LIFT: A Functional Data-Parallel IR for High-Performance GPU Code Generation

www.lift-project.org

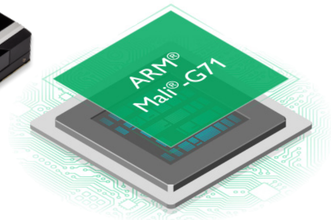
Michel Steuwer · Toomas Remmelg · Christophe Dubach

THE UNIVERSITY *of* EDINBURGH

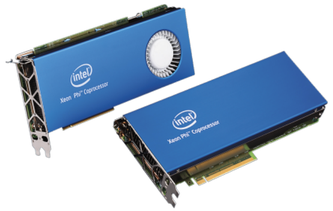
Wouldn’t it be great …

- if we could write parallel software *once* and achieve efficiency and high performance *everywhere?*

CPU



GPU



Accelerator

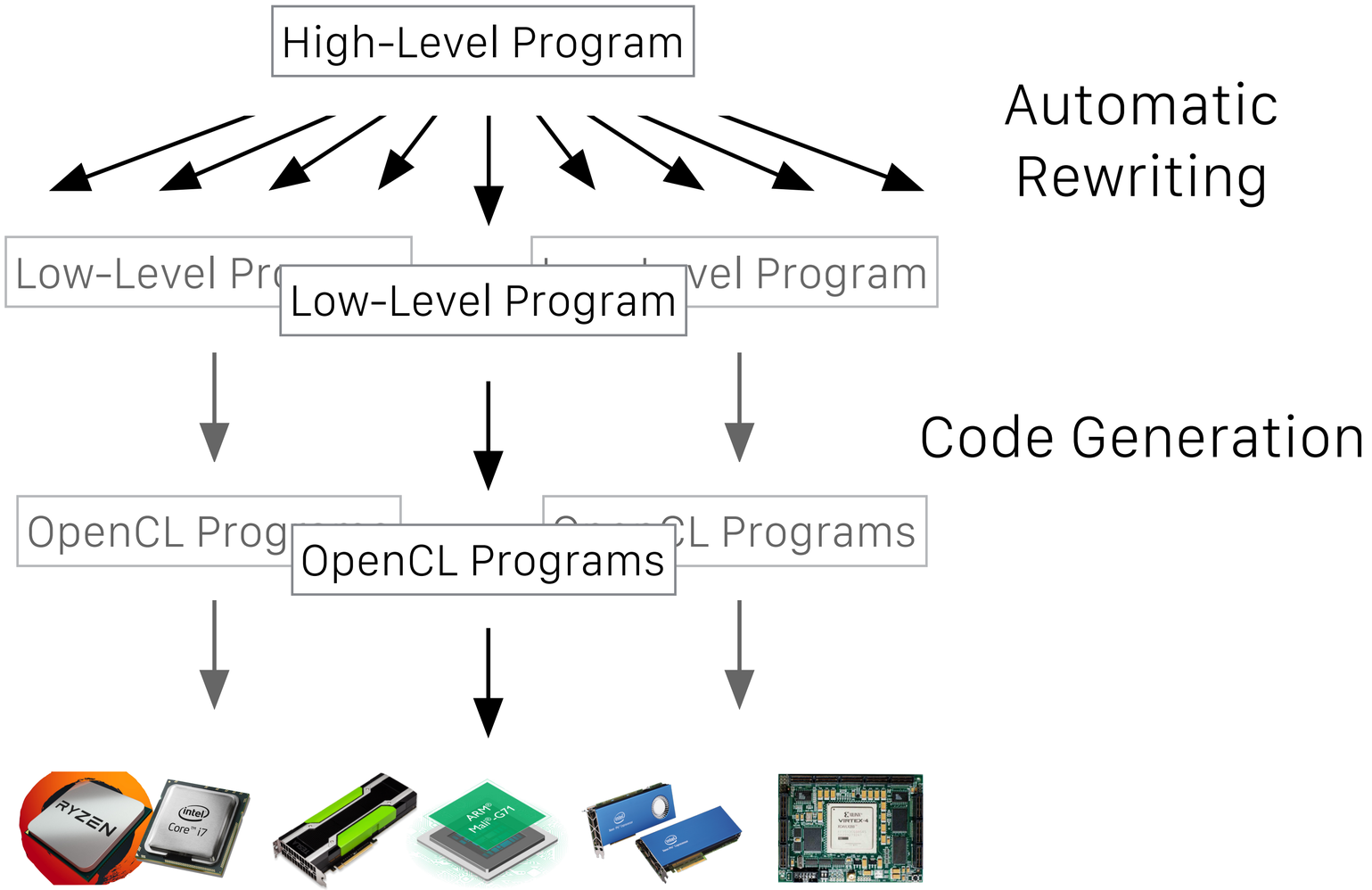
FPGA

- Instead, programs are optimized manually for every device.

Problem:

Existing imperative approaches are not performance portable!

Performance Portability in LIFT



Example Matrix Multiplication

High-Level Program

```
1  map(λ arow .
2   map(λ bcol .
3     reduce(+, 0) ∘ map(×) ∘ zip(arow, bcol)
4   , transpose(B))
5  , A)
```

Apply tiling rules

Automatic Rewriting

```
1  untile ∘ map(λ rowOfTilesA .
2   map(λ colOfTilesB .
3    toGlobal(copy2D) ∘
4    reduce(λ (tileAcc, (tileA, tileB)) .
5     map(map(+)) ∘ zip(tileAcc) ∘
6     map(λ as .
7      map(λ bs .
8       reduce(+, 0) ∘ map(×) ∘ zip(as, bs)
9      , toLocal(copy2D(tileB)))
10    , toLocal(copy2D(tileA)))
11   ,0, zip(rowOfTilesA, colOfTilesB))
12  ) ∘ tile(m, k, transpose(B))
13 ) ∘ tile(n, k, A)
```

Low-Level Program

... ...

