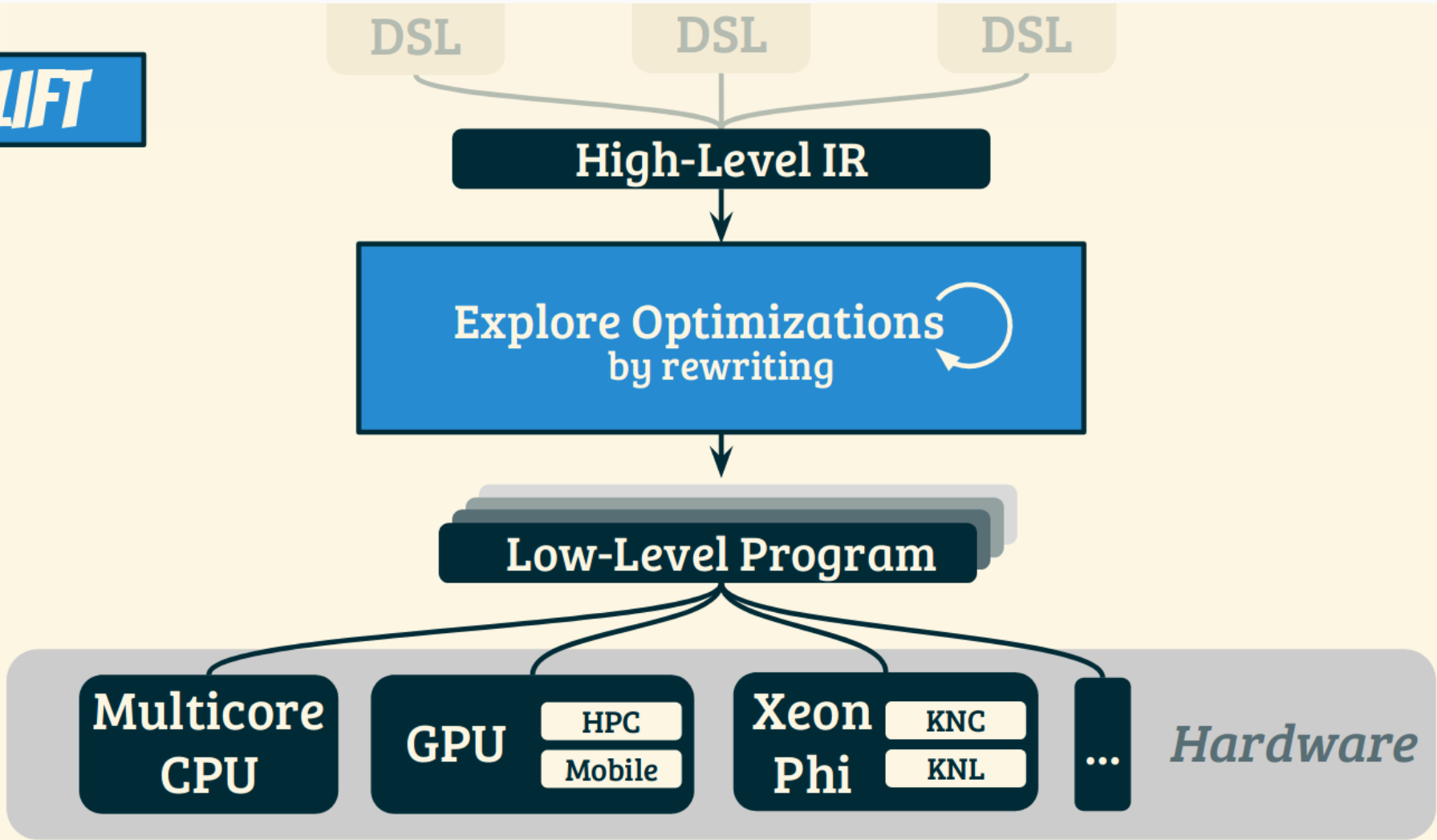
Lift Tutorial: Rewriting and Exploration

Bastian Hagedorn

Lift Overview

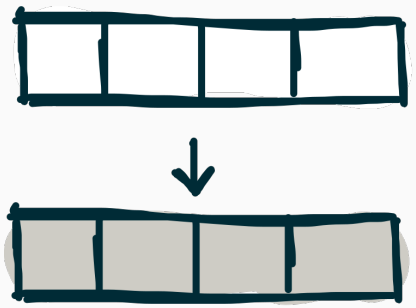


Introduction

Rewrite Rules

Rewrite Rules transform expressions without changing semantics

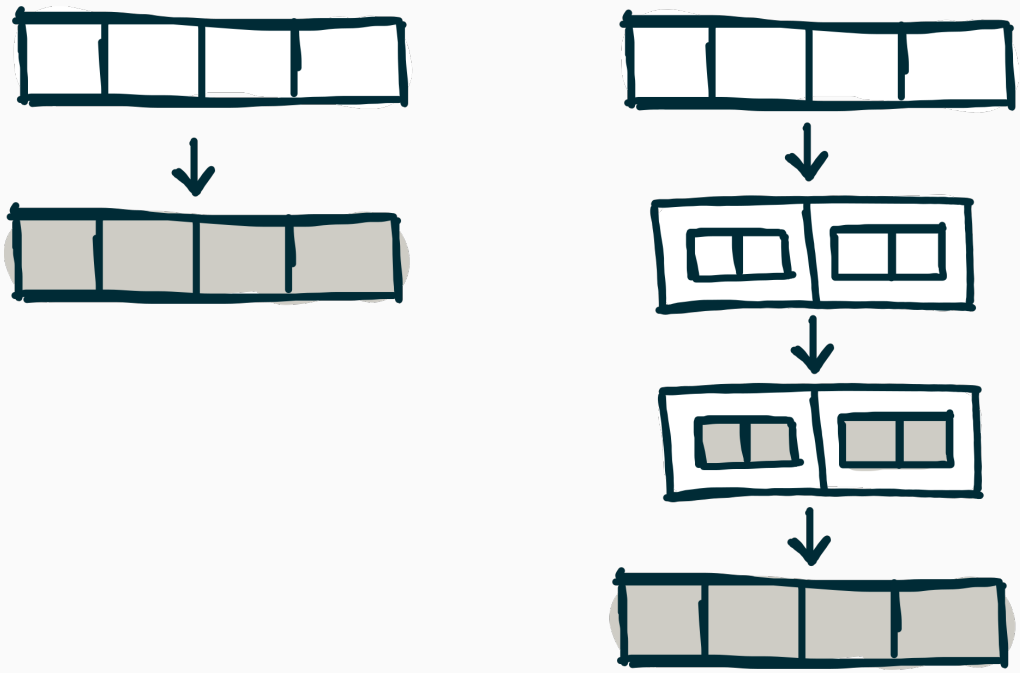
$$map(f) = join \circ map(map(f)) \circ split(n)$$



