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");
  }
};
```