SPU 二次开发指北
Using SPU in Research

Wen-jie Lu

2024-02-26

To Use Customized Protocols

- Situations:
  - new developed sub-functions or a better construction
- Possible ways:
  - Python hijack
  - Intrinsic calls
  - C++ programming

Working Scenario: Evaluating GELU

- Gaussian Error Linear Unit (GELU) $\text{Gelu}(x) = 0.5x(1 + \tanh(\sqrt{2/\pi} \cdot (x + 0.044715 \cdot x^3))$
- By default, SPU will (approximately) evaluate tanh(*) which is expensive to use
- How can we switch to a customized Gelu protocol ?
- Difficulty:
  - The Gelu function has been broken into multiple pieces of smaller functions.
  - Pattern matching the Gelu graph from the whole program is tedious
- Solution: Python context hijack

Python Context Hijack

- Replacing the target function on-the-fly

```python
## Define a 'large' program that calls jax.nn.gelu
def program_that_use_gelu(x):
    y = jax.numpy.square(x)
    return jax.nn.gelu(x + y)
```

1

```python
## A customized and faster gelu protocol from BOLT
def customized_gelu(x):
    # 0.5*x + \sum_{i=0}^{4} a_i*|x|^i
    is_neg = x < 0.0
    abs_x = jax.lax.select(is_neg, -x, x)
    is_inside_approx = abs_x < 3.0

    coeffs = jnp.array(
        [
            0.0002743776353465,
            -0.03798164612714154,
            0.5410550166368381,
            -0.18352506127082727,
            0.020848611754127593,
        ]
    )

    x2 = jax.numpy.square(abs_x)
    x4 = jax.numpy.square(x2)
    x3 = abs_x * x2
    middle_seg = (
        0.5 * x
        + coeffs[0]
        + coeffs[1] * abs_x
        + coeffs[2] * x2
        + coeffs[3] * x3
        + coeffs[4] * x4
    )

    # x > 0 and |x| >= 3 => x > 3
    cond = jax.numpy.logical_and(~is_neg, ~is_inside_approx)
    ret = cond * x
    # |x| < 3
    ret = is_inside_approx * middle_seg + ret
    # If not hit (i.e., x < -3), then return 0
    return ret
```

2

3

```python
from contextlib import contextmanager


@contextmanager
def hack_context(enabled: bool = True):
    if not enabled:
        yield
        return
    # hijack some target functions
    raw_gelu = jax.nn.gelu
    jax.nn.gelu = customized_gelu
    yield
    # recover back
    jax.nn.gelu = raw_gelu
```

Replacing all the “jax.nn.gelu” call in the program on-the-fly

4

```python
x = np.random.randn(100000) * 8
## Disable hijack: costs 61MB in ABY3
with hack_context(enabled=False):
    spu_gelu = ppsim.sim_jax(sim, program_that_use_gelu)
    z1 = spu_gelu(x)

## Enable hijack: costs 35MB in ABY3
with hack_context(enabled=True):
    spu_gelu = ppsim.sim_jax(sim, program_that_use_gelu)
    z1 = spu_gelu(x)
```

Python Context Hijack

- Replacing the target function on-the-fly
- Good: Barely need to change the given program (e.g,. taking from HuggingFace)
  - NOTE: some (pointer) function is not hijack-able, e.g., activation functions. We still need to change the Python source code.
- Bad:
  - Lack of fine-grained control, replacing none-or-all.
  - No low-level control is possible
  - Still lack of time-profiling for the hijack function (since we are still working above the Op level)

Intrinsic Calls

- A piece of C++ codes that can be called directly from Python
  - Checkout [spu/spu/intrinsic at main · secretflow/spu (github.com)]
  - Entry: [spu/libspu/device/pphlo/pphlo_intrinsic_executor.cc at main · secretflow/spu (github.com)]
- Workflow:
  - Run `python spu/intrinsic/add_new_intrinsic.py <function_name>`
    - Will generate some python wrapper in `spu/intrinsic/`
    - NOTE: Need manually add
  - Write your C++ codes and add a dispatch in `pphlo_intrinsic_executor.cc`
    - May need some config to Bazel build file to compile the C++ program

