Accelerating Halide on an FPGA by using CIRCT and Calyx as an intermediate step to go from a high-level and software-centric IRs down to RTL

Sergi Granell Escalfet

May 15, 2023

Master Degree in Innovation and Research in Informatics - High Performance Computing
Facultat d'Informàtica de Barcelona
Universitat Politècnica de Catalunya - BarcelonaTech

Table of Contents

Introduction

Image and array processing

- Image processing and array processing play an essential role in modern life:
  - Applying filters to the images that we upload to social media
  - Running object detection algorithms on self-driving cars
- $\uparrow$ **Sophistication** modern image processing pipelines, resolution image sensors, real-time video processing $\implies \uparrow$ demand for highly **efficient** image processing pipeline implementations
- **Diversity** of targets: from a small device such as a smartphone, smartwatch or edge device to large data center and HPC systems
- Optimizing these algorithms can be **complex** and often results in **non-portable code**
  - Hand-tuned C and assembly for a specific architecture
  - Implementations optimized for an x86 multicore and a modern GPU have little resemblance

Domain Specific Languages (DSLs)

- DSLs: programming languages specialized to a **particular application domain**
- **Abstraction**: they provide a higher level of abstraction tailored to the specific domain
  - Making it easier for developers to express complex concepts and ideas in a concise and natural way
- **Expressiveness**: by focusing on a specific domain, DSLs enable developers to express their intent more directly, resulting in more readable and maintainable code
- **Productivity**: they simplify the development process. Focus on solving domain-specific problems rather than low-level implementation details
- **Performance**: they can be optimized for the specific domain, potentially allowing more efficient execution and better performance
- For the image/array processing application domain $\Longrightarrow$ **Halide**

Halide

Halide

- Main idea: **decouple** the ***algorithm*** definition ("what needs to be computed") from its ***schedule*** ("how it should be computed")
- W/o changing algorithm, explore different optimizations strategies (loop nesting and loop fusion, tiling, recomputation and storage balancing, vectorization, parallelism, . . .)

```
void box_filter_3x3(const Image &in, Image &blury) {
  __m128i one_third = _mm_set1_epi16(21846);
  #pragma omp parallel for
  for (int yTile = 0; yTile < in.height(); yTile += 32) {
    __m128i a, b, c, sum, avg;
    __m128i blurx[(256/8)*(32+2)]; // allocate tile blurx array
    for (int xTile = 0; xTile < in.width(); xTile += 256) {
      __m128i *blurxPtr = blurx;
      for (int y = -1; y < 32+1; y++) {
        const uint16_t *inPtr = &(in[yTile+y][xTile]);
        for (int x = 0; x < 256; x += 8) {
          a = _mm_loadu_si128((__m128i*)(inPtr-1));
          b = _mm_loadu_si128((__m128i*)(inPtr+1));
          c = _mm_load_si128((__m128i*)(inPtr));
          sum = _mm_add_epi16(_mm_add_epi16(a, b), c);
          avg = _mm_mulhi_epi16(sum, one_third);
          _mm_store_si128(blurxPtr++, avg);
          inPtr += 8;
        }
      }
      blurxPtr = blurx;
      for (int y = 0; y < 32; y++) {
        __m128i *outPtr = (__m128i *)(&(blury[yTile+y][xTile]));
        for (int x = 0; x < 256; x += 8) {
          a = _mm_load_si128(blurxPtr+(2*256)/8);
          b = _mm_load_si128(blurxPtr+256/8);
          c = _mm_load_si128(blurxPtr++);
          sum = _mm_add_epi16(_mm_add_epi16(a, b), c);
          avg = _mm_mulhi_epi16(sum, one_third);
          _mm_store_si128(outPtr++, avg);
        }
      }
    }
  }
}
```

Hand-optimized C++. ×11 faster than naive impl., 0.9 ms/megapixel.

$\Longrightarrow$

```
Halide::Func blurx, blury;
Halide::Var x, y, xi, yi;

// The algorithm
blurx(x, y) = (   in(x-1, y) +    in(x, y) +    in(x+1, y))/3;
blury(x, y) = (blurx(x, y-1) + blurx(x, y) + blurx(x, y+1))/3;

// The schedule
blury.tile(x, y, xi, yi, 256, 32)
     .vectorize(xi, 8)
     .parallel(y);
blurx.compute_at(blury, x)
     .store_at(blury, x)
     .vectorize(x, 8);
```

0.9 ms/megapixel.

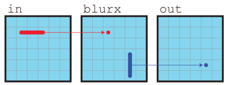
Scheduling trade-offs (1)

Blur 3x3 filter algorithm

```
bx(x, y) = in(x-1, y) + in(x, y) + in(x+1, y)
by(x, y) = bx(x, y-1) + bx(x, y) + bx(x, y+1)
```

Breadth-first strategy

```
Buffer bx(in.width(), in.height());
for (int y = 0; y < in.height(); y++)
  for (int x = 0; x < in.width(); x++)
    bx(x, y) = (in(x-1, y) + in(x, y) + in(x+1, y)) / 3;
for (int y = 0; y < in.height(); y++)
  for (int x = 0; x < in.width(); x++)
    by(x, y) = (bx(x, y-1) + bx(x, y) + bx(x, y+1)) / 3;
```



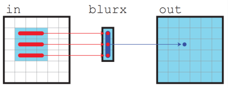
✗✗ Producer-consumer locality

✓✓ Parallelization

✓✓ Recomputation

Total fusion/inline strategy

```
for (int y = 0; y < in.height(); y++)
  for (int x = 0; x < in.width(); x++)
    out(x, y) = (in(x-1, y-1) + in(x, y-1) + in(x+1, y-1) +
                 in(x-1,   y) + in(x,   y) + in(x+1,   y) +
                 in(x-1, y+1) + in(x, y+1) + in(x+1, y+1)) / 9;
```



