Bridging High-Level Synthesis and Application-Specific Arithmetic: The Case Study of Floating-Point Summations

Yohann Uguen    Florent de Dinechin    Steven Derrien
September, FPL2017

University of Lyon, INSA Lyon, Inria, CITI
F-69621 Villeurbanne, France
{Yohann.Uguen, Florent.de-Dinechin}@insa-lyon.fr

University Rennes 1, IRISA
Rennes, France
Steven.Derrien@univ-rennes1.fr
Circuit: from specification to implementation?

\[ \sum_{i=0}^{N-1} x_i \times y_i \]
Arithmeticians point of view: highly tuned hardware, custom datatypes

✓ Low resource
✓ Low latency
✓ Accuracy control

✗ Difficult to use
  - operator nature
  - FloPoCo’s interface
  - integration of the operator
Compiler designers point of view: follow language semantics and datatypes

```c
float sumOfProduct(float x[N], float y[N]){
    float sum = 0;
    for (int i=0; i<N; i++)
        sum += x[i]*y[i];
    return sum;
}
```

✓ Ease of use

✗ Moderate resource

✗ Moderate latency

✗ No accuracy control
Compiler designers point of view: follow language semantics and datatypes

```c
float sumOfProduct(float x[N], float y[N]){
    double sum = 0;
    for (int i=0; i<N; i++)
        sum += x[i]*y[i];
    return sum;
}
```

- ✓ Ease of use
- × Moderate resource
- × Moderate latency
- × No accuracy control
Naive solution: interface FloPoCo operators with HLS

```c
float sumOfProduct(float x[N], float y[N]){
    float sum = 0;
    for (int i=0; i<N; i++)
        sum += x[i]*y[i];
    return sum;
}
```

- ✓ Custom datatypes
- ✓ Better resource/accuracy control
- ✗ Vivado HLS strongly discourages it
- ✗ Sub-optimal compiler optimizations
- ✗ Can’t simulate circuit in C
Proposed method: embed FloPoCo’s spirit in HLS compilers

✓ Ease of use
✓ Low resource
✓ Low latency
✓ Accuracy control

- Transform variable datatypes when specified
- Better than custom datatypes: a high-level pragma
- Vivado HLS compliant

HLS tools can perform IR optimizations
**INTRODUCTION**

High-level HLS C

GeCoS source-to-source compiler

arithmetic optimization plugin

context-specific arithmetic operators

Low-level HLS C

HLS (Vivado)

modified source code

Low-level HLS C

VHDL
The arithmetic side:
Exact accumulator in Vivado HLS

The compiler side:
Source-to-source transformations using GeCoS

Evaluation

Conclusion
THE ARITHMETIC SIDE:
EXACT ACCUMULATOR IN VIVADO HLS
ACCUMULATOR - ARCHITECTURE

- Fixed-point accumulator: custom application specific format
- Based on [An FPGA-specific approach to floating-point accumulation and sum-of-products, 2008]
- Minor improvements
  - support for sub-normals
  - increased modularity
- Each box tailored to application requirements
**Fixed-Point vs Floating Point Accumulator**

Infinitely accurate fixed-point accumulator

Addend

Floating-point accumulator

Shifted significand

Infinitely accurate fixed-point accumulator

Floating-point accumulator

Accurate Result

FP Acc = 50.125

Fixed-Point Acc = 50.20166015625

9/18
Fixed-point vs Floating Point Accumulator

Infinitely accurate fixed-point accumulator

Addend

Floating-point accumulator

Shifted significand

\[
\begin{array}{c}
\times 0 \\
+ \times 1 \\
\hline
\text{Acc}
\end{array}
\]

\[
\begin{array}{c}
1010000000 \\
1010011000 \\
\hline
1010000000
\end{array}
\]

Infinitely accurate fixed-point accumulator

Floating-point accumulator

\[
\begin{array}{c}
010010011100 \\
1010000000
\end{array}
\]
**Fixed-point vs Floating-point Accumulator**

Infinitely accurate fixed-point accumulator

Addend

Floating-point accumulator

Shifted significand

Infinitely accurate fixed-point accumulator

Floating-point accumulator

Accumulation:
- **x0**: 0 0 0 0 0 1 1 0 0 0
- **x1**: 0 0 0 0 0 1 1 1 0 0 0

Result:
- **x2**: 0 1 0 1 0 0 0 0 1 1 0 0 1

Accuracy:
- Exact Result: 50.2017822265625
- FP Acc: 50.125
- Fixed-Point Acc: 50.20166015625
Infinitely accurate fixed-point accumulator

Addend

\[ x_0 = 1010000000 \]
\[ + x_1 = 1010011000 \]
\[ + x_2 = 1000110011 \]

\[ = 1011101000000000 \]

Infinitely accurate fixed-point accumulator

Floating-point accumulator

\[ x_2 = 010100011001 \]
\[ \text{acc} = 1011101000 \]

\[ = 010100011110110 \]
 FIXED-POINT VS FLOATING POINT ACCUMULATOR

Infinitely accurate fixed-point accumulator

Shifted significand

\[\begin{align*}
\times 0 & \quad 1 0 1 0 0 0 0 0 0 0 \\
\times 1 & \quad 1 0 1 0 0 1 1 0 0 \\
\times 2 & \quad 1 0 0 0 1 1 0 0 1 \\
\hline
= & \quad 1 0 1 1 1 0 1 1 0 0 0 0 0 0
\end{align*}\]