Rewrite Rules

Rewrite Rules transform expressions without changing semantics

$$map(f) = join \circ map(map(f)) \circ split(n)$$

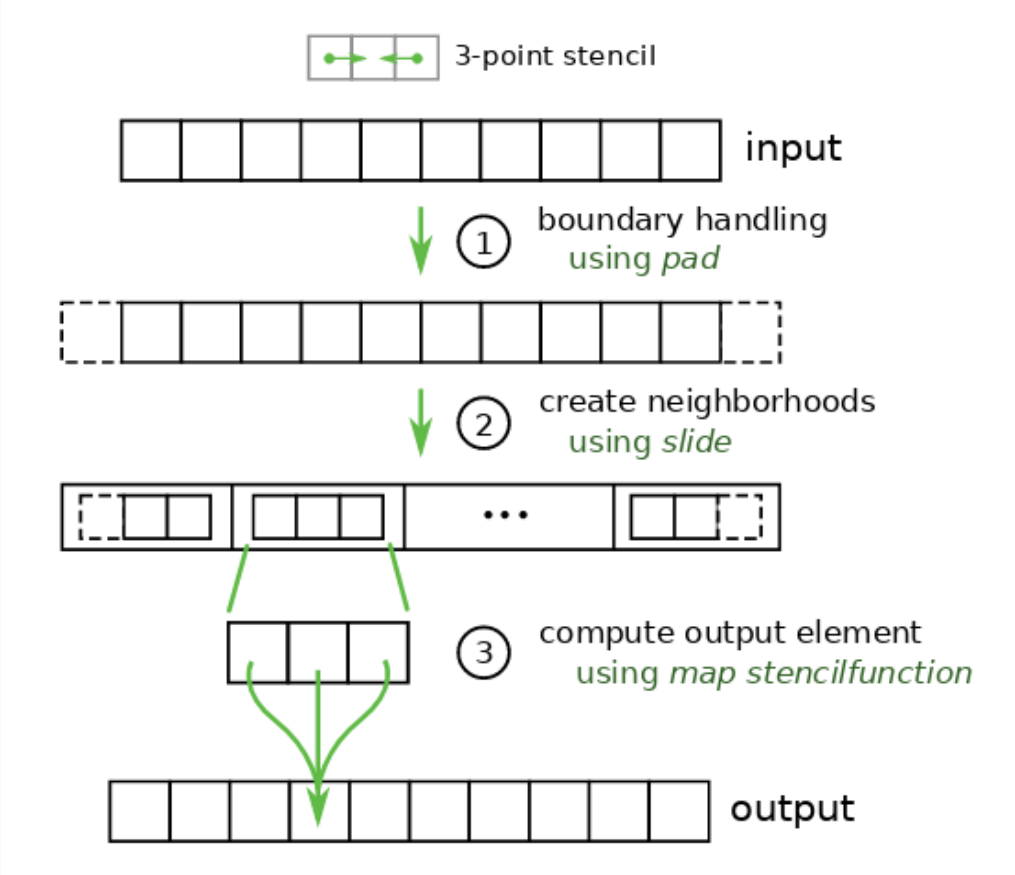


Idea: Encode optimization as semantics-preserving rules

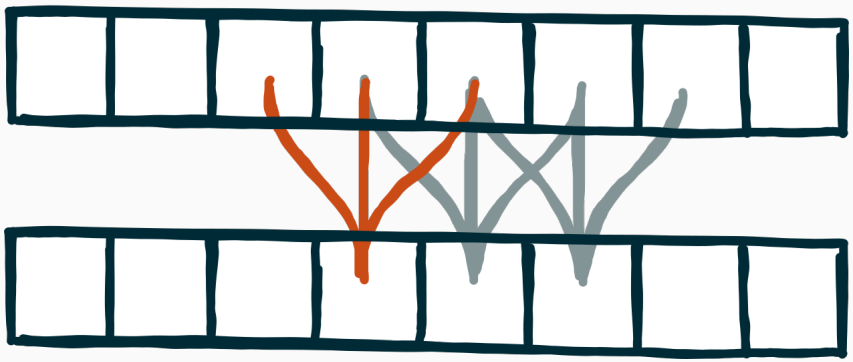
A Concrete Example

A Simple Example

```
val highLevel = fun(
  ArrayType(Float, N), input ⇒
    Map(Reduce(add, 0.0f)) o
      Slide(3,1) o
        Pad(1,1,clamp) $ input )
```



