Automatically Optimizing the Latency, Area, and Accuracy for HLS

Xitong Gao, John Wickerson, George A. Constantinides

Circuits and Systems Research Group

Department of Electrical and Electronic Engineering

Imperial College London, United Kingdom
Numerical Programs on FPGAs

Present
• RTL implementations
• Error-prone
• Slow to develop
• High development costs

Future
• Synthesize C programs with HLS
• Easy to debug & verify
• Comparable performance
• Design space exploration
Numerical Programs

• Often consist of floating-point computations
  • Long latency
  • A lot of resources

• Often spend most of their time in loops
  • Loop pipelining
Motivating Example

Dot product:

```c
float d = 0.0f;
for (int i = 0; i < 1024; i++)
    d = d + A[i] * B[i];
```
Motivating Example

```c
float d = 0.0f;
for (int i = 0; i < 1024; i++)
    d = d + A[i] * B[i];
```
Motivating Example

```c
float d = 0.0f;
for (int i = 0; i < 1024; i += 2) {
    d = d + A[i]*B[i];
    d = d + A[i+1]*B[i+1];
}
```
Motivating Example

```c
float d = 0.0f;
for (int i = 0; i < 1024; i += 2)
    d = d + (A[i]*B[i] + A[i+1]*B[i+1]);
```
Problems with Arithmetic Equivalences

Single-precision floating-point:

\[
((1 + 2^{-24}) + 2^{-24}) - 1
\]

\[
(1 + (2^{-24} + 2^{-24})) - 1
\]
Problems with Arithmetic Equivalences

Single-precision floating-point:

\[(1 + 2^{-24}) + 2^{-24} - 1 = 0 \quad \text{Round-off Error: } 2^{-23}\]

\[(1 + (2^{-24} + 2^{-24})) - 1 = 0.00000012\ldots \quad \text{Round-off Error: 0}\]
Problems with Arithmetic Equivalences

Many of the arithmetic rules do not hold under floating-point arithmetic:

• **Associativity**: \((a + b) + c \neq a + (b + c)\)
• **Distributivity**: \(a \times (b + c) \neq a \times b + a \times c\)
• **Square diff.**: \(a^2 - b^2 \neq (a + b) \times (a - b)\)
• Many more...
Problems with Arithmetic Equivalences

HLS tools (such as LegUp and Vivado HLS) have limited use of them and do not use them by default. 

-funsafe-math-optimizations

Can we use them to our advantage safely?
Round-off Errors: Equivalence?

SOAP3:

Automatically and simultaneously optimizes latency, resources, and accuracy of numerical programs by exploiting arithmetic rules and many more for HLS.
Learn to use SOAP in 10 Seconds

#define N 1000
#define TSTEPS 20

for (int t = 0; t < TSTEPS; t++)
    for (int i = 1; i < N - 1; i++)
        for (int j = 1; j < N - 1; j++)
Learn to use SOAP in 10 Seconds

#define N 1000
#define TSTEPS 20

#pragma soap input "
float A[N][N] = [0.0, 1.0]
#pragma soap output A

for (int t = 0; t < TSTEPS; t++)
  for (int i = 1; i < N - 1; i++)
    for (int j = 1; j < N - 1; j++)
Push one button...
Wait a few minutes...
This is what you get

- Explores a huge number of equivalent programs.
- Produces a set of optimized programs - 3D Pareto frontier.

**Best latency and accuracy**
Comparison

- Optimizing the example code for latency and accuracy:

<table>
<thead>
<tr>
<th>Improvements</th>
<th>Vivado HLS (Expression Balance)</th>
<th>SOAP + VHLS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Run time (s)</td>
<td>1.2x</td>
<td>7.0x</td>
</tr>
<tr>
<td>Resources (LUTs)</td>
<td>1.3x</td>
<td>3.5x</td>
</tr>
<tr>
<td>Accuracy (Worst-case error)</td>
<td>???????</td>
<td>2.5x</td>
</tr>
</tbody>
</table>

![Graph showing data points and lines](image-url)
Tool flow

1. C Program
2. C to MIR translation
3. Analysis Passes
4. Optimizer
5. Code Generation
6. Transformation Rules
7. Pareto-optimal Programs
Tool flow

Analysis Passes

C Program

C to MIR translation

Optimizer

Code Generation

Pareto-optimal Programs

Transformation Rules

- Associativity
- Distributivity
- Partial loop unrolling
- Memory access

......
Tool flow

Analysis Passes
- Accuracy Analysis
- Schedule Analysis

C Program
→
C to MIR translation

Optimizer
→
Code Generation

Pareto-optimal Programs

Transformation Rules
- Associativity
- Distributivity
- Partial loop unrolling
- Memory access
MIR: Yet another IR?

• An expression-like IR that specifically tackles the functional equivalence of numerical programs
  • – *how* a program is executed
  • + *effect of executing* the program

• Greatly reduces the size of search space, but optimal solutions are kept
MIR: Straight-line code

\[
\begin{align*}
y &= x + 1; \\
x &= y; \\
y &= y \times 2; \\
x &= x + 3;
\end{align*}
\]

\[
\begin{align*}
x &= x + 1; \\
y &= 2 \times x; \\
x &= x + 3;
\end{align*}
\]

Infinitely many equivalent programs...
MIR: Conditionals

```plaintext
x = x + 1;
if (b) {
    x = 2 * x;
}
```
MIR: Loops

- Loops are infinite depth expressions with recurring structure.

```c
x = 0;
y = 0.0f;
while (x < n) {
x = x + 1;
y = y + a / (x * x);
}
```
MIR: Array accesses

- In our MIR, we capture the read and write operations with *update* and *access* operators respectively.

\[
A[i + 1] = 2 \times A[i];
\]
Results

**PolyBench and Livermore loops**, prioritize latency:

- **Latency**
  - up to 13x
  - average 7x

- **Accuracy**
  - up to 8x
  - average 4x

- **Resources costs**
  - up to 4x
  - average 3x
Conclusion

• SOAP3
  • Arithmetic rules
  • Memory access rules
  • Standard program equivalences
  • Optimize numerical C programs for accuracy, resources and latency

• Future work
  • Fixed-point with Multiple precisions
  • Relational Abstract Domains
Tool & Results

https://admk.github.io/soap/