Pass

2022年10月20日 13:30

enhance_channel

对于kernel size大于1*1的conv2d，将维度延拓到input channel上，新的input feature map的channel数量等于原input channel数量*kernel宽*kernel高，新的kernel size为1*1，意在提高input channel较少的卷积层在OPU上的并行度

执行该变换的前提是，原input channel数量*kernel宽*kernel高 不超过DRAM带宽

完全的等价变换，没有考虑stride，若stride不等于1，则新的input feature map中有些元素没有意义

```
void enhance_channel(BasicBlock &bb, Arch* A) {
  for (auto &inst : bb) {
    if (llvm::isa<Conv2dInst>(inst)) {
      Conv2dInst* I = llvm::cast<Conv2dInst>(&inst);
      if ((I->r > 1 || I->s > 1) && I->r * I->s * I->c < A->getElementsPerDRAMAddr()) {
        I->setAttr("channel enhanced", 1);
        I->setAttr("raw K", I->k);
        I->setAttr("raw C", I->c);
        I->setAttr("raw R", I->r);
        I->setAttr("raw S", I->s);
        // Parameter
        I->h -= I->r - 1;
        I->w -= I->s - 1;
        I->c *= I->r * I->s;
        I->r = 1;
        I->s = 1;
        // Input tensorshape
        auto mx = I->getInput(0);
        assert(mx->getName().find("x") != std::string::npos);
        mx->setTensorshape(I->n, I->c, I->h, I->w);
        // Weight tensorshape
        auto mw = I->getInput(1);
        assert(mw->getName().find("w") != std::string::npos);
        mw->setTensorshape(I->k, I->c, I->r, I->s);
        // TODO: layout transform
      }
    }
  }
}
```

lower_complex

将conv2d以tile为单位分解为load matmul store等简单指令

1.查找在buffer size约束下消耗最少cycle的tile方案

```
std::unordered_map<std::string, std::vector<int>> tiling;
FindTilingFactorConv2D(I, A, tiling);
```

```
  cycle_compute = I->r * I->s * ceil_div(h_residue, I->stride) *
  cycle_memory = (float)((h_residue + I->r - 1) * (w_residue + I
  if (h_residue != 0 || w_residue != 0)
    cycle += ceil_div(I->c, spatial_c) * ceil_div(I->k, elements
  //std::cout << h << ": " << cycle << "\n";
  if (cycle <= cycle_min) {
    cycle_min = cycle;
    tiling["h"] = {h, h_cnt, h_residue};
    tiling["w"] = {w, w_cnt, w_residue};
  }
}
```

2.按照层级建立索引

```
auto pair = loop_h->tile(tiling["h"][0]);
LoopAxis* loop_h_0 = pair.first;
LoopAxis* loop_h_1 = pair.second;
pair = loop_w->tile(tiling["w"][0]);
LoopAxis* loop_w_0 = pair.first;
LoopAxis* loop_w_1 = pair.second;
pair = loop_c->tile(tiling["c"][0]);
LoopAxis* loop_c_0 = pair.first;
LoopAxis* loop_c_1 = pair.second;
pair = loop_k->tile(tiling["k"][0]);
LoopAxis* loop_k_0 = pair.first;
LoopAxis* loop_k_1 = pair.second;
pair = loop_k_1->tile(tiling["k"][1]);
LoopAxis* loop_k_1_0 = pair.first;
LoopAxis* loop_k_1_1 = pair.second;
std::vector<LoopAxis*> loopnest =
//  {loop_n, loop_h_0, loop_w_0, loop_c_0, loop_k_0, loop_r, loop_s, loop_h_1, loop_w_1, loop_k_1_0, loop_k_1_1, loop_c_1};
  {loop_n, loop_h_0, loop_w_0, loop_k_0, loop_c_0, loop_r, loop_s, loop_h_1, loop_w_1, loop_k_1_0, loop_k_1_1, loop_c_1};
loop_r->isOnChip = true;
```

3.递归遍历索引，计算每个子任务在input feature map上h w c维度的起止点

```
  loopmap values;
  LowerConv2dToInstruction(I, loopnest, values, 0);
  //BasicBlock *bb = I->getParent();
  //bb->removeInstruction(*I);
```

Allocation

1. 消除重复的DRAM load（针对weight和bias）

遍历LoadInst，缓存在memloc_inst_map，如果找到相同的pairs，保存在remove里面

```
void Allocator::remove_same_dram_load(BasicBlock &bb, Arch *A, MemoryType Type) {
  std::unordered_map<MemoryLocation*, Instruction*> memloc_inst_map;
  std::vector<std::pair<Instruction*, Instruction*>> remove;
  for (auto &I : bb) {
    if (!llvm::isa<LoadInst>(I)) continue;
    std::vector<MemoryLocation*> mb;
    for (auto *m : I.output) {
      if (m->getType() == Type) {
        mb.push_back(I.getInput(0));
      }
    }
    for (auto *m : mb) {
      bool eq = false;
      for (auto item : memloc_inst_map) {
        if (m->equal(*item.first)) {
          eq = true;
          // merge
          remove.push_back({&I, item.second});
          break;
        }
      }
      if (!eq) {
        memloc_inst_map[m] = &I;
      }
    }
  }
```

