Tensorflow XLA

wanwei.cai@bitmain.com

2018.3

Terminology

- XLA: Accelerated Linear Algebra
- HLO: XLA high level optimizer
- IR: intermediate representation

Contents

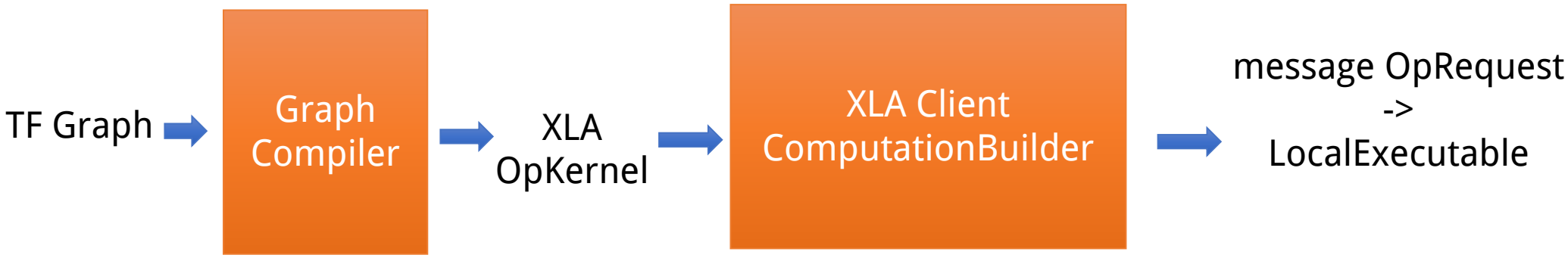
- Introduction of XLA Compilation Flow
- Optimizations of HLO IR
- How to develop XLA Bankend for ASIC

Tensorflow

- Original:
  - Graph -> OpKernel -> Map to CPU/GPU operation
- XLA:
  - Graph -> XLA OpKernel -> XLA Client Op -> Request XLA Service
  - XLA Service response -> XLA HLO IR-> HLO IR Optimization -> LLVM IR -> LLVM IR Optimization -> Map to CPU/GPU/ASIC code