Floating-point accumulator

\[\begin{align*}
0 1 0 1 0 0 & 0 1 1 1 1 0 1 1 0
\end{align*}\]
INFinitely accurate fixed−point accumulator

Addend

Floating−point accumulator

Shifted significand

Infinitely accurate fixed-point accumulator

Floating-point accumulator
**FIXED-POINT VS FLOATING-POINT ACCUMULATOR**

**Infinitely accurate fixed−point accumulator**

**Addend**

```
0 0 0 0 01 1 0 0 0 x0
0 0 0 0 01 1 11+ x1
1 0 0 0 1 1 0 0 1+ x2
01 0 1 0 1 1 1 1+ x3
```

**Shifted significand**

```
x0
+ x1
+ x2
+ x3
```

```
1 0 1 0 0 0 0 0 0
1 0 1 0 0 1 1 0 0
1 0 0 0 1 1 0 0 1
1 0 0 1 0 1 1 1 1
```

```
= 1 0 1 1 1 0 1 1 1 1 1 1
```

**Floating-point accumulator**

```
0 1 0 1 0 0 0 1 1 1 0 1 1 1
```

**Accuracy:**

<table>
<thead>
<tr>
<th>Exact Result</th>
<th>FP Acc</th>
<th>Fixed-Point Acc</th>
</tr>
</thead>
<tbody>
<tr>
<td>50.2017822265625</td>
<td>50.125</td>
<td>50.20166015625</td>
</tr>
</tbody>
</table>
Infinitely accurate fixed−point accumulator

Addend

Floating−point accumulator

Shifted significand

x0
+ x1
+ x2
+ x3
+ x4
= 

Infinitely accurate fixed-point accumulator

Floating-point accumulator
**Fixed-point vs Floating Point Accumulator**

### Finite accuracy fixed-point accumulator

**Shifted significand**

| x0 | 1 0 1 0 0 0 0 0 0 |
| x1 | 1 0 1 0 0 1 1 0 0 |
| x2 | 1 0 0 0 1 1 0 0 1 |
| x3 | 1 0 0 1 0 1 1 1 1 |
| x4 | 1 1 0 1 0 0 1 0 0 |

**Accumulator**

| 1 1 0 0 1 0 0 1 1 1 1 |

---

### Floating-point accumulator

**Accumulator**

| 1 1 0 1 0 0 1 0 0 0 1 |

---

Accuracy:

- **Exact Result**: 50.2017822265625
- **FP Acc**: 50.125
- **Fixed-Point Acc**: 50.20166015625
**FIXED-POINT VS FLOATING POINT ACCUMULATOR**

**Addend**

```
x5 0 0 1 0 1 0 0 1 1 1 0 0 1 0
```

**Shifted significand**

```
x0 1 0 1 0 0 0 0 0 0
+x1 1 0 1 0 0 1 1 0 0
+x2 1 0 0 0 1 1 0 0
+x3 1 0 0 1 0 1 1 1
+x4 1 1 0 1 0 0 1 0
+x5 1 0 1 1 1 0 0 1 0
```

```
= 1 1 0 0 1 0 0 1 1 1 0 1 0
```

**Finite accuracy fixed-point accumulator**

```
0 0 1 0 1 0 0 1 1 1 0 0 1 0
```

**Floating-point accumulator**

```
1 0 1 1 1 0 0 1 0
```

**Accuracy:**

- **Exact Result:** 50.2017822265625
- **FP Acc:** 50.125
- **Fixed-Point Acc:** 50.20166015625
**FIXED-POINT VS FLOATING POINT ACCUMULATOR**

### Accuracy:

- **Exact Result** = 50.2017822265625
- **FP Acc** = 50.125
- **Fixed-Point Acc** = 50.20166015625
Vivado HLS compliant C

Same results as FloPoCo operator in ~ 50 lines of code in terms of:

- LUTs
- Registers
- DSPs
THE COMPILER SIDE:
SOURCE-TO-SOURCE TRANSFORMATIONS USING GECOS
#define N 100000
float computeSum(float in1[N], float in2[N]){
    float sum = 0;
    for (int i=1; i<N-1; i++){
        sum += in1[i]*in2[i-1];
        sum += in1[i];
        sum += in2[i+1];
    }
    return sum;
}
Use of a pragma

```c
#define N 100000
float computeSum(float in1[N], float in2[N]){
    float sum = 0;
    #pragma FPacc VAR=sum MaxAcc=300000.0 epsilon=1e-15
    for (int i=1; i<N-1; i++){
        sum += in1[i]*in2[i-1];
        sum += in1[i];
        sum += in2[i+1];
    }
    return sum;
}
```
Build a GeCoS plug-in
- Read C code
- Transform to compiler IR
- Retrieve DFG associated to pragma
- Apply transformations
- Regenerate C code

```c
sum += in1[i]*in2[i-1];
sum += in1[i];
sum += in2[i+1];
```
original DFG - generated circuit

