RegisterComputationPlacer(
                           se::cuda::kCudaPlatformId,
                           &CreateComputationPlacer);
  xla::ComputationPlacer::RegisterComputationPlacer(
                           se::executorplugin::kExecutorPlatformId,
                           &CreateComputationPlacer);
  return true;
}

static bool module_initialized = InitModule();
```

再ビルド

```
$ bazel build --config=opt
//tensorflow/tools/pip_package:build_pip_package

$ sudo cp -p
bazel-bin/tensorflow/python/_pywrap_tensorflow_internal.so
/usr/local/lib/python2.7/dist-packages/tensorflow/python/_p
ywrap_tensorflow_internal.so
```

with tf.device("device:XLA_EXEC:0"):

platform Executor present with 1 visible devices
platform Host present with 1 visible devices
XLA service 0x4030a50 executing computations on platform Executor. Devices:
StreamExecutor device (0): Executor, 1.0

XLA_CPU と同じように、StreamExecutorが呼ばれた！

この後、エラーメッセージが表示されたが。。。

```
Traceback (most recent call last):
  File "test_jit.py", line 24, in <module>
    test_xla()
```

```
  File "test_jit.py", line 20, in test_xla
    result = sess.run(y, {x: [1.5, 0.5]})
  File ".../tensorflow/python/client/session.py", line 895, in run
    run_metadata_ptr)
  File ".../tensorflow/python/client/session.py", line 1124, in _run
    feed_dict_tensor, options, run_metadata)
  File ".../tensorflow/python/client/session.py", line 1321, in _do_run
    options, run_metadata)
  File ".../tensorflow/python/client/session.py", line 1340, in _do_call
    raise type(e)(node_def, op, message)
tensorflow.python.framework.errors_impl.UnimplementedError: Implicit
broadcasting is currently unsupported in HLO evaluator Shape Mismatch: f32[2]
vs f32[2] vs f32[]:
         [[Node: cluster_0/_0/_1 = _XlaLaunch[Nresources=0, Targs=[DT_FLOAT],
Tconstants=[], Tresults=[DT_FLOAT],
function=cluster_0[_XlaCompiledKernel=true, _XlaNumConstantArgs=0,
_XlaNumResourceArgs=0],
_device="/job:localhost/replica:0/task:0/device:XLA_EXEC:0"](_arg_x_0_0/_3)
]]
```

ExecuteOnStream

```
HloEvaluator evaluator;
TF_ASSIGN_OR_RETURN(std::unique_ptr<Literal> output,
                    evaluator.Evaluate(computation, arg_literals_ptrs));
```

ここでエラーが発生して、途中で return している

xla/service/hlo_evaluator.cc の Evaluateメソッドの
  TF_RETURN_IF_ERROR(computation->Accept(this));
がエラーになる

ここでは、サンプルコードなので、実際には、
ExecuteOnStreamには、
デバイスに対応したコードを実装をする必要がある

ありがとうございました

Twitter : @Vengineer

ブログ : Vengineerの戯言

http://blogs.yahoo.co.jp/verification_engineer

TensorFlow XLAの衝撃

2017年2月20日

http://blogs.yahoo.co.jp/verification_engineer/71016304.html