A single-source C++20 HLS flow for function evaluation on FPGA

Luc Forget, Gauthier Harnisch, Ronan Keryell
Adaptive Embeded Computing Group – AMD, San Jose, California, USA
Florent de Dinechin
University of Lyon, INSA Lyon, Inria, CITI, France

July 6, 2022

Table of contents

Arbitrary arithmetic types in C++

- C++ provides a few standard integral types:
  - Signed or not: `unsigned int`, `signed int`
  - Different width: `int8_t`, `uint16_t`, `int32_t`, `uint64_t`, ... if supported
- C23 adds fundamental N-bit type: `_BitInt(22)`
- And a small set of floating point types:
  - `float` – IEEE binary32 if supported
  - `double` – IEEE binary64 if supported
  - `long double` – IEEE binary128 if supported, or anything, really
  - `_Decimal{32,64,128}` (C23) for IEEE decimal floating point, if supported

Limited to what the CPU supports "natively"

Arbitrary arithmetic types in C++

- C++ provides a few standard integral types:
  - Signed or not: `unsigned int`, `signed int`
  - Different width: `int8_t`, `uint16_t`, `int32_t`, `uint64_t`, ... if supported
- C23 adds fundamental *N*-bit type: `_BitInt(22)`
- And a small set of floating point types:
  - `float` – IEEE binary32 if supported
  - `double` – IEEE binary64 if supported
  - `long double` – IEEE binary128 if supported, or anything, really
  - `_Decimal{32,64,128}` (C23) for IEEE decimal floating point, if supported

Limited to what the CPU supports "natively"

AMD

Arbitrary arithmetic types in C++

- C++ provides a few standard integral types:
  - Signed or not: `unsigned int`, `signed int`
  - Different width: `int8_t`, `uint16_t`, `int32_t`, `uint64_t`, ... if supported
- C23 adds fundamental N-bit type: `_BitInt(22)`
- And a small set of floating point types:
  - `float` – IEEE binary32 if supported
  - `double` – IEEE binary64 if supported
  - `long double` – IEEE binary128 if supported, or anything, really
  - `_Decimal{32,64,128}` (C23) for IEEE decimal floating point, if supported

Limited to what the CPU supports "natively"

AMD

Arbitrary arithmetic types in C++

- C++ provides a few standard integral types:
  - Signed or not: `unsigned int`, `signed int`
  - Different width: `int8_t`, `uint16_t`, `int32_t`, `uint64_t`, ... if supported
- C23 adds fundamental N-bit type: `_BitInt(22)`
- And a small set of floating point types:
  - `float` – IEEE binary32 if supported
  - `double` – IEEE binary64 if supported
  - `long double` – IEEE binary128 if supported, or anything, really
  - `_Decimal{32,64,128}` (C23) for IEEE decimal floating point, if supported

Limited to what the CPU supports "natively"

Arbitrary arithmetic types in C++

- C++ provides a few standard integral types:
  - Signed or not: `unsigned int`, `signed int`
  - Different width: `int8_t`, `uint16_t`, `int32_t`, `uint64_t`, ... if supported
- C23 adds fundamental N-bit type: `_BitInt(22)`
- And a small set of floating point types:
  - `float` – IEEE binary32 if supported
  - `double` – IEEE binary64 if supported
  - `long double` – IEEE binary128 if supported, or anything, really
  - `_Decimal{32,64,128}` (C23) for IEEE decimal floating point, if supported

Limited to what the CPU supports "natively"

Arbitrary arithmetic types in C++

- C++ provides a few standard integral types:
  - Signed or not: `unsigned int`, `signed int`
  - Different width: `int8_t`, `uint16_t`, `int32_t`, `uint64_t`, ... if supported
- C23 adds fundamental *N*-bit type: `_BitInt(22)`
- And a small set of floating point types:
  - `float` – IEEE binary32 if supported
  - `double` – IEEE binary64 if supported
  - `long double` – IEEE binary128 if supported, or anything, really
  - `_Decimal{32,64,128}` (C23) for IEEE decimal floating point, if supported