Overlapped Tiling



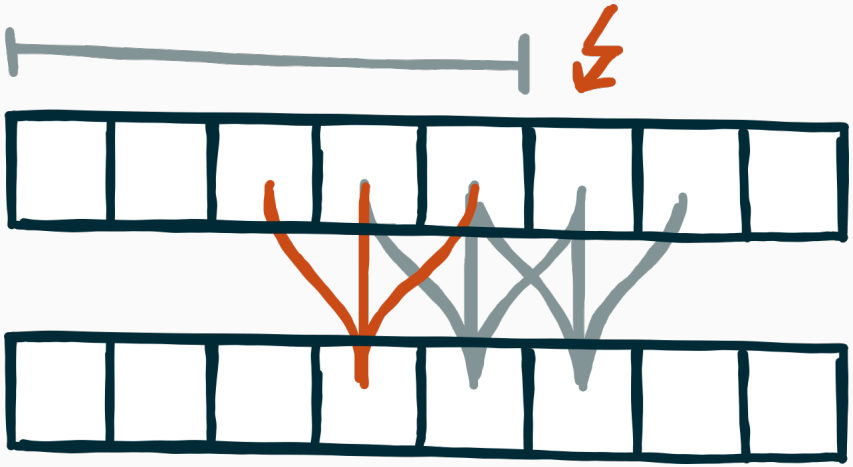
- **Exploit Locality**

  Close neighborhoods share elements that can be grouped in tiles

- **Shared Memory**

  Fast memory can be used to cache tiles

Overlapped Tiling



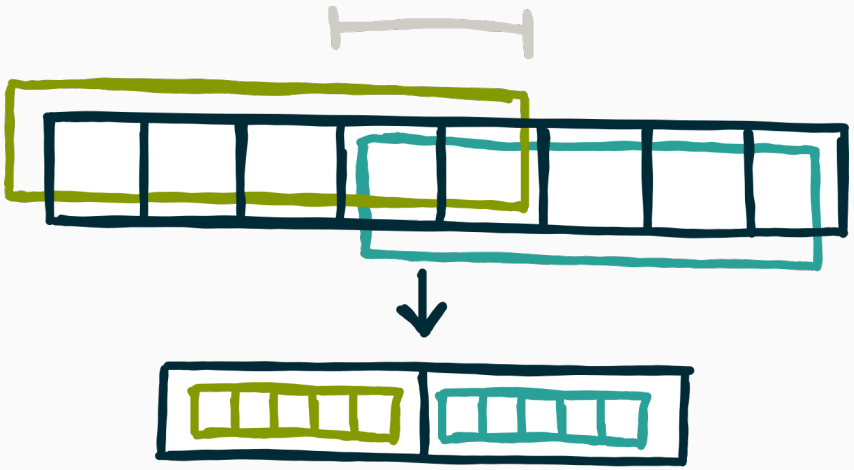
- **Exploit Locality**

  Close neighborhoods share elements that can be grouped in tiles

- **Shared Memory**

  Fast memory can be used to cache tiles

Overlapped Tiling



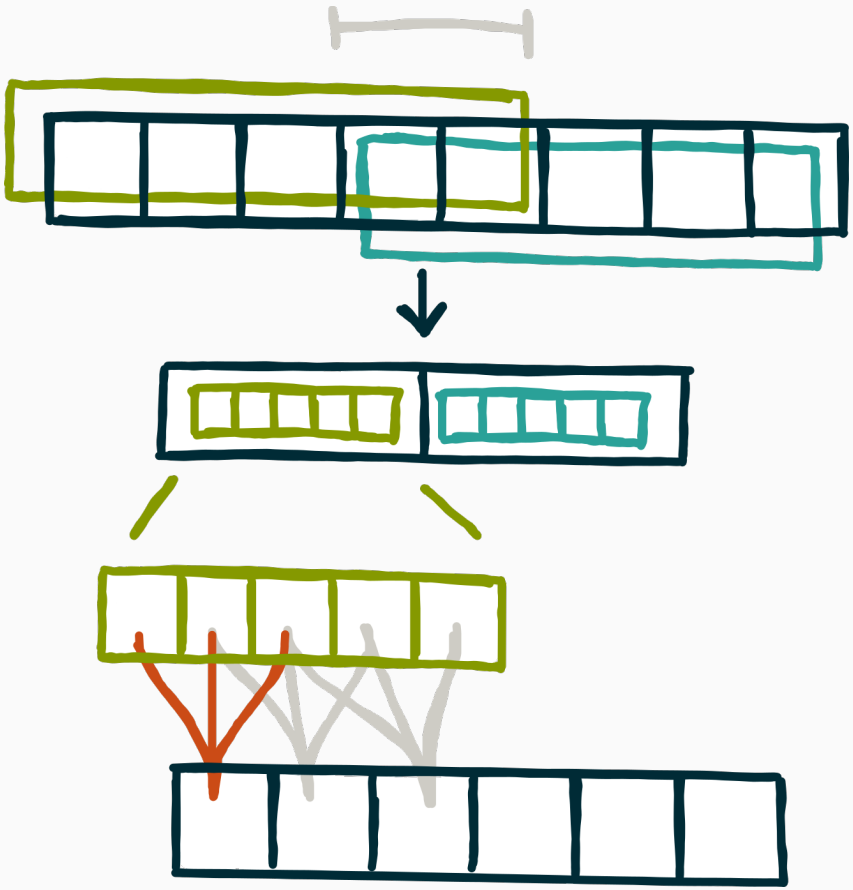
- **Exploit Locality**

  Close neighborhoods share elements that can be grouped in tiles

- **Shared Memory**

  Fast memory can be used to cache tiles

Overlapped Tiling



- **Exploit Locality**

  Close neighborhoods share elements that can be grouped in tiles

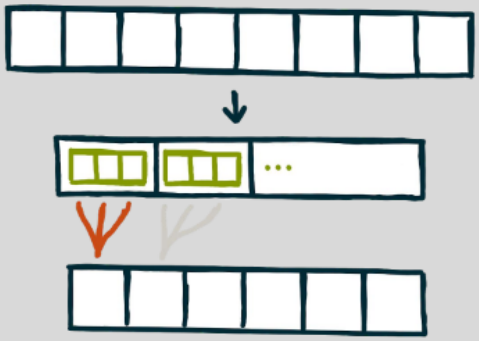
- **Shared Memory**

  Fast memory can be used to cache tiles

Overlapped Tiling Rewrite Rule

overlapped tiling rule

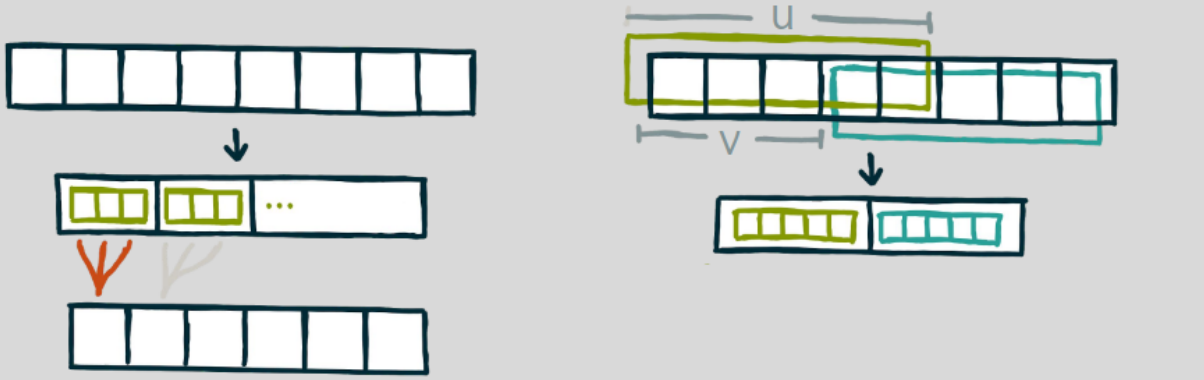
```
map(f, slide(3,1,input))
```



Overlapped Tiling Rewrite Rule

overlapped tiling rule

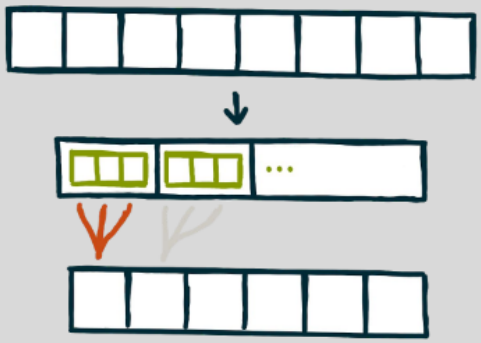
map(f, ***slide***(3,1,input)) ↦ ***slide***(u,v,input)

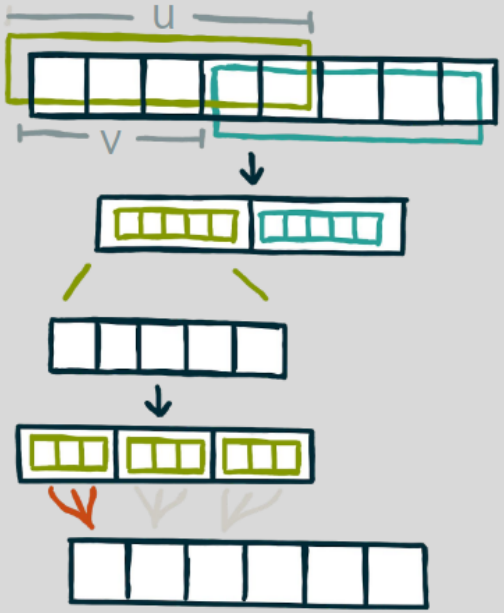


Overlapped Tiling Rewrite Rule

overlapped tiling rule

$$\texttt{map}(f,\ \texttt{slide}(3,1,input)) \mapsto \texttt{join}(\texttt{map}(tile \Rightarrow \texttt{map}(f,\ \texttt{slide}(3,1,tile)),\ \texttt{slide}(u,v,input)))$$





Implementation

Applying Overlapped Tiling

```
// replace this
Map(f) o Slide(n,s)
```

```
// with this
Join() o
 Map(fun(tile =>
  Map(f) o Slide(n,s) $
      tile)) o
   Slide(u,v)
```

Applying Overlapped Tiling

```
// replace this
Map(f) o Slide(n,s)
```

```
val expression1 = fun(
  ArrayType(Float, N), input ⇒
    Map(Reduce(add, 0.0f)) o
     Slide(3, 1) o
      Pad(1, 1, clamp) $ input )
```

```
// with this
Join() o
 Map(fun(tile =>
  Map(f) o Slide(n,s) $
      tile)) o
   Slide(u,v)
```