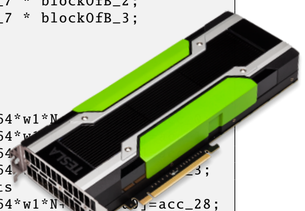
Example Matrix Multiplication

Low-Level Program

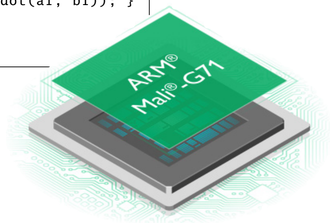
Code Generation

OpenCL Programs

```
kernel mm_amd_opt(global float * A, B, C,
                  int K, M, N) {
local float tileA[512]; tileB[512];

private float acc_0;      ...; acc_31;
private float blockOfB_0; ...; blockOfB_3;
private float blockOfA_0; ...; blockOfA_7;

int lid0 = local_id(0); lid1 = local_id(1);
int wid0 = group_id(0); wid1 = group_id(1);

for (int w1=wid1; w1<M/64; w1+=num_grps(1)) {
 for (int w0=wid0; w0<N/64; w0+=num_grps(0)) {

  acc_0 = 0.0f; ...; acc_31 = 0.0f;
  for (int i=0; i<K/8; i++) {
   vstore4(vload4(lid1*M/4+2*i*M+16*w1+lid0,A)
           ,16*lid1+lid0, tileA);
   vstore4(vload4(lid1*N/4+2*i*N+16*w0+lid0,B)
           ,16*lid1+lid0, tileB);
   barrier(...);

   for (int j = 0; j<8; j++) {
    blockOfA_0 = tileA[0+64*j+lid1*8];
    ... 6 more statements
    blockOfA_7 = tileA[7+64*j+lid1*8];
    blockOfB_0 = tileB[0 +64*j+lid0];
    ... 2 more statements
    blockOfB_3 = tileB[48+64*j+lid0];

    acc_0  += blockOfA_0 * blockOfB_0;
    acc_1  += blockOfA_0 * blockOfB_1;
    acc_2  += blockOfA_0 * blockOfB_2;
    acc_3  += blockOfA_0 * blockOfB_3;
    ... 24 more statements
    acc_28 += blockOfA_7 * blockOfB_0;
    acc_29 += blockOfA_7 * blockOfB_1;
    acc_30 += blockOfA_7 * blockOfB_2;
    acc_31 += blockOfA_7 * blockOfB_3;
   }
   barrier(...);

  }

  C[ 0+8*lid1*N+64*w0+64*w1*N
  C[16+8*lid1*N+64*w0+64*w
  C[32+8*lid1*N+64*w0+64
  C[48+8*lid1*N+64*w0+64
  ... 24 more statements
  C[ 0+8*lid1*N+64*w0+64*w1*N+7*N+lid0]=acc_28;
  C[16+8*lid1*N+64*w0+64*w1*N+7*N+lid0]=acc_29;
  C[32+8*lid1*N+64*w0+64*w1*N+7*N+lid0]=acc_30;
  C[48+8*lid1*N+64*w0+64*w1*N+7*N+lid0]=acc_31;
} } }
```



```
kernel void mm(global float4* const A,
               global float4* const B,
               global float2* C, uint n) {
  uint i = get_global_id(0);
  uint j = get_global_id(1);
  uint nv4 = n >> 2;
  float4 ab = (float4)(0.0f);
  for (uint k = 0; k < nv4; ++k) {
    float4 a0 = A[ 2*i   *nv4+k];
    float4 a1 = A[(2*i+1)*nv4+k];
    float4 b0 = B[ 2*j   *nv4+k];
    float4 b1 = B[(2*j+1)*nv4+k];
    ab += (float4)(dot(a0, b0), dot(a0, b1),
                   dot(a1, b0), dot(a1, b1)); }
  uint ix = 2*i*(n>>1) + j;
  C[ix]          = ab.s01;
  C[ix + (n>>1)] = ab.s23; }
```



