Reading-Paper-And-Its-Code

OPU: An FPGA-based Overlay Processor for Convolutional Neural Networks

in folder src: its main and tests code

Part1 tests.cc Time:2022-10-13

tests.cc文件通过ipa初始化后测试两个功能：1、inner_product 2、accumulate
均通过调用ipa中成员函数和tests.cc内置golden case进行对比来测试
下例中是inner_product功能的测试

- 函数定义

```
//in file 'tests.cc' inner_product function definition
std::vector<acc_type> inner_product(std::vector<acc_type> ifm, std::vector<acc_type> wgt_a, std::vector<acc_
{
    int wgt_a_size = wgt_a.size();
    int wgt_b_size = wgt_b.size();
    int ifm_size = ifm.size();
    assert(wgt_a_size == wgt_b_size);
    assert(ifm_size == wgt_b_size);

    std::vector<acc_type> res;
    for (int i = 0; i < ifm_size; ++i)
    {
        res.push_back(ifm[i] * wgt_a[i]);
        if (PRECISION == 8)
            res.push_back(ifm[i] * wgt_b[i]);
    }
    return res;
}

//in file 'accelerator.h' inner_product definition
 // Fetch data from compute buffer and get ipa results
void Forward(int output_num) {
    std::vector<acc_type> ifm = fm_buf_.AsVec(0, nMac, PRECISION);
    std::vector<acc_type> wgt_a = wgt_buf_a_.AsVec(0, nMac, PRECISION);
    std::vector<acc_type> wgt_b = wgt_buf_b_.AsVec(0, nMac, PRECISION);

    std::vector<acc_type> psum_local;
    int output_num_iter = PRECISION == 8 ? (output_num / 2): output_num;
    size_t step = nMac / output_num_iter;
    for (int i = 0; i < output_num_iter; i++) {
      acc_type sum = std::inner_product(ifm.begin() + i * step,
              ifm.begin() + (i + 1) * step, wgt_a.begin() + i * step, 0);
      psum_local.push_back(sum);
      if(PRECISION == 8){
          //返回作为两个序列乘积而生成的元素的总和。步调一致地检查两个序列,将来自两个序列的元素相乘,将相乘的结果求
      //inner_product(beg1, end1, beg2, init)
          sum = std::inner_product(ifm.begin() + i * step,
              ifm.begin() + (i + 1) * step, wgt_b.begin() + i * step, 0);
        psum_local.push_back(sum);
      }
    }

    // Concatenate at front 在前面串接
    psum.insert(psum.begin(), psum_local.begin(), psum_local.end());
}
```

- 函数调用

```
/*
     * Part 1 : inner-product
     */
    // Run IPA to compute inner-product between its inputs: fm_buf, wgt_buf_a, wgt_buf_b
    ipa_.Forward(pe_num * 2);

    ASSERT_EQ(pe_num * 2, ipa_.GetOutputNum());

    // Get inner_product golden case for comparison
    std::vector<acc_type> ifm = ipa_.fm_buf_.AsVec(0, pe_num, PRECISION);
    std::vector<acc_type> wt_a = ipa_.wgt_buf_a_.AsVec(0, pe_num, PRECISION);
    std::vector<acc_type> wt_b = ipa_.wgt_buf_b_.AsVec(0, pe_num, PRECISION);
    std::vector<acc_type> ip_res = inner_product(ifm, wt_a, wt_b);
    std::vector<acc_type> psum_ = ipa_.psum;

    // Compare algorithmic output and IPA (i.e. hardware) output
    for (int i = 0; i < ip_res.size(); ++i)
        ASSERT_EQ(ipa_.psum[i], ip_res[i]);
```

其中各项数值在tests.cc文件开头已被默认

```
#define PRECISION 8
using acc_type = int;
using input_type = int8_t;
using output_type = int16_t;
```

Part2 accelerator.h Time：2022-10-13

- 代码中的参数化部分

```
// ========== Parameterization Start =========

#define PRECISION 8 // 8/16

#if PRECISION == 8
#define ACC_SIZE 26 // 2*PRECISION+10 = 26/42
#define ACC_STORAGE 32 // full-precision storage = 32/64
#define NEG_MASK 0x80 // 0x80/0x800 - 8000?
#define INPUT_MASK 0xFF // 0xFF/0xFFFF
using input_type = int8_t; // int8_t/int16_t
using output_type = int16_t; // int16_t/int32_t
using acc_type = int; // int/long
// TODO: check if reversal needed
#define REVERSE_BYTES(value) REVERSE_BYTES16(value)
#endif

#if PRECISION == 16
#define PRECISION 16 // 8/16
#define ACC_SIZE 42 // 2*PRECISION+10 = 26/42
#define ACC_STORAGE 64 // full-precision storage = 32/64
#define NEG_MASK 0x8000 // 0x80/0x800 - 8000?
#define INPUT_MASK 0xFFFF // 0xFF/0xFFFF
using input_type = int16_t; // int8_t/int16_t
using output_type = int32_t; // int16_t/int32_t
using acc_type = long; // int/long
// TODO: check if reversal needed
#define REVERSE_BYTES(value) REVERSE_BYTES32(value)
#endif

// ========== Parameterization End =========
```

- 参数列表

参数名称	参数含义
PRECISION	精度，Q:为什么=16时不计算wgt_b A:=16指的是OPU系统， 当系统精度只有8时需要用两个8位数组来存储16位数值

- 头文件中的类（Memop、SRAM、IPA组件）

  Define template components for hardware modules, including SRAM, IPA

  - Memory class

```
/*
 * MemOp class encapsulates source/destination address and size for memory operation
 */
class MemOp {
 public:
  uint32_t src_addr;
  uint32_t size;
  uint32_t dst_addr;
  MemOp(uint32_t src_addr, uint32_t size, uint32_t dst_addr) {
    this->src_addr = src_addr;
    this->size = size;
    this->dst_addr = dst_addr;
  }
};
```

- SRAM class

  SRAM 类的结构编写参考的是[https://github.com/apache/tvm-vta/blob/main/src/sim/sim_driver.cc]

