HIGH PERFORMANCE STENCIL CODE GENERATION WITH LIFT

Bastian Hagedorn | Larisa Stoltzfus | Michel Steuwer | Sergei Gorlatch | Christophe Dubach
**Why Stencil Computations?**

Stencil computations are a class of kernels which update *neighboring* array elements according to a fixed pattern, called *stencil*.

Frequently occur in:

- Medical Imaging
- Physics Simulations
- Machine Learning
- PDE Solvers
WHY STENCIL COMPUTATIONS?

Stencil compute time:

- HPC Center München: 49% (2017)
- HPC Center Stuttgart: 68% (2016)

Frequently occur in:

- Medical Imaging
- Machine Learning
- Physics Simulations
- PDE Solvers
Yet Another Stencil Paper?
### Domain Specific Languages

- PATUS
- Pochoir
- PARTANS
- Halide
- ...  
- DSL
EXPLOITING DOMAIN KNOWLEDGE

Hardware

- Multicore CPU
- GPU
  - HPC
  - Mobile
- Xeon Phi
  - KNC
  - KNL

DSL

- PATUS
- Pochoir
- PARTANS
- Halide
- ...
Exploiting Domain Knowledge

Stencil DSLs

Multicore CPU

GPU
- HPC
- Mobile

Xeon Phi
- KNC
- KNL

... Hardware
EXPLOITING DOMAIN KNOWLEDGE

- Linear Algebra DSLs
- Stencil DSLs
- N-Body DSLs

Hardware:
- Multicore CPU
- GPU
- Xeon Phi
- HPC
- Mobile
- KNC
- KNL

...

...
APPROACHING PERFORMANCE PORTABILITY

- Linear Algebra DSLs
- Stencil DSLs
- N-Body DSLs

UNIVERSAL HIGH PERFORMANCE CODE GENERATOR

- Multicore CPU
- GPU: HPC, Mobile
- Xeon Phi: KNC, KNL

Hardware
APPROACHING PERFORMANCE PORTABILITY

Linear Algebra DSLs  Stencil DSLs  N-Body DSLs  ...

LIFT

Multicore CPU  GPU  Xeon Phi

HPC  Mobile  KNC  KNL  ...

Hardware
Explore Optimizations by rewriting [CASES'16]
High-Level IR

Explore Optimizations by rewriting

Low-Level Program

Multicore CPU
GPU
Xeon Phi

HPC Mobile KNC KNL

Hardware

[CASES'16]
High-Level IR

Explore Optimizations by rewriting

Low-Level Program

Multicore CPU

GPU

Xeon Phi

HPC Mobile KNC KNL...

Hardware

[CGO'17]

[CASES'16]

Code Generation

DSL

Code Generation

DSL

LIFT
2. High-level Programming
1. Low-level Optimizations
G. High Performance
Lift's High-level Primitives

- map(□→□) → □
- reduce(⊕) → □
- split(n) → □
- join → □
- zip → □
LIFT'S HIGH-LEVEL PRIMITIVES

map(□→□) → □□□□
reduce(⊕) → □
split(n) → □□□□
join → □□□□
zip → □□□□

dotproduct.lift

a b
LIFT'S HIGH-LEVEL PRIMITIVES

- **map(\(\square \to \square\))**
- **reduce(\(\oplus\))**
- **split\((n)\)**
- **join**
- **zip**

**dotproduct.lift**

\[
\text{dotproduct.lift} = \text{reduce}(+, 0, \text{map}(*, \text{zip}(a, b)))
\]
LIFT'S HIGH-LEVEL PRIMITIVES

- \( \text{map}(\square \rightarrow \square) \)
- \( \text{reduce}(\oplus) \)
- \( \text{split}(n) \)
- \( \text{join} \)
- \( \text{zip} \)

\[
\text{map}(\ast, \text{zip}(a,b))
\]
**LIFT'S HIGH-LEVEL PRIMITIVES**

- `map( □ → □ )`  
- `reduce( ⊕ )`  
- `split(n)`  
- `join`  
- `zip`  

```
reduce(+, 0, map(*, zip(a, b)))
```
**Lift's High-Level Primitives**

- **map** (□ → □)
- **reduce** (⊕)
- **split** (n)
- **join**
- **zip**

Can we express stencil computations in Lift?