LIFT Project Overview

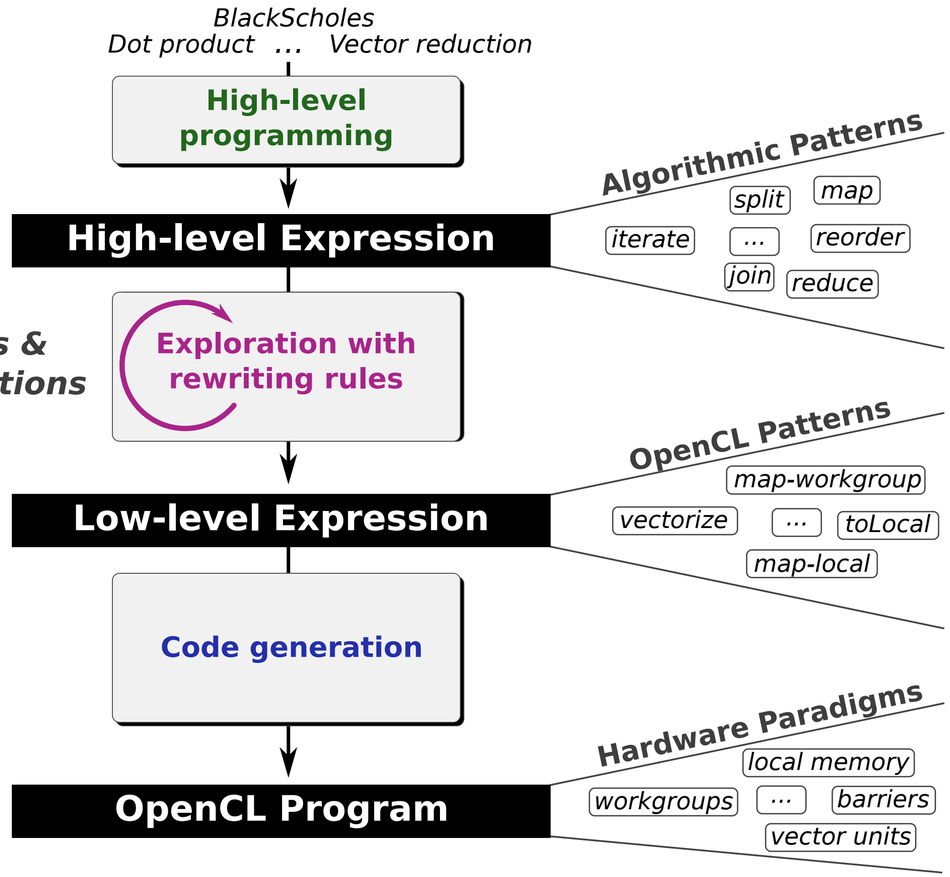
[ICFP 2015]

[GPGPU 2016]

[CASES 2016]

Algorithmic choices & Hardware optimizations

[CGO 2017]



www.lift-project.org

The LIFT Intermediate Language

Algorithmic Patterns

$\mathbf{mapSeq}(f, [x_n, \cdots, x_2, x_1]) = [f(x_1), f(x_2), \cdots, f(x_n)]$

$\mathbf{reduceSeq}(z, f, [x_n, \cdots, x_2, x_1]) = f(\cdots(f(f(z, x_1), x_2)\cdots), x_n)$

$\mathbf{id}([x_n, \cdots, x_2, x_1]) = [x_n, \cdots, x_2, x_1]$

$\mathbf{iterate}^m(f, [x_n, \cdots, x_2, x_1]) = \underbrace{f(\cdots(f(}_{m \text{ times}}[x_n, \cdots, x_2, x_1])))$

Parallel Patterns

$\mathbf{mapGlb}^{\{0,1,2\}}$ $\mathbf{mapWrg}^{\{0,1,2\}}$ $\mathbf{mapLcl}^{\{0,1,2\}}$

Address Space Patterns

toGlobal **toLocal** **toPrivate**

Vectorize Patterns

$\mathbf{asVector}([x_1, x_2, \cdots, x_n]) = \overrightarrow{x_1, x_2, \ldots, x_n}$, x_i is scalar

$\mathbf{asScalar}(\overrightarrow{x_1, x_2, \ldots, x_n}) = [x_1, x_2, \cdots, x_n]$

$\mathbf{mapVec}(f, \overrightarrow{x_1, x_2, \ldots, x_n}) = \overrightarrow{f(x_1), f(x_2), \ldots, f(x_n)}$

Data Layout Patterns

$\mathbf{split}^m([x_1, x_2, \cdots, x_n]) = [[x_1, x_2, \cdots], \cdots, [\cdots, x_n]]$ (each inner array has length m)

$\mathbf{join}([[x_1, x_2, \cdots], \cdots, [\cdots, x_n]]) = [x_1, x_2, \cdots, x_n]$

$\mathbf{zip}([x_1, x_2, \cdots, x_n], [y_1, y_2, \cdots, y_n]) = [(x_1,y_1), (x_2,y_2), \cdots, (x_n,y_n)]$

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

$\mathbf{gather}(f, [x_{f(1)}, x_{f(2)}, \cdots, x_{f(n)}]) = [x_1, x_2, \cdots, x_n]$

$\mathbf{scatter}(f, [x_1, x_2, \cdots, x_n]) = [x_{f(1)}, x_{f(2)}, \cdots, x_{f(n)}]$

$\mathbf{slide}(size, step, [x_1, x_2, \cdots, x_n])$ — windows of $size$ starting every $step$ elements

Dot Product in the LIFT IL

```
partialDot(x: [float]N, y: [float]N) = {

  join(mapWrg0(λ → t1
    join(toGlobal(mapLcl0(mapSeq(id)))(split1(
    iterate6(λ → t2
      join(mapLcl0(toLocal(mapSeq(id)),
                   reduceSeq(add, 0, split2(t2)))),
      join(mapLcl0(toLocal(mapSeq(id)),
                   reduceSeq(multAndSumUp, 0, split2(t1)))))))))
  , split128(zip(x, y))))

}
```