  其中的SRAM模板

```
template<int bits, int size, int banks>
class SRAM {
 public:
  static const int dBytes = bits * banks / 8;
  using DType = typename std::aligned_storage<dBytes, dBytes>::type; //别名指定

  SRAM() {
    data_ = new DType[size];
  }

  ~SRAM() {
    delete [] data_;
  }

  //基本上用于xx->BeginPtr(0)
  // Get the i-th index
  void* BeginPtr(uint32_t index) {
    // CHECK_LT(index, kMaxNumElem);
    return &(data_[index]);
  }

  // Copy data from disk file to SRAM
  //src folder中暂无调用
  void InitFromFile(std::string filename) {}

  // Copy data from disk file to SRAM with address offset and size specified
  //   - src_wl : word length corresponding to mem.src and mem.size
  //src folder中暂无调用
  template<uint32_t src_wl>
  void LoadFromFile(MemOp mem, const char* filename) {}

  // Copy data from in-memory pointer to SRAM
  //将
  template<int src_bits>
  void Load(MemOp mem, void* src) {
    int8_t* src_t = reinterpret_cast<int8_t*>(src);
    size_t bytes = mem.size * src_bits / 8;
    //将src_t复制bytes大小到data_ + mem.dst_addr中
    std::memcpy(data_ + mem.dst_addr, src_t, bytes);
  }

  // Get data as std::vector<acc_type> to compute
  //将SRAM中以_data+addr地址为首的剩下的的data转化为vector<acc_type>输出
  std::vector<acc_type> AsVec(int addr, int vec_size, int src_wl) {}

 private:
  DType* data_;
};
```

- SRAM的成员函数的应用
  - void* BeginPtr(uint32_t index)

```
    //get wgt addr
    //一般参数为0或所需参数的首地址常量
    wgt_addr = compute->ker_addr_s + p;
    input_type* wgt = reinterpret_cast<input_type*>(wgt_ram->BeginPtr(wgt_addr));
    for (int ii = 0; ii < pe_bytes; ii++) {
        // For 16-bit, use only a (don't split kernel)
        wgt_src_a[ii] = wgt[2 * ii];
        // For 8-bit, split kernel into a and b
        if(PRECISION == 8){
          wgt_src_b[ii] = wgt[2 * ii + 1];
        }
    }

    // Load bias
    ipa_.adder_buf_b_.Load<1024*(PRECISION/8)>(MemOp(0, 1, 0), bias_ram->BeginPtr(0));
    std::vector<acc_type> bias = ipa_.adder_buf_b_.AsVec(0, 2*pe, PRECISION*2);
```

- void Load(MemOp mem, void* src)

```
//将wgt_a的值传入ipa_下名为wgt_buf_a的SRAM中
ipa_.wgt_buf_a_.Load<pe_num * PRECISION>(MemOp(0, 1, 0), wgt_a);
```

```
- std::vector<acc_type> AsVec(int addr, int vec_size, int src_wl)
```

```
std::vector<acc_type> ifm = fm_buf_.AsVec(0, nMac, PRECISION);
```

重点

- IPA class

```
  /*
 * IPA class is parameterized with
 *  - mac : number of multiply-accumulate units (DSP macros on FPGA) per PE
 *  - pe : number of PE (processing unit), which is a bunch of macs followed
 *         by adder tree for reduction
 *  - operand_bits : word lengths for mac operands (8 by default)
 */
template<int mac, int pe, int operand_bits>
class IPA {
 public:
  static const int nMac = mac * pe;
  // Compute buffer
  using PE_buf_t = SRAM<mac*operand_bits, 1, pe>;
  PE_buf_t fm_buf_;
  PE_buf_t wgt_buf_a_;
  PE_buf_t wgt_buf_b_;
  using Acc_buf_t = SRAM<ACC_STORAGE*(2*pe/(PRECISION/8)), 1, 1>; // 2pe 32-bit adders for 8-bit, pe 64-bit
  Acc_buf_t adder_buf_a_;
  Acc_buf_t adder_buf_b_;

  std::vector<acc_type> psum;
  std::vector<acc_type> adder_b;

  // Get current valid ipa output number
  int GetOutputNum() {
    return psum.size();
  }

//非常重要的函数：卷积函数—从计算缓冲区取数据并进行ipa
// Fetch data from compute buffer and get ipa results
  void Forward(int output_num) {
    std::vector<acc_type> ifm = fm_buf_.AsVec(0, nMac, PRECISION);
    std::vector<acc_type> wgt_a = wgt_buf_a_.AsVec(0, nMac, PRECISION);
    std::vector<acc_type> wgt_b = wgt_buf_b_.AsVec(0, nMac, PRECISION);

    std::vector<acc_type> psum_local;
    int output_num_iter = PRECISION == 8 ? (output_num / 2): output_num;
    size_t step = nMac / output_num_iter;
    for (int i = 0; i < output_num_iter; i++) {
      acc_type sum = std::inner_product(ifm.begin() + i * step,
              ifm.begin() + (i + 1) * step, wgt_a.begin() + i * step, 0);
      psum_local.push_back(sum);
      if(PRECISION == 8){
               //返回作为两个序列乘积而生成的元素的总和。步调一致地检查两个序列,将来自两个序列的元素相乘,将相乘的
               //inner_product(beg1, end1, beg2, init)
               sum = std::inner_product(ifm.begin() + i * step,
              ifm.begin() + (i + 1) * step, wgt_b.begin() + i * step, 0);
        psum_local.push_back(sum);
      }
    }

    // Concatenate at front 在前面串接
    psum.insert(psum.begin(), psum_local.begin(), psum_local.end());
  }

// Output control for accumulation
  void Accumulate(int fm_lshift_num, int pe_num, bool cut, void* dst,
    std::ofstream& os, bool dump) {
    std::vector<acc_type> adder_a(2*pe_num/(PRECISION/8) - psum.size(), 0);
    for (int i = 0; i < psum.size(); i++) {
      acc_type value = psum[i];
```

```
      adder_a.push_back(
        Saturate(value << fm_lshift_num, value > 0, ACC_SIZE));
    }
    // std::vector<int> gdb = wgt_ram.AsVec(wgt_addr, 1024, 8);
    #ifdef DEBUG_OUT_ADDER_B
        writeOut<acc_type, PRECISION>(os, adder_b, dump);
    #endif
    #ifdef DEBUG_OUT_ADDER_A
        writeOut<acc_type, PRECISION>(os, adder_a, dump);
    #endif
    psum.clear();
    std::vector<acc_type> res;
    auto ia = adder_a.begin();
    auto ib = adder_b.begin();
    for (int ii = 0; ii < 2*pe_num/(PRECISION/8); ii++) {
      acc_type p = *(ia++) + *(ib++);
      res.push_back(p);
    }
    // ACC_TYPE--cut->2*PRECISION (eg: for 8 bit we cut from 26b -> 16, for 16 bit we cut from 42b -> 32)
    for (auto &item : res) {
           item = Saturate(item >> 10, item > 0, 2*PRECISION);
    }
    #ifdef DEBUG_CUT_16
        writeOut<acc_type, PRECISION/2>(os, res, dump);
    #endif
    if (cut) {
      // 2*PRECISION--round->PRECISION (eg: for 8 bit cut from 16->8, for 16 bit cut from 32->16)
      for (auto &item : res) {
        bool positive = item > 0;
        if (item & NEG_MASK)
          item = (item >> PRECISION) + 1;
        else
          item = item >> PRECISION;
        item = Saturate(item, positive, PRECISION);
      }
    }
    #ifdef DEBUG_CUT_8
        writeOut<acc_type, PRECISION/4>(os, res, dump);
    #endif
    // write to dst* of temp buffer
    std::vector<output_type> trunc;

    for (auto item : res) {
      // -> little endian
      output_type value = static_cast<output_type>(item);
      value = REVERSE_BYTES(value);
      trunc.push_back(value);
    }
    std::memcpy(dst, &trunc[0], 2*pe_num*(PRECISION*2)/8);  // byte count
  }
};
```

```
//OPU系统的最终抽象体Device，其涵盖了之前所有的类；
//而且使用了其他cc和h的函数与类，集大成体
//【先阅读一遍】->【去阅读其他h和cc代码】->【回头再重新捋一遍】
```