Limited to what the CPU supports "natively"

Arbitrary arithmetic types in C++

- C++ provides a few standard integral types:
  - Signed or not: `unsigned int`, `signed int`
  - Different width: `int8_t`, `uint16_t`, `int32_t`, `uint64_t`, ... if supported
- C23 adds fundamental N-bit type: `_BitInt(22)`
- And a small set of floating point types:
  - `float` – IEEE binary32 if supported
  - `double` – IEEE binary64 if supported
  - `long double` – IEEE binary128 if supported, or anything, really
  - `_Decimal{32,64,128}` (C23) for IEEE decimal floating point, if supported

Limited to what the CPU supports "natively"

Arbitrary arithmetic types in C++

- C++ provides a few standard integral types:
  - Signed or not: `unsigned int`, `signed int`
  - Different width: `int8_t`, `uint16_t`, `int32_t`, `uint64_t`, ... if supported
- C23 adds fundamental *N*-bit type: `_BitInt(22)`
- And a small set of floating point types:
  - `float` – IEEE binary32 if supported
  - `double` – IEEE binary64 if supported
  - `long double` – IEEE binary128 if supported, or anything, really
  - `_Decimal{32,64,128}` (C23) for IEEE decimal floating point, if supported

Limited to what the CPU supports “natively”

AMD

Arbitrary arithmetic types *not* in C++

Some of the types that do not have native C++ support:

- All the variation around 16 bits floating point:
  - IEEE `binary16`
  - Google `bfloat16`
  - IBM `DLFloat16`
- Arbitrary precision floating point format
- Logarithmic Number System with many bases
- Exotic and bizarre encodings
  - Posit
  - Unum

Can we offer C++ types that represent these types ?
Can we have these types working with HLS ?
Without needing *Frankenstein*-like HLS-RTL integration ?
With good QOR ?

Arbitrary arithmetic types *not* in C++

Some of the types that do not have native C++ support:

- All the variation around 16 bits floating point:
  - IEEE `binary16`
  - Google `bfloat16`
  - IBM `DLFloat16`
- Arbitrary precision floating point format
- Logarithmic Number System with many bases
- Exotic and bizarre encodings
  - Posit
  - Unum

Can we offer C++ types that represent these types ?
Can we have these types working with HLS ?
Without needing *Frankenstein*-like HLS-RTL integration ?
With good QOR ?

Arbitrary arithmetic types *not* in C++

Some of the types that do not have native C++ support:

- All the variation around 16 bits floating point:
  - IEEE `binary16`
  - Google `bfloat16`
  - IBM `DLFloat16`
- Arbitrary precision floating point format
- Logarithmic Number System with many bases
- Exotic and bizarre encodings
  - Posit
  - Unum

Can we offer C++ types that represent these types ?
Can we have these types working with HLS ?
Without needing *Frankenstein*-like HLS-RTL integration ?
With good QOR ?

Arbitrary arithmetic types *not* in C++

Some of the types that do not have native C++ support:

- All the variation around 16 bits floating point:
  - IEEE `binary16`
  - Google `bfloat16`
  - IBM `DLFloat16`
- Arbitrary precision floating point format
- Logarithmic Number System with many bases
- Exotic and bizarre encodings
  - Posit
  - Unum

Can we offer C++ types that represent these types ?
Can we have these types working with HLS ?
Without needing *Frankenstein*-like HLS-RTL integration ?
With good QOR ?

Arbitrary arithmetic types *not* in C++

Some of the types that do not have native C++ support:

- All the variation around 16 bits floating point:
  - IEEE `binary16`
  - Google `bfloat16`
  - IBM `DLFloat16`
- Arbitrary precision floating point format
- Logarithmic Number System with many bases
- Exotic and bizarre encodings
  - Posit
  - Unum

Can we offer C++ types that represent these types ?
Can we have these types working with HLS ?
Without needing *Frankenstein*-like HLS-RTL integration ?
With good QOR ?