Step 3

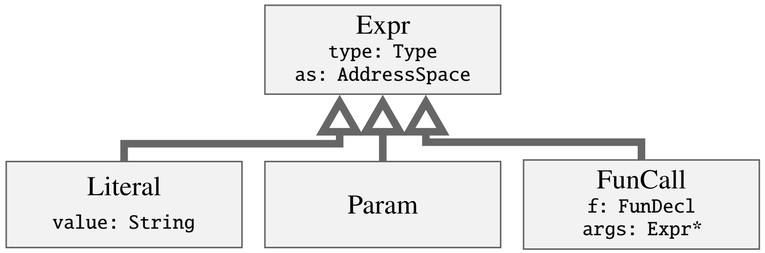
Step 2

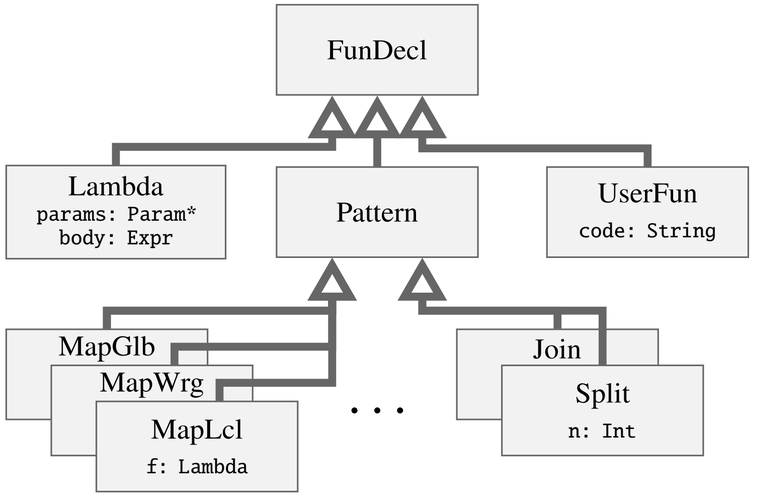
Step 1

The LIFT Intermediate Representation

Step 1

```
λ x →
 join(
  mapLcl⁰(
   toLocal(mapSeq(id)),
   reduceSeq(multAndSumUp, 0, split²(x))))
```

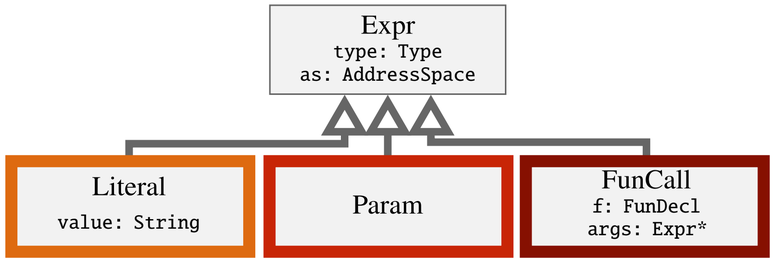


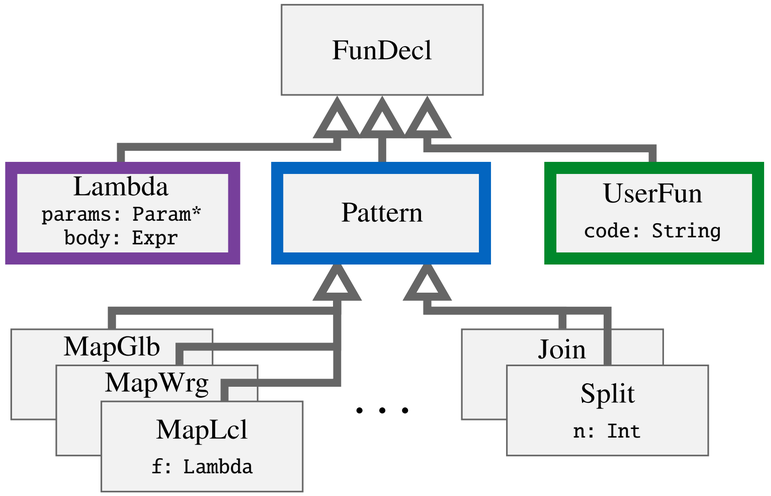


The LIFT Intermediate Representation

Step 1

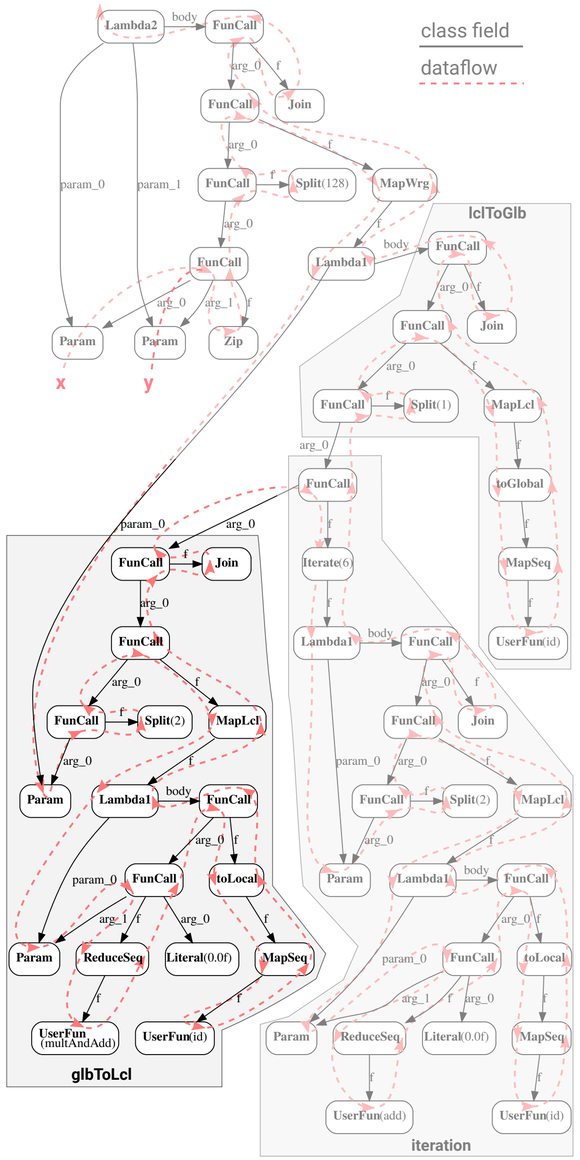
```
λ x0 →
 join(λ x6 →
  mapLcl⁰(λ x5 →
   toLocal(λ x4 → mapSeq(λ x3 → id(x3), x4), x5),
   reduceSeq(λ x1,x2 → multAndSumUp(x1,x2), 0, split²(x0))), x6)
```