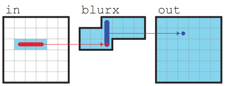
✓✓ Producer-consumer locality

✓✓ Parallelization

✗✗ Recomputation

Sliding window strategy

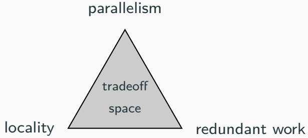
```
Buffer bx(in.width(), 3);
for (int y = -1; y < in.height(); y++)
  for (int x = 0; x < in.width(); x++)
    bx(x, (y + 1) % 3) = (in(x-1, y+1) + in(x, y+1) + in(x+1, y+1)) / 3;
    if (y < 1)
      continue;
    out(x, y-1) = (bx(x, 0) + bx(x, 1) + bx(x, 2) / 3;
```



✓✓ Producer-consumer locality

✗✗ Parallelization

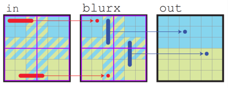
✓✓ Recomputation



Scheduling trade-offs (2)

Tiling strategy

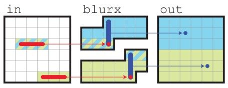
```
for (int y0 = 0; y0 < in.height() / 32; y0++)
  for (int x0 = 0; x1 < in.width() / 32; x0++)
    Buffer bx(32, 32);
    for (int y1 = 0; y1 < 32; y1++)
      for (int x1 = 0; x1 < 32; x1++)
        bx(x1, y1) = (in(x0*32+x1-1, y0*32+y1) +
                      in(x0*32+x1,   y0*32+y1) +
                      in(x0*32+x1+1, y0*32+y1)) / 3;
    for (int y1 = 0; y1 < 32; y1++)
      for (int x1 = 0; x1 < 32; x1++)
        out(x0*32+x1, y0*32+y1) = (bx(x1, y1-1) +
                                   bx(x1, y1)   +
                                   bx(x1, y1+1)) / 3;
```



- ✓ Producer-consumer locality
- ✓✓ Parallelization
- ✗ Recomputation

Sliding window within tiling strategy

```
for (int y0 = 0; y0 < in.height() / 8; y0++)
  Buffer bx(in.width(), 3);
  for (int y1 = -1; y1 < 8; y1++)
    for (int x = 0; x < in.width(); x++)
      bx(x, (y1+1)%3) = (in(x-1, y0*8+y1+1) +
                         in(x,   y0*8+y1+1) +
                         in(x+1, y0*8+y1+1)) / 3;
      if (y1 < 1)
        continue;
      out(x, y0*8+y1-1) = (bx(x, 0) + bx(x, 1) + bx(x, 2)) / 3;
```



- ✓ Producer-consumer locality
- ✓ Parallelization
- ✓ Recomputation

Note

The best scheduling choice differs depending on each target architecture and the computational characteristics of the image pipeline stages.

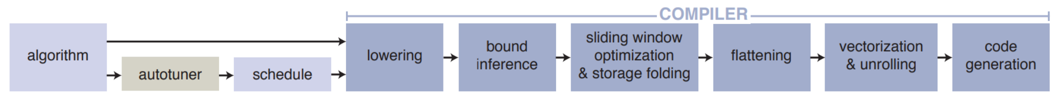
Halide scheduling example

1. Function's *domain* is *tiled* into 64×64-sized tiles
2. Outer two loops of *tiling* are *fused* together into a single loop (`tile_index`)
3. *Fused* loop is *parallelized*
4. Each *tile* is *tiled* again with 4×2-sized sub-tiles
5. Sub-tile innermost x-loop (`x_vectors`) is *vectorized* with the same factor as `x_vectors` (4): no iterations will be performed at this nesting level, the whole loop will be vectorized
6. Sub-tile y-loop (`y_pairs`) is fully *unrolled* with a factor matching the sub-tile vertical size (2), therefore eliminating the sub-tile inner loops in favor of *unrolling* and *vectorization*

```
// Algorithm
Func gradient(x, y) = x + y;

// Schedule
Var x_outer, y_outer, x_inner, y_inner, tile_index;
gradient.tile(x, y, x_outer, y_outer, x_inner, y_inner, 64, 64)
        .fuse(x_outer, y_outer, tile_index)
        .parallel(tile_index);

Var x_inner_outer, y_inner_outer, x_vectors, y_pairs;
gradient.tile(x_inner, y_inner, x_inner_outer, y_inner_outer, x_vectors, y_pairs, 4, 2)
        .vectorize(x_vectors)
        .unroll(y_pairs);

Buffer<int32_t> result = gradient.realize({350, 250});
```

Halide compilation flow

1. **Lowering and loop synthesis**: given the *schedule*, it generates the loop nests and allocations required to evaluate the pipeline, beginning from the output.
2. **Bounds inference**: recursively back from the output and using interval analysis, for each function, it evaluates the bounds of the dimensions based on the bounds required by its caller and the indices it is called with.
3. **Sliding window optimization and storage folding**: traverses the loop nests seeking opportunities for sliding window optimizations (when the results of a function are to be stored by a serial loop at a higher loop nesting level than its computation).
4. **Flattening**: multi-dimensional loads, stores, and allocations are flattened into their linear single-dimensional equivalent.
5. **Vectorization and unrolling**: converts loops that were scheduled as vectorized or unrolled into the corresponding loops. During vectorization, occurrences of a loop index are replaced with a special value $ramp(n)$ which represents the vector $[0, 1, \ldots, n-1]$.
6. **Back-end code generation**: low-level optimizations are performed and machine code is emitted for the resulting pipeline. After running constant-folding and dead-code elimination passes, the Halide IR is ready to be lowered with a `CodeGen` backend. The primary backends use LLVM for code generation.



