# 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 an 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

We use four lists to perform the analysis, the kernel_references will have all the references to the kernel parameters (doesn't matter which type they are), the kernel_pointers list will contain all the pointers (directly defined or inside a struct) that come from the kernel parameters. The shm_buffers list will contain all the selected buffers that must go to the shared memory and the shm_blacklist will contain all the blacklisted buffers.

```cpp
static unsigned int shm_width = 0;
static std::list<std::string> shm_buffers;
static std::list<std::string> shm_blacklist;
static std::list<std::string> kernel_images;
static std::list<std::string> kernel_filters;
static std::list<std::string> image_pointers;
static std::list<std::string> filter_pointers;
static std::map<std::string, std::string> conv_map;
```

The following code shows the functions that emits the copy from the global memory to the shared memory and the access in the shared memory, respectively.

```cpp
std::ostream& CCodeGen::emit_shm_image_copy(const std::string shm_name, const std::string src_buffer) {
  int extend_width = FILTER_SIZE / 2;
  int extend_height = FILTER_SIZE / 2;



  std::string idxx_string = "gidx - " + std::to_string(extend_width) + " + i";
  std::string idxy_string = "gidy - " + std::to_string(extend_height) + " + j";

  func_impl_ << endl;

  func_impl_ << "#line 100 \"shared_memory_image_copy\"" << endl;
  func_impl_ << "int gidx = blockIdx.x * blockDim.x + threadIdx.x;" << endl;
  func_impl_ << "int gidy = blockIdx.y * blockDim.y + threadIdx.y;" << endl;

  func_impl_ << "for(int i = 0; threadIdx.x + i < blockDim.x + " << extend_width * 2 << "; i += blockDim.x) {" <<
  func_impl_ << "int idxx = " << idxx_string << ";" << endl;

  func_impl_ << "for(int j = 0; threadIdx.y + j < blockDim.y + " << extend_height * 2 << "; j += blockDim.y) {" <
  func_impl_ << "int idxy = " << idxy_string << ";" << endl;

  func_impl_ << "if(idxx < image_width_reg && idxy < image_height_reg) {" << up << endl;

  func_impl_ << shm_name << "[(threadIdx.y + j) * " << shm_width << " + threadIdx.x + i] = " << \
                src_buffer << "[idxy * image_width_reg + idxx];" << down << endl;

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

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

    func_impl_ << endl << "double *adjusted_" << shm_name << " = &" << shm_name << \
                          "[(" << extend_height << " - blockIdx.y * blockDim.y) * " << shm_width << \
                          " + " << extend_width << " - blockIdx.x * blockDim.x];" << endl;

    return func_impl_;
}

std::ostream& CCodeGen::emit_shm_image_access(const std::string shm_name, std::string x, std::string y) {
    func_impl_ << "&" << "adjusted_" << shm_name << "[(" << y  << ") * " << shm_width << " + (" << x << ")]";

    return func_impl_;
}

std::ostream& CCodeGen::emit_shm_filter_copy(const std::string shm_name, const std::string src_buffer) {
    std::string idx_string;

    func_impl_ << endl;

    func_impl_ << "#line 200 \"shared_memory_filter_copy\"" << endl;

#ifdef SEP_FILTER
    idx_string = "i";

    func_impl_ << "for(int i = 0; i < " << FILTER_SIZE << "; i++) {" << up << endl;
    func_impl_ << shm_name << "[" << idx_string << "] = " << src_buffer << "[" << idx_string << "];" << down << end

    func_impl_ << "}" << endl;
#else
    idx_string = "(threadIdx.y + j) * " + std::to_string(FILTER_SIZE) + " + threadIdx.x + i";

    func_impl_ << "for(int i = 0; i < " << FILTER_SIZE << "; i += blockDim.x) {" << up << endl;
    func_impl_ << "for(int j = 0; j < " << FILTER_SIZE << "; j += blockDim.y) {" << up << endl;
    func_impl_ << "if(threadIdx.x + i < " << FILTER_SIZE << " && " << endl << \
                  "    threadIdx.y + j < " << FILTER_SIZE << ") {" << up << endl;

    func_impl_ << shm_name << "[" << idx_string << "] = \\\"" << endl << \
                  "   " << src_buffer << "[" << idx_string << "];" << down << endl;

    func_impl_ << "}" << down << endl;
    func_impl_ << "}" << down << endl;
    func_impl_ << "}" << endl;
#endif

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

    return func_impl_;
}

std::ostream& CCodeGen::emit_shm_filter_access(const std::string shm_name, std::string index) {
    func_impl_ << "&" << shm_name << "[" << index << "]";

    return func_impl_;
}
```