Intrinsic Calls

- A piece of C++ codes tha
  - Checkout [spu/spu/intrinsic at main · secretfl](#)
  - Entry: [spu/libspu/device/pphlo/pphlo_intrins](#)
- Workflow:
  - Run `python spu/intrinsic/
    - Will generate some python
    - NOTE: Need manually add
  - Write your C++ codes anc
    - May need some config to

```cpp
std::vector<Value> intrinsic_dispatcher(SPUContext* ctx, llvm::StringRef name,
                                        absl::Span<const Value> inputs) {
  // FIXME: This should be something register by protocol
  if (name == "example_binary") {
    SPDLOG_INFO("Binary example, input0 = {}, input1 = {}", inputs[0],
                inputs[1]);

    Shape result_shape = {inputs[0].shape()[0] + inputs[1].shape()[0],
                          inputs[0].shape()[1] + inputs[1].shape()[1]};

    auto zeros = kernel::hlo::Constant(ctx, 0, result_shape);

    if (inputs[0].isSecret() || inputs[1].isSecret()) {
      zeros = kernel::hlo::Cast(ctx, zeros, VIS_SECRET, inputs[0].dtype());
    } else {
      zeros = kernel::hlo::Cast(ctx, zeros, VIS_PUBLIC, inputs[0].dtype());
    }

    return {zeros};
  }

  if (name == "customized_gelu") {
    SPU_ENFORCE(inputs.size() == 1 && inputs[0].isFxp());
    return {customized_gelu(ctx, inputs[0])};
  }
  // DO-NOT-EDIT: Add_DISPATCH_CODE

  // Default: Identity function
  if (name == "example") {
    SPDLOG_INFO("Calling example intrinsic");
    return {inputs.begin(), inputs.end()};
  }
  SPU_THROW("Unhandled intrinsic call {}", name.str());
}
```

Intrinsic Calls

```python
## A customized and faster gelu protocol from BOLT
def customized_gelu(x):
    # 0.5*x + \sum_{i=0}^{4} a_i*|x|^i
    is_neg = x < 0.0
    abs_x = jax.lax.select(is_neg, -x, x)
    is_inside_approx = abs_x < 3.0

    coeffs = jnp.array(
        [
            0.0002743776353465,
            -0.03798164612714154,
            0.5410550166368381,
            -0.18352506127082727,
            0.020848611754127593,
        ]
    )

    x2 = jax.numpy.square(abs_x)
    x4 = jax.numpy.square(x2)
    x3 = abs_x * x2
    middle_seg = (
        0.5 * x
        + coeffs[0]
        + coeffs[1] * abs_x
        + coeffs[2] * x2
        + coeffs[3] * x3
        + coeffs[4] * x4
    )

    # x > 0 and |x| >= 3 => x > 3
    cond = jax.numpy.logical_and(~is_neg, ~is_inside_approx)
    ret = cond * x
    # |x| < 3
    ret = is_inside_approx * middle_seg + ret
    # If not hit (i.e., x < -3), then return 0
    return ret
```

From Python to Low-level C++ code

5 truncations

Only 1 truncation

```cpp
Value customized_gelu(SPUContext* ctx, const Value& x) {
  using namespace spu::kernel;
  auto is_neg =
      hal::f_less(ctx, x, hal::constant(ctx, 0.0F, x.dtype(), x.shape()));
  auto abs_x =
      hal::_mux(ctx, is_neg, hal::_negate(ctx, x), x).setDtype(x.dtype());
  auto is_inside_range =
      hal::f_less(ctx, abs_x, hal::constant(ctx, 3.0F, x.dtype(), x.shape()));

  // seg = a*|x|^4 + b*|x|^3 + c*|x|^2 + d*|x| + e + 0.5x
  std::vector<float> coeffs = {0.0002743776353465, -0.03798164612714154,
                               0.5410550166368381, -0.18352506127082727,
                               0.020848611754127593};

  float scale = 1L << ctx->getFxpBits();
  auto x2 = hal::f_mul(ctx, abs_x, abs_x);
  auto x4 = hal::f_mul(ctx, x2, x2);
  auto x3 = hal::f_mul(ctx, abs_x, x2);

  auto seg_4 =
      hal::_mul(ctx, x4, hal::constant(ctx, coeffs[4], x.dtype(), x.shape()));
  auto seg_3 =
      hal::_mul(ctx, x3, hal::constant(ctx, coeffs[3], x.dtype(), x.shape()));
  auto seg_2 =
      hal::_mul(ctx, x2, hal::constant(ctx, coeffs[2], x.dtype(), x.shape()));
  auto seg_1 = hal::_mul(ctx, abs_x,
                         hal::constant(ctx, coeffs[1], x.dtype(), x.shape()));
  auto seg_0 = hal::constant(ctx, coeffs[0] * scale, x.dtype(), x.shape());
  auto seg = hal::_trunc(
      ctx,
      hal::_add(
          ctx, hal::_mul(ctx, x, hal::constant(ctx, 0.5, x.dtype(), x.shape())),
          hal::_add(
              ctx, seg_0,
              hal::_add(ctx, seg_1,
                        hal::_add(ctx, seg_2, hal::_add(ctx, seg_3, seg_4))))));
  // x > 3
  auto cond = hal::_and(ctx, hal::logical_not(ctx, is_neg),
                        hal::logical_not(ctx, is_inside_range));
  auto gelu = hal::_mul(ctx, cond, x);
  // // -3 <= x  <= 3
  gelu = hal::_add(ctx, gelu, hal::_mul(ctx, is_inside_range, seg));
  gelu.setDtype(x.dtype());

  return gelu;
}
```