Applying Overlapped Tiling

```
// replace this
Map(f) o Slide(n,s)
```

```
val expression1 = fun(
  ArrayType(Float, N), input ⇒
    Map(Reduce(add, 0.0f)) o
     Slide(3, 1) o
      Pad(1, 1, clamp) $ input )
```

```
val f = Reduce(add, 0.0f)
val expression2 = fun(
  ArrayType(Float, N), input ⇒
    Map(f) o Slide(3, 1) o
      Pad(1, 1, clamp) $ input )
```

```
// with this
Join() o
 Map(fun(tile =>
  Map(f) o Slide(n,s) $
      tile)) o
   Slide(u,v)
```

Applying Overlapped Tiling

```
// replace this
Map(f) o Slide(n,s)
```

```
// with this
Join() o
 Map(fun(tile =>
  Map(f) o Slide(n,s) $
      tile)) o
   Slide(u,v)
```

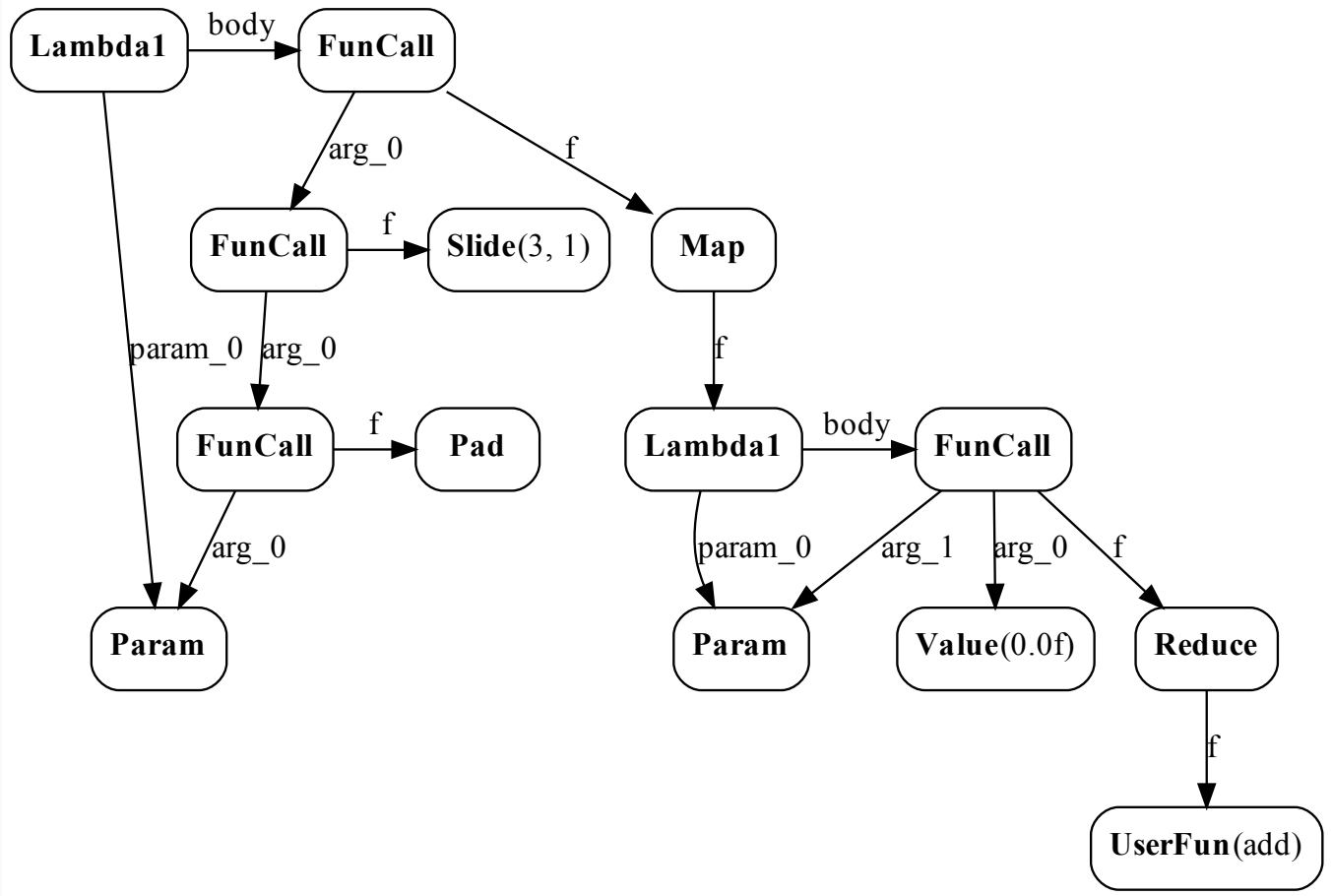
```
val expression1 = fun(
  ArrayType(Float, N), input ⇒
    Map(Reduce(add, 0.0f)) o
     Slide(3, 1) o
      Pad(1, 1, clamp) $ input )
```

```
val f = Reduce(add, 0.0f)
val expression2 = fun(
  ArrayType(Float, N), input ⇒
    Map(f) o Slide(3, 1) o
      Pad(1, 1, clamp) $ input )
```

```
val expression3 = fun(
  ArrayType(Float, N), input ⇒
    Join() o
      Map(fun(tile ⇒
        Map(f) o Slide(3, 1) $ tile)) o
        Slide(u, v) o
          Pad(1, 1, clamp) $ input )
```