按照remove，修改原有的instruction

```
  std::cout << bb.getName() << " ";
  std::cout << "#instructions: " << bb.size() << " -> ";
  for (auto item : remove) {
    for (auto inst_succ : item.first->succ) {
      inst_succ->removeDependency(item.first);
      inst_succ->addDependency(item.second);
      inst_succ->removeInput(item.first->getOutput(0));
      inst_succ->addInput(item.second->getOutput(0));
    }
    bb.removeInstruction(*item.first);
  }
  std::cout << bb.size() << " after removing redundant load from DRAM to "
    << MemoryType2String(Type) << "\n";
}
```

2.为feature map, weight, bias分配DRAM地址

以feature map为例，便利读取json文件时创建的mem_locs表，根据其占用的DRAM大小，累加得到地址，依次设定offset

```cpp
void Allocator::dram_allocation(Function &f, Arch *A) {
  std::vector<MemoryLocation*> mem_locs = f.MemoryLocations();
  std::unordered_map<std::string, int> tensor_name_offset_map;
  // fmap
  int addr = A->dram_fmap_addr_offset;
  for (auto *m : mem_locs) {
    std::string name = m->getName();
    if (name.find("x") != std::string::npos) {
      m->setOffset(addr);
      tensor_name_offset_map[name] = addr;
      std::cout << name << " -> DRAM " << addr << "\n";
      auto shape = m->getTensorshape();
      int n = shape[0];
      int c = shape[1];
      int h = shape[2];
      int w = shape[3];
      c = ceil((float)c / A->getElementsPerDRAMAddr()) * A->getElementsPerDRAMAddr();
      addr += n * c * h * w / A->getElementsPerDRAMAddr();
    }
  }
```

3.为feature map, weight, bias分配buffer地址

以WB(weight buffer)为例，此外还有FB(feature buffer), BB(bias buffer)

目前所有的DRAM load都存放于buffer的起始地址，简单但不能高效地利用buffer，十分依赖于load merge优化，避免重复load相同的weight和bias

```cpp
void Allocator::allocate_wb(BasicBlock &bb, Arch *A) {
  int wb_depth = A->getWBDepth();
  int elements_per_dram_addr = A->getElementsPerDRAMAddr();
  int offset = 0;
  int total_bank = A->getWBBank();  // 2
  int bank_id = 0;
  std::vector<std::pair<Instruction*, Instruction*>> mergeto;
  for (auto &I : bb) {
    if (!llvm::isa<LoadInst>(I)) continue;
    for (auto *m : I.output) {
      if (m->getType() == MemoryType::WB) {
        // check w addr needed by m
        auto *mi = I.getInput(0);
        int dram_addr_cnt = mi->getStepSize();
        int wb_addr_cnt = dram_addr_cnt * elements_per_dram_addr / A->w_buffer_bw_bytes;
        // assign physical offset/step for m
        /*
        if (offset + wb_addr_cnt * mi->getStep() > wb_depth) {
          offset = 0;
          bank_id = (bank_id + 1) % total_bank;  // modulo
        }
        */
        m->setOffset(/*offset*/0);  // for each load from DRAM to weight buffer, we simply start
        m->setStepSize(wb_addr_cnt);
        m->setStep(mi->getStep());
        m->setStride(mi->getStep());
        m->setBankId(bank_id);
        bank_id = (bank_id + 1) % total_bank;  // switch for each load
        offset += wb_addr_cnt * mi->getStep();
      }
    }
  }
```

Scheduling

分配执行单元，判断依赖关系，规划指令起止周期，展开matmult运算，指令重排

```cpp
int Scheduler::run(Module &module) {
  Arch *A = module.getArchModel();
  int cycle = 0;
  for (auto &f : module) {
    for (auto &bb : f) {
      std::cout << bb.getName() << "\n";
      assign_engine(bb);
      add_true_dependency(bb);
      add_anti_dependency(bb);
      add_output_dependency(bb);
      get_latency(bb, A);
      cycle += inst_sched(bb, A);
      unroll_matmult(bb);  // codegen related : slow down the sorting, assign sync id without sorting instruction?
      insertion_sort(bb);
    }
  }
  std::cout << "Total latency = "
    << (float)cycle / A->getFreq() * 1000 << "ms (" << A->getFreq() << "Hz)\n";
  return 0;
}
```

