Lift: The Language, The IR and Code Generation

Naums Mogers

April 2nd, 2018

University of Edinburgh

Table of contents

LIFT – An Intermediate Language

Algorithmic Patterns

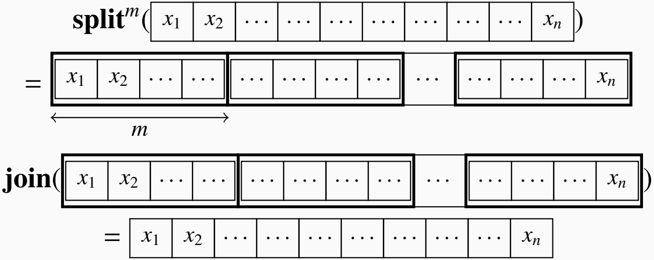
$$\textbf{mapSeq}(f, \boxed{x_n \mid \cdots \mid x_2 \mid x_1}) = \boxed{f(x_1) \mid f(x_2) \mid \cdots \mid f(x_n)}$$

$$\textbf{reduceSeq}(z, f, \boxed{x_n \mid \cdots \mid x_2 \mid x_1}) = \boxed{f(\cdots(f(f(z, x_1), x_2)\cdots), x_n)}$$

$$\textbf{id}(\boxed{x_n \mid \cdots \mid x_2 \mid x_1}) = \boxed{x_n \mid \cdots \mid x_2 \mid x_1}$$

$$\textbf{iterate}^m(f, \boxed{x_n \mid \cdots \mid x_2 \mid x_1}) = \underbrace{f(\cdots(f(}_{m\text{ times}}\boxed{x_n \mid \cdots \mid x_2 \mid x_1})))$$

Data Layout Patterns



- Do not perform any computation
- Reorganize the data layout (`View`)

Data Layout Patterns

$$\textbf{gather}(f, \boxed{x_{f(1)} \mid x_{f(2)} \mid \cdots \mid x_{f(n)}}) = \boxed{x_1 \mid x_2 \mid \cdots \mid x_n}$$

$$\textbf{scatter}(f, \boxed{x_1 \mid x_2 \mid \cdots \mid x_n}) = \boxed{x_{f(1)} \mid x_{f(2)} \mid \cdots \mid x_{f(n)}}$$

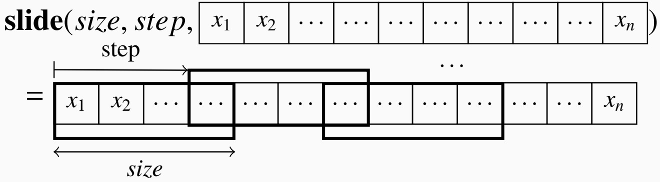
```scala
val transposeFunction = (outerSize: ArithExpr, innerSize: ArithExpr) =>
(i: ArithExpr, _) => {
  val col = (i % innerSize) * outerSize
  val row = i / innerSize

  row + col
}

val Transpose = Split(N) o Gather(IndexFunction.transposeFunction(M, N)) o Join()
```

For examples of **`Gather`** and **`Scatter`** indexing functions, see *src/main/ir/ast/package.scala*

$$\mathbf{zip}(\begin{array}{|c|c|c|c|}\hline x_1 & x_2 & \cdots & x_n \\ \hline\end{array}, \begin{array}{|c|c|c|c|}\hline y_1 & y_2 & \cdots & y_n \\ \hline\end{array})$$

$$= \begin{array}{|c|c|c|c|}\hline (x_1,y_1) & (x_2,y_2) & \cdots & (x_n,y_n) \\ \hline\end{array}$$

$$\mathbf{get}_i((x_1, x_2, \ldots, x_n)) = x_i$$



Parallel Patterns

- `mapWrg`$(0-2)$
- `mapLcl`$(0-2)$
- `mapGlb`$(0-2)$

- `mapWarp`
- `mapLane`

Address Space Patterns

toGlobal **toLocal** **toPrivate**

```
MapWrg(MapLcl(toLocal(MapSeq(id))) $ X
```

- These primitives decouple the decision of *where* to store data from the decision of *how* the data is produced.

Vectorize Pattern

$$\textbf{asVector}(\boxed{x_1 \mid x_2 \mid \cdots \mid x_n}) = \overrightarrow{x_1, x_2, \ldots, x_n},\ x_i \text{ is scalar}$$

$$\textbf{asScalar}(\overrightarrow{x_1, x_2, \ldots, x_n}) = \boxed{x_1 \mid x_2 \mid \cdots \mid x_n}$$

- During code generation, the LIFT compiler transforms f into a vectorized form using OpenCL built-in vectorized arithmetic operations whenever possible.
    - In other cases, f is applied to each scalar in the vector.