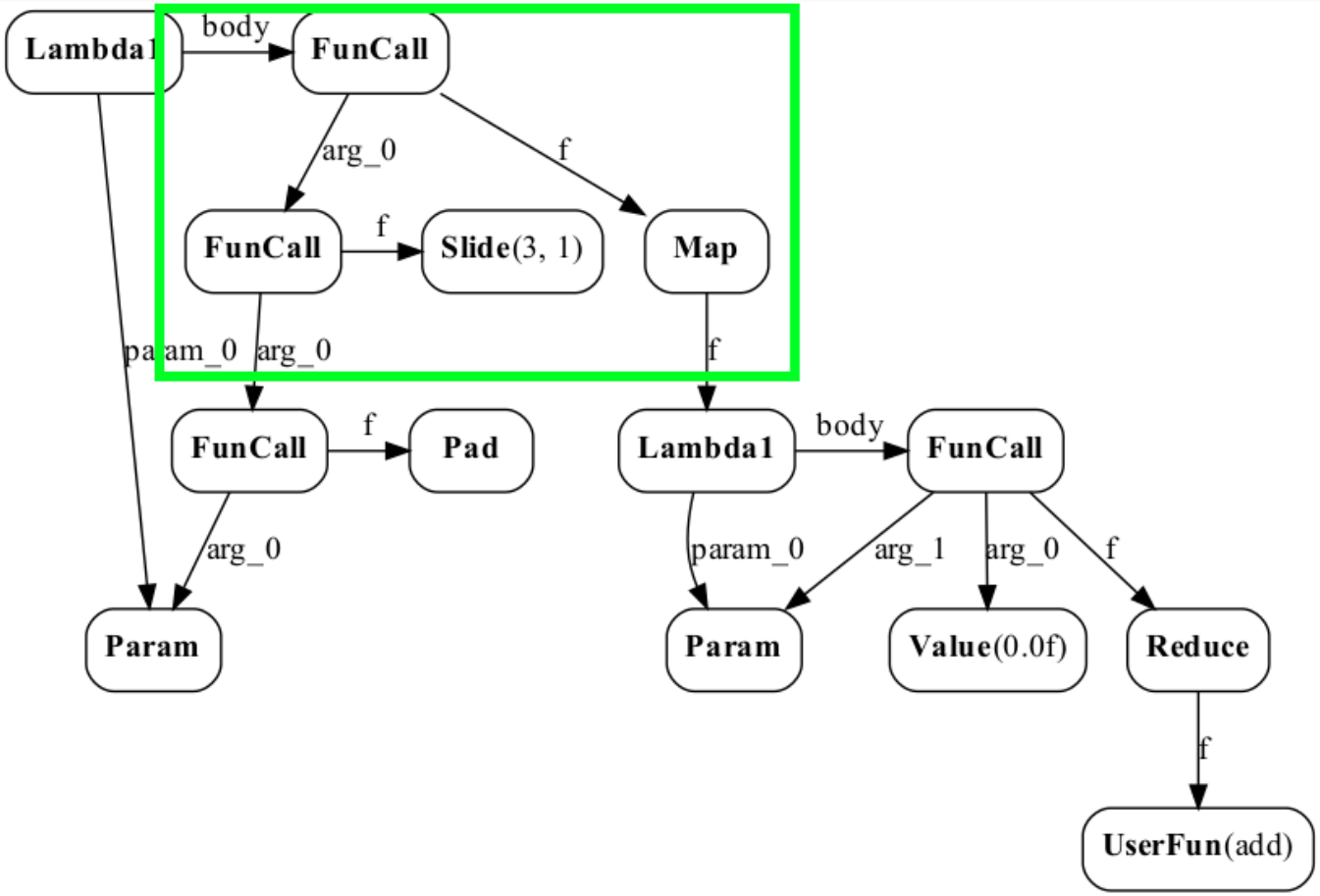
AST

```
val f = Reduce(add, 0.0f)
val expression2 = fun(
  ArrayType(Float, N), input ⇒
    Map(f) o Slide(3, 1) o
      Pad(1, 1, clamp) $ input )
```



AST

```
val f = Reduce(add, 0.0f)
val expression2 = fun(
  ArrayType(Float, N), input ⇒
    Map(f) o Slide(3, 1) o
      Pad(1, 1, clamp) $ input )
```



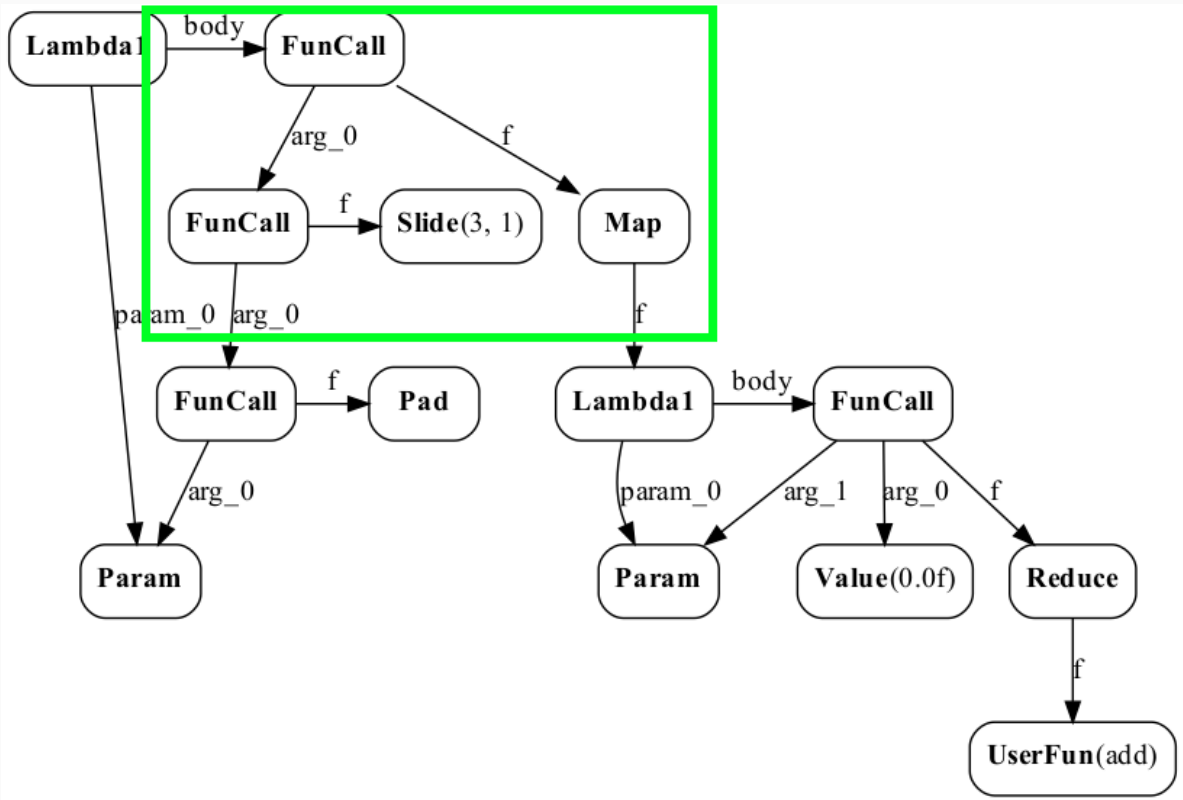
Rewrite Rules in Lift: `Rule` and `Rewrite`