- Device class

```
 /*
 * Device class is the top abstraction for OPU overlay
 */
class Device {
 public:
  uint32_t ins_pc{0};
  // Special purpose registers
  OPURegFile reg_;
  // Scratchpad memory
  using Ins_ram_t = SRAM<32, 2<<15, 1>;
  using Fm_ram_t = SRAM<512, 4096, 1>; // TODO: increase depth?
  using Wgt_ram_t = SRAM<512, 64, 16>; // TODO: increase depth?
  using Bias_ram_t = SRAM<1024, 1, 1>;
  Ins_ram_t ins_ram_;
  Fm_ram_t fm_ram_a_;
  Fm_ram_t fm_ram_b_;
  Wgt_ram_t  wgt_ram_a_;
  Wgt_ram_t  wgt_ram_b_;
  Bias_ram_t bias_ram_a_;
  Bias_ram_t bias_ram_b_;
  // IPA
  using IPA_t = IPA<16, 32, PRECISION>;
  IPA_t ipa_;
  // Partial sum buffer
  using Tmp_buf_t = SRAM<64*2*PRECISION, 2<<11, 1>;
  Tmp_buf_t tmp_buf_;
  // DDR
  using DRAM = VirtualMemory;
  DRAM dram;

  // dw
  using Wgt_ram_dw_t = SRAM<1024, 64, 16>; // TODO: make some change later? Unsure
  Wgt_ram_dw_t wgt_ram_dw_;
  using IPA_DW_t = IPA<16, 64, PRECISION>;
  IPA_DW_t ipa_dw_;

  // Double buffering
  std::vector<Fm_ram_t*> fm_ram_vec_;
  std::vector<Wgt_ram_t*> wgt_ram_vec_;
  std::vector<Bias_ram_t*> bias_ram_vec_;
  std::vector<Ins_ram_t*> ins_ram_vec_;
  size_t fm_ram_id{0};
  size_t wgt_ram_id{0};
  size_t bias_ram_id{0};
  size_t ins_ram_id{0};

  // Control flow
  std::queue<std::vector<OPUGenericInsn*>> event_q;
  OPUDDRLDInsn* load_ins_;
  OPUDDRSTInsn* store_ins_;
  OPUComputeInsn* compute_ins_;
  std::vector<OPUGenericInsn*> ins_vec_;
  void DependencyUpdate(OPUShortInsn* ins);
  void DependencyUpdateUtil();
  std::vector<OPUGenericInsn*>
    DependencyCheck(std::vector<OPUGenericInsn*> ins_vec);
  // Global variables
  bool layer_start_{false};
  bool compute_finish {false};
  bool compute_cnt_enable {false};
  int compute_cnt {0};
```

```
  bool load_single {false};
  int fc_tmp_addr {0};

  // Function wrappers
  void Run(int cnt = INT_MAX);
  void FetchInsn();
  // void RunInsn(OPUGenericInsn* ins);
  void RunLoad(OPUDDRLDInsn* load);
  void RunStore(OPUDDRSTInsn* store);
  void RunPostProcess(OPUDDRSTInsn* store);
  void RunPadding(OPUDDRSTInsn* store);
  void RunCompute(OPUComputeInsn* compute);
  void RunComputeDW(OPUComputeInsn* compute);
  void RunComputeFC(OPUComputeInsn* compute);
  // Sub-function wrappers
  void Pooling(std::vector<acc_type>& data_o, std::vector<acc_type> data_i, int type,
    bool st, bool ed, int window_size);
  void Activation(std::vector<acc_type>& data, int type);
  void ResidualAdd(std::vector<acc_type>& data, bool enable, int addr);

  // Utility function
  std::string GetInsName(OPUGenericInsn* ins);
  bool IsComplete();

  // Debug
  std::ofstream os;
  bool dump {false};
  bool skip_exec {false};
  Profiler profiler;

  Device() {
    dram.Init();
    // srams
    //ins_ram_.InitFromFile("tinyyolo/ins.bin");
    //ins_ram_.InitFromFile("yolov3/ins.bin");
    ins_ram_.InitFromFile(INS_FILE_PATH);
    ins_ram_vec_ = {&ins_ram_};
    fm_ram_vec_ = {&fm_ram_a_, &fm_ram_b_};
    wgt_ram_vec_ = {&wgt_ram_a_, &wgt_ram_b_};
    bias_ram_vec_ = {&bias_ram_a_, &bias_ram_b_};
    // coarse-grained generic instructions
    load_ins_ = new OPUDDRLDInsn();
    store_ins_ = new OPUDDRSTInsn();
    compute_ins_ = new OPUComputeInsn();
    ins_vec_ = {load_ins_, store_ins_, compute_ins_};

    os.open("out.txt");
  }
};
```

Reading-Paper-And-Its-Code

OPU: An FPGA-based Overlay Processor for Convolutional Neural Networks

in folder src: its main and tests code

Part3 driver.cc Time:2022-10-14

Device作为最高层抽象，模拟了整个OPU overlay，包括其中的component、instruction、flow和layer等；
除了compiler部分，当导入已编译的数据后，通过driver.cc中的main函数初始化即可开始OPU的模拟。

- main function Part

```
//initialize the class Device
Device* dev = new Device();

//初始化flow: IF,COMPUTE,STORE 加载数据
//剩下的均与指令相关
dev->RunLoad(load);
dev->RunCompute(compute);
dev->RunStore(store);
dev->FetchInsn();
dev->Run();
```