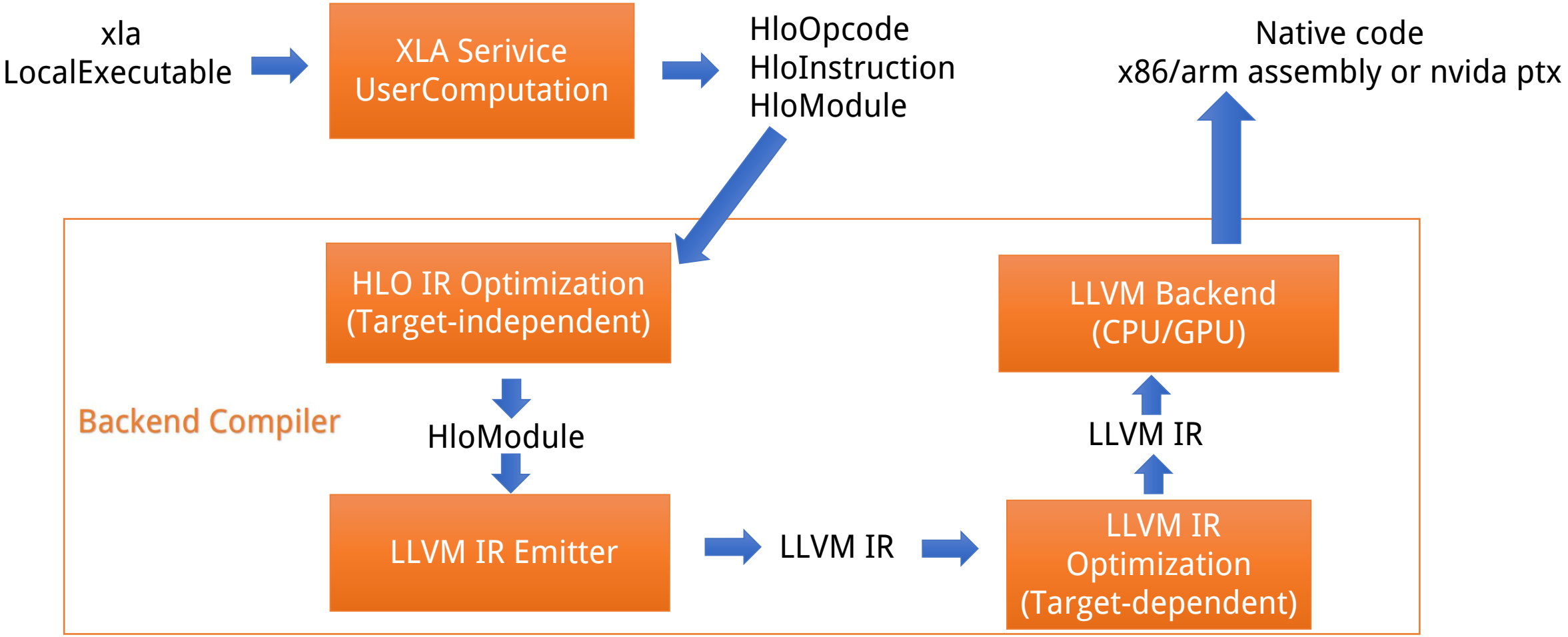
XLA Compilation Flow

- Client



XLA Compilation Flow cont.

- Service



TF Graph

- TF Graph is generated from user tf program
- example of user program

```
from six.moves import xrange

import numpy as np
import tensorflow as tf

device_name = "/gpu:0"

cww_b = tf.Variable(tf.zeros([4]))
cww_W = tf.Variable([[1, 2, 3], [4, 5, 6]], dtype=tf.float32)
cww_x = tf.placeholder(name="x", dtype=tf.float32)
cww_mul = tf.matmul(cww_W, cww_x) + cww_b
cww_relu = tf.nn.relu(cww_mul)

config = tf.ConfigProto()
config.graph_options.optimizer_options.global_jit_level = tf.OptimizerOptions.ON_1
with tf.Session(config=config) as s:
    init = tf.global_variables_initializer()
    s.run(init)

    for step in xrange(0, 1):
        input = np.array([[1, 1, 1, 1], [2, 2, 2, 2], [3, 3, 3, 3]])
        result = s.run(cww_relu, feed_dict={cww_x: input})
        print(step, result)
```

XLA OpKernel

- Location: compiler/tf2xla/kernels/*
- XLA OpKernels are a set of implementations of tf OpKernel to be solved by XLA Compilation Device
  - AddNOp
  - ArgOp
  - BatchMatMulOp
  - MatMulOp
  - ...

OpRequest

- Location: compiler/xla/xla_data.proto
- OpRequest is used by CS communication
  - BinaryOpRequest (e.g. , Add and Dot)
  - BroadcastRequest
  - CallRequest
  - …

XLA HLO IR

- Location: compiler/xla/service/hlo_opcode.h
- XLA HLO IR is generated from xla::Executable by XLA Service UserComputation, and includes:
  - HloModule (composed by HloInstructions)
  - HloInstruction (created from HloOpcode)
  - HloOpcode
    - kAbs, kAdd, kBatchNormTraining, kDot ...
  - the example of HLO IR is as follow:

```
%constant = f32[] constant(0) # metadata=op_type: "Relu" op_name: "Relu"
%arg0 = f32[2,3]{1,0} parameter(0)
%arg2 = f32[3,4]{1,0} parameter(2)
%dot = f32[2,4]{1,0} dot(f32[2,3]{1,0} %arg0, f32[3,4]{1,0} %arg2) # metadata=op_type: "MatMul" op_name: "MatMul"
%arg1 = f32[4]{0} parameter(1)
%broadcast = f32[2,4]{1,0} broadcast(f32[4]{0} %arg1), dimensions={1} # metadata=op_type: "Add" op_name: "add"
%add = f32[2,4]{1,0} add(f32[2,4]{1,0} %dot, f32[2,4]{1,0} %broadcast) # metadata=op_type: "Add" op_name: "add"
%maximum = f32[2,4]{1,0} maximum(f32[] %constant, f32[2,4]{1,0} %add) # metadata=op_type: "Relu" op_name: "Relu"
%tuple = (f32[2,4]{1,0}) tuple(f32[2,4]{1,0} %maximum)
%get-tuple-element = f32[2,4]{1,0} get-tuple-element((f32[2,4]{1,0}) %tuple), index=0
```