Demo: Show `Rule` and `Rewrite`

Pattern Matching the AST

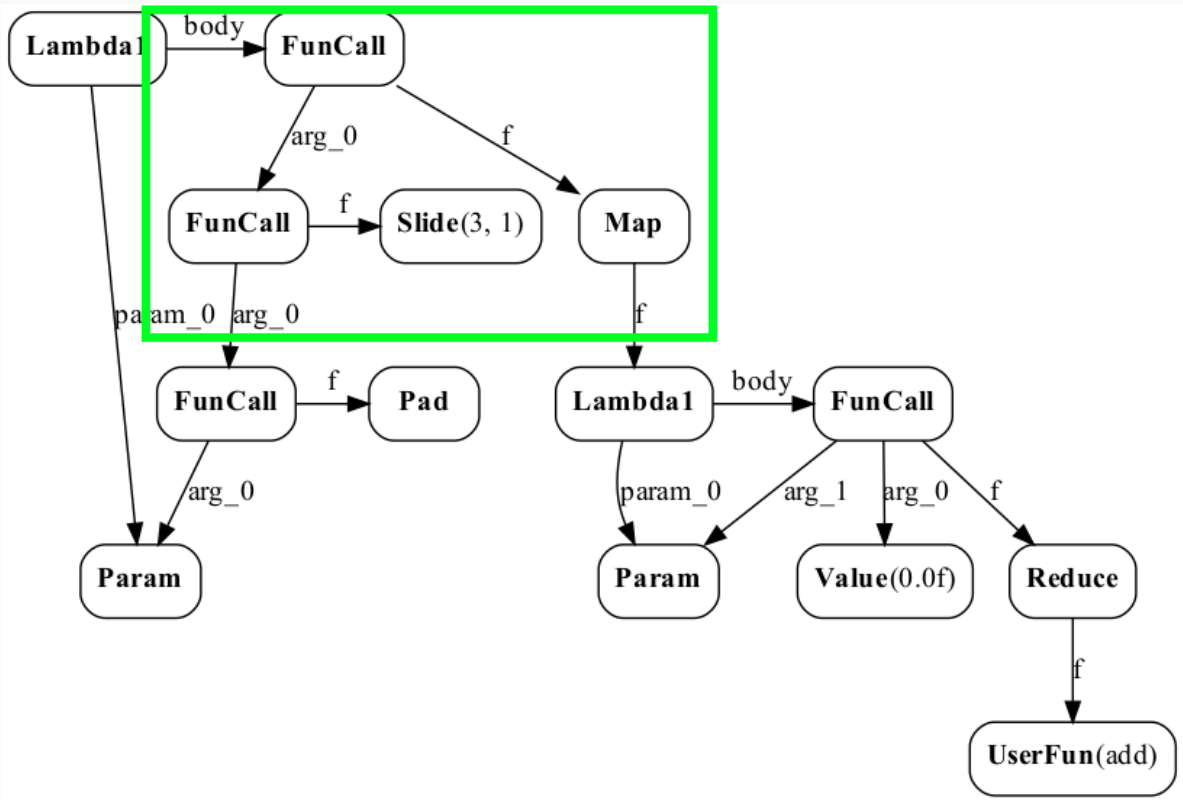
```
val tileStencils2 =
  Rule("Map(f) o Slide(n,s) ⇒ Join() o Map(Map(f) o Slide(n,s)) o Slide(u,v)", {
    case FunCall(Map(f), FunCall(Slide(n,s), arg)) ⇒ {

      Join() o Map(Map(f) o Slide(n, s)) o Slide(u, v) $ arg
    }
  })
```



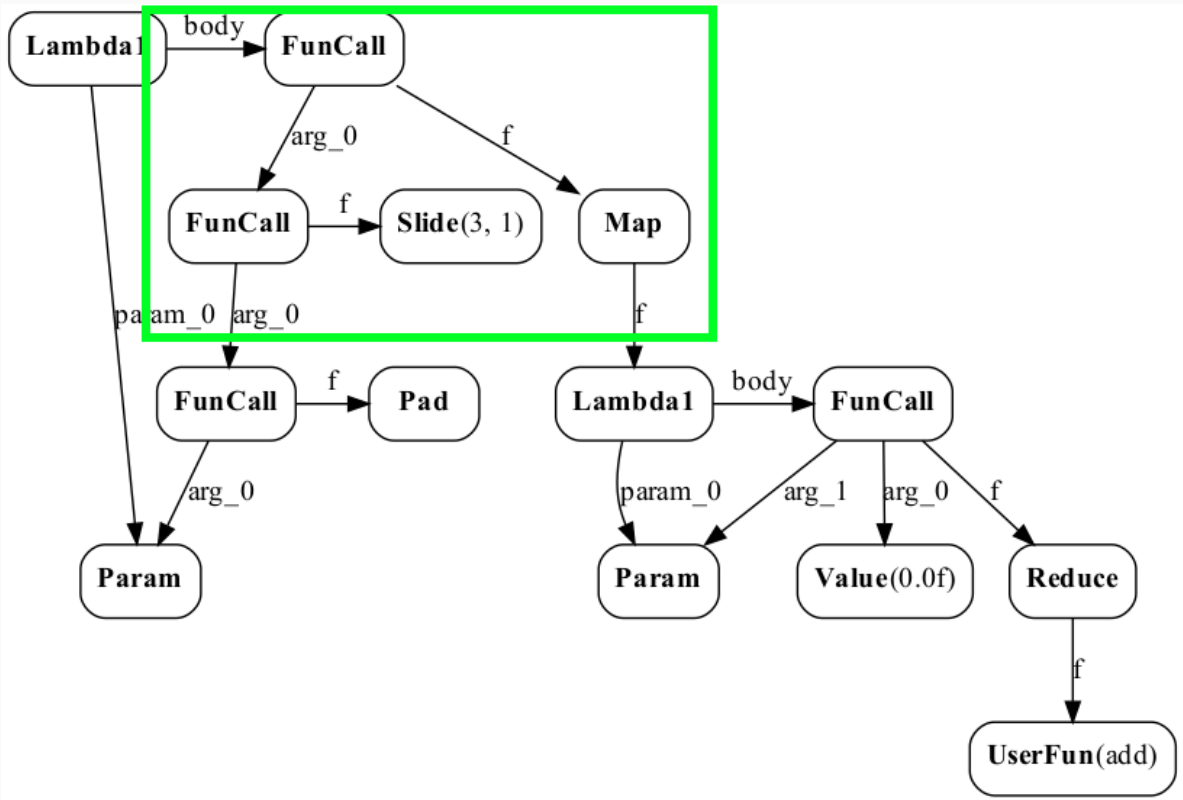
Pattern Matching the AST

```
val tileStencils2 =
  Rule("Map(f) o Slide(n,s) ⇒ Join() o Map(Map(f) o Slide(n,s)) o Slide(u,v)", {
    case FunCall(Map(f), FunCall(Slide(n,s), arg)) ⇒ {
      val u = ?         // tileSize
      val v = u + n-s   // tileStep
      Join() o Map(Map(f) o Slide(n, s)) o Slide(u, v) $ arg
    }
  })
```