To avoid the filter data to be used for shared memory, we still ignore all the

parameters with type filter from the kernel parameters. This must be changed
in next work.

```cpp
std::stringstream type_stream;

type_stream << param->type();

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

if(type_stream.str().compare("image") == 0 && !list_contains(kernel_images, param->unique_name())) {
  image_width_name = param->unique_name() + ".e2";
  image_height_name = param->unique_name() + ".e3";
  kernel_images.push_back(param->unique_name());
}

if(type_stream.str().compare("Buffer") == 0 && !list_contains(kernel_images, param->unique_name())) {
  kernel_images.push_back(param->unique_name());
}

if(param->type()->isa<PtrType>() && !list_contains(image_pointers, param->unique_name())) {
  image_pointers.push_back(param->unique_name());
}
```

The following code emits the shared memory declaration in the kernel using
the filter dimension macros.

```cpp
if(bdimx != 0 && bdimy != 0 && bdimz != 0) {
  shm_width = bdimx + (FILTER_SIZE / 2) * 2;
  func_impl_ << endl << "__shared__ double ds_img[" << (shm_width * (bdimy + (FILTER_SIZE / 2) * 2)) << "];";

  #ifdef SEP_FILTER
    func_impl_ << endl << "__shared__ double ds_filter[" << FILTER_SIZE << "];";
  #else
    func_impl_ << endl << "__shared__ double ds_filter[" << (FILTER_SIZE * FILTER_SIZE) << "];";
  #endif
}
```

The following code is executed before the code generation, it perform our
analysis of possible candidates to be optimized using the shared memory.

```cpp
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>()) {
        if(list_contains(kernel_images, aggop->agg()->unique_name())) {
```

```cpp
        if(aggop->type()->isa<StructType>() && !list_contains(kernel_images, primop_name)) {
          kernel_images.push_back(primop_name);
        } else if(aggop->type()->isa<PtrType>() && !list_contains(image_pointers, primop_name)) {
          image_pointers.push_back(primop_name);
        }
      }

      if(list_contains(kernel_filters, aggop->agg()->unique_name())) {
        if(aggop->type()->isa<StructType>() && !list_contains(kernel_filters, primop_name)) {
          kernel_filters.push_back(primop_name);
        } else if(aggop->type()->isa<PtrType>() && !list_contains(filter_pointers, primop_name)) {
          filter_pointers.push_back(primop_name);
        }
      }
    }
  } else if(auto conv = primop->isa<ConvOp>()) {
    if(conv->isa<Bitcast>()) {
      if(list_contains(image_pointers, conv->from()->unique_name())) {
        image_pointers.push_back(primop_name);
      }

      if(list_contains(filter_pointers, conv->from()->unique_name())) {
        filter_pointers.push_back(primop_name);
      }
    }
  } else if(auto lea = primop->isa<LEA>()) {
    if(list_contains(image_pointers, lea->ptr()->unique_name())) {
      conv_map.insert(std::pair<std::string, std::string>(lea->ptr()->unique_name(), primop_name));
      image_pointers.push_back(primop_name);
    }

    if(list_contains(filter_pointers, lea->ptr()->unique_name())) {
      conv_map.insert(std::pair<std::string, std::string>(lea->ptr()->unique_name(), primop_name));
      filter_pointers.push_back(primop_name);
    }
  } else if(auto load = primop->isa<Load>()) {
    auto ptr_name = load->ptr()->unique_name();
    auto blacklisted = list_contains(shm_blacklist, ptr_name);

    if(!blacklisted && (list_contains(image_pointers, ptr_name) || list_contains(filter_pointers, ptr_name))) {
      shm_buffers.push_back(ptr_name);
    }
  } else if(auto store = primop->isa<Store>()) {
    auto ptr_name = store->ptr()->unique_name();

    if(list_contains(shm_buffers, ptr_name)) {
      shm_buffers.remove(ptr_name);
    }

    shm_blacklist.push_back(ptr_name);
  }
  }
}

func_impl_ << endl;
func_impl_ << "int image_width_reg = " << image_width_name << ";" << endl;
func_impl_ << "int image_height_reg = " << image_height_name << ";" << endl;
```