Halide IR

- Halide IR nodes have an explicit type described by `enum IRNodeType`. Examples:
  - `IntImm` to create integer immediates
  - `Add` to represent additions
  - `Store` and `Load` to perform memory accesses
- `struct IRNode`: `IRNodeType` + reference count. Virtual method `accept` to implement the **visitor pattern**.
  - `IRVisitor`: traverse the IR nodes and perform some action on it (like generating code), but without modifying them.
  - `IRMutator`: traverse the IR nodes to modify them.
- Two kinds of IR nodes, analogously to C:
  - **Expressions** (`ExprNode`): represent some value and have some type (e.g. `x + 3`)
  - **Statements** (`StmtNode`): side-effecting pieces of code that do not represent a value (e.g. `assert(x > 3)`, `store`).
- Type system: signed and unsigned ints, IEEE fp numbers, opaque pointers (like `void *`) and `bfloat`

`IntImm` node definition

```
struct IntImm : public ExprNode<IntImm> {
  int64_t value;

  static const IntImm *make(Type t, int64_t value);

  static const IRNodeType _node_type = IRNodeType::IntImm;
};
```

`IfThenElse` node definition

```
struct IfThenElse : public StmtNode<IfThenElse> {
  Expr condition;
  Stmt then_case, else_case;

  static Stmt make(Expr condition, Stmt then_case, Stmt else_case = Stmt());

  static const IRNodeType _node_type = IRNodeType::IfThenElse;
};
```

Multi-Level Intermediate Representation (MLIR)

- Multi-Level Intermediate Representation (MLIR): open-source compiler infrastructure project; provides common IR to represent **multiple levels of abstractions** maintaining a **unified interface**.
- Under LLVM's umbrella.
- Address challenges in building compilers and optimizing code generation for modern high-performance computing and machine-learning applications.
  - Many compilation and system design problems are better modeled at a **higher**- or **lower-level abstraction**. Languages that use LLVM end up developing their IR to solve domain-specific problems. ML frameworks also use domain-specific abstractions ("ML graphs").
  - Makes it **easy** to define and **introduce new abstraction levels** and provides the infrastructure to use them to solve common compiler engineering problems.
- MLIR infrastructure provides:
  1. Standardized Static Single Assignment (**SSA**)-based IR data structures.
  2. Declarative system for defining IR **dialects**.
  3. Wide range of common infrastructure: documentation, parsing and printing logic, multithreaded compilation support, **pass management**, etc.

MLIR Dialects

- **Dialect**: collection of related **operations**, **attributes** and **types** used to represent a particular domain.
  - *Attributes*: mechanism for specifying constant data on operations in places where a variable is never allowed (such as the comparison predicate of a `cmpi` operation).
- MLIR allows for multiple dialects (even those outside of the main code tree) to **co-exist** together within one *module*. Dialects are produced and consumed by certain ***passes***.
- Examples:
  - ***Arith***: arithmetic dialect, holds basic integer and floating point mathematical operations which include: unary, binary, and ternary arithmetic ops, bitwise and shift ops, cast ops, and compare ops. Operations in this dialect also accept vectors and tensors of integers or floats.
  - ***Func***: creation of high-level function abstractions and function calls.
  - ***Memref***: memory reference, provides a collection of operations and types focused on representing and manipulating multi-dimensional arrays, or tensors, in memory.
  - ***SCF***: structured control flow, which includes operations such as loops and conditionals.
  - ***Vector***: supports multi-dimensional vector types and custom operations on them.
  - ***Affine***: affine expressions and affine loops that allows polyhedral model compilation, analysis and optimizations.

- IR is **generic** enough to represent ASTs in a language frontend, generated instructions in a target-specific backend, HLS constructs, circuits (CIRCT), etc.
- IR is based on a graph-like data structure:
  - Nodes: ***Operations***
  - Edges: ***Values***

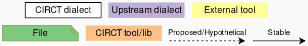
  Each *Value* is the result of exactly one *Operation* or ***Block Argument*** and has a ***Value Type*** defined by the type system.
- Three forms: human-readable (`.mlir`), in-memory and serialized.

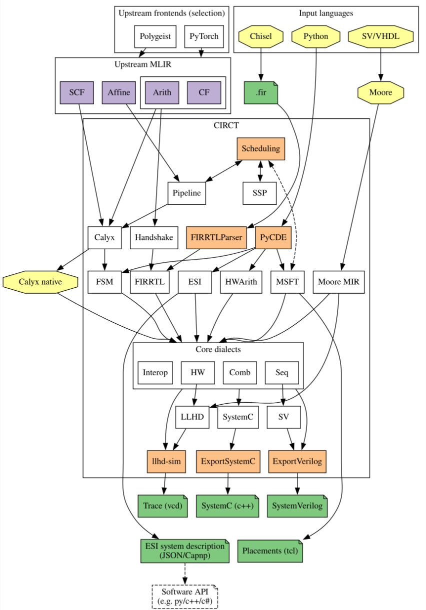
```
func.func @myfunc(%buffer: memref<?xi32>, %size: index) {
  %c0 = arith.constant 0 : index
  %c1 = arith.constant 1 : index
  scf.for %i = %c0 to %size step %c1 {
    %0 = memref.load %buffer[%i] : memref<?xi32>
    %1 = arith.muli %0, %0 : i32
    memref.store %1, %buffer[%i] : memref<?xi32>
  }
  return
}
```

