HeteroGen: Transpiling C to Heterogeneous HLS Code with Automated Test Generation and Program Repair

Qian Zhang, Jiyuan Wang, Harry Xu, Miryung Kim

ASPLOS ’22, Feb 28, Lausanne, Switzerland
Heterogeneous computing is becoming popular
HLS-C is not standard C/C++

- Resource **finization**
- Hardware expertise and **pragmas** for optimization

Manual *rewriting* for synthesizability and optimization
Developer tools are not available

C/C++  
HLS-C

• Resource **finization**
• Hardware expertise and **pragmas** for optimization

Manual **rewriting** for synthesizability and optimization

No developer tools for code **translation**
int KNN()
{
    // Calculate distance
    for (i = 0 to number)
    {
        dist[i] = l2norm(data[i], dim);
    }

    // Top 1 nearest neighbor
    ...
}

7x speedup on FPGA.

What is HLS Development Process Today?

1. Performance profiling
2. Kernel function identification
3. Manual rewriting for HLS compatibility
4. Differential testing with input samples
5. Performance Optimization

- 6 minutes HLS compilation
- 8 minutes CPU-FPGA simulation
- 2.5 hours FPGA synthesis

Performance Optimization
HLS-C requires specifying *bitwidth* for each type

```c
float vecdot(
    float a[],
    float b[],
    int n) {
    for (int i = 0; i < n; i++)
        sum += a[i] * b[i];
    return sum;
}
```

```c
float vecdot(
    float a[],
    float b[],
    fpga_int<7> n) {
    for (fpga_int<7> i = 0; i < n; i++)
        sum += a[i] * b[i];
    return sum;
}
```

C Program

HLS-C Program
HLS-C uses a custom floating point type

C Program

```c
float vecdot(
    float a[],
    float b[],
    fpga_int<7> n) {
    for (fpga_int<7> i = 0; i < n; i++)
        sum += a[i] * b[i];
    return sum;
}
```

HLS-C Program

```c
fpga_float<8,15> vecdot(
    fpga_float<8,15> a[],
    fpga_float<8,15> b[],
    fpga_int<7> n) {
    for (fpga_int<7> i = 0; i < n; i++)
        sum += a[i] * b[i];
    return sum;
}
```
HLS-C requires *finitizing* resources

```c
struct Node {
    Node *left, *right;
    int val;
};
void init(Node **root) {
    *root = (Node *)malloc(sizeof(Node)),
}
void delete_tree(Node *root) {
    free(root);
}
void traverse(Node *curr) {
    if (curr == NULL) return;
    int ret = visit(curr->val);
    traverse(curr->left);
    traverse(curr->right);
}
```

C Program

```c
Node Node_arr[NODE_ARR_SIZE];
struct Node {
    Node *left, *right;
    int val;
};
void init(Node_ptr *root) {
    *root = (Node_ptr)node_malloc(sizeof(Node_ptr))
}
void delete_tree(Node_ptr root) {
    node_free(root);
}
void traverse(Node_ptr curr) {
    stack<context>
    s(STACK_SIZE);
    while (!s.empty()) {
    }
}
```

HLS-C Program

4 errors
Automated Program Repair (2008~current)

Iterative code update for error removal

Fault Localization

Candidate Repair Generation

Fitness Evaluation

Oracle

Fix Patterns
Challenges: Long Compilation Time

Iterative code update for error removal

- Fault Localization
- Candidate Repair Generation
- Fitness Evaluation

Oracle

Fix Patterns

0.2s in GenProg vs. 14 minutes in HLS

Challenges: Check Behavior Equivalence

Iterative code update for error removal

- Fault Localization
- Candidate Repair Generation
- Fitness Evaluation

Oracle

Fix Patterns
HeteroGen Overview

1. Fuzzing-based test generation
2. Profiling to generate initial version

Concrete Test Inputs

3. Error-based fault localization
4. Dependence-based repair exploration

HLS Errors and Fix Patterns

5. Fitness evaluation via differential execution

Iterative code update for error repair
Observation 1: HLS repairs have patterns.

- HLS compatibility rewrite patterns from real-world user posts from Xilinx.
- We represent the repair edits as a set of parameterized AST edits to be concretized to a given context.

static_stack($a1:val)$

An error study based on 1000 posts.
Observation 2: HLS repairs require dependent edits.

```c
struct Node {
    Node *left, *right;
    int val;
};

void init(Node **root) {
    *root = (Node *)malloc(sizeof(Node));
}

void delete_tree(Node *root) {
    free(root);
}

void traverse(Node *curr) {
    if (curr == NULL) return;
    int ret = visit(curr->val);
    traverse(curr->left);
    traverse(curr->right);
}
```

C Program
Observation 2: HLS repairs require dependent edits.

```c
struct Node {
    Node *left, *right;
    int val;
};
void init(Node **root) {
    *root = (Node *)malloc(sizeof(Node));
}
void delete_tree(Node *root) {
    free(root);
}
void traverse(Node *curr) {
    if (curr == NULL) return;
    int ret = visit(curr->val);
    traverse(curr->left);
    traverse(curr->right);
}
```