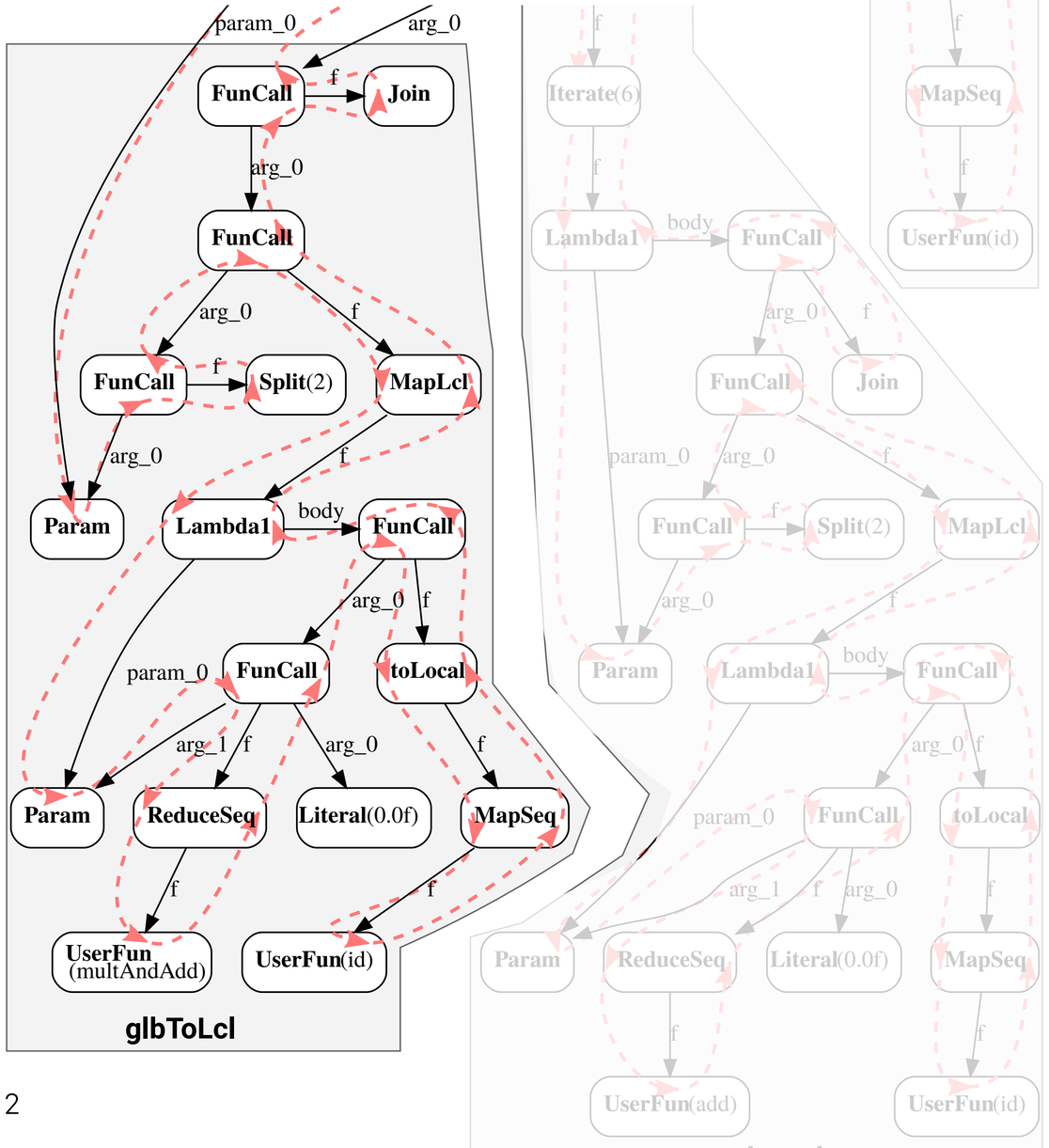
Dot Product in the Lift IR

```
λ x0 →
 join(λ x6 →
  mapLcl⁰(λ x5 →
   toLocal(λ x4 →
     mapSeq(λ x3 →
       id(x3), x4), x5),
   reduceSeq(λ x1,x2 →
     multAndSumUp(x1,x2),
     0,
     split²(x0))), x6)
```