Circuit IR Compilers and Tools (CIRCT)

- **C**ircuit **IR** **C**ompilers and **T**ools (CIRCT): project built **on top of MLIR**. Provides a set of tools and libraries to help with the **design** and verification of **digital circuits**.
- Adds new **hardware-oriented dialects** such as:
    - *`hw`*: generic HW dialect where other dialect operations are instantiated.
    - *`comb`*: models digital combinational logic.
    - *`seq`*: models digital sequential logic.
    - *`fsm`*: models finite-state machines.
    - *`sv`*: represents various SystemVerilog-specific constructs.
    - *`calyx`*: represents Calyx IR types and operations.
- Provides a static scheduling infrastructure:
    - A *problem* is created from the IR (such as *ModuloProblem*).
    - A *scheduler* solves the problem (list scheduler, LP-based schedulers, etc).
    - `AffineToLoopSchedule` pass uses Calyx operator library for operation latencies and lowers to `LoopSchedule` dialect.
- `circt::createExportVerilogPass()` takes IR and emits SystemVerilog code.
    - Style and options controlled by `struct circt::LoweringOptions`.
- Under LLVM's umbrella.
- SiFive contributing to CIRCT (order of magnitude faster than the current Chisel compiler).

CIRCT Dialects and conversion passes





Calyx

- Many hardware DSLs **re-engineer** a new **intermediate language** (IL) and compiler to generate the HW.
- ***Calyx*** is a **shared IL** along a compiler infrastructure that implements useful optimizations and analyses, so that new hardware DSLs can use it as an **intermediate step** to quickly generate hardware designs.
- In Calyx, ***components*** correspond to hardware modules (with input and output ports). Each *component* has three distinct sections:
  - `cells`: the instantion of hardware sub-components that form the *component* being defined.
  - `wires`: set of connection between the sub-components. They can be organized into `groups`.
  - `control`: imperative control flow that defines the *component*'s execution schedule (when each `group` executes).

```
component name(inputs) -> (outputs) {
  cells { ... }
  wires { ... }
  control { ... }
}
```

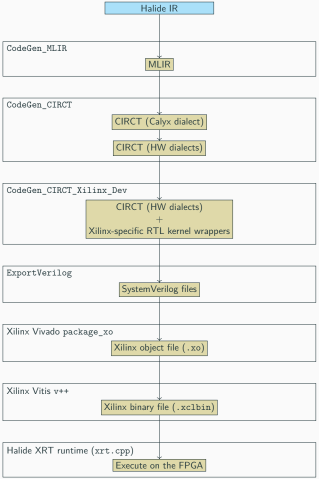
- Each `group` has `go` and `done` ports. `go` triggers execution of the group, `done` finishes it. The `control` section uses those signals to orchestrate `group` execution.
- `Control` statements: `seq`, `par`, `if` and `while`.
- An *assignment* can optionally have a `guard` expression:

```
group mygroup {
  reg3.in = cmp.out ? reg1.out;
  reg3.in = !cmp.out ? reg2.out;
  reg3.write_en = 1'b1;
  mygroup[done] = reg3.done;
}
```

Methodology

Methodology

- Halide down to RTL (Xilinx FPGAs bitstream + XRT) is complex and involves **many steps**.
- **Top-down** and **incremental and iterative** design methodology. Start from simple Halide pipeline and keep adding complexity (new Halide IR nodes), then add MLIR conversion for them.
  - Start with `output(x) = x + 42;`, Halide IR nodes: `Add`, `IntImm`, `For` and `Store`.



Halide MLIR *CodeGen*

Marking loops to be offloaded to an accelerator

1. `InjectAcceleratorOffload` `IRMutator` traverses IR and marks `For` IR nodes with new `enum DeviceAPI::XRT`.
2. For each marked `For` loop:
   2.1 Pass the loop `Stmt` and kernel name to virtual class called `CodeGen_Accelerator_Dev`
      - `CodeGen_Xilinx_Dev` for `enum DeviceAPI::XRT`

   2.2 Replace the loop with a call to the Halide runtime that has been implemented for XRT (`xrt.cpp`):
      - `halide_xrt_run` to start kernel execution
      - But also `halide_xrt_initialize_kernels` and `halide_xrt_finalize_kernels` before/after the kernel execution to load/unload the kernel (FPGA bitstream).
3. `CodeGen_Xilinx_Dev` uses `CodeGen_MLIR` internally to generate the high-level MLIR code to be transformed into RTL.

CodeGen_MLIR: Halide IR to MLIR conversion: Basics

- Kernel arguments converted into func's FuncOp:
  - Non-buffer arguments: mlir_type_of converts Halide type to MLIR type.
  - Buffer arguments: two MLIR arguments are generated:
    1. 64-bit integer: base offset of the buffer within the assigned AXI interface. Written by the host code prior to kernel execution.
    2. MemRefType (Memref dialect): needed by MLIR to perform load/store accesses. Gets converted into a *Calyx external memory interface* later on. Before accessing the base offset is added. Symbol name has ".buffer" suffix.
- Subclass of IRVisitor, CodeGen_MLIR::Visitor, walks IR tree and emits MLIR.
  - Has a void visit(const <NodeType> *) for each Halide IR node type
- "Scoped" symbol table maps string → mlir::Value.
  - sym_push, sym_pop, sym_get
- Helper methods codegen for Expr and Stmt:

```
mlir::Value codegen(const Expr &e);
void codegen(const Stmt &s);
```

They call accept method of the Expr/Stmt passing this as the IRVisitor* argument, which calls the corresponding visit method of the CodeGen_MLIR::Visitor class for that Expr or Stmt node type. E.g. codegen(myIntImm) calls IRVisitor::visit(const IntImm *).