Arbitrary arithmetic types *not* in C++

Some of the types that do not have native C++ support:

- All the variation around 16 bits floating point:
  - IEEE `binary16`
  - Google `bfloat16`
  - IBM `DLFloat16`
- Arbitrary precision floating point format
- Logarithmic Number System with many bases
- Exotic and bizarre encodings
  - Posit
  - Unum

Can we offer C++ types that represent these types ?
Can we have these types working with HLS ?
Without needing *Frankenstein*-like HLS-RTL integration ?
With good QOR ?

Arbitrary arithmetic types *not* in C++

Some of the types that do not have native C++ support:

- All the variation around 16 bits floating point:
  - IEEE `binary16`
  - Google `bfloat16`
  - IBM `DLFloat16`
- Arbitrary precision floating point format
- Logarithmic Number System with many bases
- Exotic and bizarre encodings
  - Posit
  - Unum

Can we offer C++ types that represent these types ?

Can we have these types working with HLS ?

Without needing *Frankenstein*-like HLS-RTL integration ?

With good QOR ?

Arbitrary arithmetic types *not* in C++

Some of the types that do not have native C++ support:

- All the variation around 16 bits floating point:
  - IEEE `binary16`
  - Google `bfloat16`
  - IBM `DLFloat16`
- Arbitrary precision floating point format
- Logarithmic Number System with many bases
- Exotic and bizarre encodings
  - Posit
  - Unum

Can we offer C++ types that represent these types ?

Can we have these types working with HLS ?

Without needing *Frankenstein*-like HLS-RTL integration ?

With good QOR ?

Arbitrary arithmetic types *not* in C++

Some of the types that do not have native C++ support:

- All the variation around 16 bits floating point:
  - IEEE `binary16`
  - Google `bfloat16`
  - IBM `DLFloat16`
- Arbitrary precision floating point format
- Logarithmic Number System with many bases
- Exotic and bizarre encodings
  - Posit
  - Unum

Can we offer C++ types that represent these types ?
Can we have these types working with HLS ?
Without needing *Frankenstein*-like HLS-RTL integration ?
With good QOR ?

Arbitrary arithmetic types *not* in C++

Some of the types that do not have native C++ support:

- All the variation around 16 bits floating point:
  - IEEE `binary16`
  - Google `bfloat16`
  - IBM `DLFloat16`
- Arbitrary precision floating point format
- Logarithmic Number System with many bases
- Exotic and bizarre encodings
  - Posit
  - Unum

Can we offer C++ types that represent these types ?
Can we have these types working with HLS ?
Without needing *Frankenstein*-like HLS-RTL integration ?
With good QOR ?

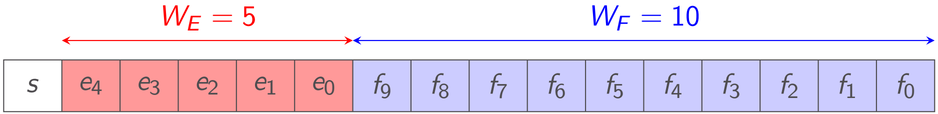
Floating point number

Library proposes template type `FPNumber<`W_E`, `W_F`>`

Floating point number

Library proposes template type FPNumber<$\mathtt{W_E}$, $\mathtt{W_F}$>

IEEE binary16 $= \mathrm{FP}<5, 10>$



Floating point number

Library proposes template type $\texttt{FPNumber<}\texttt{W}_\texttt{E}\texttt{, }\texttt{W}_\texttt{F}\texttt{>}$