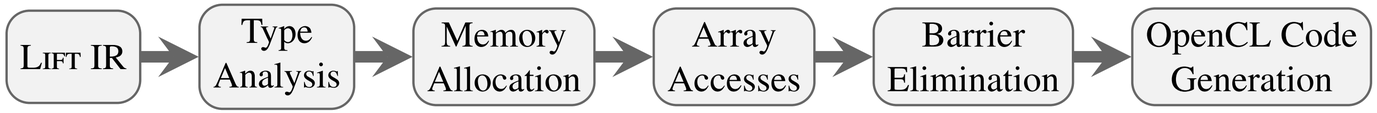


Dot Product in the LIFT IR

```
λ x0 →
 join(λ x6 →
  mapLcl⁰(λ x5 →
   toLocal(λ x4 →
     mapSeq(λ x3 →
       id(x3), x4), x5),
   reduceSeq(λ x1,x2 →
     multAndSumUp(x1,x2),
     0,
     split²(x0))), x6)
```



Compilation of LIFT IR to OpenCL



- Type Analysis:
  Inference of datatypes including shapes and length of multi-dimensional arrays
- Memory Allocation:
  Inference of address space and memory allocation for non data layout patterns
- **Array Accesses:**
  Generation of explicit, flat OpenCL array accesses from LIFT patterns
  Simplification of generated array indices
- Barrier Elimination:
  Identifying and removing of superfluous memory barriers
- OpenCL Code Generation:
  Emitting matching OpenCL code for each pattern;
  Cheapest control flow is chosen based on type information

Multi-Dimensional Array Accesses

```
mapWrg⁰(
 λ z →join(mapLcl⁰(
   toLocal(mapSeq(id)),
   reduceSeq(λ a,xy → a+(xy₀*xy₁) , 0, split²(z)))),
 split¹²⁸(zip(x, y)) )
```

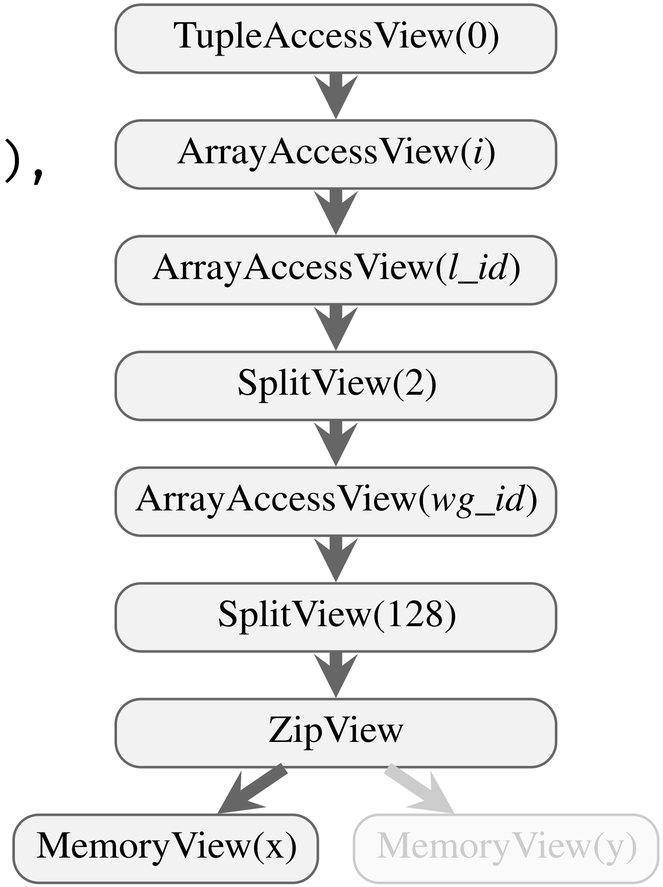
↓ ?

```
…
for (…) {
  a = a +
          x[(2 * l_id) + (128 * wg_id) + i]
        *
          y[(2 * l_id) + (128 * wg_id) + i];
}
…
```