`CodeGen_MLIR`: Halide IR to MLIR conversion: Basic nodes

- `Let`, `LetStmt`: represent the "let" construct found in many functional programming languages.

```
void CodeGen_MLIR::Visitor::visit(const Let *op) {
  sym_push(op->name, codegen(op->value));
  value = codegen(op->body);
  sym_pop(op->name);
}
```

- `IntImm`, `UIntImm`, `FloatImm`: numeric immediates. `mlir_type_of` to convert it to the corresponding `mlir::Type`.

```
void CodeGen_MLIR::Visitor::visit(const IntImm *op) {
  mlir::Type type = mlir_type_of(op->type);
  mlir::IntegerAttr val = builder.getIntegerAttr(type, op->value);
  value = builder.create<mlir::arith::ConstantOp>(type, val);
}
```

- `Add`, `Sub`: addition/subtraction of two `Expr`.

```
void CodeGen_MLIR::Visitor::visit(const Add *op) {
  if (op->type.is_int_or_uint())
    value = builder.create<mlir::arith::AddIOp>(codegen(op->a), codegen(op->b));
  else if (op->type.is_float())
    value = builder.create<mlir::arith::AddFOp>(codegen(op->a), codegen(op->b));
}
```

- `EQ`, `NE`: equality and inequality comparison operations.

```
void CodeGen_MLIR::Visitor::visit(const EQ *op) {
  if (op->a.type().is_int_or_uint())
    value = builder.create<mlir::arith::CmpIOp>(mlir::arith::CmpIPredicate::eq,
                                                codegen(op->a), codegen(op->b));
  else if (op->a.type().is_float())
    value = builder.create<mlir::arith::CmpFOp>(mlir::arith::CmpFPredicate::OEQ,
                                                codegen(op->a), codegen(op->b));
}
```

CodeGen_MLIR: Halide IR to MLIR conversion: For loop

- For: the only loop construct that Halide IR has.

```cpp
void CodeGen_MLIR::Visitor::visit(const For *op) {
  mlir::Value min = codegen(op->min);
  mlir::Value max = builder.create<mlir::arith::AddIOp>(min, codegen(op->extent));
  mlir::Value lb = builder.create<mlir::arith::IndexCastOp>(builder.getIndexType(), min);
  mlir::Value ub = builder.create<mlir::arith::IndexCastOp>(builder.getIndexType(), max);
  mlir::Value step = builder.create<mlir::arith::ConstantIndexOp>(1);

  mlir::scf::ForOp forOp = builder.create<mlir::scf::ForOp>(lb, ub, step);
  {
    mlir::OpBuilder::InsertionGuard guard(builder);
    builder.setInsertionPointToStart(&forOp.getLoopBody().front());

    mlir::Value i = forOp.getInductionVar();
    sym_push(op->name, builder.create<mlir::arith::IndexCastOp>(max.getType(), i));
    codegen(op->body);
    sym_pop(op->name);
  }
}
```

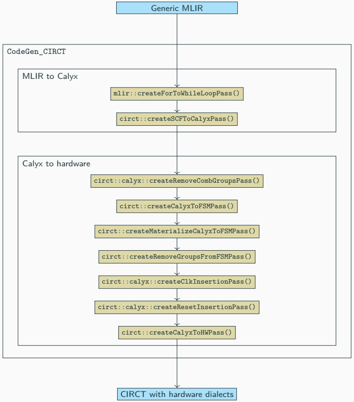
- Load, Store: memory accesses.

```
void CodeGen_MLIR::Visitor::visit(const Load *op) {
  Expr index;
  if (op->type.is_scalar())
    index = op->index;
  else if (Expr ramp_base = strided_ramp_base(op->index); ramp_base.defined())
    index = ramp_base;
  else
    user_error << "Unsupported load.";

  mlir::Value baseIndex = sym_get(op->name);
  mlir::Value indexI64 = builder.create<mlir::arith::ExtUIOp>(builder.getI64Type(), codegen(index));
  mlir::Value address = builder.create<mlir::arith::AddIOp>(baseIndex, indexI64);
  mlir::Value addrIdx = builder.create<mlir::arith::IndexCastOp>(builder.getIndexType(), address);

  mlir::Value buffer = sym_get(op->name + ".buffer");

  if (op->type.is_scalar())
    value = builder.create<mlir::memref::LoadOp>(buffer, mlir::ValueRange{addrIdx});
  else
    value = builder.create<mlir::vector::LoadOp>(mlir_type_of(op->type), buffer, mlir::ValueRange{addrIdx});
}
```

Lowering MLIR to CIRCT

CodeGen_CIRCT: Generates generic RTL kernel

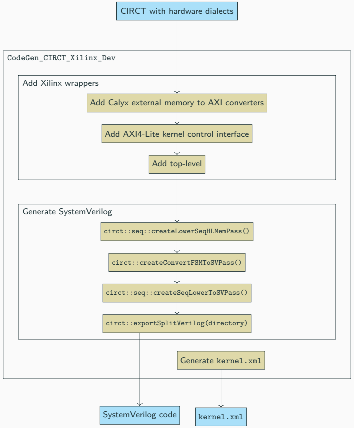
- `MemRefType` arguments transformed into *Calyx external memory interface*
- New features implemented:
  - Passing custom argument names to Calyx
  - Support for sequential-reads memories (`read_en`)and variable memory-access sizes (`access_size`)
  - Support more `arith` dialect operations such as `MinSIOp`
  - Adding initial vector support (`calyx::AssignOp` modified to assign flattened bits $\leftrightarrow$ vectors)
  - Implement basic `vector` dialect operations (such as `vector::SplatOp`, a scalar $\rightarrow$ vector broadcast)