HLO IR Optimization

- HLO IR Optimizations are triggered by Backend Compiler, such as xla::gpu::GpuCompiler
  - input and output are both HloModule
  - optimizations includes:
    - simplification
    - algsimp (algebra simplification)
    - reshape-motion
    - constant_folding
    - convolution-folding
    - transpose-folding
    - cse (common-subexpression elimination)
    - dce (dead-computation elimination)
    - fusion
    - fusion merger
    - ...

HLO IR Optimization cont.

- example of fusion optimization

```
%constant = f32[] constant(0) # metadata=op_type: "Relu" op_name: "Relu"
%arg0 = f32[2,3]{1,0} parameter(0)
%arg2 = f32[3,4]{1,0} parameter(2)
%dot = f32[2,4]{1,0} dot(f32[2,3]{1,0} %arg0, f32[3,4]{1,0} %arg2) # metadata=op_type: "MatMul" op_name: "MatMul"
%arg1 = f32[4]{0} parameter(1)
%broadcast = f32[2,4]{1,0} broadcast(f32[4]{0} %arg1), dimensions={1} # metadata=op_type: "Add" op_name: "add"
%add = f32[2,4]{1,0} add(f32[2,4]{1,0} %dot, f32[2,4]{1,0} %broadcast) # metadata=op_type: "Add" op_name: "add"
%maximum = f32[2,4]{1,0} maximum(f32[] %constant, f32[2,4]{1,0} %add) # metadata=op_type: "Relu" op_name: "Relu"
```

```
%constant = f32[] constant(0) # metadata=op_type: "Relu" op_name: "Relu"
%arg0 = f32[2,3]{1,0} parameter(0)
%arg2 = f32[3,4]{1,0} parameter(2)
%dot = f32[2,4]{1,0} dot(f32[2,3]{1,0} %arg0, f32[3,4]{1,0} %arg2) # metadata=op_type: "MatMul" op_name: "MatMul"
%arg1 = f32[4]{0} parameter(1)
%fusion = f32[2,4]{1,0} fusion:kLoop(f32[] %constant, f32[2,4]{1,0} %dot, f32[4]{0} %arg1) # metadata=op_type: "Relu" op_name: "Relu"
  fused_computation (constant.param_0: f32[], dot.param_1: f32[2,4], arg1.param_2: f32[4]) -> f32[2,4] {
    %constant.param_0 = f32[] parameter(0)
    %dot.param_1 = f32[2,4]{1,0} parameter(1)
    %arg1.param_2 = f32[4]{0} parameter(2)
    %broadcast = f32[2,4]{1,0} broadcast(f32[4]{0} %arg1.param_2), dimensions={1} # metadata=op_type: "Add" op_name: "add"
    %add = f32[2,4]{1,0} add(f32[2,4]{1,0} %dot.param_1, f32[2,4]{1,0} %broadcast) # metadata=op_type: "Add" op_name: "add"
    %maximum = f32[2,4]{1,0} maximum(f32[] %constant.param_0, f32[2,4]{1,0} %add) # metadata=op_type: "Relu" op_name: "Relu"
  }
```

LLVM IR

(Location: compiler/xla/service/gpu/ir_emitter.cc)

