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

www.lift-project.org

Michel Steuwer · Toomas Remmelg · Christophe Dubach
Wouldn’t it be great ...

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

- Instead, programs are optimized manually for every device.

**Problem:**
Existing imperative approaches are not performance portable!
Performance Portability in LIFT

High-Level Program

Automatic
Rewriting

Low-Level Program

Code Generation

OpenCL Programs
Example Matrix Multiplication

High-Level Program

Automatic Rewriting

Low-Level Program

1. `map(λ arow .
   map(λ bcol .
       reduce(+, 0) ◦ map(×) ◦ zip(arow, bcol)
         , transpose(B))
       , A)`

Apply tiling rules

1. `until ◦ map(λ rowOfTilesA .
   map(λ colOfTilesB .
       toGlobal(copy2D) ◦
       reduce(λ (tileAcc, (tileA, tileB)) .
           map(map(+) ◦ zip(tileAcc) ◦
               map(λ as .
                   map(λ bs .
                       reduce(+, 0) ◦ map(×) ◦ zip(as, bs)
                         , toLocal(copy2D(tileB)))
                       , toLocal(copy2D(tileA))))
                   , 0, zip(rowOfTilesA, colOfTilesB))
         ) ◦ tile(m, k, transpose(B))
       ) ◦ tile(n, k, A)`

...  ...  ...
Example Matrix Multiplication

Low-Level Program

Code Generation

OpenCL Programs
LIFT Project Overview

Figure 5.3: Overview of our code generation approach. Problems expressed with high-level algorithmic patterns are systematically transformed into low-level OpenCL patterns using a rule rewriting system. OpenCL code is generated by mapping the low-level patterns directly to the OpenCL programming model representing hardware paradigms.

We argue that the root of the problem lies in a gap in the system stack between the high-level algorithmic patterns on the one hand and low-level hardware optimizations on the other hand. We propose to bridge this gap using a novel pattern-based code generation technique. A set of rewrite rules systematically translates high-level algorithmic patterns into low-level hardware patterns. The rewrite rules express different algorithmic and optimization choices. By systematically applying the rewrite rules semantically equivalent, low-level expressions are derived from high-level algorithmic expressions written by the application developer. Once derived, high-performance code based on these expressions can be automatically generated. The next section introduces an overview of our approach.

ICFP 2015
GPGPU 2016
CASES 2016
CGO 2017

www.lift-project.org
The LIFT Intermediate Language

Algorithmic Patterns

\[
\begin{align*}
\text{mapSeq}(f, x_0 \ldots x_n) &= f(x_0) f(x_1) \ldots f(x_n) \\
\text{reduceSeq}(z, f, x_0 \ldots x_n) &= z \cdot (f(z \cdot x_1), x_2 \ldots, x_n) \\
\text{id}(x_0 \ldots x_n) &= x_0 \ldots x_n \\
\text{iterate}^m(f, x_0 \ldots x_n) &= \frac{f(\cdots (f(x_0 \ldots x_2), x_1) \ldots)}{m \text{ times}} \\
\end{align*}
\]

Parallel Patterns

\[
\begin{align*}
\text{mapGb}^{[0,1,2]} \quad \text{mapWr}^{[0,1,2]} \quad \text{mapLc}^{[0,1,2]} \\
\end{align*}
\]

Address Space Patterns

\text{toGlobal} \quad \text{toLocal} \quad \text{toPrivate}

Vectorize Patterns

\[
\begin{align*}
\text{asVector}(x_1 x_2 \ldots x_n) &= x_1, x_2, \ldots, x_n, \text{ } x_j \text{ is scalar} \\
\text{asScalar}(x_1 x_2 \ldots x_n) &= x_1 x_2 \ldots x_n \\
\text{mapVec}(f, x_1 x_2 \ldots x_n) &= f(x_1), f(x_2), \ldots, f(x_n) \\
\end{align*}
\]

Data Layout Patterns

\[
\begin{align*}
\text{split}^m(x_0 \ldots x_n) &= (x_0 x_1 \ldots x_{m-1}, x_m \ldots x_n) \\
\text{join}(x_0 \ldots x_n) &= x_0 \ldots x_n \\
\text{zip}(x_0 \ldots x_n, y_0 \ldots y_n) &= (x_0, y_0) \ldots (x_n, y_n) \\
\text{get}_i((x_0, \ldots, x_n)) &= x_i \\
\text{slide}(\text{size}, \text{step}, x_0 \ldots x_n) &= \begin{array}{c}
\vdots \\
x_0 \quad x_1 \quad \ldots \\
\end{array} \\
\end{align*}
\]

Finally, the intermediate language also supports four algorithms used as building blocks to express programs. Besides the well-known primitives, the input array can be transformed into a vectorized form. This transformation is straightforward and is applied in parallel.