View Construction

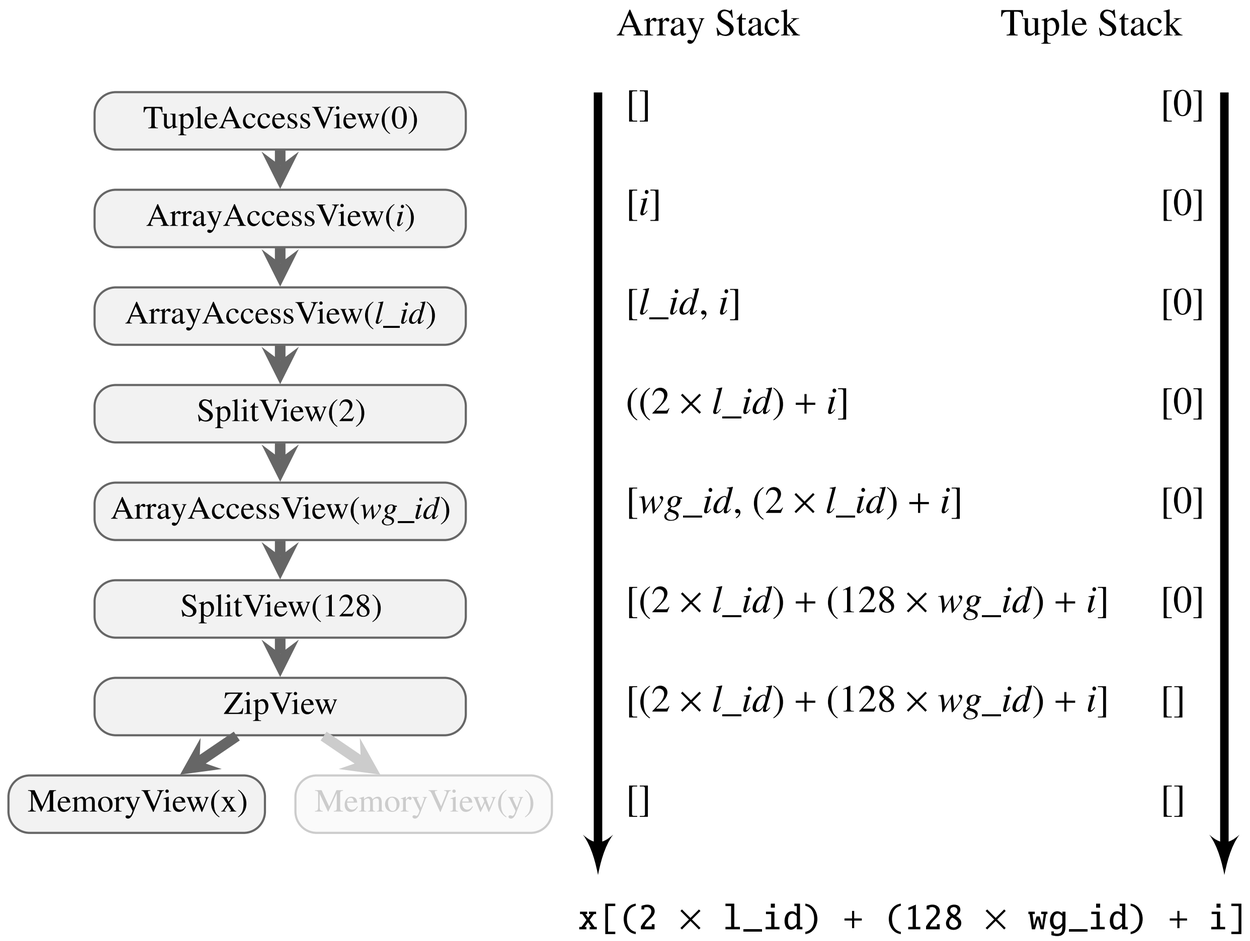
```
mapWrg⁰(
 λ z →join(mapLcl⁰(
   toLocal(mapSeq(id)),
   reduceSeq(λ a,xy → a+(xy₀*xy₁) , 0, split²(z)))),
 split¹²⁸(zip(x, y)) )
```

- Data patterns are used to construct a compiler internal data structure: *View*
- Every data pattern has a corresponding view recording how to access memory
- Views are constructed by traversing the AST



View Consumption

- When consuming the *Views* two stacks are maintained
- The *Array Stack* keeps track which element to access in an array
- The *Tuple Stack* keeps track which array to access



Simplifying Array Accesses

- Straightforward generation of arrays accesses leads to long (really long) array indices

```
1 ((((wg_id×M+l_id)/M)+(((wg_id×M+l_id) mod M)×N))/N)×N+(((wg_id×M+l_id)/M)+(((wg_id×M+l_id) mod M)×N)) mod N
2 ((  wg_id           +           l_id       ×N) /N)×N+(  wg_id           +           l_id       ×N)  mod N
3                                 l_id              ×N+   wg_id
```

- Set of arithmetic rules are used for simplification

$$x/y = 0, \quad \text{if } x < y \text{ and } y \neq 0$$

$$(x \times y + z)/y = x + z/y, \quad \text{if } y \neq 0$$

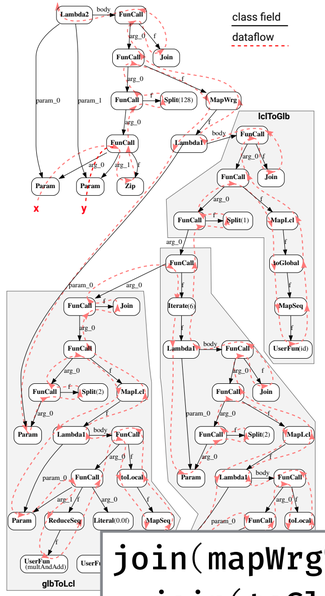
$$x \bmod y = x, \quad \text{if } x < y \text{ and } y \neq 0$$

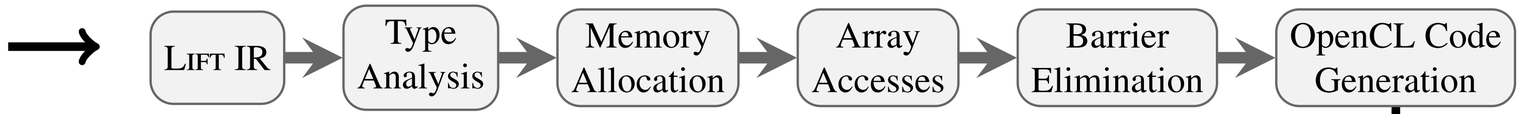
$$(x/y) \times y + x \bmod y = x, \quad \text{if } y \neq 0$$