DFG

Circuit

in2[i-1] → Float Mul → Float Add → Float Add → sum → sum

in2[i+1] → Float Add → Float Add → Float Add → sum

in1[i] → Float Mul → Float Add → Float Add → Float Add → sum

in2[i+1] → Float Add → Float Add → Float Add → sum

DFG

Circuit

5

7

7

7

sum
This CANNOT be obtained using FloPoCo
TRANSFORMED DFG - GENERATED CIRCUIT

This CANNOT be obtained using FloPoCo
EVALUATION
**Evaluation of the previous example**

- After place-and-route
- For 100MHz
- On Xilinx Kintex 7
- Using Vivado HLS 2016.3
- Using Vivado 2016.3

<table>
<thead>
<tr>
<th></th>
<th>Original no pragma</th>
<th>Transformed epsilon=1e-7</th>
<th>Transformed epsilon=1e-10</th>
<th>Transformed epsilon=1e-19</th>
</tr>
</thead>
<tbody>
<tr>
<td>LUTs</td>
<td>538</td>
<td>693</td>
<td>824</td>
<td>1400</td>
</tr>
<tr>
<td>DSPs</td>
<td>5</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Latency (N=100K)</td>
<td>2000K</td>
<td>100K</td>
<td>100K</td>
<td>100K</td>
</tr>
</tbody>
</table>

Single cycle accumulation improves latency.
FP benchmark suite

- 10 kernels
- 5 contains accumulations

Methodology

- Implemented a golden reference (MPFR) for accuracy
- Synthetized the 5 kernels w/ and w/o transformations
- Accumulator tailored to be just more accurate than original benchmarks
Three types of results

1. One kernel gets +33% LUT usage, -60% DSPs for a 400% speed improvement

2. Two kernels different loop sizes: between +24% and +26% LUT usage, -60% DSPs for a speed improvement between 480% and 840%

3. Two kernels have less than 1% difference for resource usage but unpredictable latency
   - Better latency observed by simulation
   - Benchmarks should be completely re-written to be suitable for HLS
CONCLUSION
Arithmetic optimizations and HLS are compatible

- At least for sums-of-products
- Tool distributed with GeCoS

Future work

- Support more operators
- Make more use of compiler knowledge such as loop counts when possible
- Integration to a HLS tool

Thank you
Exact Multiplier

Sign 1  XOR  Exponent 1  Mantissa 1

Sign 2  Exponent 2  Mantissa 2

Mantissa 1  Exponent 1  MultSign

MultExponent

W_e

W_f

ExactProduct

Mantissa 2  Exponent 2

subnormal detection

0/1  0/1

MultMantissa

×

W_e + 1

2 × W_f + 2

We
<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Type</th>
<th>LUTs</th>
<th>DSPs</th>
<th>Latency</th>
<th>Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Livermore</td>
<td>Original</td>
<td>384</td>
<td>5</td>
<td>80K</td>
<td>11 bits</td>
</tr>
<tr>
<td></td>
<td>Transformed</td>
<td>576 (+33%)</td>
<td>2 (-60%)</td>
<td>20K (/4)</td>
<td>13 bits</td>
</tr>
<tr>
<td>LU-8</td>
<td>Original</td>
<td>809</td>
<td>5</td>
<td>82</td>
<td>8-23 bits</td>
</tr>
<tr>
<td></td>
<td>Transformed</td>
<td>1007 (+24%)</td>
<td>2 (-60%)</td>
<td>17 (/4.8)</td>
<td>23 bits</td>
</tr>
<tr>
<td>LU-45</td>
<td>Original</td>
<td>819</td>
<td>5</td>
<td>452</td>
<td>8-23 bits</td>
</tr>
<tr>
<td></td>
<td>Transformed</td>
<td>1034 (+26%)</td>
<td>2 (-60%)</td>
<td>54 (/8.4)</td>
<td>23 bits</td>
</tr>
<tr>
<td>Scholes</td>
<td>Original</td>
<td>15640</td>
<td>175</td>
<td>N/A</td>
<td>19 bits</td>
</tr>
<tr>
<td></td>
<td>Transformed</td>
<td>15923 (+1%)</td>
<td>175 (+0%)</td>
<td>N/A</td>
<td>23 bits</td>
</tr>
<tr>
<td>Fourier</td>
<td>Original</td>
<td>34596</td>
<td>64</td>
<td>N/A</td>
<td>6 bits</td>
</tr>
<tr>
<td></td>
<td>Transformed</td>
<td>34681 (+1%)</td>
<td>59 (-8%)</td>
<td>N/A</td>
<td>11 bits</td>
</tr>
</tbody>
</table>
ACCUMULATOR - ARCHITECTURE

Exponent  Mantissa  Sign

MaxMSBX - w_e
Shift value
MaxMSBX - LSBA + 1
Negate

FloatToFix

Registers

Accumulator

Negate

FixToFloat

LZC + Shifter

Exponent  Mantissa  Sign

w_e  w_f  w_e  w_f