Wrapping the generic RTL kernel for Xilinx FPGAs

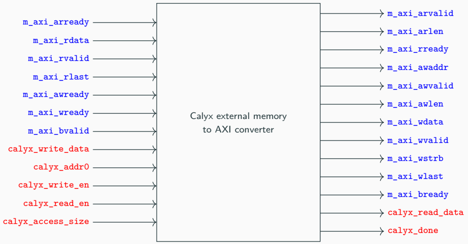
CIRCT to Xilinx RTL

- Wraps generic RTL kernel with necessary Xilinx-specific logic: *Calyx external memory* to AXI converters and AXI4-Lite subordinate **control logic** specified by XRT.
- Generates SystemVerilog code and `kernel.xml` file needed by Vitis `v++`.

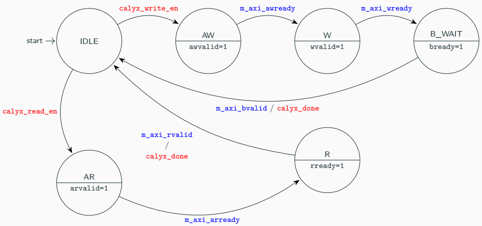


Calyx external memory to AXI converter

- Buffer kernel arguments are accessed through *Calyx external memory interface*.
  - Each buffer is mapped into a different interface. This module converts it to AXI.
- FSM handles two-way handshake of AXI transfers.



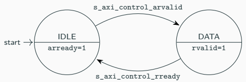
(a) *Calyx external memory* to AXI converter



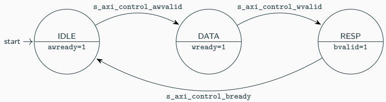
(b) *Calyx external memory* to AXI converter FSM

Kernel control logic

- Support seamless integration with Xilinx runtime libraries: XRT-managed kernel control requirements.
- Kernel acts as an AXI4-Lite subordinate and exposes a set of registers that are used by the host to control the kernel execution and set up the kernel.
  - Control register with `start` and `done` bits.
  - Interrupt registers to enable/mask interrupts.
  - At offset `+0x10` kernel arguments.
- Interrupt line signals events kernel $\rightarrow$ host to avoid continuous polling.



(a) Control interface read handler FSM.



(b) Control interface write handler FSM.

Halide runtime XRT backend

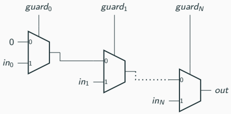
Halide runtime XRT backend (`xrt.cpp`)

- Implements `struct halide_device_interface_impl_t` interface:
  - `halide_xrt_device_malloc`: allocate buffers that the device can access.
  - `halide_xrt_device_free`: deallocate them.
  - `halide_xrt_copy_to_device`: copy a buffer to device/flush CPU cache so that all changes are visible to the device.
  - `halide_xrt_copy_to_host`: copy buffer to host/flush all the relevant device buffers and invalidate CPU cache so that all changes are visible to the host.
- Also implements functions called directly when `For` loop is offloaded to the accelerator:
  - `halide_xrt_initialize_kernels`: loads a kernel into the device.
  - `halide_xrt_finalize_kernels`: unload the kernel.
  - `halide_xrt_run`: starts kernel execution with the specified arguments.
- Allocating buffers depends on loading kernel first, so they are **lazily allocated**: just prior to kernel execution.
- Uses XRT's C API. Seamless integration with Halide.

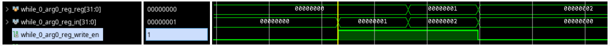
Bugs and issues

Bugs and issues

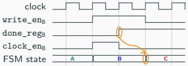
- MLIR and CIRCT are still experimental projects.
- MLIR SCF → Calyx pass and Calyx → HW dialects had **never been tested on real hardware** (or even cycle-accurate simulator)
- Numerous bugs were found. Waveform debugging was essential.
- Bugs were fixed, submitted to upstream, and already merged:
  - Add support for multiple `calyx::AssignOp` with guards to the same destination



  - Clock-enable with the `done` signal when writing to Calyx registers



$\Longrightarrow$



  - `calyx::NotLibOp` was lowered incorrectly (*XOR* with 0s instead of 1s)
  - Read/write-enable signals of external memories were left unconnected

Evaluation

Evaluation

- Avnet Ultra96-V2 Board with 2 GB LPDDR4:

Processor Core	Quad-core Arm® Cortex®-A53 MPCore™ up to 1.5GHz
Memory w/ECC	L1 Cache 32KB I/D per core, L2 Cache 1MB, on-chip Memory 256KB
Graphics Processing Unit	Mali™-400 MP2 up to 667MHz
Memory	L2 Cache 64KB
DRAM Interface	x16: DDR4 w/o ECC; x32/x64: DDR4, LPDDR4, DDR3, DDR3L, LPDDR3 w/ ECC
High-Speed Connectivity	PCIe® Gen2 x4, 2x USB3.0, SATA 3.1, DisplayPort, 4x Tri-mode Gigabit Ethernet

Programmable Functionality	**System Logic Cells (K)**	154
	CLB Flip-Flops (K)	141
	CLB LUTs (K)	71
Memory	**Max. Distributed RAM (Mb)**	1.8
	Total Block RAM (Mb)	7.6
	UltraRAM (Mb)	-
Clocking	**Clock Management Tiles (CMTs)**	3
Integrated IP	**DSP Slices**	360