IEEE binary16 $= \text{FP}<5, 10>$

$W_E = 5$ $W_F = 10$

s	e_4	e_3	e_2	e_1	e_0	f_9	f_8	f_7	f_6	f_5	f_4	f_3	f_2	f_1	f_0

bfloat16 $= \text{FP}<8,7>$

$W_E = 8$ $W_F = 7$

s	e_7	e_6	e_5	e_4	e_3	e_2	e_1	e_0	f_6	f_5	f_4	f_3	f_2	f_1	f_0

Floating point number

Library proposes template type `FPNumber<W_E, W_F>`

IEEE binary16 = FP<5, 10>

$W_E = 5$ $\qquad$ $W_F = 10$

s	e_4	e_3	e_2	e_1	e_0	f_9	f_8	f_7	f_6	f_5	f_4	f_3	f_2	f_1	f_0

bfloat16 = FP<8,7>

$W_E = 8$ $\qquad$ $W_F = 7$

s	e_7	e_6	e_5	e_4	e_3	e_2	e_1	e_0	f_6	f_5	f_4	f_3	f_2	f_1	f_0

My completely custom float type = FP<7, 9>

$W_E = 7$ $\qquad$ $W_F = 9$

s	e_6	e_5	e_4	e_3	e_2	e_1	e_0	f_8	f_7	f_6	f_5	f_4	f_3	f_2	f_1	f_0

The posit encoding scheme

A posit encoding is parametrized by the word size N and the exponent shift size W_{es}. For a positive value, the code is made of the following fields :

- The first sign bit s is set to zero
- The range encoded by the length r of a sequence of identical bits b ended by $\bar{b}$
- The exponent shift es on W_{es} bits
- The remaining $N-(k+2+W_{es})$ bits are the significand bits f

The encoded value is

$$v = 1.f \cdot 2^{k2^{W_{es}}+es}$$

$$k = \begin{cases} -r & \text{if } b = 0 \\ r-1 & \text{if } b = 1 \end{cases}$$

Negative values are encoded as 2's complement of their opposite

The posit encoding scheme – simple case ($N = 8,\ W_{\text{es}} = 0$)

- Word size N
- Exponent: computed from variable length sequence r of identical bits
- Remaining bits: fraction bits

The posit encoding scheme – simple case ($N = 8$, $W_{es} = 0$)

- Word size N
- Exponent: computed from variable length sequence r of identical bits
- Remaining bits: fraction bits

$r = 1$

0	1	0	1	0	0	0	1

$$1.10001 \times 2^{1-1} = 1.53125$$

The posit encoding scheme – simple case ($N = 8,\ W_{es} = 0$)

- Word size N
- Exponent: computed from variable length sequence r of identical bits
- Remaining bits: fraction bits

$r = 1$

0	1	0	1	0	0	0	1

$1.10001 \times 2^{1-1} = 1.53125$

$r = 3$

0	1	1	1	0	0	0	1

$1.001 \times 2^{3-1} = 4.5$

The posit encoding scheme – simple case ($N = 8,\ W_{es} = 0$)

- Word size N
- Exponent: computed from variable length sequence r of identical bits
- Remaining bits: fraction bits

$r = 1$

0	1	0	1	0	0	0	1

$$1.10001 \times 2^{1-1} = 1.53125$$

$r = 3$

0	1	1	1	0	0	0	1

$$1.001 \times 2^{3-1} = 4.5$$

$r = 5$

0	1	1	1	1	1	0	1

$$1.1 \times 2^{5-1} = 24$$

The posit encoding scheme – simple case ($N = 8,\ W_{es} = 0$)

- Word size N
- Exponent: computed from variable length sequence r of identical bits
- Remaining bits: fraction bits

$r = 1$

0	1	0	1	0	0	0	1

$$1.10001 \times 2^{1-1} = 1.53125$$

$r = 3$

0	1	1	1	0	0	0	1

$$1.001 \times 2^{3-1} = 4.5$$

$r = 5$

0	1	1	1	1	1	0	1

$$1.1 \times 2^{5-1} = 24$$

$r = 7$

0	1	1	1	1	1	1	1

$$1 \times 2^{7-1} = 64$$

Custom type usage example

```
using posit_t = PositNumber<8, 2>;
auto adder(posit_t a, posit_t b) {
    return a + b;
}
```

Custom type usage example

```cpp
using posit_t = PositNumber<8, 2>;
auto adder(posit_t a, posit_t b) {
    return a + b;
}
```