- LLVM IR is generated from HLO IR by ir_emitter
  - HandleDot
  - HandleBitcast
  - ...

```
entry:
  %_arg1.typed = bitcast i8* %_arg1.raw to [4 x float]*
  %_fusion.typed = bitcast i8* %_fusion.raw to [2 x [4 x float]]*
  %_dot.raw = getelementptr inbounds i8, i8* %temp_buffer, i64 0
  %_dot.typed = bitcast i8* %_dot.raw to [2 x [4 x float]]*
  %0 = call i32 @llvm.nvvm.read.ptx.sreg.ctaid.x(), !range !2
  %block_id = zext i32 %0 to i64
  %1 = call i32 @llvm.nvvm.read.ptx.sreg.tid.x(), !range !3
  %thread_id = zext i32 %1 to i64
  %2 = mul nuw nsw i64 %block_id, 8
  %linear_index = add nuw nsw i64 %2, %thread_id
  %3 = icmp ult i64 %linear_index, 8
  br i1 %3, label %in_bounds-true, label %in_bounds-after

in_bounds-after:                                  ; preds = %in_bounds-true, %entry
  ret void

in_bounds-true:                                   ; preds = %entry
  %4 = udiv i64 %linear_index, 1
  %5 = urem i64 %4, 4
  %6 = udiv i64 %linear_index, 4
  %7 = urem i64 %6, 2
  %8 = load float, float* @_constant.typed
  %9 = bitcast [2 x [4 x float]]* %_dot.typed to float*
  %10 = getelementptr inbounds float, float* %9, i64 %linear_index
  %11 = load float, float* %10, !alias.scope !4, !noalias !7
  %12 = getelementptr inbounds [4 x float], [4 x float]* %_arg1.typed, i64 0, i64 %5
  %13 = load float, float* %12, !invariant.load !9, !noalias !10
  %14 = fadd fast float %11, %13
  %15 = call fast float @llvm.maxnum.f32(float %8, float %14)
  %16 = bitcast [2 x [4 x float]]* %_fusion.typed to float*
  %17 = getelementptr inbounds float, float* %16, i64 %linear_index
  store float %15, float* %17, !alias.scope !7, !noalias !4
  br label %in_bounds-after
```

Target-dependent Optimization in LLVM

- LLVM IR example after optimization of GPU LLVM

```
%temp_buffer5 = addrspacecast i8* %temp_buffer to i8 addrspace(1)*
%_fusion.raw3 = addrspacecast i8* %_fusion.raw to i8 addrspace(1)*
%_arg1.raw1 = addrspacecast i8* %_arg1.raw to i8 addrspace(1)*
%0 = tail call i32 @llvm.nvvm.read.ptx.sreg.tid.x(), !range !3
%thread_id = zext i32 %0 to i64
%_arg1.typed = bitcast i8 addrspace(1)* %_arg1.raw1 to [4 x float] addrspace(1)*
%1 = and i32 %0, 3
%2 = zext i32 %1 to i64
%3 = bitcast i8 addrspace(1)* %temp_buffer5 to float addrspace(1)*
%4 = getelementptr inbounds float, float addrspace(1)* %3, i64 %thread_id
%5 = load float, float addrspace(1)* %4, align 4, !alias.scope !4, !noalias !7
%6 = getelementptr inbounds [4 x float], [4 x float] addrspace(1)* %_arg1.typed, i64 0, i64 %2
%7 = load float, float addrspace(1)* %6, align 4, !invariant.load !9, !noalias !10
%8 = fadd fast float %7, %5
%9 = tail call fast float @llvm.maxnum.f32(float %8, float 0.000000e+00)
%10 = bitcast i8 addrspace(1)* %_fusion.raw3 to float addrspace(1)*
%11 = getelementptr inbounds float, float addrspace(1)* %10, i64 %thread_id
store float %9, float addrspace(1)* %11, align 4, !alias.scope !7, !noalias !4
ret void
```