无非数据加载，执行指令几项。

重新阅读Device类

- 整理变量以及变量类型

变量名称	变量类型
Special purpose registers	
*reg_*	OPURegFile
Scratchpad memory	
*ins_ram_*	Ins_ram_t -> SRAM<32, 2<<15, 1>
*fm_ram_a_*	Fm_ram_t -> SRAM<512, 4096, 1>
*fm_ram_b_*	Fm_ram_t -> SRAM<512, 4096, 1>
*wgt_ram_a_*	Wgt_ram_t -> SRAM<512, 64, 16>
*wgt_ram_b_*	Wgt_ram_t -> SRAM<512, 64, 16>
*bias_ram_a_*	Bias_ram_t -> SRAM<1024, 1, 1>

变量名称	变量类型
*bias_ram_b_*	Bias_ram_t -> SRAM<1024, 1, 1>
IPA	
*ipa_*	IPA_t -> IPA<16, 32, PRECISION>
Partial sum buffer	
*tmp_buf_*	Tmp_buf_t -> SRAM<642PRECISION, 2<<11, 1>
DDR	
dram	DRAM -> VirtualMemory
dw	
*wgt_ram_dw_*	Wgt_ram_dw_t -> SRAM<1024, 64, 16>
*ipa_dw_*	IPA_DW_t -> IPA<16, 64, PRECISION>
Double buffering	
*fm_ram_vec_*	std::vector<Fm_ram_t*>
*wgt_ram_vec_*	std::vector<Wgt_ram_t*>
*bias_ram_vec_*	std::vector<Bias_ram_t*>
*ins_ram_vec_*	std::vector<Ins_ram_t*>
fm_ram_id	size_t
wgt_ram_id	size_t
bias_ram_id	size_t
ins_ram_id	size_t
Control flow	
Global variables	
Function wrappers	
Sub-function wrappers	
Utility function	
Debug	

instruction.h中的类其成员函数均为赋值，并无操作

for each layer:
dev->FetchInsn() // fetch one instruction block 获取一整个指令块
dev->Run(#phases) // run until no new events pop out by default

▼ 初步学习TVM
  ▼ TVM是如何将对应模型转化为TVM IR的
    - the 1st-level func [from_onnx]
    ▼ the 2nd-level func [from_onnx] in onnx graph g
      - GraphProto对象
      - Turn onnx GraphProto into helper class GP -> GraphProtoMember Parse
    ▼ the 3nd-level func [_convert_operator] in helper class GP
      - 举例Conv的convert_map
    - Conclusion: convert process func-level

初步学习TVM

TVM是如何将对应模型转化为TVM IR的

the 1st-level func [from_onnx]

基于onnx模型，TVM采用

```
mod, params = relay.frontend.from_onnx(onnx_model, shape_dict)
```

函数来加载IR
mod 输出

```
def @main(%input.1: Tensor[(1, 3, 224, 224), float32], %v193: Tensor[(64, 3, 7, 7), float32], %v194: Tensor[
  %0 = nn.conv2d(%input.1, %v193, strides=[2, 2], padding=[3, 3, 3, 3], kernel_size=[7, 7]);
  %1 = nn.bias_add(%0, %v194);
  %2 = nn.relu(%1);
```

> 可以看到这个mod其实就是一个函数（Relay Function），函数的输入就是ONNX模型中所有输入Tensor的shape信息，不仅包含真实的输入input.1，还包含带权重OP的权重Tensor的shape信息，比如卷积层的weight和bias。对应的mod, params = relay.frontend.from_onnx(onnx_model, shape_dict)这里的params则保存了ONNX模型所有OP的权重信息，以一个字典的形式存放，字典的key就是权重Tensor的名字，而字典的value则是TVM的Ndarry，存储了真实的权重（通过Numpy作为中间数据类型转过来的）。

the 2nd-level func [from_onnx] in onnx graph g

该函数中关键语句为

```
with g:
    mod, params = g.from_onnx(graph, opset)
return mod, params
```

GraphProto对象

其中graph为onnx protobuf 对象，其内部具有如下的一些信息，不限于node, attribute, input等：

```
ir_version: 8
producer_name: "onnx-example"
graph {
  node {
    input: "X"
    input: "pads"
    input: "value"
    output: "Y"
    op_type: "Pad"
    attribute {
      name: "mode"
      s: "constant"
      type: STRING
    }
  }
  name: "test-model"
  input {
    name: "X"
    type {
      tensor_type {
        elem_type: 1
        shape {
          dim {
            dim_value: 3
          }
          dim {
            dim_value: 2
          }
        }
      }
    }
  }
  input {
    name: "pads"
    type {
      tensor_type {
        elem_type: 1
        shape {
          dim {
            dim_value: 1
          }
          dim {
            dim_value: 4
          }
        }
      }
    }
  }
  input {
    name: "value"
    type {
      tensor_type {
        elem_type: 1
        shape {
          dim {
            dim_value: 1
          }
        }
      }
    }
  }
```

```
  output {
    name: "Y"
    type {
      tensor_type {
        elem_type: 1
        shape {
          dim {
            dim_value: 3
          }
          dim {
            dim_value: 4
          }
        }
      }
    }
  }
}
opset_import {
  version: 15
}
```

GraphProto类定义如下：

```
class GraphProto(object):
    """A helper class for handling Relay expression copying from pb2.GraphProto.
    Definition: https://github.com/onnx/onnx/blob/master/onnx/onnx.proto
        Parameters
    ----------
    shape : dict of str to tuple, optional
        The input shape to the graph
    dtype : str or dict of str to str
        The input types to the graph
    """

    def __init__(self, shape, dtype):
        self._nodes = {}
        self._params = {}
        self._renames = {}
        self._num_input = 0
        self._num_param = 0
        self._shape = shape if shape else {}
        self._dtype = dtype

    def from_onnx(self, graph, opset):
    ...
```

Turn onnx GraphProto into helper class GP -> GraphProtoMember Parse

ONNX结构分析

> 这里我把需要重点了解的对象列出来
>
> - ModelProto
> - GraphProto
> - NodeProto
> - AttributeProto
> - ValueInfoProto