Low-level IRs

All LIFT primitives are either:

- High-level, capturing rich information about the algorithmic structure of programs
- Low-level and platform-specific (OpenCL, OpenCL for FPGAs, OpenMP, etc)

Writing an Application

General Steps

- Determine input parameters
- Initialise input data
    - If testing, initialise comparison data
- Craft or translate the algorithm of interest
- Create an OpenCL kernel from your algorithm

Data Input to Lift Algorithms

- Lift can take in arrays or scalars as input parameters

```
val liftLambda = fun(
  ArrayType(Float, SizeVar("N")),
  ArrayType(Float, weights.length),
  ...
)
```

- Single entry point for arrays into functions
  - Multiple arrays can be zipped together (but must be the same size!)

```
fun(neighbourhood) =>
{
    ...
    $ Zip(weights, neighbourhood)
}
```

Initialising Data in Scala

- Create arrays of data to pass into Lift algorithms in Scala

```
val stencilValues = Array.tabulate(nx,ny,nz) { (i,j,k) => (i + j + k + 1).toFloat }
```

- Our examples are all in unit tests, which include data to compare against - often from the same algorithm in Scala

```
assertEquals(dotProductScala(lift,right), output.sum, 0.0f)
```

Developing an Algorithm

The goal is not for Lift to be programmed in directly.

However, functionality for new types of algorithms must be added in and tested. In doing so, there are a few things to keep in mind:

- Lift allows multiple inputs, but there is only one data entry point to the main algorithm (can contain tuples)
- The algorithm itself must eventually map values back to global memory
- The result will be returned in a single array (however, this array can also contain tuples)

Simple Example: 1D Jacobi Stencil

```
val jacobi1Dstencil = fun(
  ArrayType(Float, N),
  (input) => {
    Map(Reduce(add, 0.0f)) o
      Slide(3, 1) o
        Pad(1, 1, clamp) $ input
  }
)
```

Creating an OpenCL kernel

- To compile your Lift kernel to OpenCL, run **`[opencl.executor]Compile(<kernel>)`**
  - This kernel can then be saved as a string or file

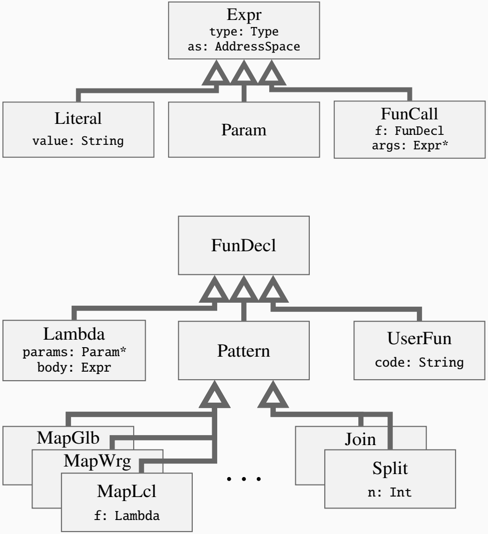
```
Compile(lambda)
```

- To execute the kernel straight away (compiling will happen behind the scenes), run **`[opencl.executor]Execute(<options>)`**
  `[Array[type]](lambda, ..inputs..)`

```
val (output, runtime) = Execute(inputData.length)[Array[Float]](stencilLambda, inputData, stencilWeights
```

LIFT Intermediate Representation

Class diagram



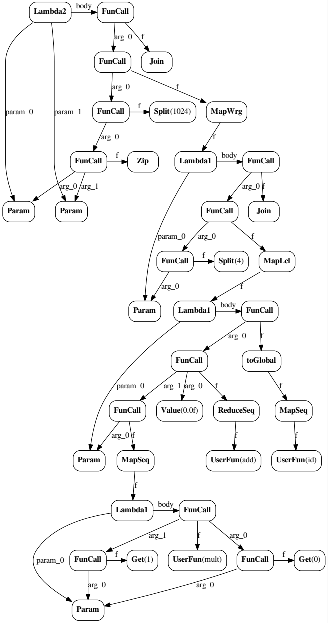
- **Expressions** represent values and have a type associated with.
- **Function declarations** represent callable entities: lambdas, patterns and user functions.

Dot product example

```scala
val dotProductLift = fun(
  ArrayTypeWSWC(Float, N),
  ArrayTypeWSWC(Float, N),
  (left, right) => {
    Join() o MapWrg(
      Join() o
      MapLcl(
        toGlobal(MapSeq(id)) o
        ReduceSeq(add, 0.0f) o
        MapSeq(mult)) o
      Split(4)
    ) o Split(1024) $ Zip(left, right)
  })
```

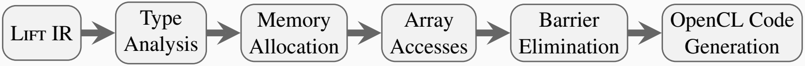
For more dot product variations, see
src/test/tutorial/applications/DotProduct.scala