Native Code Generation

- Use LLVM Backend for CPU/GPU
  - input is HLO IR
  - output is x86/arm assembly or nvidia ptx

```
PTX:
//
// Generated by LLVM NVPTX Back-End
//

.version 4.2
.target sm_50
.address_size 64

 // .globl       _fusion

.visible .entry _fusion(
 .param .u64 _fusion_param_0,
 .param .u64 _fusion_param_1,
 .param .u64 _fusion_param_2
)
.maxntid 8, 1, 1
{
 .reg .f32       %f<5>;
 .reg .b32       %r<3>;
 .reg .b64       %rd<12>;

 ld.param.u64    %rd1, [_fusion_param_0];
 ld.param.u64    %rd2, [_fusion_param_2];
 cvta.to.global.u64      %rd3, %rd2;
 ld.param.u64    %rd4, [_fusion_param_1];
 cvta.to.global.u64      %rd5, %rd4;
 cvta.to.global.u64      %rd6, %rd1;
 mov.u32         %r1, %tid.x;
 and.b32         %r2, %r1, 3;
 mul.wide.u32    %rd7, %r1, 4;
 add.s64         %rd8, %rd3, %rd7;
 ld.global.nc.f32        %f1, [%rd8];
 mul.wide.u32    %rd9, %r2, 4;
 add.s64         %rd10, %rd6, %rd9;
 ld.global.nc.f32        %f2, [%rd10];
 add.f32         %f3, %f2, %f1;
 max.f32         %f4, %f3, 0f00000000;
 add.s64         %rd11, %rd5, %rd7;
 st.global.f32   [%rd11], %f4;
 ret;
}
```

Contents

- Introduction of XLA Compilation Flow
- Optimizations of HLO IR
- How to develop XLA Bankend for ASIC

Optimizations of HLO IR

- simplification
- algsimp (algebra simplification)
- reshape-motion
- constant_folding
- convolution-folding
- transpose-folding
- cse (common-subexpression elimination)
- dce (dead-computation elimination)
- fusion
- fusion merger
- ...

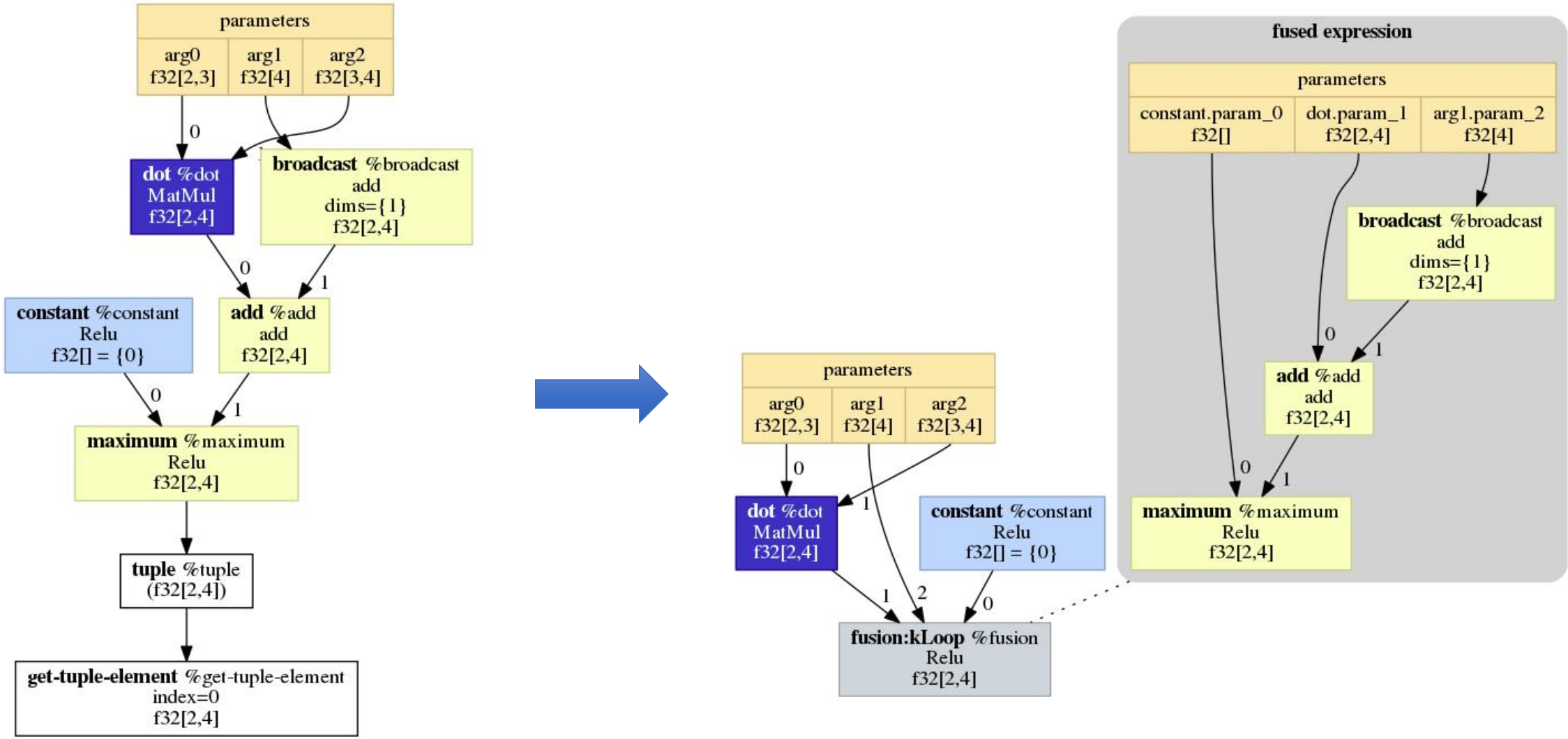
HLO IR Optimization cont.