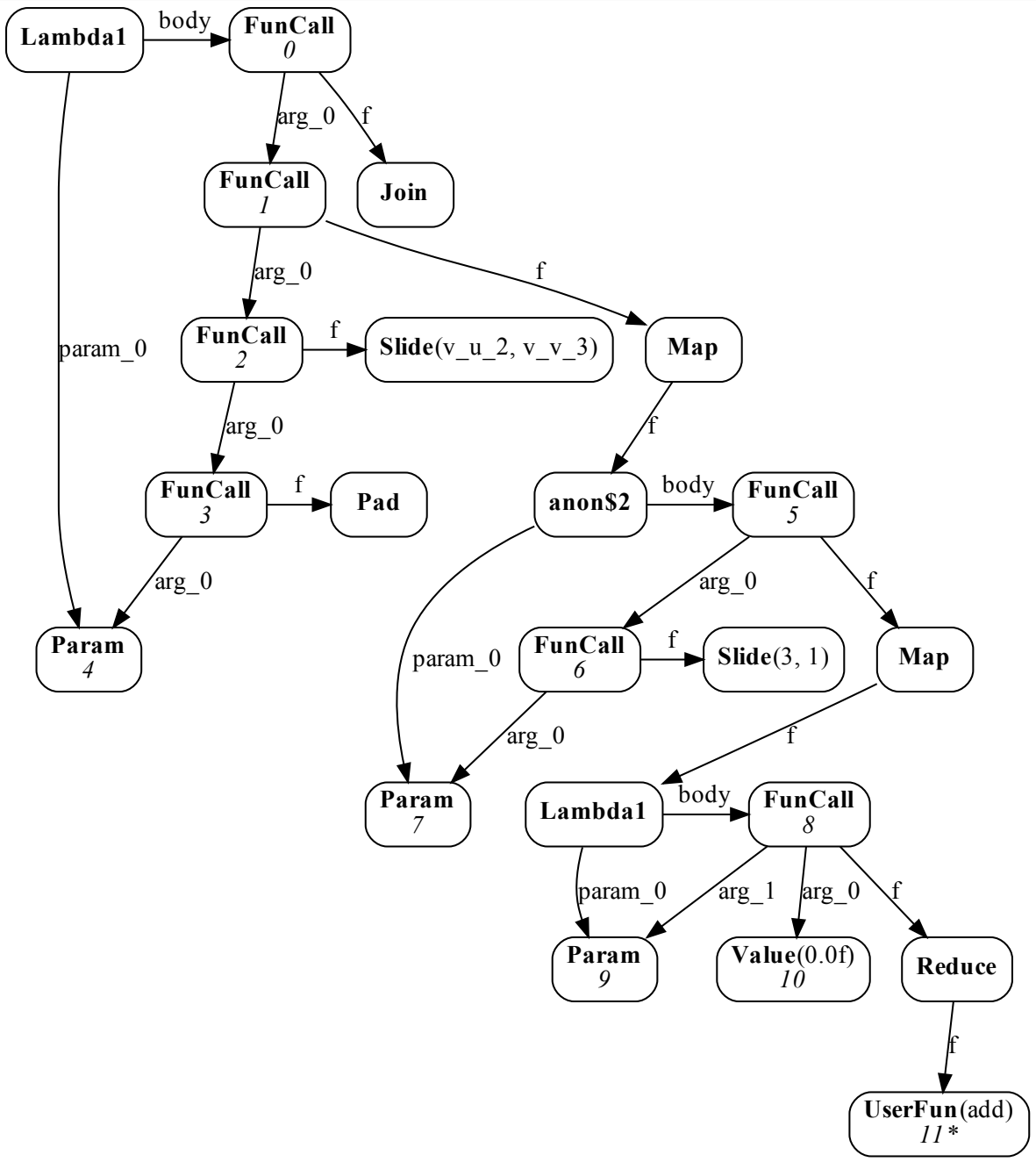
Pattern Matching the AST

```
val tileStencils =
  Rule("Map(f) o Slide(n,s) ⇒ Join() o Map(Map(f) o Slide(n,s)) o Slide(u,v)", {
    case funCall@FunCall(Map(_), slideCall@FunCall(Slide(_,_), _)) ⇒
      val tiled = Rewrite.applyRuleAt(funCall, Rules.slideTiling, slideCall)
      val moved = Rewrite.applyRuleAt(tiled, EnablingRules.movingJoin, tiled)
      val fused = Rewrite.applyRuleAtId(moved, 1, FusionRules.mapFusion)
      fused
  })
```



Pattern Matching the AST

```
val expression3 = fun(
  ArrayType(Float, N), input ⇒
    Join() o
      Map(fun(tile ⇒
        Map(f) o Slide(3, 1) $ tile)) o
        Slide(u, v) o
          Pad(1, 1, clamp) $ input )
```



Rewriting Workflow

```
val expression3 = fun(
  ArrayType(Float, N), input ⇒
    Join() o
      Map(fun(tile ⇒
        Map(f) o Slide(3, 1) $ tile)) o
        Slide(u, v) o
          Pad(1, 1, clamp) $ input )
```

Demo: `OpenCLRules.mapGlb`, `OpenCLRules.mapWrg`

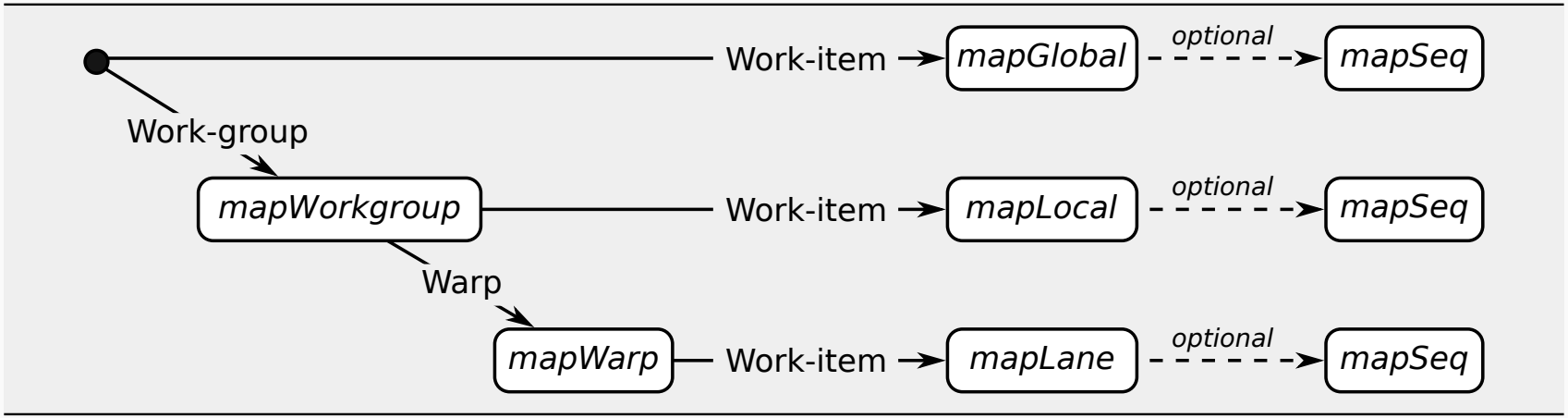
1. **Algorithmic Rewriting**
   E.g., introducing tiles (using overlapped tiling rule)
2. **OpenCL Rewriting**
   Explicitly make use of memory and thread hierarchy
3. **Parameter Tuning**
   Resolve and tune numerical parameters (e.g., ?)

Rewriting Workflow

```
val expression3 = fun(
  ArrayType(Float, N), input ⇒
    Join() o
      Map(fun(tile ⇒
        Map(f) o Slide(3, 1) $ tile)) o
        Slide(u, v) o
          Pad(1, 1, clamp) $ input )
```

Demo: `OpenCLRules.mapGlb`, `OpenCLRules.mapWrg`

1. **Algorithmic Rewriting**
   E.g., introducing tiles (using overlapped tiling rule)
2. **OpenCL Rewriting**
   Explicitly make use of memory and thread hierarchy
3. **Parameter Tuning**
   Resolve and tune numerical parameters (e.g., ?)