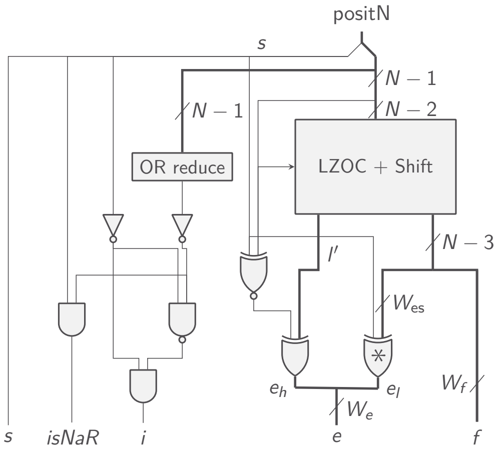


Figure: Posit to internal FP representation decoder

Custom type usage example

```
using posit_t = PositNumber<8, 2>;
auto adder(posit_t a, posit_t b) {
    return a + b;
}
```

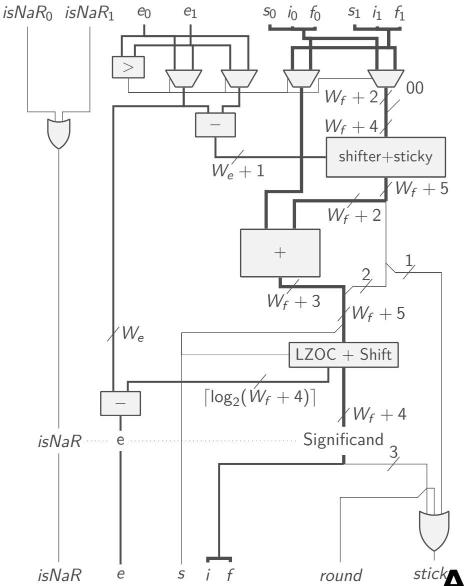


Figure: Perform lossless compressed addition

Custom type usage example

```cpp
using posit_t = PositNumber<8, 2>;
auto adder(posit_t a, posit_t b) {
    return a + b;
}
```

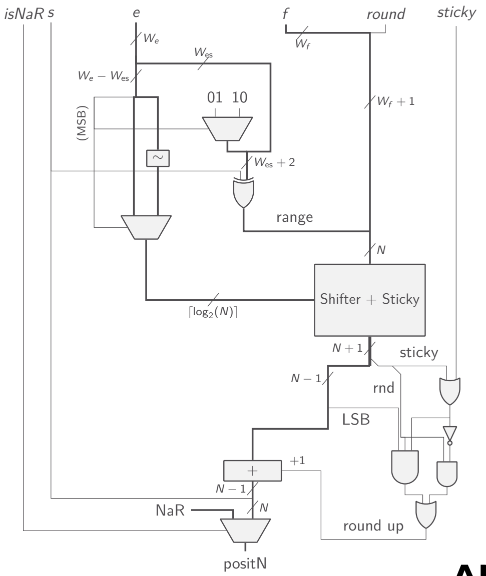


Figure: Convert the result back to Posit

- Logarithmic Number System (LNS): number represented by its logarithm
  - Stored value v in fixed-point represents 2^v.
  - *e.g.* stored value of 2.75 represents $2^{2.75} \approx 6.72717$
- Product simplified (addition of the logarithms)
- Addition require computation of logarithm and power function

In base 2 with $X > Y$, and L_X and L_Y the representation of X and Y respectively:

$$\begin{aligned} L_{X+Y} &= \log_2(2^{L_X} + 2^{L_Y}) \\ &= \log_2\left(2^{L_X}(1 + 2^{L_Y - L_X})\right) \\ &= \log_2(2^{L_X}) + \log_2(1 + 2^{L_Y - L_X}) \\ &= L_X + f_{\oplus}(L_Y - L_X), \qquad \text{with} \quad f_{\oplus}(r) = \log_2(1 + 2^r) \end{aligned}$$