- example of fusion optimization

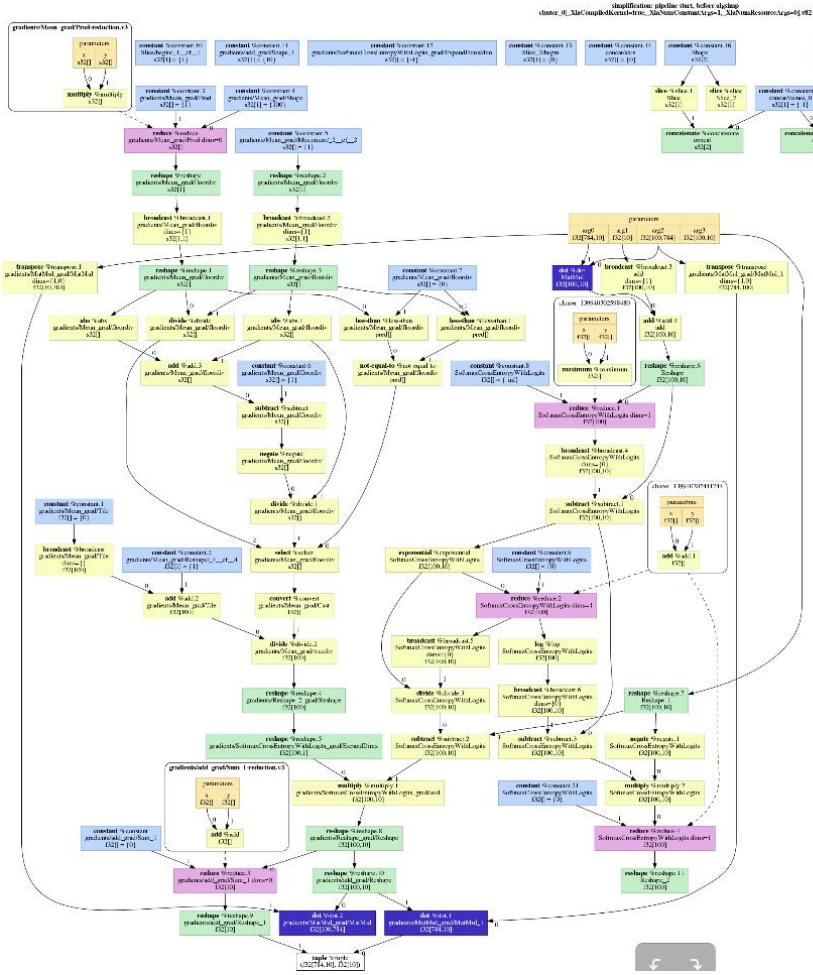
```
%constant = f32[] constant(0) # metadata=op_type: "Relu" op_name: "Relu"
%arg0 = f32[2,3]{1,0} parameter(0)
%arg2 = f32[3,4]{1,0} parameter(2)
%dot = f32[2,4]{1,0} dot(f32[2,3]{1,0} %arg0, f32[3,4]{1,0} %arg2) # metadata=op_type: "MatMul" op_name: "MatMul"
%arg1 = f32[4]{0} parameter(1)
%broadcast = f32[2,4]{1,0} broadcast(f32[4]{0} %arg1), dimensions={1} # metadata=op_type: "Add" op_name: "add"
%add = f32[2,4]{1,0} add(f32[2,4]{1,0} %dot, f32[2,4]{1,0} %broadcast) # metadata=op_type: "Add" op_name: "add"
%maximum = f32[2,4]{1,0} maximum(f32[] %constant, f32[2,4]{1,0} %add) # metadata=op_type: "Relu" op_name: "Relu"
```