Let's look at a simple stencil example...
What are Stencil Computations?

3-point-stencil.c

```c
for (int i = 0; i < N; i++) {
    int sum = 0;
    for (int j = -1; j <= 1; j++) {
        int pos = i + j;
        pos = pos < 0 ? 0 : pos;
        pos = pos > N - 1 ? N - 1 : pos;
        sum += A[pos];
    }
    B[i] = sum;
}
```
WHAT ARE STENCIL COMPUTATIONS?

3-point-stencil.c

```c
for (int i = 0; i < N; i++) {
    int sum = 0;
    for (int j = -1; j <= 1; j++) {
        int pos = i + j;
        pos = pos < 0 ? 0 : pos;
        pos = pos > N - 1 ? N - 1 : pos;
        sum += A[pos];
    }
    B[i] = sum;
}```
What are Stencil Computations?

3-point-stencil.c

```c
for (int i = 0; i < N; i++) {
    int sum = 0;
    for (int j = -1; j <= 1; j++) {
        int pos = i + j;
        pos = pos < 0 ? 0 : pos;
        pos = pos > N - 1 ? N - 1 : pos;
        sum += A[pos];
    }
    B[i] = sum;
}
```
**Stencil Computations in Lift**

- **map**: $\Box \rightarrow \Box 
- **reduce**: $\oplus$
- **split**: $(\text{n})$
- **join**: $\Box \Box$
- **zip**: $\Box \Box 

3-point-stencil.lift
Stencil Computations in Lift

- \( \text{map}(\Box \rightarrow \Box) \)
- \( \text{reduce}(\oplus) \)
- \( \text{split}(n) \)
- \( \text{join} \)
- \( \text{zip} \)
- \( \text{stencil} \)

3-point-stencil.lift

Add specialized primitive: Job done?
Stencil computations in Lift

- `map(\rightarrow)`
- `reduce(\oplus)`
- `split(n)`
- `join`
- `zip`
- `stencil`

Add specialized primitive: Job done?

- **No Reuse** of existing primitives and optimizations
- **Domain-specific** rather than generic
- **Multidimensional?** is it composable?
Decomposing Stencil Computations

3-point-stencil.c

```c
for (int i = 0; i < N ; i ++) {
    int sum = 0;
    for (int j = -1; j <= 1; j ++) {
        int pos = i + j;
        pos = pos < 0 ? 0 : pos;
        pos = pos > N - 1 ? N - 1 : pos;
        sum += A[ pos ];
    }
    B[ i ] = sum ;
}
```
Decomposing Stencil Computations

3-point-stencil.c

for (int i = 0; i < N; i++) {
    int sum = 0;
    for (int j = -1; j <= 1; j++) { // (a)
        int pos = i + j;
        pos = pos < 0 ? 0 : pos;
        pos = pos > N - 1 ? N - 1 : pos;
        sum += A[pos];
    }
    B[i] = sum;
}

(a) access neighborhoods for every element
Decomposing Stencil Computations

3-point-stencil.c

```c
for (int i = 0; i < N ; i ++) {
    int sum = 0;
    for (int j = -1; j <= 1; j ++) {   // ( a )
        int pos = i + j;
        pos = pos < 0 ? 0 : pos;        // ( b )
        pos = pos > N - 1 ? N - 1 : pos;
        sum += A[ pos ]; }
    B[ i ] = sum ; }
```

(a) access neighborhoods for every element
(b) specify boundary handling
Decomposing Stencil Computations

(a) access neighborhoods for every element
(b) specify boundary handling
(c) apply stencil function to neighborhoods
Decomposing Stencil Computations

3-point-stencil.c