LNS

- Logarithmic Number System (LNS): number represented by its logarithm
  - Stored value v in fixed-point represents 2^v.
  - *e.g.* stored value of 2.75 represents $2^{2.75} \approx 6.72717$
- Product simplified (addition of the logarithms)
- Addition require computation of logarithm and power function

In base 2 with $X > Y$, and L_X and L_Y the representation of X and Y respectively:

$$
\begin{aligned}
L_{X+Y} &= \log_2(2^{L_X} + 2^{L_Y}) \\
&= \log_2\left(2^{L_X}(1 + 2^{L_Y - L_X})\right) \\
&= \log_2(2^{L_X}) + \log_2(1 + 2^{L_Y - L_X}) \\
&= L_X + f_\oplus(L_Y - L_X), \qquad \text{with} \quad f_\oplus(r) = \log_2(1 + 2^r)
\end{aligned}
$$

LNS

- Logarithmic Number System (LNS): number represented by its logarithm
  - Stored value v in fixed-point represents 2^v.
  - *e.g.* stored value of 2.75 represents $2^{2.75} \approx 6.72717$
- Product simplified (addition of the logarithms)
- Addition require computation of logarithm and power function

In base 2 with $X > Y$, and L_X and L_Y the representation of X and Y respectively:

$$\begin{aligned} L_{X+Y} &= \log_2(2^{L_X} + 2^{L_Y}) \\ &= \log_2\left(2^{L_X}(1 + 2^{L_Y - L_X})\right) \\ &= \log_2(2^{L_X}) + \log_2(1 + 2^{L_Y - L_X}) \\ &= L_X + f_\oplus(L_Y - L_X), \qquad \text{with} \quad f_\oplus(r) = \log_2(1 + 2^r) \end{aligned}$$

LNS

- Logarithmic Number System (LNS): number represented by its logarithm
  - Stored value v in fixed-point represents 2^v.
  - *e.g.* stored value of 2.75 represents $2^{2.75} \approx 6.72717$
- Product simplified (addition of the logarithms)
- Addition require computation of logarithm and power function

In base 2 with $X > Y$, and L_X and L_Y the representation of X and Y respectively:

$$
\begin{aligned}
L_{X+Y} &= \log_2(2^{L_X} + 2^{L_Y}) \\
&= \log_2\left(2^{L_X}(1 + 2^{L_Y - L_X})\right) \\
&= \log_2(2^{L_X}) + \log_2(1 + 2^{L_Y - L_X}) \\
&= L_X + f_\oplus(L_Y - L_X), \qquad \text{with} \quad f_\oplus(r) = \log_2(1 + 2^r)
\end{aligned}
$$

LNS

- Logarithmic Number System (LNS): number represented by its logarithm
  - Stored value v in fixed-point represents 2^v.
  - *e.g.* stored value of 2.75 represents $2^{2.75} \approx 6.72717$
- Product simplified (addition of the logarithms)
- Addition require computation of logarithm and power function

In base 2 with $X > Y$, and L_X and L_Y the representation of X and Y respectively:

$$\begin{aligned} L_{X+Y} &= \log_2(2^{L_X} + 2^{L_Y}) \\ &= \log_2\left(2^{L_X}(1 + 2^{L_Y - L_X})\right) \\ &= \log_2(2^{L_X}) + \log_2(1 + 2^{L_Y - L_X}) \\ &= L_X + f_\oplus(L_Y - L_X), \qquad \text{with} \quad f_\oplus(r) = \log_2(1 + 2^r) \end{aligned}$$

Motivation: how to compute composed functions?

How to compute a value for

$$f_{\oplus} : r \mapsto \log_2(1 + 2^r)$$

Composition of elementary functions not OK:

- No control on error (or need high precision on intermediates for worst case scenario)
- Quite slow (sum of the latencies of individual functions)

We can do something better !

Motivation: how to compute composed functions?

How to compute a value for

$$f_{\oplus} : r \mapsto \log_2(1 + 2^r)$$

Composition of elementary functions not OK:

- No control on error (or need high precision on intermediates for worst case scenario)
- Quite slow (sum of the latencies of individual functions)

We can do something better !