The code generator during code generation, the function \( f \) must be nested inside of the \( \text{split}^m \) primitive projects a component of a tuple.
Dot Product in the LIFT IL

\[
partialDot(x: \mathbb{R}^n, y: \mathbb{R}^n) = \{ \\
\text{join}(\text{mapWrg}^0(\lambda \to t1) \\
\text{join}(\text{toGlobal}(\text{mapLcl}^0(\text{mapSeq}(id)))(\text{split}^1( \\
\text{iterate}^6(\lambda \to t2) \\
\text{join}(\text{mapLcl}^0(\text{toLocal}(\text{mapSeq}(id)), \\
\text{reduceSeq}(\text{add}, 0, \text{split}^2(t2)))), \\
\text{join}(\text{mapLcl}^0(\text{toLocal}(\text{mapSeq}(id)), \\
\text{reduceSeq}(\text{multAndSumUp}, 0, \text{split}^2(t1)))), \\
\text{split}^{128}(\text{zip}(x, y)))))) \\
\} \\
\]
The LIFT Intermediate Representation

Step 1

\[ \lambda x \rightarrow \]

\[
\text{join(}
\text{mapLcl}\theta(
\text{toLocal(mapSeq(id))},
\text{reduceSeq(multAndSumUp, 0, split}^2(x))))
\]
The LIFT Intermediate Representation

Step 1

\[
\lambda \, x_0 \rightarrow \\
\text{join}(\lambda \, x_6 \rightarrow \\
\text{mapLcl}^0(\lambda \, x_5 \rightarrow \\
\text{toLocal}(\lambda \, x_4 \rightarrow \text{mapSeq}(\lambda \, x_3 \rightarrow \text{id}(x_3), x_4), x_5), \\
\text{reduceSeq}(\lambda \, x_1, x_2 \rightarrow \text{multAndSumUp}(x_1, x_2), 0, \text{split}^2(x_0)), x_6)
\]
Dot Product in the LIFT IR

\[
\lambda \ x0 \rightarrow \\
\text{join} (\lambda \ x6 \rightarrow \\
\text{mapLcl}_\theta(\lambda \ x5 \rightarrow \\
\text{toLocal}(\lambda \ x4 \rightarrow \\
\text{mapSeq}(\lambda \ x3 \rightarrow \\
\text{id}(x3),\ x4),\ x5),\ \\
\text{reduceSeq}(\lambda \ x1,x2 \rightarrow \\
\text{multAndSumUp}(x1,x2),\ \\
0,\ \\
\text{split}^2(x0))),\ x6)
\]
Dot Product in the LIFT IR

\[ \lambda \ x0 \rightarrow \]
\[ \text{join}(\lambda \ x6 \rightarrow) \]
\[ \text{mapLcl}^\theta(\lambda \ x5 \rightarrow) \]
\[ \text{toLocal}(\lambda \ x4 \rightarrow) \]
\[ \text{mapSeq}(\lambda \ x3 \rightarrow) \]
\[ \text{id}(x3), x4), x5), \]
\[ \text{reduceSeq}(\lambda \ x1,x2 \rightarrow) \]
\[ \text{multAndSumUp}(x1,x2), \]
\[ \theta, \]
\[ \text{split}^2(x0)), x6) \]
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

mapWrg^0(
    \lambda \ z \rightarrow \text{join}(\text{mapLcl}^0(\n        \text{toLocal}(\text{mapSeq}(\text{id})),\n        \text{reduceSeq}(\lambda \ a, xy \rightarrow a + (xy_0*xy_1), 0, \text{split}^2(z))),\n        \text{split}^{128}(\text{zip}(\text{x}, \text{y})))
)

\downarrow \ ?

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

...
View Construction

mapWrgθ(
    λ z → join(mapLclθ(
        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

\[
\begin{align*}
\text{TupleAccessView(0)} & \quad \text{ArrayAccessView(i)} \\
\text{ArrayAccessView(l_id)} & \quad \text{SplitView(2)} \\
\text{ArrayAccessView(wg_id)} & \quad \text{SplitView(128)} \\
\text{ZipView} & \quad \text{MemoryView(x)} \\
\end{align*}
\]
**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

![Diagram of View Consumption]

Array Stack | Tuple Stack
--- | ---
[] | [0]
[i] | [0]
[l_id, i] | [0]
[(2 × l_id) + i] | [0]
[wg_id, (2 × l_id) + i] | [0]
[(2 × l_id) + (128 × wg_id) + i] | [0]
[(2 × l_id) + (128 × wg_id) + i] | []
[0] | []

x[(2 × l_id) + (128 × wg_id) + i]
Simplifying Array Accesses

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

\[
1 (((wg_id\times M + l_id)/M) + (((wg_id\times M + l_id) \mod M)\times N))/N + (((wg_id\times M + l_id)/M) + (((wg_id\times M + l_id) \mod M)\times N)) \mod N
\]

\[
2 (w_id + l_id \times N)/N + (w_id + l_id \times N) \mod N
\]

\[
3 l_id \times N + w_id
\]