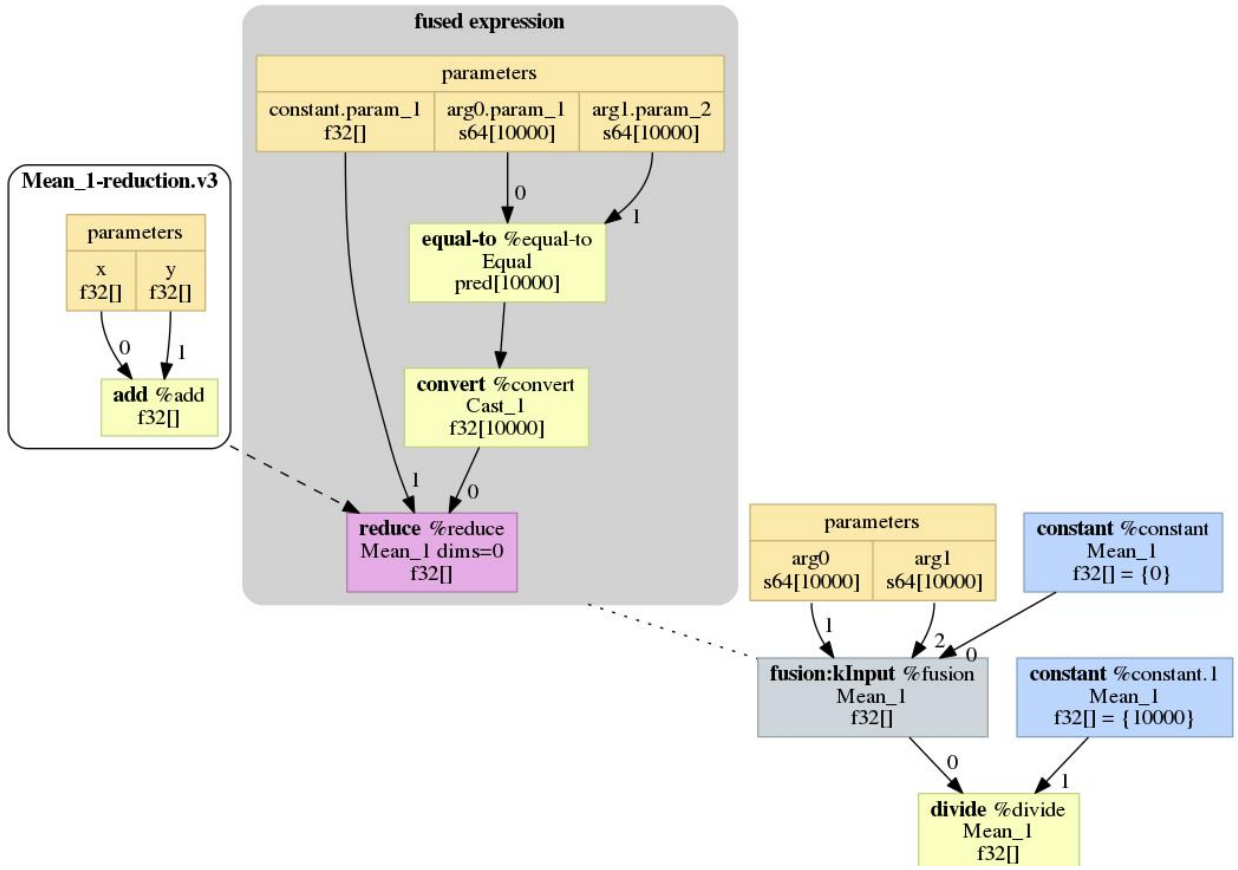
```
%constant = f32[] constant(0) # metadata=op_type: "Relu" op_name: "Relu"
%arg0 = f32[2,3]{1,0} parameter(0)
%arg2 = f32[3,4]{1,0} parameter(2)
%dot = f32[2,4]{1,0} dot(f32[2,3]{1,0} %arg0, f32[3,4]{1,0} %arg2) # metadata=op_type: "MatMul" op_name: "MatMul"
%arg1 = f32[4]{0} parameter(1)
%fusion = f32[2,4]{1,0} fusion:kLoop(f32[] %constant, f32[2,4]{1,0} %dot, f32[4]{0} %arg1) # metadata=op_type: "Relu" op_name: "Relu"
  fused_computation (constant.param_0: f32[], dot.param_1: f32[2,4], arg1.param_2: f32[4]) -> f32[2,4] {
    %constant.param_0 = f32[] parameter(0)
    %dot.param_1 = f32[2,4]{1,0} parameter(1)
    %arg1.param_2 = f32[4]{0} parameter(2)
    %broadcast = f32[2,4]{1,0} broadcast(f32[4]{0} %arg1.param_2), dimensions={1} # metadata=op_type: "Add" op_name: "add"
    %add = f32[2,4]{1,0} add(f32[2,4]{1,0} %dot.param_1, f32[2,4]{1,0} %broadcast) # metadata=op_type: "Add" op_name: "add"
    %maximum = f32[2,4]{1,0} maximum(f32[] %constant.param_0, f32[2,4]{1,0} %add) # metadata=op_type: "Relu" op_name: "Relu"
  }
```