Finally, the next code emits the shared memory access when necessary (it's still necessary to fix the indexes).

```
if (conv->isa<Bitcast>()) {
    func_impl_ << "union { ";
    emit_addr_space(func_impl_, dst_type);
    emit_type(func_impl_, dst_type) << " dst; ";
    emit_addr_space(func_impl_, src_type);
    emit_type(func_impl_, src_type) << " src; ";
    func_impl_ << "} u" << def_name << ";" << endl;
    func_impl_ << "u" << def_name << ".src = ";
    emit(conv->from()) << ";" << endl;
    func_impl_ << def_name << " = u" << def_name << ".dst;";

    if(list_contains(image_pointers, def_name)) {
      std::map<std::string, std::string>::iterator i;

      if((i = conv_map.find(def_name)) != conv_map.end()) {
        if(list_contains(shm_buffers, i->second)) {
          emit_shm_image_copy("ds_img", def_name);
        }
      }
    }

    if(list_contains(filter_pointers, def_name)) {
      std::map<std::string, std::string>::iterator i;

      if((i = conv_map.find(def_name)) != conv_map.end()) {
        if(list_contains(shm_buffers, i->second)) {
          emit_shm_filter_copy("ds_filter", def_name);
        }
      }
    }
}

if (auto lea = def->isa<LEA>()) {
    if (is_texture_type(lea->type())) { // handle texture fetches
        emit_type(func_impl_, lea->ptr_pointee()) << " " << def_name << ";" << endl;
        func_impl_ << def_name << " = tex1Dfetch(";
        emit(lea->ptr()) << ", ";
        emit(lea->index()) << ");";
    } else {
        if (lea->ptr_pointee()->isa<TupleType>() || lea->ptr_pointee()->isa<StructType>()) {
            emit_type(func_impl_, lea->type()) << " " << def_name << ";" << endl;
            func_impl_ << def_name << " = &";
            emit(lea->ptr()) << "->e";
            emit(lea->index()) << ";";
        } else if (lea->ptr_pointee()->isa<DefiniteArrayType>()) {
            emit_type(func_impl_, lea->type()) << " " << def_name << ";" << endl;
            func_impl_ << def_name << " = &";
            emit(lea->ptr()) << "->e[";
            emit(lea->index()) << "];";
        } else {
            if(list_contains(shm_buffers, def_name)) {
              func_impl_ << "#line 100 \"shared_memory_access\"" << endl;
            }

            emit_addr_space(func_impl_, lea->ptr()->type());
```

```cpp
        emit_type(func_impl_, lea->type()) << " " << def_name << ";" << endl;
        func_impl_ << def_name << " = ";

        if(list_contains(shm_buffers, def_name)) {
            std::map<std::string, std::string>::iterator i;
            auto is_filter = list_contains(filter_pointers, def_name);

            /*
            for(i = conv_map.begin(); i != conv_map.end(); ++i) {
              if(i->second.compare(def_name) && list_contains(filter_pointers, i->first)) {
                is_filter = true;
              }
            }
            */

            if(!is_filter) {
              emit_shm_image_access(
                "ds_img",
                lea->index()->unique_name() + " % image_width_reg",
                lea->index()->unique_name() + " / image_width_reg"
              );
            } else {
              emit_shm_filter_access("ds_filter", lea->index()->unique_name());
            }

            func_impl_ << ";";
        } else {
            emit(lea->ptr()) << " + ";
            emit(lea->index()) << ";";
        }
    }
}

insert(def, def_name);
return func_impl_;
}
```

# 4 Things to do

- Perform analysis of the image dimensions by identifying the computing of gid_x and gid_y and then finding the "less than" comparison with the image dimensions

- Adjust the index in the shared memory access

- Emit the shared memory copy after the bitcast of the target pointer

- Use the filter saving its data into texture memory, the challenge is to identify which read-only (in relation to the analysis) buffer is the filter and which one is the image