Intrinsic Calls from Hijack

- Intrinsic (i.e., Python wrapper) is still a Python call
- Thus can also be used in context hijack

```python
from contextlib import contextmanager

import spu.intrinsic as intrinsic


@contextmanager
def hack_context(enabled: bool = True, use_intrinsic: bool = False):
    if not enabled:
        yield
        return
    # hijack some target functions
    raw_gelu = jax.nn.gelu
    if use_intrinsic:
        jax.nn.gelu = intrinsic.customized_gelu
    else:
        jax.nn.gelu = customized_gelu
    yield
    # recover back
    jax.nn.gelu = raw_gelu
```

```python
x = np.random.randn(100000) * 8
## Python hijack: costs 35MB in ABY3
with hack_context(enabled=True, use_intrinsic=False):
    spu_gelu = ppsim.sim_jax(sim, program_that_use_gelu)
    z1 = spu_gelu(x)

## C++ hijack: costs 27MB in ABY3
with hack_context(enabled=True, use_intrinsic=True):
    spu_gelu = ppsim.sim_jax(sim, program_that_use_gelu)
    z1 = spu_gelu(x)
```

Improvements due to fine-grained truncation control

Intrinsic Calls from Hijack

- Intrinsic (i.e., Python wrapper) is still a Python call
- Thus can also be used in context hijack
- Turn on ‘config.enable_pphlo_profile = True’ can see the profiling for intrinsic

```
[api.cc:163] [Profiling] SPU execution program_that_use_gelu completed, input processing took 1.08
[api.cc:191] HLO profiling: total time 0.15015229200000002
[api.cc:196] - pphlo.add, executed 1 times, duration 0.000888167s, send bytes 0
[api.cc:196] - pphlo.custom_call, executed 1 times, duration 0.14534825s, send bytes 28100000
[api.cc:196] - pphlo.free, executed 2 times, duration 7.8333e-05s, send bytes 0
[api.cc:196] - pphlo.multiply, executed 1 times, duration 0.003837542s, send bytes 4000000
[api.cc:191] HAL profiling: total time 0.14943770700000003
```

Advanced Topic 1: Full Control

Advanced Topic 1: Full Control

- Intrinsic already provides the access to \`SPUContext\` from which we can fully control the MPC back-end
  - E.g., \`ctx->prot()\` to obtain the handler of the underlying MPC back-end
- For example, to compute the square, cubic and quad term from [x]
  - That is [x] => [x^2], [x^3] and [x^4].

```
static std::array<Value, 3> ComputeUptoPower4(SPUContext* ctx, const Value& x) {
  if (not x.isSecret() || ctx->config().protocol() != ProtocolKind::BUMBLEBEE) {
    auto x2 = f_square(ctx, x);
    auto x4 = f_square(ctx, x2);
    auto x3 = f_mul(ctx, x, x2);
    return {x2, x3, x4};
  }

  KernelEvalContext kctx(ctx);
  auto [_x2, _x3, _x4] =
      spu::mpc::bumblebee::ComputeUptoPower4(&kctx, x.data());
  return {Value(_x2, x.dtype()), Value(_x3, x.dtype()), Value(_x4, x.dtype())};
}
```