```c
for (int i = 0; i < N; i++) {
    int sum = 0;
    for (int j = -1; j <= 1; j++) {   // (a)
        int pos = i + j;
        pos = pos < 0 ? 0 : pos;       // (b)
        pos = pos > N - 1 ? N - 1 : pos;
        sum += A[pos]; }
    B[i] = sum; }
```

(a) access **neighborhoods** for every element
(b) specify **boundary handling**
(c) apply **stencil function** to neighborhoods
Stencil Computations in Lift

- **map** (□→□) → 
- **reduce** (⊕) → 
- **split** (n) → 
- **join** → 
- **zip** → 

3-point-stencil.lift

???
**Stencil Computations in Lift**

- **map(□→□)**:  
  - Input:  
  - Output:  

- **reduce(⊕)**:  
  - Input:  
  - Output:  

- **split(n)**:  
  - Input:  
  - Output:  

- **join**:  
  - Input:  
  - Output:  

- **zip**:  
  - Input:  
  - Output:  

---

### 3-point-stencil.lift

- **Reuse map**: allows to reuse existing rewrite rules
- **Simplicity**: one primitive per task
- **Multidimensional**: easily composable

- **Map**:  
  - Input:  
  - Output:  

---

### Diagram Details

- **3-point-stencil.lift**
  - **Reuse map**: allows to reuse existing rewrite rules
  - **Simplicity**: one primitive per task
  - **Multidimensional**: easily composable

---

*Note: The diagram illustrates the stencil computations in Lift, showing how the operations are applied and the resulting structures.*
BOUNDARY HANDLING USING PAD

**pad (reindexing)**

\[
\text{clamp}(i, n) = (i < 0) ? 0 : ((i >= n) ? n-1 : i)
\]

\[
\text{pad}(1, 1, \text{clamp}, [a, b, c, d]) = [a, a, b, c, d, d]
\]

**pad-reindexing.lift**

\[
\text{pad-reindexing.lift}(\text{clamp}, [a, b, c, d]) = [a, a, b, c, d, d]
\]

**pad (constant)**

\[
\text{constant}(i, n) = C
\]

\[
\text{pad}(1, 1, \text{constant}, [a, b, c, d]) = [C, a, b, c, d, C]
\]

**pad-constant.lift**

\[
\text{pad-constant.lift}(\text{constant}, [a, b, c, d]) = [C, a, b, c, d, C]
\]
Neighborhood Creation Using **Slide**

- **size**
- **step**

\[
\text{slide}(3, 1, [a, b, c, d, e]) =
\[[a, b, c], [b, c, d], [c, d, e]]
\]
Applying Stencil Function Using **Map**

```plaintext
map(nbh =>
  reduce(add, 0.0f, nbh))
```
Putting it Together

\[
\begin{align*}
\text{map}(\square \rightarrow \square) & \rightarrow \begin{array}{c}
\square \\
\square \\
\square \\
\square \\
\square \\
\square \\
\square \\
\square \\
\square \\
\square
\end{array} \\
\text{reduce}(\oplus) & \rightarrow \begin{array}{c}
\square \\
\square \\
\square
\end{array} \\
\text{split}(n) & \rightarrow \begin{array}{c}
\square \\
\square
\end{array} \\
\text{join} & \rightarrow \begin{array}{c}
\square \\
\square \\
\square
\end{array} \\
\text{zip} & \rightarrow \begin{array}{c}
\square \\
\square \\
\square \\
\square \\
\square \\
\square \\
\square \\
\square \\
\square \\
\square
\end{array} \\
\text{pad}(l,r,b) & \rightarrow \begin{array}{c}
\square \\
\square \\
\square \\
\square \\
\square \\
\square \\
\square \\
\square \\
\square \\
\square
\end{array} \\
\text{slide}(n,s) & \rightarrow \begin{array}{c}
\square \\
\square \\
\square \\
\square \\
\square \\
\square \\
\square \\
\square \\
\square \\
\square
\end{array}
\end{align*}
\]

\[
\text{stencil1D.lift}
\]

\[
\text{def } \text{stencil1D} = \\
\text{fun}(A =>) \\
\text{map(reduce(add, 0.0f),) \\
\text{slide(3,1,} \\
\text{pad(1,1,clamp,A))))})
\]
MULTIDIMENSIONAL STENCIL COMPUTATIONS

are expressed as compositions of intuitive, generic 1D primitives
MULTIDIMENSIONAL STENCIL COMPUTATIONS

are expressed as compositions of intuitive, generic 1D primitives
Multidimensional stencil computations are expressed as compositions of intuitive, generic 1D primitives.

\[
\text{pad}_2(1,1,\text{clamp},\text{input})
\]
Multidimensional Boundary Handling using $\text{PAD}_2$

\[
\text{input} \quad \begin{array}{|c|c|c|c|} \hline \hline \end{array}
\]

\[\text{pad}_2 = \text{map}(\text{pad}(\text{pad}(l,r,b,\text{input})))\]
**MULTIDIMENSIONAL BOUNDARY HANDLING USING \( \text{PAD}_2 \)**

\[
\text{pad}_2 = \text{pad}(l, r, b, \text{input})
\]
Multidimensional boundary handling using $\text{pad}_2$

$$\text{pad}_2 = \text{map}(\text{pad}(l, r, b, \text{pad}(l, r, b, \text{input})))$$
Multidimensional stencil computations are expressed as compositions of intuitive, generic 1D primitives:

$$\text{pad}_2(1,1, \text{clamp}, \text{input})$$
MULTIDIMENSIONAL STENCIL COMPUTATIONS

are expressed as compositions of intuitive, generic 1D primitives

\[ \text{slide}_2(3,1, \text{pad}_2(1,1,\text{clamp},\text{input})) \]
Multidimensional stencil computations are expressed as compositions of intuitive, generic 1D primitives.

\[ \text{map}_2(\text{sum}, \text{slide}_2(3,1, \text{pad}_2(1,1,\text{clamp},\text{input}))) \]
Multidimensional stencil computations are expressed as compositions of intuitive, generic 1D primitives.

\[
\text{map}_3(\text{sum}, \text{slide}_3(3,1, \text{pad}_3(1,1,\text{clamp},\text{input})))
\]
Multidimensional stencil computations are expressed as compositions of intuitive, generic 1D primitives.

\[
\text{map}_3(\text{sum}, \text{slide}_3(3,1, \text{pad}_3(1,1,\text{clamp},\text{input})))
\]

Advantages:
- Compact Language
- Reuse Rewrites
- Simple Compilation
1. Low-level optimizations
2. High-level programming
G. High performance
## REUSING EXISTING REWRITE RULES

### Divide & Conquer

