Thorin Report - Shared Memory Code Generation

Rafael Ravedutti Lucio Machado

November 2017

1 Introduction

This report shows the methods used for allowing Thorin code generator to emit the optimized version of the Gaussian Filter (and other simple filters) using shared memory and texture memory. The first part explains the idea used for generating the shared memory and texture memory code and the second shows the modifications included in the Thorin backend code generator.

2 Idea

Thorin is a CPS graph based higher-order intermediate representation, this means that its code generator will iterate over the graph generating its continuations, parameters and primops. For each vertex in the graph, Thorin must generate a unique name to avoid conflicts, our idea is to trace some of these vertex by the name to identify the input image and the filter in the kernels.

In the first part, we defined a macro with the filter dimensions (for now), then at each kernel declaration, we get the kernel block dimensions to emit the shared memory buffer declaration. We also retrieve all the structures and pointers in the kernel parameters for further evaluation of which will be input buffers to store in the faster memories.

Following the analysis, we go through the kernel primops to check the parameters behavior, if a extraction, a bitcast or a LEA operation is performed, we keep inserting the targets in the list so they also are evaluated.

When we find a load operation in one of our targets, we insert them in the shared memory candidates list if they weren't blacklisted. A target is blacklisted when a store operation is performed in it (we don't want to store writable kernel buffers in the shared memory).

After the analysis, for each bitcasting operation with the targets we emit the shared memory copy of the original buffer in the slow global memory to the faster memory by splitting the buffer in blocks of the kernel block dimension. Then we iterate over each buffer blocks with each thread transferring one element (per iteration) from the global memory to the faster memory. Finally, for each LEA operation with a target, we change the target reference to the shared memory with the offsets (x and y axis) included in the indexes.

3 Modifications

```
static std::list<std::string> shm_buffers;
static std::list<std::string> shm_blacklist;
static std::list<std::string> kernel_references;
static std::list<std::string> kernel_pointers;

std::ostream& CCodeGen::emit_shm_copy(
  const std::string shm_name,
  const std::string src_buffer,
  const std::string width,
  const std::string height
) {
  int extend_width = FILTER_WIDTH / 2;
  int extend_height = FILTER_HEIGHT / 2;

  std::string idxx_string = \
    "((blockIdx.x * blockDim.x + threadIdx.x) - " + std::to_string(extend_width) + " + i)";
  std::string idxy_string = \
    "((blockIdx.y * blockDim.y + threadIdx.y) - " + std::to_string(extend_height) + " + j)";

  func_impl_ << endl;

  func_impl_ << "for(int i = 0; i < blockDim.x + " << extend_width * 2 << "; i += blockDim.x) {" << up << endl;
  func_impl_ << "for(int j = 0; j < blockDim.y + " << extend_height * 2 << "; j += blockDim.y) {" << up << endl;
  func_impl_ << "if(threadIdx.x + i < blockDim.x + " << extend_width * 2 << " && " << endl << \
                "   threadIdx.y + j < blockDim.y + " << extend_height * 2 << " && " << endl << \
                "   " << idxx_string << " >= 0 && " << endl << \
                "   " << idxx_string << " < " << width << " && " << endl << \
                "   " << idxy_string << " >= 0 && " << endl << \
                "   " << idxy_string << " < " << height << ") {" << up << endl;

  func_impl_ << shm_name << "[threadIdx.x + i][threadIdx.y + j] = \\" << endl << \
                "  " << src_buffer << "[" << idxy_string << " * " << width << " + " << \
                idxx_string << "];" << down << endl;

  func_impl_ << "}" << down << endl;
  func_impl_ << "}" << down << endl;
  func_impl_ << "}" << endl;

  func_impl_ << endl << "__syncthreads();" << endl;

  return func_impl_;
}

std::ostream& CCodeGen::emit_shm_access(const std::string shm_name, std::string x, std::string y) {
  int extend_width = FILTER_WIDTH / 2;
  int extend_height = FILTER_HEIGHT / 2;

  func_impl_ << shm_name << "[" << x << " + " << extend_width << " - blockIdx.x * blockDim.x][" \
                                << y << " + " << extend_height << " - blockIdx.y * blockDim.y]";

  return func_impl_;
}
```

```
std::stringstream type_stream;

type_stream << param->type();

if(type_stream.str().compare("filter") != 0) {
  kernel_references.push_back(param->unique_name());
}

if(param->type()->isa<PtrType>()) {
  kernel_pointers.push_back(param->unique_name());
}

if(bdimx != 0 && bdimy != 0 && bdimz != 0) {
  func_impl_ << endl << "__shared__ double ds_img[" << (bdimx + (FILTER_WIDTH / 2) * 2) << "][" << (bdimy + (FILT
}

for(const auto& block : schedule) {
  auto continuation = block.continuation();
  if(continuation->empty()) {
    continue;
  }

  assert(continuation == scope.entry() || continuation->is_basicblock());

  for(auto primop : block) {
    auto primop_name = var_name(primop);

    if(auto aggop = primop->isa<AggOp>()) {
      if(aggop->isa<Extract>()) {
        auto found = std::find(kernel_references.begin(), kernel_references.end(), aggop->agg()->unique_name());

        if(found != kernel_references.end()) {
          if(aggop->type()->isa<StructType>()) {
            kernel_references.push_back(primop_name);
          } else if(aggop->type()->isa<PtrType>()) {
            kernel_pointers.push_back(primop_name);
          }
        }
      }
    } else if(auto conv = primop->isa<ConvOp>()) {
      if(conv->isa<Bitcast>()) {
        auto found = std::find(kernel_pointers.begin(), kernel_pointers.end(), conv->from()->unique_name());

        if(found != kernel_pointers.end()) {
          kernel_pointers.push_back(primop_name);
        }
      }
    } else if(auto lea = primop->isa<LEA>()) {
      auto found = std::find(kernel_pointers.begin(), kernel_pointers.end(), lea->ptr()->unique_name());

      if(found != kernel_pointers.end()) {
        kernel_pointers.push_back(primop_name);
      }
    } else if(auto load = primop->isa<Load>()) {
      auto ptr_name = load->ptr()->unique_name();
      auto found = std::find(kernel_pointers.begin(), kernel_pointers.end(), ptr_name);
      auto blacklisted = std::find(shm_blacklist.begin(), shm_blacklist.end(), ptr_name) != shm_blacklist.end();
      if(!blacklisted && found != kernel_pointers.end()) {
        shm_buffers.push_back(ptr_name);
      }
    } else if(auto store = primop->isa<Store>()) {
      auto ptr_name = store->ptr()->unique_name();
      auto found = std::find(shm_buffers.begin(), shm_buffers.end(), ptr_name);

      if(found != shm_buffers.end()) {
        shm_buffers.remove(ptr_name);
      }

      shm_blacklist.push_back(ptr_name);
    }
  }
}

auto found = std::find(shm_buffers.begin(), shm_buffers.end(), def_name);

if(found != shm_buffers.end()) {
    emit_shm_access("ds_img", lea->index()->unique_name(), lea->index()->unique_name());
    func_impl_ << ";";
} else {
    emit(lea->ptr()) << " + ";
    emit(lea->index()) << ";";
}
```