> - TensorProto
>
> 我用尽可能简短的语言描述清楚上述几个Proto之间的关系：当我们将ONNX模型load进来之后，得到的是一个ModelProto，它包含了一些版本信息，生产者信息和一个非常重要的GraphProto；在GraphProto中包含了四个关键的repeated数组，分别是node(NodeProto类型)，input(ValueInfoProto类型)，output(ValueInfoProto类型)和initializer(TensorProto类型)，其中node中存放着模型中的所有计算节点，input中存放着模型所有的输入节点，output存放着模型所有的输出节点，initializer存放着模型所有的权重；那么节点与节点之间的拓扑是如何定义的呢？非常简单，每个计算节点都同样会有input和output这样的两个数组(不过都是普通的string类型)，通过input和output的指向关系，我们就能够利用上述信息快速构建出一个深度学习模型的拓扑图。最后每个计算节点当中还包含了一个AttributeProto数组，用于描述该节点的属性，例如Conv层的属性包含group，pads和strides等等，具体每个计算节点的属性、输入和输出可以参考这个Operators.md文档。
>
> 需要注意的是，刚才我们所说的GraphProto中的input输入数组不仅仅包含我们一般理解中的图片输入的那个节点，还包含了模型当中所有权重。举个例子，Conv层中的W权重实体是保存在initializer当中的，那么相应的会有一个同名的输入在input当中，其背后的逻辑应该是把权重也看作是模型的输入，并通过initializer中的权重实体来对这个输入做初始化(也就是把值填充进来)

from_onnx func中按如下顺序处理GraphProtoMember:

GraphProtoMember	Type
initializer	TensorProto
input	ValueInfoProto
node	NodeProto
output	ValueInfoProto

Q1: 处理member的顺序能否进行交换?

将四个不同type的数组按同样结构GPMember保存在helper class GP中，存储的代码如下：

- Part1: initializer

```
## in from_onnx func
        # 解析网络的输入到relay中，又叫参数，onnx的initializer就是用来保存模型参数的，即权重
        for init_tensor in graph.initializer:
            if not init_tensor.name.strip():
                raise ValueError("Tensor's name is required.")
            # 具体实现就是先把这个TensorProto使用get_numpy函数获得值，再reshape到特定形状，再基于这个numpy构造tv
            '''
            def _parse_array(self, tensor_proto):
                np_array = get_numpy(tensor_proto).reshape(tuple(tensor_proto.dims))
                return _nd.array(np_array)
            '''
            array = self._parse_array(init_tensor)
            # 前面解释过，如果设置冻结参数，则将这个参数设置为Relay中的常量OP
            if self._freeze_params:

                self._nodes[init_tensor.name] = _expr.const(array)
            else:
                # g中_nodes为字典，name->[name, shape, dtype],
                self._params[init_tensor.name] = array
                self._nodes[init_tensor.name] = new_var(
                    init_tensor.name,
                    shape=self._params[init_tensor.name].shape,
                    dtype=self._params[init_tensor.name].dtype,
                )
```

- Part2: input
  处理onnx model的input，onnx model的input可以通过

```
        # 解析ONNX模型的输入 graph为onnx model.GraphProto
        for i in graph.input:
            # 获取i这个输入的名字，shape，数据类型以及shape每个维度对应的名字
            i_name, i_shape, d_type, i_shape_name = get_info(i)
```

语句完成；
graph.input里面既有权重的同名输入，也有节点，即需要区分node和param

```
# 判断i这个输入是权重参数还是输入
        if i_name in self._params:
            # i is a param instead of input
            self._num_param += 1
            self._params[i_name] = self._params.pop(i_name)
            self._nodes[i_name] = new_var(
                i_name, shape=self._params[i_name].shape, dtype=self._params[i_name].dtype
            )
        # 输入节点已经在Relay IR中了就不用处理了
        elif i_name in self._nodes:
            continue
        else:
            # 真正的输入节点，依赖用户进行指定
            self._num_input += 1
            self._input_names.append(i_name)
            if i_name in self._shape:
                i_shape = self._shape[i_name]
            else:
                if "?" in str(i_shape):
                    warning_msg = (
                        "Input %s has unknown dimension shapes: %s. "
                        "Specifying static values may improve performance"
                        % (i_name, str(i_shape_name))
                    )
                    warnings.warn(warning_msg)
            if isinstance(self._dtype, dict):
                dtype = self._dtype[i_name] if i_name in self._dtype else d_type
            else:
                dtype = d_type
            self._nodes[i_name] = new_var(i_name, shape=i_shape, dtype=dtype)
        self._inputs[i_name] = self._nodes[i_name]
```

- Part3: node
  通过前两个Part处理完参数和输入的一些非法情况和不支持的操作后，可以保证node中所有算子都是可以被支持的了。

对node进行信息提取并对op进行转换，其中

node	NodeProto Type
input	String Type
output	String Type

node中存储input和output来保存指向关系。

```
# 到这里说明这个ONNX模型的所有算子都被Relay支持，可以正常进行转换了
        for node in graph.node:
            op_name = node.op_type
            # 解析attribute参数
            attr = self._parse_attr(node.attribute)
            # 创建并填充onnx输入对象。
            inputs = onnx_input()
            for i in node.input:
                if i != "":
                    # self._renames.get(i, i)用来获取ONNX Graph每个节点的输入
                    inputs[i] = self._nodes[self._renames.get(i, i)]
                else:
                    inputs[i] = None
            #目前inputs的初始化使用的是下面的语句
            #inputs = [self._nodes[self._renames.get(i, i)] for i in node.input]
            i_name = self._parse_value_proto(node)
            node_output = self._fix_outputs(op_name, node.output)
            attr["tvm_custom"] = {}
            attr["tvm_custom"]["name"] = i_name
            attr["tvm_custom"]["num_outputs"] = len(node_output)
            # 执行转换操作
            op = self._convert_operator(op_name, inputs, attr, opset)

            ...
```

the 3nd-level func [_convert_operator] in helper class GP

_convert_operator function如下:

```
def _convert_operator(self, op_name, inputs, attrs, opset):
        """将ONNX的OP转换成Relay的OP
        转换器必须为不兼容的名称显式指定转换，并将处理程序应用于运算符属性。

        Parameters
        ----------
        op_name : str
            Operator 名字，比如卷积，全连接
        inputs : tvm.relay.function.Function的list
            List of inputs.
        attrs : dict
            OP的属性字典
        opset : int
            Opset version

        Returns
        -------
        sym : tvm.relay.function.Function
            Converted relay function

        return value sym的type和输入参数inputs type一样，其中_get_convert_map返回一系列定义好的映射，如：
        def _get_convert_map(opset):
          return {
            # defs/experimental
            'Identity': Renamer('copy'),
            # 'Affine'
            'ThresholdedRelu': ThresholdedRelu.get_converter(opset),
            'ScaledTanh': ScaledTanh.get_converter(opset),
            'ParametricSoftplus': ParametricSoftPlus.get_converter(opset),
            'ConstantOfShape': ConstantOfShape.get_converter(opset),
            # 'GivenTensorFill'
            'FC': AttrCvt('dense', ignores=['axis', 'axis_w']),
            'Scale': Scale.get_converter(opset),
            "Conv": Conv.get_converter(opset),
        """
        convert_map = _get_convert_map(opset)
        if op_name in _identity_list:
            sym = get_relay_op(op_name)(*inputs, **attrs)
        elif op_name in convert_map:
            sym = convert_map[op_name](inputs, attrs, self._params)
        else:
            raise NotImplementedError("Operator {} not implemented.".format(op_name))
        return sym
```