Fusion for the example in slice 7



All Optimizations for mnist softmax





Contents

- Introduction of XLA Compilation Flow
- Optimizations of HLO IR
- How to develop XLA Bankend for ASIC

How to develop XLA Bankend for ASIC

- StreamExecutor: It is conceptually the "handle" for a device.
- xla::Compiler: This class encapsulates the compilation of an HLO computation into an xla::Executable
- xla::Executable: This class is used to launch a compiled computation on the platform.
- xla::TransferManager: This class enables backends to provide platform-specific mechanisms for constructing XLA literal data from given device memory handles. In other words, it helps encapsulate the transfer of data from the host to the device and back

How to develop XLA Bankend for ASIC

- StreamExecutor: It is conceptually the "handle" for a device.
  - AllocateArray
  - SynchronousMemcpyH2D
  - GetKernel

How to develop XLA Bankend for ASIC

- xla::Compiler: This class encapsulates the compilation of an HLO computation into an xla::Executable
  - https://www.tensorflow.org/performance/xla/operation_semantics
  - ir_emitter of AISC
    - such as HandleDot：implemented by bmkernel

How to develop XLA Bankend for ASIC

- xla::Executable: This class is used to launch a compiled computation on the platform.
  - implemented by bmtap runtime (PCIE/SOC/CModel)