1.assign_engine

根据指令类型，分配DMA或PE单元

```cpp
void assign_engine(BasicBlock &bb) {
  for (auto &I : bb) {
    if (llvm::isa<LoadInst>(I) ||
        llvm::isa<StoreInst>(I)) {
      I.setEngine(EngineType::DMA);
    } else if (llvm::isa<MatmultInst>(I)){
      I.setEngine(EngineType::PE);
    } else {
      I.setEngine(EngineType::UNKNOWN);
    }
  }
}
```

2.add_true_dependency 先写后读依赖，a指令的输出影响b指令的输入

add_output_dependency 先写后写依赖，两条指令写入同一地址

add_anti_dependency 先读后写依赖，读取完成前不得对数据做出修改

三者构建原理类似，通过建立索引并查找实现

```cpp
void add_true_dependency(BasicBlock &bb) {
  std::unordered_map<std::string, Instruction*> last_def;
  for (auto &I : bb) {
    for(auto *mi : I.input) {
      std::string mname = getMemLocName(mi);
      auto it = last_def.find(mname);
      if (it != last_def.end()) {
        I.addDependency(it->second);
      }
    }
    // update last def
    for (auto *mo : I.output) {
      if (mo->getType() == MemoryType::DRAM)
        continue;
      last_def[getMemLocName(mo)] = &I;
    }
  }
}
```

3.Get_latency

计算每条指令的执行延迟，并以latency属性保存

以load和store为例

```cpp
void get_latency(BasicBlock &bb, Arch *A) {
  int dram_latency = A->getDRAMLatency();
  int pe_latency = A->getPELatency();
  for (auto &I : bb) {
    // assume after allocation
    if (llvm::isa<LoadInst>(I)) {
      auto mi = I.getInput(0);  // assume uni-operand and load from DRAM
      int latency = mi->step * (mi->step_size + dram_latency);
      I.setAttr("latency", latency);
    } else if (llvm::isa<StoreInst>(I)) {
      auto mo = I.getOutput(0);
      int latency = mo->step * (mo->step_size + dram_latency);
      I.setAttr("latency", latency);
```

4.inst_sched
计算每条指令的起始cycle数和终止cycle数

Completed 已完成指令
Ready 可执行指令
Active 正在执行指令

```cpp
  std::unordered_map<Instruction*, bool> completed;

  std::vector<Instruction*> ready;
  std::vector<Instruction*> active;
```

循环的每一个cycle对应OPU一个时钟cycle
依赖条件全部执行完成的指令会被放入ready，当闲置执行单元数量足够时，ready状态的指令会进入active执行状态，并扣除对应的执行单元数量

```cpp
  while (!ready.empty() || !active.empty()) {
    if (!ready.empty()) {
      std::vector<Instruction*> issue;
      for (auto *inst : ready) {
        if (rmap[EngineType2String(inst->getEngine())] > 0) {
          rmap[EngineType2String(inst->getEngine())]--;
          issue.push_back(inst);
          inst->setAttr("start cycle", cycle);
          //std::cout << inst->toString() << "\n";
          active.push_back(inst);
          //i++;if (i == 8) exit(1);
        }
      }
      for (auto *inst : issue) {
        ready.erase(std::find(ready.begin(), ready.end(), inst));
      }
    }
    cycle++;
```

当某一指令执行完毕，即当前cycle数大于等于指令的起始cycle数+指令latency，将其放入remove中清除active状态，并返还对应的执行单元
同时遍历该指令的后续指令，如果某一后续指令的所有依赖全部处理完毕，则将其放入ready准备执行

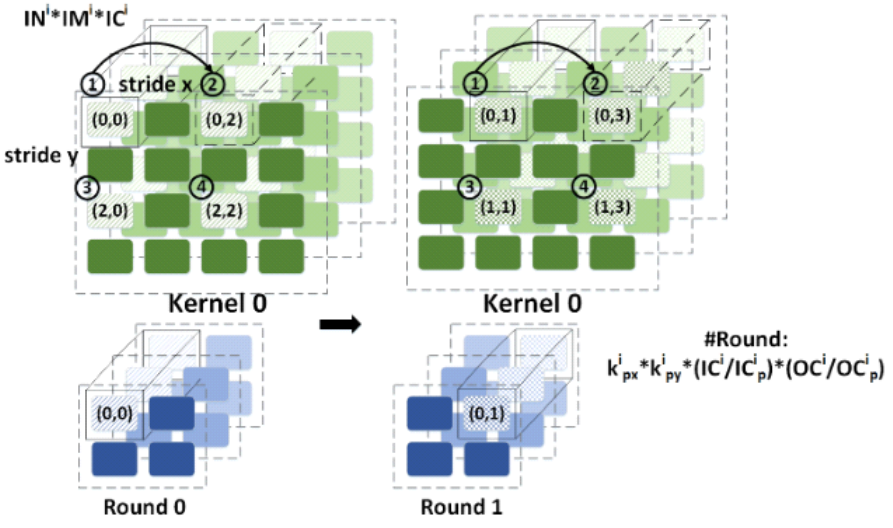
```cpp
std::vector<Instruction*> remove;
for (auto *I : active) {
  if (I->getAttr("start cycle") + I->getAttr("latency") <= cycle) {
    remove.push_back(I);
    completed[I] = true;
    I->setAttr("end cycle", cycle);
    rmap[EngineType2String(I->getEngine())]++;
    for (auto *succ : I->succ) {
      // need to check if all pred are completed
      bool all_pred_complete = true;
      for (auto *pred : succ->pred) {
        all_pred_complete &= completed[pred];
      }
      if (all_pred_complete) {
        ready.push_back(succ);
      }
    }
  }
}
for (auto *I : remove) {
  active.erase(std::find(active.begin(), active.end(), I));
}
```