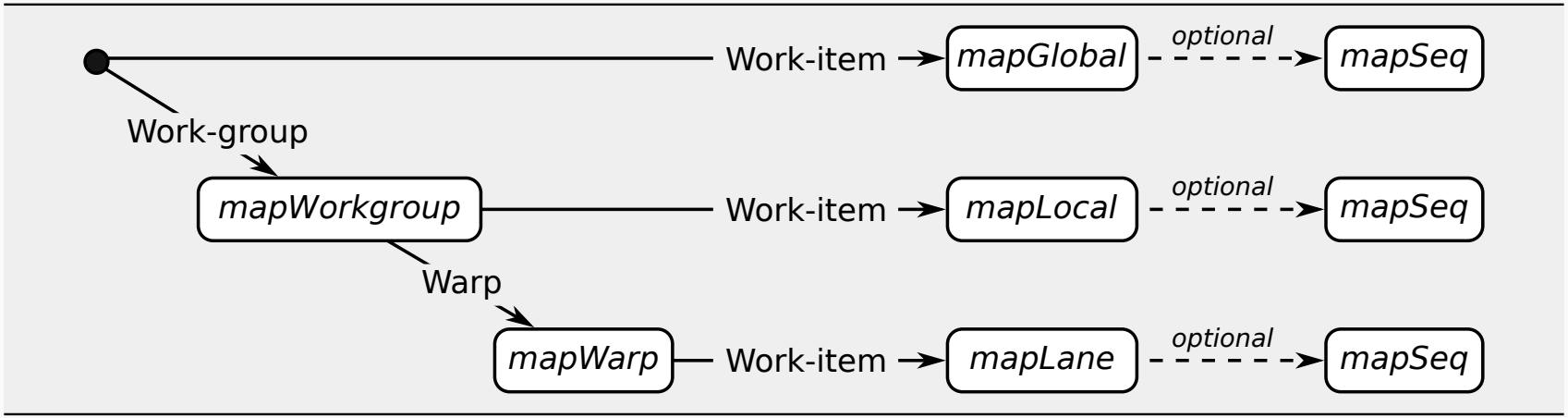


Rewriting Workflow

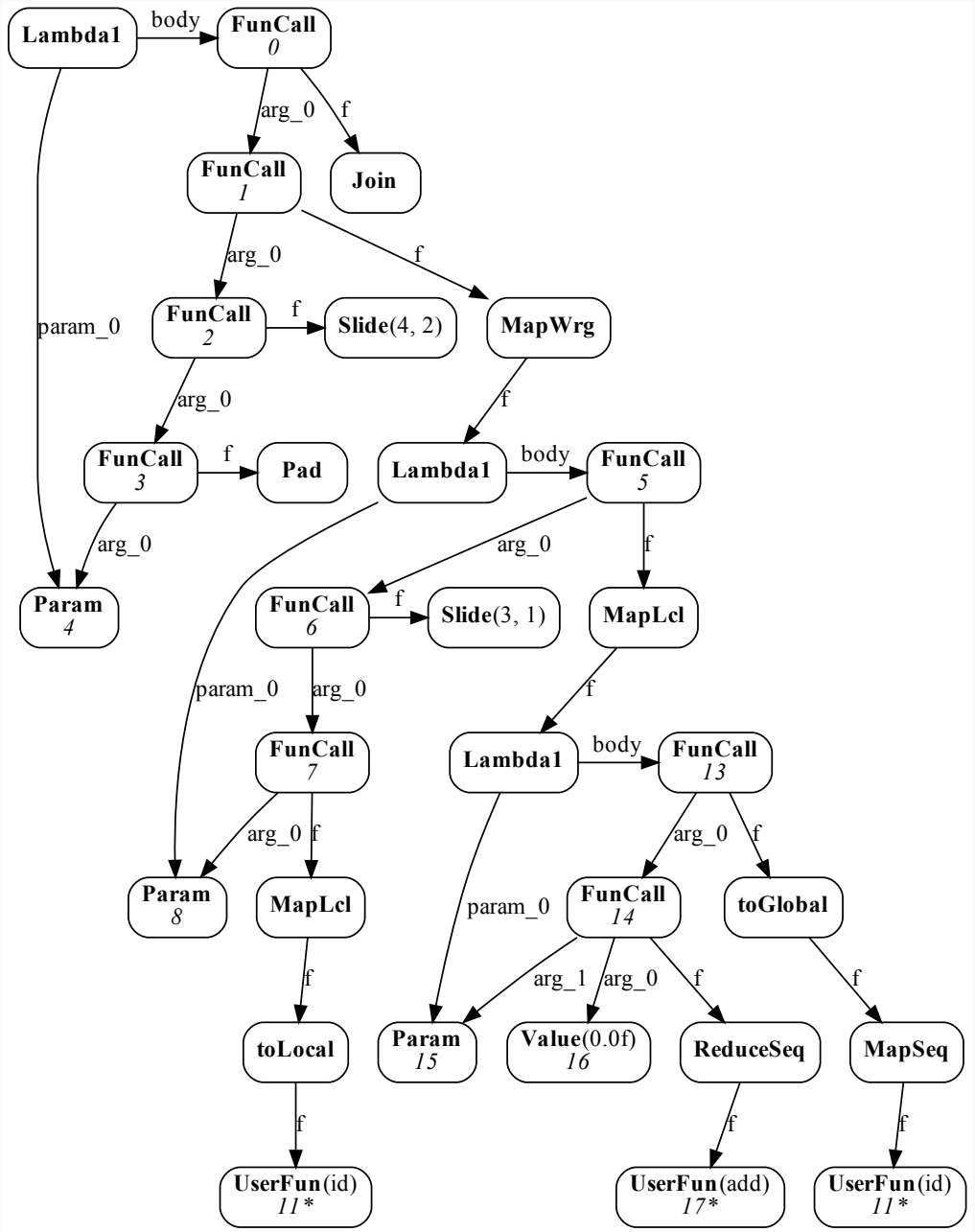
```
val expression4 = fun(
  ArrayType(Float, N), input ⇒
    Join() o
      MapWrg(fun(tile ⇒
        MapLcl(f) o Slide(3, 1) $ tile)) o
        Slide(u, v) o
          Pad(1, 1, clamp) $ input )
```

Demo: `OpenCLRules.mapGlb`, `OpenCLRules.mapWrg`

1. **Algorithmic Rewriting**
   E.g., introducing tiles (using overlapped tiling rule)
2. **OpenCL Rewriting**
   Explicitly make use of memory and thread hierarchy
3. **Parameter Tuning**
   Resolve and tune numerical parameters (e.g., ?)



Lowered Expression



Generated Kernel

```
float id(float x) { return x; }
float add(float x, float y) { return x + y; }
kernel void KERNEL(const global float *restrict IN, global float *OUT, int N) {
    // tile in local memory
    local float TILE[4];
    float acc;

    for (int groupId = get_group_id(0); (groupId < (N / 2));
         groupId = (groupId + get_num_groups(0))) {
        for (int localId = get_local_id(0); (localId < 4);
             localId = (localId + get_local_size(0))) {

            // fill tile in local memory
            TILE[localId] =
                id(IN[(((-1 + localId + (2 * groupId)) >= 0)
                           ? (((-1 + localId + (2 * groupId)) < N) ?
                               (-1 + localId + (2 * groupId)) : (-1 + N)) : 0)]);
        }

        // synchronize threads
        barrier(CLK_LOCAL_MEM_FENCE);
        for (int localId = get_local_id(0); (localId < 2);
            localId = (localId + get_local_size(0))) {

            acc = 0.0f
                // perform stencil computation in each tile
                for (int i = 0; (i < 3); i = (1 + i)) {
                acc = add(acc, TILE[(i + localId)]);
            }

            OUT[(localId + (2 * groupId))] = id(acc);
        }
        barrier(CLK_GLOBAL_MEM_FENCE);
    }
}
```

Exploration

Overview

Automatic application of rewrite rules:

1. **Algorithmic Rewriting**

   `HighLevelRewrite`

2. **OpenCL Rewriting**

   `MemoryMappingRewrite`

3. **Tuning of Numerical Parameters**

   `ParameterRewrite`

Exploration

Demo: Show full exploration demo

1. show exploration config-files and explain heuristics
2. execute `HighLevelRewrite` and examine results
3. execute `MemoryMappingRewrite` and examine results
4. execute `ParameterRewrite` and examine results
5. execute generated kernels using `Harness`
6. show tuning of kernels using OpenTuner