C Program
Observation 2: HLS repairs require dependent edits.

```c
struct Node {
    Node *left, *right;
    int val;
};
void init(Node **root) {
    *root = (Node *)malloc(sizeof(Node)); }
void delete_tree(Node *root) {...
    free(root); }
void traverse(Node *curr) {
    if (curr == NULL) return;
    int ret = visit(curr->val);
    traverse(curr->left);
    traverse(curr->right);
}

C Program
```
Observation 2: HLS repairs require dependent edits.

```c
struct Node {
    Node *left, *right;
    int val;
};
void init(Node **root) {
    *root = (Node *)malloc(sizeof(Node));
}
void delete_tree(Node *root) {...
    free(root);
}
void traverse(Node *curr) {
    if (curr == NULL) return;
    int ret = visit(curr->val);
    traverse(curr->left);
    traverse(curr->right);
}
```
Optimization 1: Early Repair Candidate Rejection

- **Early Candidate Rejection**
  - If a repair does not conform to HLS coding styles, it does not need to be compiled

**14 mins** full synthesis and simulation **vs.**

**1 second** conformance checking

```c
void foo (...) {
    int8 array1[M];
    int12 array2[N];
    ...
    #pragma HLS unroll skip_exit_check factor=4
    loop_2: for(i=0;i<M;i++) {
        array1[i] = ...;
        array2[i] = ...;
        ...
    }
    ...
}
```
Optimization 2: Expedite Search using Dependence

- Dependence-guided search helps construct both valid edits and also prune the search space of potential repairs
  - 1
  - 1 -> 2
  - 1 -> 3
  - 1 -> 2 -> 5
## Evaluation: Code Edit

<table>
<thead>
<tr>
<th>Subject</th>
<th>Edits (LOC)</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Manual</td>
<td>HeteroGen</td>
<td></td>
</tr>
<tr>
<td>Signal Transmission</td>
<td>78</td>
<td>69</td>
<td></td>
</tr>
<tr>
<td>Arithmetic Computation</td>
<td>8</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>Merge Sort</td>
<td>276</td>
<td>356</td>
<td></td>
</tr>
<tr>
<td>Image Processing</td>
<td>136</td>
<td>32</td>
<td></td>
</tr>
<tr>
<td><strong>Graph Traversal</strong></td>
<td><strong>144</strong></td>
<td><strong>438</strong></td>
<td></td>
</tr>
<tr>
<td>Matrix Multiplication</td>
<td>25</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>Bubble Sort</td>
<td>45</td>
<td>25</td>
<td></td>
</tr>
<tr>
<td>Linked List</td>
<td>156</td>
<td>298</td>
<td></td>
</tr>
<tr>
<td>Face Detection</td>
<td>3272</td>
<td>144</td>
<td></td>
</tr>
<tr>
<td>Digit Recognition</td>
<td>61</td>
<td>35</td>
<td></td>
</tr>
</tbody>
</table>

HeteroGen preserves test behavior for all subjects. It automates up to 438 line edits, reducing HLS rewriting effort.
Evaluation: Speedup

Time (minute)

% of invocations

WithoutChecker

HeteroGen  WithoutDependence

P1  P2  P3  P4  P5  P6  P7  P8  P9  P10
Evaluation: Speedup

1. Dependence-based repair search enable 35x speedup.
2. HeteroGen obviates the need of invoking the full HLS tool chain by 75%, which results in 4x speedup.
Evaluation: Summary

90%  
Effectiveness  
HeteroGen produces an HLS-compatible version for 9 out of 10.

97%  
Coverage  
Auto-generated inputs cover 97%, while pre-existing tests reach 36% coverage.

35X  
Speed-up  
Dependence-based search contributes to 35X speedup than the one without.

~438 lines  
Automation  
It automates upto 438 lines.

1.64X  
Latency  
It produces a HLS version 1.63X faster than the original C
Summary

- HeteroGen automatically translates C/C++ to HLS code by search-based code repair.
- HeteroGen speeds up the repair process with two optimization:
  - Apply code edits with dependence to reduce search space.
  - Code style check to avoid unnecessary compilation process.
- HeteroGen ensures the correctness of translated code by automated testing.
- HeteroGen on Github: [https://github.com/UCLA-SEAL/HeteroGen](https://github.com/UCLA-SEAL/HeteroGen)
Developer Tools - Test, Refactor, and Repair

**HeteroRefactor**  
(ICSE 2020)  
Automated refactoring, achieves performance improving and resource saving for FPGA program

**HeteroFuzz**  
(ESEC/FSE 2021)  
754X faster in exposing the same set of distinct platform portability errors than naive fuzzing

**HeteroGen**  
(ASPLOS 2022)

Developer Tools for Heterogeneous Computing:  
https://github.com/UCLA-SEAL
Thanks for listening!

Q&A