Corresponding AST



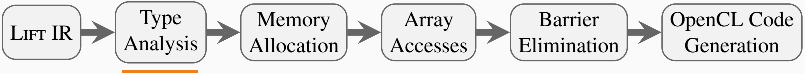
LIFT compilation

Compilation stages



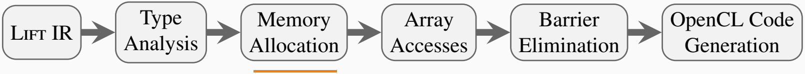
- Compile: *src/main/opencl/executor/Compile.scala:44*
  - Type checking: *src/main/ir/TypeChecker.scala:39*
    - Example Pattern.checkType(): *src/main/opencl/ir/pattern/ReduceSeq.scala:11*
  - Generate: *src/main/opencl/generator/OpenCLGenerator.scala:176*
    - Memory address space inference: *src/main/opencl/ir/InferOpenCLAddressSpace.scala:18*
    - Domain-specific range inference: *src/main/opencl/generator/RangesAndCounts.scala:26*
    - Memory allocation: *src/main/ir/Type.scala:559*
    - Loop unrolling: *src/main/opencl/generator/ShouldUnroll.scala:50*
    - Barrier elimination: *src/main/opencl/generator/BarrierElimination.scala:41*
    - Views (array Accesses): *src/main/ir/view/View.scala:585*

LIFT type system



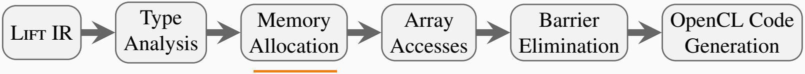
- Lift has a *dependent* type system
- Scalar types: **`int`**, **`float`**, etc
- Vector types corresponding to OpenCL types **`int2`**, **`float4`**, etc
- Tuples
    - *Represented as* **`structs`** *in the generated OpenCL code*
- Arrays
    - Can be nested
    - Carry information about the size and capacity of each dimension in their type
    - This information is represented by arithmetic expressions (more on this later)

Memory allocation



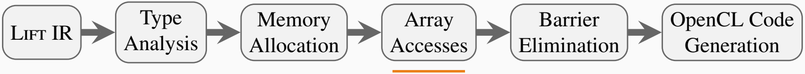
- The naive approach would be to allocate a new output buffer for every **`FunCall`** AST node
- We only allocate memory to the nodes where the called function contains a **`UserFun`**
- The address space is inferred from **`FunCall`**

Memory allocation



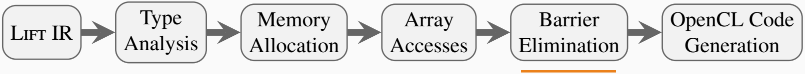

```
input  : Lambda expression representing a program
output : Expressions annotated with address space information

inferAddressSpaceProg(lambda)
foreach param in lambda.params do
    if param.type is ScalarType then param.as = PrivateMemory;
    else param.as = GlobalMemory;
inferASExpr(lambda.body, null)

inferASExpr(expr, writeTo)
switch expr.type do
    case Literal expr.as = PrivateMemory;
    case Param assert (expr.as != null);
    case FunCall
        foreach arg in expr.args do
            inferASExpr(arg, writeTo)
        switch expr.f.type do
            case is UserFun
                if writeTo != null then expr.as = writeTo;
                else expr.as = inferASFromArgs(expr.args);
            case is Lambda inferASFunCall(expr.f, expr.args, writeTo);
            case is toPrivate
             inferASFunCall(expr.f.lambda, expr.args, PrivateMemory);
            case is toLocal
             inferASFunCall(expr.f.lambda, expr.args, LocalMemory);
            case is toGlobal
             inferASFunCall(expr.f.lambda, expr.args, GlobalMemory);
            case is Reduce
             inferASFunCall(expr.f.f, expr.args, expr.f.init.as);
            case is Iterate or Map
             inferASFunCall(expr.f.f, expr.args, writeTo);
            otherwise do  expr.as = expr.args.as;
```

Array accesses



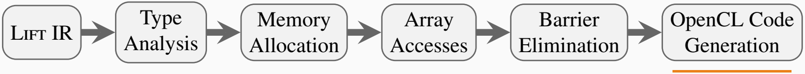
- In LIFT IR, arrays are accessed implicitly based on the patterns
- This eliminates arbitrary memory accesses and the associated problems
- However, expressing (efficient) pattern-transformed accesses is not obvious
- ...which is where **`Views`** come to the rescue (but more on that later)

Barrier elimination



- We start by synchronizing after each occurrence of a parallel **`Map`**
- Then we remove barriers one by one in cases when it can be inferred that they are not required
  - When the data is not shared (i.e. **`Split`**, **`Join`**, **`Gather`** and **`Scatter`** are not used)
  - When the two parallel **`Maps`** are executed independently in separate branches of **`Zip`**

OpenCL code generation



- The AST is traversed recursively
- No OpenCL code is generated for the patterns that only affect **`View`**
- Low-level optimizations such as loop unrolling are applied to simplify the control flow using the information on *ranges* inferred from the patterns such as **`mapLcl`**

The end

Slides are available at
http://www.lift-project.org/ispass2018