$$(x \times y) \bmod y = 0, \quad \text{if } y \neq 0$$

$$(x + y) \bmod z = (x \bmod z + y \bmod z) \bmod z, \quad \text{if } z \neq 0$$

Compilation Flow of Dot Product

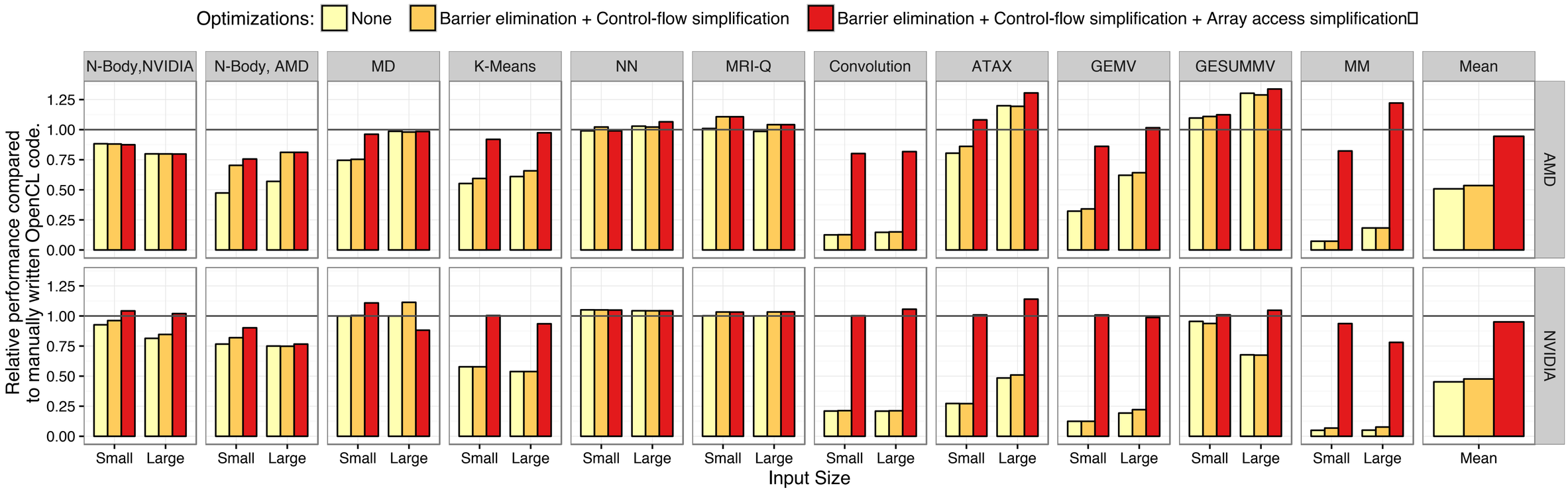





```
join(mapWrg⁰(λ → t1
  join(toGlobal(mapLcl0(mapSeq(id)))(split¹(
  iterate⁶(λ → t2
    join(mapLcl⁰(toLocal(mapSeq(id)),
              reduceSeq(add, 0, split²(t2)))),
  join(mapLcl⁰(toLocal(mapSeq(id)),
            reduceSeq(multAndSumUp, 0, split²(t1)))))))))
, split¹²⁸(zip(x, y))))
```

```
1  kernel void KERNEL(const global float *restrict x,
2                     const global float *restrict y,
3                     global float *z, int N) {
4    local float tmp1[64];  local float tmp2[64];
5    local float tmp3[32];
6    float acc1; float acc2;
7    for (int wg_id  = get_group_id(0); wg_id < N/128;
8              wg_id += get_num_groups(0)) {
9      { int l_id = get_local_id(0);
10       acc1 = 0.0f;
11       for (int i = 0; i < 2; i += 1) {
12         acc1 = multAndSumUp(acc1,
13                      x[2 * l_id + 128 * wg_id + i],
14                      y[2 * l_id + 128 * wg_id + i]); }
15       tmp1[l_id] = id(acc1); }
16     barrier(CLK_LOCAL_MEM_FENCE);
17     int size = 64;
18     local float *in = tmp1; local float *out = tmp2;
19     for (int iter = 0; iter < 6; iter += 1) {
20       if (get_local_id(0) < size / 2) {
21         acc2 = 0.0f;
22         for (int i = 0; i < 2; i += 1) {
23           acc2 = add(acc2, in[2 * l_id + i]); }
24         out[l_id] = id(acc2); }
25       barrier(CLK_LOCAL_MEM_FENCE);
26       size = size / 2;
27       in   = (out == tmp1) ? tmp1 : tmp3;
28       out  = (out == tmp1) ? tmp3 : tmp1;
29       barrier(CLK_LOCAL_MEM_FENCE); }
30     if (get_local_id(0) < 1) {
31       z[wg_id] = id(tmp3[l_id]); }
32     barrier(CLK_GLOBAL_MEM_FENCE); } }
```

Experimental Evaluation



Performance of LIFT generated Code on par with OpenCL code

Optimizations crucial for achieving high performance

The LIFT Team

Christophe Dubach
Lecturer
University of Edinburgh

Michel Steuwer
Postdoc
University of Edinburgh

Toomas Remmelg
PhD Student
University of Edinburgh

Adam Harries
PhD Student
University of Edinburgh

Bastian Hagedorn
PhD Student
University of Münster

Larisa Stoltzfus
PhD Student
University of Edinburgh

Federico Pizzuti
PhD Student
University of Edinburgh

Naums Mogers
PhD Student
University of Edinburgh

LIFT is Open-Source Software

Papers and more infos at: lift-project.org

CGO Artifact at: gitlab.com/michel-steuwer/cgo_2017_artifact

Source code at: github.com/lift-project/lift