The cases that we are not optimizing, simply turn to default APIs

The cases that we interest, we can call the optimized function

Advanced Topic 2: C++ Programming

Advanced Topic 2: C++ Only Programming

- The C++ module under `libspu/kernel/hal/` can be used directly

```
spu::Value logistic_hessian(spu::SPUContext* ctx, const spu::Value& prob) {
  using namespace spu::kernel;
  SPU_ENFORCE(prob.isFxp());
  // hessian = (1 - prob) * prob
  // We explicitly compute prob - prob^2 because square is faster than mul.
  spu::Value prob_square = hal::f_mul(ctx, prob, prob);
  spu::Value hessian = hal::f_sub(ctx, prob, prob_square);
  return hessian;
}
```

Advanced Topic 2: C++ Only Programming

- For complicated function (e.g., multi-head attention), we can leverage the Python to dump the Intermediate Representation (IR) and call the IR in C++ directly.
- Use regex to handle the shape information

```cpp
// tensor.shape NxHxWxC
// valid padding
spu::Value MaxPool2D(spu::SPUContext* ctx, const spu::Value& tensor,
                     int64_t window_size, int64_t stride) {
  SPU_ENFORCE_EQ(tensor.shape().ndim(), 4L, "needs 4D tensor NxHxWxC");

  // Generate via jax.nn.maxpool(x)
  std::string jit = R"(
    func.func @main(%arg0: tensor<INPUT_SHAPE!pphlo.secret<f32>>) -> tensor<OUTPUT_SHAPE!pphlo.secret<f32>> {
    %0 = pphlo.constant dense<0xFF800000> : tensor<f32>
    %1 = pphlo.convert %0 : (tensor<f32>) -> tensor<!pphlo.secret<f32>>
    pphlo.free %0 : tensor<f32>
    %2 = "pphlo.reduce_window"(%arg0, %1) ({
    ^bb0(%arg1: tensor<!pphlo.secret<f32>>, %arg2: tensor<!pphlo.secret<f32>>):
      %3 = pphlo.greater %arg1, %arg2
      : (tensor<!pphlo.secret<f32>>, tensor<!pphlo.secret<f32>>) -> tensor<!pphlo.secret<i1>>
      %4 = pphlo.select %3, %arg1, %arg2
      : (tensor<!pphlo.secret<i1>>, tensor<!pphlo.secret<f32>>, tensor<!pphlo.secret<f32>>) -> tensor<!pphlo.secret<f32>>
      pphlo.free %3 : tensor<!pphlo.secret<i1>>
      pphlo.return %4 : tensor<!pphlo.secret<f32>>
    }) {window_dilations = array<i64: 1, 1, 1, 1>,
        window_dimensions = array<i64: 1, WIN_SZE, WIN_SZE, 1>,
        window_strides = array<i64: 1, STRIDE, STRIDE, 1>}
        : (tensor<INPUT_SHAPE!pphlo.secret<f32>>, tensor<!pphlo.secret<f32>>) -> tensor<OUTPUT_SHAPE!pphlo.secret<f32>>
    pphlo.free %1 : tensor<!pphlo.secret<f32>>
    return %2 : tensor<OUTPUT_SHAPE!pphlo.secret<f32>>
  }
  )";

  spu::Shape output_shape = tensor.shape();
  for (size_t d : {1, 2}) {
    output_shape[d] = (tensor.shape()[d] - window_size + stride) / stride;
  }

  jit = std::regex_replace(jit, std::regex("INPUT_SHAPE"),
                           ToPPHLOShape(tensor.shape()));
  jit = std::regex_replace(jit, std::regex("OUTPUT_SHAPE"),
                           ToPPHLOShape(output_shape));
  jit = std::regex_replace(jit, std::regex("WIN_SZE"),
                           std::to_string(window_size));
  jit = std::regex_replace(jit, std::regex("STRIDE"), std::to_string(stride));

  spu::device::SymbolTable sym_table;
  sym_table.setVar("tensor", tensor);

  spu::ExecutableProto executable;
  executable.set_name("maxpool_2d");
  executable.set_code(jit);
  std::vector<std::string> input_names = {"tensor"};
  std::vector<std::string> output_names = {"output"};
  *executable.mutable_input_names() = {input_names.begin(), input_names.end()};
  *executable.mutable_output_names() = {output_names.begin(),
                                        output_names.end()};

  spu::device::pphlo::PPHloExecutor executor;
  spu::device::execute(&executor, ctx, executable, &sym_table);
  return sym_table.getVar("output");
}
```