- Set of arithmetic rules are used for simplification

\[
x/y = 0, \quad \text{if } x < y \text{ and } y \neq 0
\]
\[
(x \times y + z)/y = x + z/y, \quad \text{if } y \neq 0
\]
\[
x \mod y = x, \quad \text{if } x < y \text{ and } y \neq 0
\]
\[
(x/y) \times y + x \mod y = x, \quad \text{if } y \neq 0
\]
\[
(x \times y) \mod y = 0, \quad \text{if } y \neq 0
\]
\[
(x + y) \mod z = (x \mod z + y \mod z) \mod z, \quad \text{if } z \neq 0
\]
Compilation Flow of Dot Product

```
kernel void KERNEL(const global float *restrict x,
                 const global float *restrict y,
                 global float *z, int N) {
  local float tmp1[64]; local float tmp2[64];
  local float tmp3[32];
  float acc1; float acc2;
  for (int wg_id = get_group_id(0); wg_id < N/128; 
       wg_id += get_num_groups(0)) {
    int l_id = get_local_id(0);
    acc1 = 0.0f;
    for (int i = 0; i < 2; i += 1) {
      acc1 = multiAndSumUp(acc1, 
                           x[2 * l_id + 128 * wg_id + i],
                           y[2 * l_id + 128 * wg_id + i]); }
  tmp1[l_id] = id(acc1); }
  barrier(CLK_LOCAL_MEM_FENCE);
  int size = 64;
  local float *in = tmp1; local float *out = tmp2;
  for (int iter = 0; iter < 6; iter += 1) {
    if (get_local_id(0) < size / 2) {
      acc2 = 0.0f;
      for (int i = 0; i < 2; i += 1) {
        acc2 = add(acc2, in[2 * l_id + i]); }
      out[l_id] = id(acc2); }
    barrier(CLK_LOCAL_MEM_FENCE);
    size = size / 2;
    in = (out == tmp1) ? tmp1 : tmp3;
    out = (out == tmp1) ? tmp3 : tmp1;
    barrier(CLK_LOCAL_MEM_FENCE); }
  if (get_local_id(0) < 1) {
    z[wg_id] = id(tmp3[l_id]); }
  barrier(CLK_GLOBAL_MEM_FENCE); }
```
Experimental Evaluation

<table>
<thead>
<tr>
<th>Optimizations:</th>
<th>None</th>
<th>Barrier elimination + Control-flow simplification</th>
<th>Barrier elimination + Control-flow simplification + Array access simplification</th>
</tr>
</thead>
<tbody>
<tr>
<td>N-Body, NVIDIA</td>
<td><img src="image" alt="Graph" /></td>
<td><img src="image" alt="Graph" /></td>
<td><img src="image" alt="Graph" /></td>
</tr>
<tr>
<td>N-Body, AMD</td>
<td><img src="image" alt="Graph" /></td>
<td><img src="image" alt="Graph" /></td>
<td><img src="image" alt="Graph" /></td>
</tr>
<tr>
<td>MD</td>
<td><img src="image" alt="Graph" /></td>
<td><img src="image" alt="Graph" /></td>
<td><img src="image" alt="Graph" /></td>
</tr>
<tr>
<td>K-Means</td>
<td><img src="image" alt="Graph" /></td>
<td><img src="image" alt="Graph" /></td>
<td><img src="image" alt="Graph" /></td>
</tr>
<tr>
<td>NN</td>
<td><img src="image" alt="Graph" /></td>
<td><img src="image" alt="Graph" /></td>
<td><img src="image" alt="Graph" /></td>
</tr>
<tr>
<td>MRI-Q</td>
<td><img src="image" alt="Graph" /></td>
<td><img src="image" alt="Graph" /></td>
<td><img src="image" alt="Graph" /></td>
</tr>
<tr>
<td>Convolution</td>
<td><img src="image" alt="Graph" /></td>
<td><img src="image" alt="Graph" /></td>
<td><img src="image" alt="Graph" /></td>
</tr>
<tr>
<td>ATAX</td>
<td><img src="image" alt="Graph" /></td>
<td><img src="image" alt="Graph" /></td>
<td><img src="image" alt="Graph" /></td>
</tr>
<tr>
<td>GEMV</td>
<td><img src="image" alt="Graph" /></td>
<td><img src="image" alt="Graph" /></td>
<td><img src="image" alt="Graph" /></td>
</tr>
<tr>
<td>GESUMMV</td>
<td><img src="image" alt="Graph" /></td>
<td><img src="image" alt="Graph" /></td>
<td><img src="image" alt="Graph" /></td>
</tr>
<tr>
<td>MM</td>
<td><img src="image" alt="Graph" /></td>
<td><img src="image" alt="Graph" /></td>
<td><img src="image" alt="Graph" /></td>
</tr>
</tbody>
</table>

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