举例Conv的convert_map

> 我们以卷积层为例来看看ONNX的OP是如何被转换成Relay表达式的。卷积OP一般有输入，权重，偏置这三个项，对应了下面函数中的inputs[0],inputs[1],inputs[2]。而auto_pad这个属性是ONNX特有的属性，TVM的Relay卷积OP不支持这种属性，所以需要将ONNX 卷积OP需要Pad的数值计算出来并分情况进行处理（这里有手动对输入进行Pad以及给Relay的卷积OP增加一个padding参数两种做法，具体问题具体分析）。然后需要注意的是在这个转换函数中inputs[0]是Relay IR，而不是真实的数据，我们可以通过打印下面代码中的inputs[0]看到。

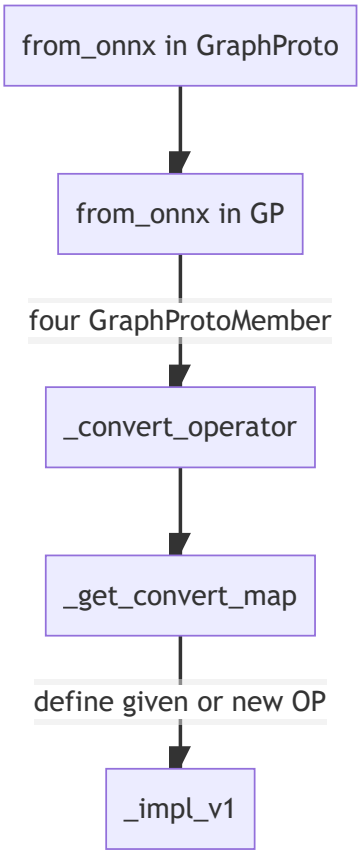
```
class Conv(OnnxOpConverter):
    """Operator converter for Conv."""

    @classmethod
    def _impl_v1(cls, inputs, attr, params):
        # Use shape of input to determine convolution type.
        data = inputs[0]
        input_shape = infer_shape(data)
        ndim = len(input_shape)
        if "auto_pad" in attr:
            attr["auto_pad"] = attr["auto_pad"].decode("utf-8")
            if attr["auto_pad"] in ("SAME_UPPER", "SAME_LOWER"):
                # Warning: Convolution does not yet support dynamic shapes,
                # one will need to run dynamic_to_static on this model after import
                data = autopad(data, attr["strides"], attr["kernel_shape"], attr["dilations"], ndim)
            elif attr["auto_pad"] == "VALID":
                attr["pads"] = tuple([0 for i in range(ndim - 2)])
            elif attr["auto_pad"] == "NOTSET":
                pass
            else:
                msg = 'Value {} in attribute "auto_pad" of operator Conv is invalid.'
                raise tvm.error.OpAttributeInvalid(msg.format(attr["auto_pad"]))
            attr.pop("auto_pad")

        out = AttrCvt(
            op_name=dimension_picker("conv"),
            transforms={
                "kernel_shape": "kernel_size",
                "dilations": ("dilation", 1),
                "pads": ("padding", 0),
                "group": ("groups", 1),
            },
            custom_check=dimension_constraint(),
        )([data, inputs[1]], attr, params)

        use_bias = len(inputs) == 3
        if use_bias:
            out = _op.nn.bias_add(out, inputs[2])
        return out
```

class OnnxOpConverter get_converter func相当于C++的虚函数，真正的实现是每个op class的_impl_v1 func，所以新建op只需要写该op的class再在_get_convert_map func中注册即可。
获取op后，再对op进行一些处理。

- Part4: output

```
# 解析ONNX模型的输出
        outputs = [self._nodes[self._parse_value_proto(i)] for i in graph.output]
        outputs = outputs[0] if len(outputs) == 1 else _expr.Tuple(outputs)
        # 如果需要直接返回转换后的表达式，在这里return
        if get_output_expr:
            return outputs
        # 保持来自ONNX Graph的输入和参数顺序，但仅仅包含这些需要执行转换到Relay的节点
        free_vars = analysis.free_vars(outputs)
        nodes = {v: k for k, v in self._nodes.items()}
        free_vars = [nodes[var] for var in free_vars]
        for i_name in self._params:
            if i_name in free_vars and i_name not in self._inputs:
                self._inputs[i_name] = self._nodes[i_name]
        # 根据我们的输出表达式和所有输入变量创建一个函数。
        func = _function.Function([v for k, v in self._inputs.items()], outputs)
        # 把这个函数用IRModule包起来返回，并同时返回权重参数
        return IRModule.from_expr(func), self._params
```

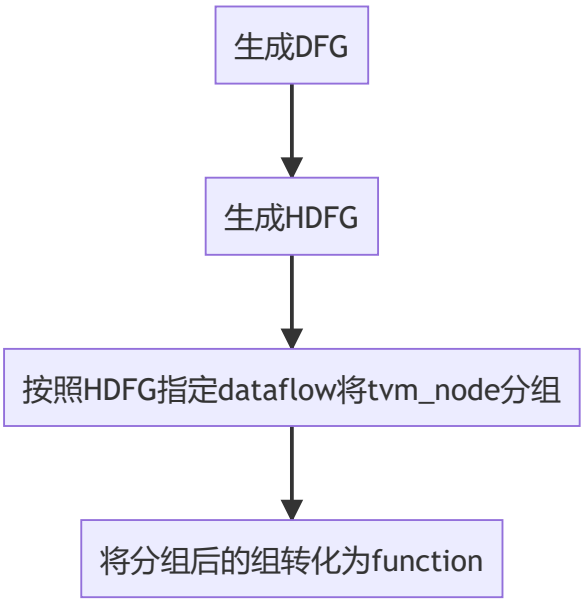
Conclusion: convert process func-level



fuse_op_hw Pass HW算子融合逻辑

1 总貌

tvm前端解析出来网络module并转化为Relay IR后，fuse_op_hw Pass做了以下几件事：



其中

class	class annotation
DFG	data flow graph，用来作为tvm node和HDFG node之间的过渡，是tvm node的映射
Creator	DFG用来完成node映射的helper class
HDFG	Hardware data flow graph，将tvm IR转化为指定OPU的IR，并进行对node的分组
GraphMatchFuser	用来完成node fusion并将group转化为function的helper class

function	function annotation
DFG::Creator::Prepare	完成tvm node到DFG node的映射，分三步：建立映射表、遍历每个node并根据类型映射、更新node父子关系
DFG::Creator::VisitExpr_	自定义重载func，通过自定义AddNode来完成对tvm node的遍历和DFG node的映射
HDFG::Create	定义指定的data flow（该data flow便是一个组，例如cc中的data flow只允许存在一个pad，第二个pad将被归为第二组）
GraphMatchFuser::Partition	算子融合func，采用dfs贪心尽可能多的将node归为一个group
HDFG::FindMatch	广度遍历后序树，input为DFG node，return为bool_match，side effect为修改last_match值以完成一趟hardware data flow
GraphMatchFuser::CollectInputs	向上递归，将没有算进来的结点归为该group（例如constant等节点为新分支节点，没办法向下遍历时访问到，只能通过向上遍历）