5.unroll_matmult

将每个tile的卷积拆分为r*s（kernel尺寸）个1*1卷积，input feature map不变

```cpp
int cycle_per_rs = (end_cycle - start_cycle) / ((r_ub - r_lb) * (s_ub - s_lb));
I->setAttr("end cycle", start_cycle + cycle_per_rs);
int offset = mw->getOffset();
int step_size = mw->getStepSize();
for (int i = r_lb; i < r_ub; i++) {
  for (int j = s_lb; j < s_ub; j++) {
    int rs_cnt = (i - r_lb) * (s_ub - s_lb) + j;
    int stride = (*mx->getAccessPattern(2)->vars.begin())->getStride();
    MemoryLocation* mx_local = mx->clone();
    MemoryLocation* mw_local = mw->clone();
    mw_local->setOffset(offset + rs_cnt * step_size);
    mw_local->setStep(1);
    mw_local->setStride(1);
    if (i == r_lb && j == s_lb) {
      mx_local->setAP(2, new AffineExpr(new LoopAxis("h_unroll", h_lb + i, h_ub - r_ub + r_lb + i + 1, stride)));
      mx_local->setAP(3, new AffineExpr(new LoopAxis("w_unroll", w_lb + j, w_ub - s_ub + s_lb + j + 1, stride)));
      mw_local->setAP(2, new AffineExpr(new LoopAxis("r_unroll", i, i + 1)));
      mw_local->setAP(3, new AffineExpr(new LoopAxis("s_unroll", j, j + 1)));
      for (auto &m : I->input) {
        if (m->getType() == MemoryType::FB) {
          m = mx_local;
        } else if (m->getType() == MemoryType::WB) {
          m = mw_local;
        }
      }
```

流程如图（图源 OPU: An FPGA-based Overlay Processorfor Convolutional Neural Networks, Page4）



6.insertion_sort

按照生成的start cycle信息重排指令

```
void insertion_sort(BasicBlock &bb) {
  int i, key, j;
  auto it = bb.begin();
  for (i = 1; i < bb.size(); i++) {
    key = getInstructionByIndex(bb, i).getAttr("start cycle");
    j = i - 1;
    while (j >= 0 && getInstructionByIndex(bb, j).getAttr("start cycle") > key) {
      j--;
    }
    auto &inst_j = getInstructionByIndex(bb, j + 1);
    auto &inst_i = getInstructionByIndex(bb, i);
    inst_i.moveBefore(inst_j);
  }
}
```

dram_layout_gen

生成DRAM Layout JSON文件

```
int DRAMLayoutGen::run(bir::Module &module) {
  std::ofstream outw("dram-weight-layout.json");
  for (auto &f : module) {
    for (auto &bb : f) {
      gen_weight_layout(bb, outw);
    }
  }
  outw.close();
  std::ofstream outb("dram-bias-layout.json");
  for (auto &f : module) {
    for (auto &bb : f) {
      gen_bias_layout(bb, outb);
    }
  }
  outb.close();
  return 0;
}
```

Codegen

按照start_cycle生成指令sync_id，最后依序生成JSON文件

```
void Codegen::generate_fsim_inst(Module &module) {
  auto mc = new LegacyMC();
  Arch *A = module.getArchModel();
  mc->init();
  std::ofstream outj(this->filename);
  for (auto &f : module) {
    for (auto &bb : f) {
      std::cout << "[BB] " << bb.getName() << " export json line code\n";
      mc->sync_map.clear();
      mc->reg_trace.clear();
      mc->next_engine_inst.clear();
      mc->findLastMatmult(bb);
      mc->emitLayerGlobalRegTrace(bb, A);
      mc->setSyncId(bb);
      mc->setSync();
      mc->instMatch();
      mc->exportSimJson(bb, outj);
    }
  }
  outj.close();
}
```