```latex
\text{map}(f, A)
```

![Diagram](image)
### REUSING EXISTING REWRITE RULES

**Divide & Conquer**

<table>
<thead>
<tr>
<th>$\text{map}(f, A)$</th>
<th>$\Rightarrow$</th>
<th>$\text{join}(\text{map}(\text{map}(f), \text{split}(n, A)))$</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image1" alt="Diagram" /></td>
<td></td>
<td><img src="image2" alt="Diagram" /></td>
</tr>
</tbody>
</table>
OPTIMIZATION: OVERLAPPED TILING

- Exploit Locality
  Close neighborhoods share elements that can be grouped in tiles

- Shared Memory
  Fast memory can be used to cache tiles
**Optimization: Overlapped Tiling**

- Exploit Locality
  Close neighborhoods share elements that can be grouped in tiles

- Shared Memory
  Fast memory can be used to cache tiles
Optimization: Overlapped Tiling

Exploit Locality

Close neighborhoods share elements that can be grouped in tiles

Shared Memory

Fast memory can be used to cache tiles
Optimization: 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 as a Rewrite Rule

Overlapped tiling rule

\( \text{map}(f, \text{slide}(3, 1, \text{input})) \)
Overlapped Tiling as a Rewrite Rule

Overlapped tiling rule

\[ \text{map}(f, \text{slide}(3,1,\text{input})) \rightarrow \text{slide}(u,v,\text{input}) \]
OVERLAPPED TILING AS A REWRITE RULE

overlapped tiling rule

\[ \text{map}(f, \text{slide}(3,1,\text{input})) \rightarrow \text{join}(\text{map}(\text{tile} \Rightarrow \text{map}(f, \text{slide}(3,1,\text{tile})), \text{slide}(u,v,\text{input}))) \]
COMPARISON WITH HAND-OPTIMIZED CODES

higher is better

Lift achieves the same performance as hand optimized code
Comparison with polyhedral compilation

Lift outperforms state-of-the-art optimizing compilers

higher is better
STENCIL COMPUTATIONS IN LIFT

We added:

- **2 Primitives**
  - pad, slide

- **1 Rewrite Rule**
  - overlapped tiling

Diagram:
- High-Level IR
- Explore Optimizations by rewriting
- Low-Level Program
- Multicore CPU
- GPU
  - HPC
  - Mobile
- Xeon Phi
  - KNC
  - KNL
- Hardware
LIFT IS OPEN SOURCE!

more info at:

lift-project.org

" Paper  Artifact Available  CGO Artifact  Source Code"

Bastian Hagedorn: b.hagedorn@wwu.de