2 Partition function细节分析

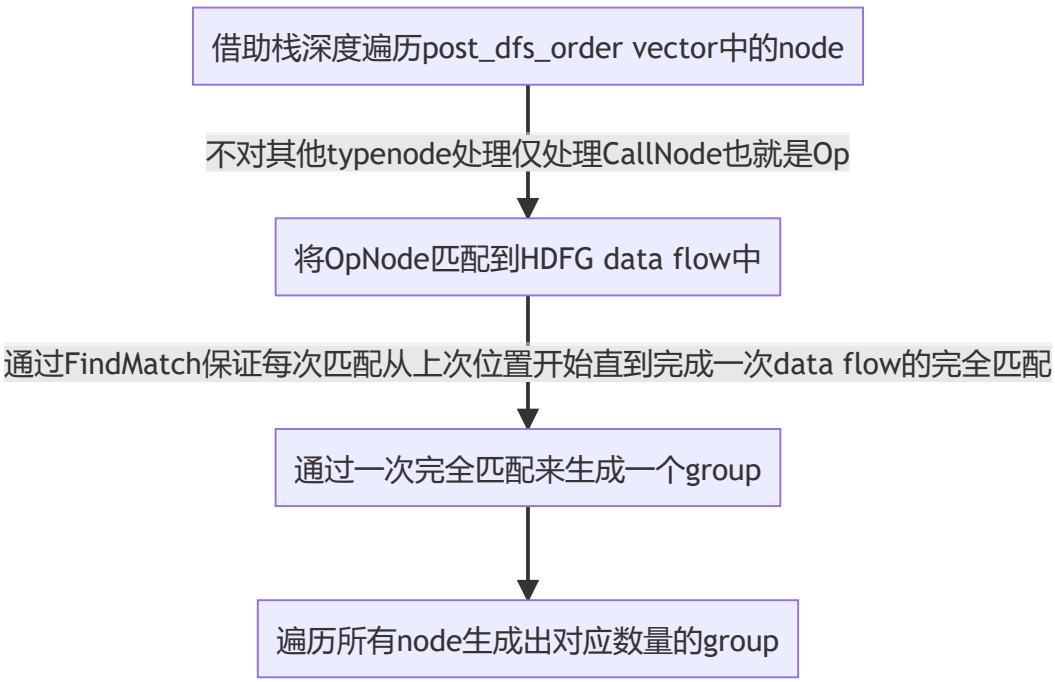
tvm的fuse_op使用了支配树来定义group division rule，而OPU通过定义hardware data flow来确定分组的规则，hardware data flow在Create func中定义：

```
// describe data flow
hdfg.root = concat;
hdfg.LinkFromTo(concat, add_pre); // concat -> add_pre
```

```
hdfg.LinkFromTo(add_pre, mul_pre);
hdfg.LinkFromTo(mul_pre, pad);
hdfg.LinkFromTo(pad, ipa);
hdfg.LinkFromTo(ipa, mul);
hdfg.LinkFromTo(mul, add);
hdfg.LinkFromTo(add, compare);
hdfg.LinkFromTo(add, pool_pad);
hdfg.LinkFromTo(add, res_add);
hdfg.LinkFromTo(pool_pad, pool);
hdfg.LinkFromTo(compare, pool);
hdfg.LinkFromTo(compare, res_add);
hdfg.LinkFromTo(pool, compare);
hdfg.LinkFromTo(pool, res_add);
hdfg.LinkFromTo(res_add, compare);
hdfg.LinkFromTo(res_add, pool);
hdfg.LinkFromTo(compare, upsample);
hdfg.LinkFromTo(pool, upsample);
hdfg.LinkFromTo(res_add, upsample);
...
```

可以得知data flow中padding和IPA(即conv2d)仅出现一次，故每一group中仅有一个pad，这在该data_flow中是关键信号。

Partition function整体逻辑flow如下：



而对于除了CallNode之外的类似VarNode、ConstantNode，均在AddToGroup func中的CollectInputs func中完成，CollectInputs func完成这样的操作：

```
void GraphMatchFuser::AddToGroup(DFG::Node* node, Group* grp) {
  gmap_[node->ref] = grp;
```

```
  // update group root with the latest operator node (post dfs order)
  grp->root_ref = node->ref;
  // add all ungrouped preds
  // aim to collect nodes that cannot be mapped via hdfg.op_map
  // e.g. tuple, const, expand_dims
  // counterexample: tuple has 2 relu inputs, which are not considered below,
  // since they should be captured in the Partition() flow
  // 这是一个向上看的过程，从Op node开始将他上面的node均判断一遍
  for (auto pred : node->pred) {
    CollectInputs(pred, grp);
  }
}

/*
 * 向上递归的过程，将上面没有算进来的结点归为该group，
 * 例如constant等节点为新分支节点，没办法向下遍历时访问到，只能通过向上遍历
 * recursively add inputs of DFG:Node to target group
 * terminate until
 *   1. a root DFG::Node from other group is met
 *   2. a grouped DFG::Node is met
 *   3. a ungrouped DFG::Node, which is a CallNode and will be grouped later
 */
void GraphMatchFuser::CollectInputs(DFG::Node* node, Group* grp) {
  auto it = root_map.find(node->ref);
  if (it != root_map.end()) {
    return;
  } else if (gmap_.find(node->ref) != gmap_.end()) {
    return;
  } else {
    if (node->ref->IsInstance<TupleNode>()) {
      os << "tuple\n";
    } else if (node->ref->IsInstance<ConstantNode>()) {
      os << "const\n";
    } else if (node->ref->IsInstance<CallNode>()) {
      const CallNode* call = static_cast<const CallNode*>(node->ref);
      os << call->op << "\n";
      if (call->op.get()->IsInstance<OpNode>()) {
        // Operators in op_map will be taken care of in Partition()
        const OpNode* op = static_cast<const OpNode*>(call->op.get());
        auto ie = hdfg.op_map.find(op->name);
        if (ie != hdfg.op_map.end()) {
          return;
        }
      }
    } else if (node->ref->IsInstance<VarNode>()) {
```

```
      os << "var\n";
    } else {
      os << "unknown\n";
    }
    gmap_[node->ref] = grp;
    os << "## " << node->index << ": added to group " << grp->index
       << " via CollectInputs() \n";
  }
  for (auto pred : node->pred) {
    CollectInputs(pred, grp);
  }
}
```