- 3 Halide kernels used to evaluate the generated RTL code
- For each kernel, resource utilization and execution time analysis performed
- Each kernel has been run three times and the average time has been taken
- The result (output data) of RTL execution binary-compared with CPU execution (“golden model”)
  - Timing of CPU execution taken with vanilla kernel without Halide scheduling directives (defaults to *inline*/*total fusion*)
- For all the kernels, the 150 MHz target frequency was met.
  - Note: 150 MHz on the FPGA, while the CPU runs at 1.5 GHz, 10 times faster

Test load kernel

- Tests the load and store from external memory functionality (32-bit ints):

```
func(x) = 5 * in(x);
func.vectorize(x, VECTORIZATION_FACTOR);
```

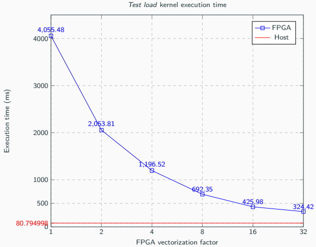
- Evaluation: vect. factors: powers of two from 1 to 32. AXI data bus widths: vector size and max (1024).
- Resource usage:

	Vectorization factor										
	None (1)		2		4		8		16		32
AXI data bus size (bits)	32	1024	64	1024	128	1024	256	1024	512	1024	1024
CLB LUTs	39.29	50.55	39.61	50.75	39.09	51.65	42.65	53.50	47.58	55.52	58.21
LUT as Logic	35.65	46.88	35.97	47.09	35.51	47.98	39.01	49.84	43.91	51.85	54.55
LUT as Memory	8.92	8.98	8.92	8.98	8.76	8.98	8.93	8.98	8.98	8.98	8.98
CLB Registers	28.81	39.36	29.43	39.41	30.00	39.50	32.09	39.69	35.00	40.05	40.78
Register as FF	28.79	39.34	29.41	39.39	29.98	39.47	32.07	39.66	34.97	40.02	40.75
Register as Latch	0.02	0.02	0.02	0.03	0.02	0.03	0.03	0.03	0.03	0.03	0.03

	Vectorization factor										
	None (1)		2		4		8		16		32
AXI data bus size (bits)	32	1024	64	1024	128	1024	256	1024	512	1024	1024
Block RAM Tile	8.56	46.76	8.56	46.76	9.49	46.76	15.05	46.76	25.46	46.76	46.76
RAMB36/FIFO	7.41	45.37	7.41	45.37	8.80	45.37	14.35	45.37	24.07	45.37	45.37
RAMB18	1.16	1.39	1.16	1.39	0.69	1.39	0.69	1.39	1.39	1.39	1.39

	Vectorization factor										
	None (1)		2		4		8		16		32
AXI data bus size (bits)	32	1024	64	1024	128	1024	256	1024	512	1024	1024
DSPs	0	0	0	0	0	0	0	0	0	0	0

  - **1024-bit AXI** leads to a **considerable increase in CLB LUT as Logic**: extra logic needed to drive the data bus wires in the *Calyx external memory to AXI converter*.
  - Native vector AXI width size: 39.29% up to 58.21% and 28.81% up to 40.78%, for vectorization factors 1 and 32, respectively.
- Execution time: buffer with 4 × 1024 × 1024 32-bit ints (16 MB in total).



Test load div int8 kernel

- Take advantage of vectorization support by using 8-bit integers:

```
func(x) = in(x) / 3;
func.vectorize(x, VECTORIZATION_FACTOR);
```

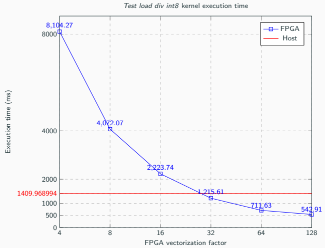
- Evaluation: vect. factors powers of two from 4 to 128, and native and 1024-bit AXI data bus width.
- Resource usage:

	Vectorization factor										
	4		**8**		**16**		**32**		**64**		**128**
AXI data bus size (bits)	32	1024	64	1024	128	1024	256	1024	512	1024	1024
CLB LUTs	39.33	51.47	39.77	53.16	39.60	53.73	43.47	55.35	49.68	58.11	62.14
LUT as Logic	35.69	47.80	36.13	49.50	36.02	50.06	39.82	51.69	46.01	54.44	58.48
LUT as Memory	8.92	8.98	8.92	8.98	8.76	8.98	8.93	8.98	8.98	8.98	8.98
CLB Registers	28.81	39.36	29.43	39.41	29.99	39.49	32.07	39.66	34.95	40.00	40.69
Register as FF	28.79	39.34	29.40	39.38	29.97	39.47	32.05	39.64	34.92	39.98	40.66
Register as Latch	0.02	0.02	0.03	0.03	0.02	0.03	0.03	0.03	0.03	0.03	0.03

	Vectorization factor										
	4		**8**		**16**		**32**		**64**		**128**
AXI data bus size (bits)	32	1024	64	1024	128	1024	256	1024	512	1024	1024
Block RAM Tile	8.56	46.76	8.56	46.76	9.49	46.76	15.05	46.76	25.46	46.76	46.76
RAMB36/FIFO	7.41	45.37	7.41	45.37	8.80	45.37	14.35	45.37	24.07	45.37	45.37
RAMB18	1.16	1.39	1.16	1.39	0.69	1.39	0.69	1.39	1.39	1.39	1.39

	Vectorization factor										
	4		**8**		**16**		**32**		**64**		**128**
AXI data bus size (bits)	32	1024	64	1024	128	1024	256	1024	512	1024	1024
DSPs	0	0	0	0	0	0	0	0	0	0	0

  - Same as before: increase in the logic when native vector AXI vs 1024-bits.
  - Division by constant, the synthesizer used logic and did **not use DSPs** to implement it.
- Execution time: buffer with $32 \times 1024 \times 1024$ 8-bit ints.