Advanced Topic 2: Hand Optimizing IR

```python
if __name__ == "__main__":
    """
    You can modify the code below for debug purpose only.
    Please DONT commit it unless it will cause build break.
    """
    config = spu_pb2.RuntimeConfig(
        protocol=spu_pb2.ProtocolKind.ABY3, field=spu_pb2.FieldType.FM64
    )
    config.enable_hal_profile = True
    config.enable_pphlo_profile = True
    sim = ppsim.Simulator(3, config)

    x = np.random.randn(128, 3)
    spu_fn = ppsim.sim_jax(sim, lambda x: jnp.argmax(x, axis=1))
    spu_fn(x)
    print(spu_fn.pphlo)
```

Dump the ArgMax IR from jax.numpy

```
reducer() {
  %8 = pphlo.greater %arg1, %arg3
  # NaN check which is not useful for MPC
  %9 = pphlo.equal %arg1, %arg1
  %10 = pphlo.not %9
  %11 = pphlo.or %8, %10
  %12 = pphlo.select %11, %arg1, %arg3
  # Stable Argmax: return the smaller index
  # when two values are equal
  %13 = pphlo.equal %arg1, %arg3
  %14 = pphlo.less %arg2, %arg4
  %15 = pphlo.and %13, %14
  %16 = pphlo.or %11, %15
  %17 = pphlo.select %16, %arg2, %arg4
  pphlo.return %12, %17
}
```

Advanced Topic 2: Hand Optimizing IR

```python
if __name__ == "__main__":
    """
    You can modify the code below for debug purpose only.
    Please DONT commit it unless it will cause build break.
    """
    config = spu_pb2.RuntimeConfig(
        protocol=spu_pb2.ProtocolKind.ABY3, field=spu_pb2.FieldType.FM64
    )
    config.enable_hal_profile = True
    config.enable_pphlo_profile = True
    sim = ppsim.Simulator(3, config)

    x = np.random.randn(128, 3)
    spu_fn = ppsim.sim_jax(sim, lambda x: jnp.argmax(x, axis=1))
    spu_fn(x)
    print(spu_fn.pphlo)
```

```
reducer()  {
  %8 = pphlo.greater %arg1, %arg3
  # NaN check which is not useful for MPC
  %9 = pphlo.equal %arg1, %arg1
  %10 = pphlo.not %9
  %11 = pphlo.or %8, %10
  %12 = pphlo.select %11, %arg1, %arg3
  # Stable Argmax: return the smaller index
  # when two values are equal
  %13 = pphlo.equal %arg1, %arg3
  %14 pphlo.less %arg2, %arg4
  % phlo.and %13, %14
  or %11, %15
  select %16, %arg2, %arg4
  %12, %17
```

Dump the ArgMax IR from jax.numpy

Some “plaintext” ops are barely meaningful for MPC

Advanced Topic 2: Hand Optimizing IR

```
reducer()  {
# NOTE(lwj): We skip the NaN check.
# When lhs = rhs, return rhs ignoring the index
# lhs > rhs
%8 = "pphlo.greater" %arg1, %arg3
# max(lhs, rhs)
%9 = "pphlo.select"%8, %arg1, %arg3
# select the max index
%10 = "pphlo.select" %8, %arg2, %arg4
"pphlo.return" %9, %10
}) {dimensions = dense<AXIS> : tensor<1xi64>}
```

simplify

```
reducer()  {
  %8 = pphlo.greater %arg1, %arg3
  # NaN check which is not useful for MPC
  %9 = pphlo.equal %arg1, %arg1
  %10 = pphlo.not %9
  %11 = pphlo.or %8, %10
  %12 = pphlo.select %11, %arg1, %arg3
  # Stable Argmax: return the smaller index
  # when two values are equal
  %13 = pphlo.equal %arg1, %arg3
  %14  pphlo.less %arg2, %arg4
  %    phlo.and %13, %14
       or %11, %15
       select %16, %arg2, %arg4
       n %12, %17
```

Some “plaintext” ops are barely meanngful for MPC