每次访问到一个OpNode时，通过pred关系向上递归遍历其他类型node，将这些node归为这个OpNode所在group。Partition中便通过这样的方式完成group分组：

```
    //只对Op_node进行操作，遇到其他类型node直接跳过，
    //比如刚开始跳过node[0]直接到node[1]:Op(nn.pad)，
    //在CollectInputs中将node[0]collect进来。
    if (node->ref->IsInstance<CallNode>()) {
      const CallNode* call = static_cast<const CallNode*>(node->ref);
      std::string opname = call->op.as<OpNode>()->name;
      //从last_match开始是因为在HDFG中定义了一块data_flow，
      //这一块data_flow便定义了一组可能的group，
      //所以会有没有find(因为一趟已经结束了)但是是能够找到Op的情况，这时候说明要开新group了。
      bool find = hdfg.FindMatch(node);
      auto it = hdfg.op_map.find(opname);
      if (find) {
        // mapped to hardware successfully, add to current group
        grp = GetLatestGroup();
        AddToGroup(node, grp);
        os << "## " << node->index << ": " << call->op
           << " added to group " << grp->index << "\n";
      } else if (it != hdfg.op_map.end()) {
        // primitive operator nodes cannot be mapped to hardware data flow
        // try restart mapping (bfs matching) from source of hardware data flow
        hdfg.last_match = nullptr;
        find = hdfg.FindMatch(node);
        if (find) {
          // successfully mapped after from hardware source
          // add node to a new group
          grp = GetLatestGroup(true);
          AddToGroup(node, grp);
          os << "## " << node->index << ": " << call->op
             << " added to group " << grp->index << "\n";
```

```
    }
  }
  ...
```

tvm节点类型问题

在hw的fuse_op具体问题中所处理的NodeType为以下几种：
- FunctionNode
- CallNode
- VarNode
- ConstantNode

暂且不管用以辅助的例如OpNode或没有使用到的例如TupleNode。
其中FunctionNode和CallNode的关系没有怎么厘清，借此笔记边整理边思考。

CallNode class如下：

```
class CallNode : public ExprNode {
 public:
  /*!
   * \brief The operator(function) being invoked 被function调用的operator
   *
   *  - It can be relay::Op which corresponds to the primitive operators.
   *  - It can also be user defined functions (Function, GlobalVar, Var).
   *  - 可以是元算子，也可以是用户自定义functions
   */
  Expr op;

  /*! \brief The arguments(inputs) of the call */
  //该call的输入参数，见如下图
  tvm::Array<relay::Expr> args;

  /*! \brief The additional attributes */
  Attrs attrs;
  ...
```

FunctionNode class如下：

```
class FunctionNode : public BaseFuncNode {
 public:
  /*! \brief Function parameters */
  tvm::Array<Var> params;
  /*!
   * \brief
   * The expression which represents the computation of the function,
   * the expression may reference the parameters, and the type of it
   * or sub-expressions may reference the type variables.
   * 表示函数计算的表达式，表达式可以引用参数，它或子表达式的类型可以引用类型变量。
   */
  Expr body;
  /*! \brief User annotated return type of the function. */
  Type ret_type;
  /*!
   * \brief Type parameters of the function.
   *  Enables the function to vary its type based on these.
   *  This corresponds to template paramaters in c++'s terminology.
   *
   * \note This can be usually empty for non-polymorphic functions.
   */
  tvm::Array<TypeVar> type_params;

  /*!
   * \brief The attributes which store metadata about functions.
   */
  tvm::Attrs attrs;
```

```
GROUP[6] ROOT:0x6ce9ff0 Op(nn.leaky_relu)
7 6:1
new_args are: Var(p0, ty=TensorType([1, 1024, 13, 13], float32))
7 7:0
7 7:0
new_args are: CallNode(Op(nn.pad), [Var(p0, ty=TensorType([1, 1024, 13, 13], float32))], relay.attrs.PadAttrs(0x6c35768), [TensorType([1, 1024, 13, 13], float32)])
new_args are: Constant([[[[ 3.19110951e-03  8.65325145e-03 -4.04290743e-02]
   [-6.63737580e-02 -6.08754754e-02 -5.59197441e-02]
   [-1.82669330e-02 -1.84959767e-03 -2.14269683e-02]]

  [[ 2.11132839e-02  4.70295101e-02  6.11791052e-02]
   [ 7.82777444e-02  4.97723222e-02  5.27830832e-02]
   [-1.67056941e-03  1.59923211e-02  1.78829078e-02]]

  [[ 6.27283975e-02 -9.14163068e-02 -5.42042404e-02]
   [ 2.04739477e-02  1.16414472e-01  3.87375355e-02]
   [ 7.95943514e-02  2.34684777e-02  6.18486665e-02]]

  ...

  [[ 2.97288410e-02 -2.78714821e-02  1.83622371e-02]
   [ 2.91654915e-02 -2.40816623e-02 -4.40047728e-03]
   [-1.41656790e-02 -3.18748020e-02 -1.72210820e-02]]

  [[ 5.22858044e-03  2.81296000e-02 -1.29548004e-02]
   [ 4.23780158e-02 -9.78771481e-04  1.58469018e-03]
   [-6.74006045e-02 -7.52917752e-02 -5.20187579e-02]]

  [[-4.33363812e-03 -4.05620262e-02  3.14398222e-02]
   [-9.78523679e-03 -2.80521289e-02  3.39381397e-03]
   [-6.44844249e-02 -5.86274406e-03 -7.98369721e-02]]]
```

对fuse_op patition的补充： group->rootref是整个group最后一个node，后序访问。

> GROUP[1] ROOT:0x9bac430 Op(nn.max_pool2d)

对每个节点的访问，找到其所在的group，如果内部arg所在的group与该节点group不同，则将该arg group中所有节点的args表线性合并，替换该节点

明天再说 看晕了

图形学部分

```
Intersect(Ray ray, BVH node){
    if (ray misses node.bbox) return;

    if (node is a leaf node){
        test intersection with all objs;
        return closest intersection;
    }

    hit1 = Intersect(ray, node.child1);
    hit2 = Intersect(ray, node.child2);

    return the closer of hit1, hit2;
}
```