  - After vect. factor of 32, it runs faster than the CPU
  - With vector. factor of 128 elements (1024-bit AXI accesses), it is $\times 2.6$ **faster**.

Test blur3x3 sliding window kernel (1)

- 3x3 blur filter on a 2D array of 32-bit ints. Tries to exploit most of the FPGA characteristics and supported features implemented.

```
// Algorithm
tmp(x, y) = input(x, y);
blur_x(x, y) = (tmp(x-1, y) + tmp(x, y) + tmp(x+1, y))/3;
blur_y(x, y) = (blur_x(x, y-1) + blur_x(x, y) + blur_x(x, y+1))/3;

// Schedule
blur_y.tile(x, y, xi, yi, TILE_SIZE, TILE_SIZE);

blur_x.store_at(blur_y, x)
      .compute_at(blur_y, yi);

tmp.store_at(blur_x, y)
   .compute_at(blur_x, x);
```

- Exploit device locality and loads from external memory by using **sliding windows within tiling** with device-local buffers.
- Evaluation: since **vector accesses to local memories were not implemented for this thesis**, **vectorization can not be used**. Different **tile sizes** will be evaluated: 8×8, 16×16 and 32×32.
  - AXI data bus width was set to the element size (32 bits)

Test blur3x3 sliding window kernel (2)

- Resource usage:

	Tile size		
	8x8	16x16	32x32
CLB LUTs	41.75	41.75	41.82
LUT as Logic	38.06	38.04	38.07
LUT as Memory	9.03	9.09	9.20
CLB Registers	29.41	29.41	29.40
Register as FF	29.35	29.35	29.35
Register as Latch	0.05	0.05	0.05

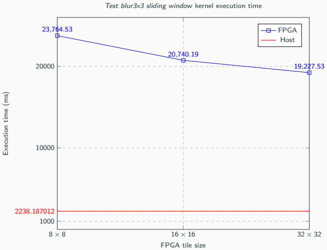
	Tile size		
	8x8	16x16	32x32
`tmp`	4×32	4×32	4×32
`blur_x`	$8 \times 4 \times 32$	$16 \times 4 \times 32$	$32 \times 4 \times 32$
Total	1152 bits	2176 bits	4224 bits

	Tile size		
	8x8	16x16	32x32
Block RAM Tile	8.56	8.56	8.56
RAMB36/FIFO	7.41	7.41	7.41
RAMB18	1.16	1.16	1.16

		Tile size		
	Category	8x8	16x16	32x32
RAMS32	CLB	462	430	430
RAMS64E	CLB	0	32	64
RAMB18E2	BLOCKRAM	5	5	5
RAMB36E2	BLOCKRAM	16	16	16

	Tile size					
	Used (count)			Utilization (%)		
	8x8	16x16	32x32	8x8	16x16	32x32
DSP48E2	9	9	9	2.50	2.50	2.50

  - CLB utilization barely changes: from control PoV the only change are loop boundaries (constants).
  - Sliding buffers allocated as distributed memories in the CLBs (as SRAMs).
  - Multiplication of $3 \times 3 = 9$ input elements for each output using DSPs. Two divs constant 3 using logic as before.
- Execution time: buffer with 4096×4096 32-bit ints.



  - **Local buffers and vectorization is currently unsupported**: kernel code not vectorized and accesses to external memory limited **32 bits**. Execution on the FPGA 10 times slower than on the CPU.
  - Execution time reduces when increasing the tile size: thanks to the **reduction of re-computation of tile boundaries**.

Conclusions and future work

Conclusions

- FPGAs are **highly flexible devices**. Enable the power of reconfigurable hardware for a wide range of applications.
- However, developing FPGA applications is a **complex task**: requires a deep understanding of both hardware and software design flows.
- **DSLs** have emerged to help simplify the development process by providing **higher levels of abstraction** and focusing on specific problem domains.
- **Halide** is a popular DSL designed for expressing **image and array processing** and computational photography algorithms.
- In this thesis, a **new backend** for Halide which **targets FPGAs** has been developed.
  - **Generic RTL kernel** is generated. Then wrapped for Xilinx FPGA devices.
- Instead of directly generating RTL or HLS code, **generic MLIR** is first generated.
  - **Can target many devices and acceleration APIs**
- **Novel flow** using **CIRCT** and **Calyx** to convert generic MLIR has been implemented.
  - Leverages the flexibility and extensibility of MLIR and CIRCT
- Results presented in this thesis show that the approach is **viable** and can be used to generate efficient FPGA code from Halide programs.
  - Still **work to do** and features to be added to improve the generated code.
- Contributed to open source projects by finding bugs, fixing them, adding and bringing the necessity of new features.

Future work

- Support vectorized accesses to local memory
- Improved support for MLIR's `arith` *min* and *max* operations
- Generalize `MemRefType` lowering
- Proper support for `scf::IfOp` in CIRCT's `SCFToCalyx` pass
- Implement lowering of `calyx::ParOp` in `CalyxToHW`
- Add floating-point support in the MLIR to RTL lowering
- Avoid useless pipeline stages after `comb` canonicalization of lowered pipelined Calyx operations
- Emit loops and memory accesses using MLIR's `affine` dialect
- Use CIRCT's static scheduling infrastructure to lower MLIR to Calyx
- Add AXI-Stream support
- Coalescing buffer to implement write-combining
- Experiment with HLS code generation from MLIR
- Halide autoschedulers for FPGA targets
- ...

Thank you for your attention.