AMD

Motivation: how to compute composed functions?

How to compute a value for

$$f_{\oplus} : r \mapsto \log_2(1 + 2^r)$$

Composition of elementary functions not OK:

- No control on error (or need high precision on intermediates for worst case scenario)
- Quite slow (sum of the latencies of individual functions)

We can do something better !

Motivation: how to compute composed functions?

How to compute a value for

$$f_{\oplus} : r \mapsto \log_2(1 + 2^r)$$

Composition of elementary functions not OK:

- No control on error (or need high precision on intermediates for worst case scenario)
- Quite slow (sum of the latencies of individual functions)

We can do something better !

Motivation: how to compute composed functions?

How to compute a value for

$$f_{\oplus} : r \mapsto \log_2(1 + 2^r)$$

Composition of elementary functions not OK:

- No control on error (or need high precision on intermediates for worst case scenario)
- Quite slow (sum of the latencies of individual functions)

We can do something better !

Mathematical function evaluators

Focus of the HEART '22 presentation: accuracy-constrained arbitrary fixed-point unary function evaluators

Given

- An arbitrary fixed-point domain $\mathcal{D}$,
  e.g. $\{k \cdot 2^{-2} \mid \forall k \in [0\,..\,15]\}$
- A unary function $f : \mathcal{D} \to \mathbb{R}$,
  e.g. $f : x \mapsto \sin(\pi \cdot x/2)$
- An accuracy constraint ϵ
- ϵ of the form 2^p

Mathematical function evaluators

Focus of the HEART '22 presentation: accuracy-constrained arbitrary fixed-point unary function evaluators

Given

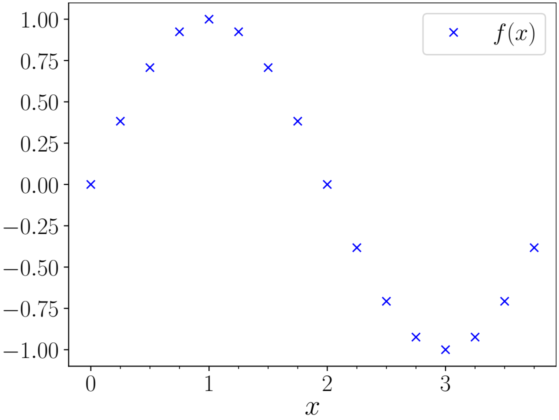
- An arbitrary fixed-point domain $\mathcal{D}$,
  e.g. $\{k \cdot 2^{-2} \mid \forall k \in [0\,..\,15]\}$
- A unary function $f : \mathcal{D} \to \mathbb{R}$,
  e.g. $f : x \mapsto \sin(\pi \cdot x/2)$
- An accuracy constraint ϵ
- ϵ of the form 2^p

Mathematical function evaluators

Focus of the HEART '22 presentation: accuracy-constrained arbitrary fixed-point unary function evaluators

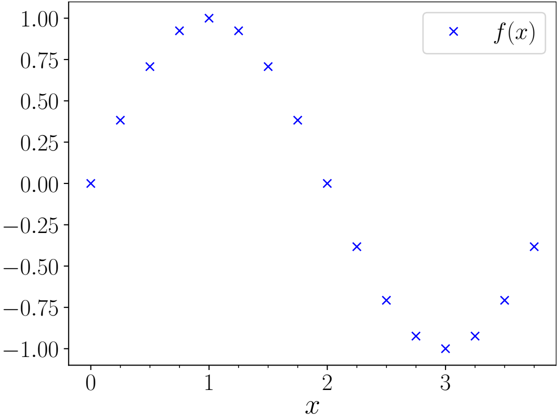
Given

- An arbitrary fixed-point domain $\mathcal{D}$, *e.g.* $\{k \cdot 2^{-2} \mid \forall k \in [0 \mathinner{..} 15]\}$
- A unary function $f : \mathcal{D} \to \mathbb{R}$, *e.g.* $f : x \mapsto \sin(\pi \cdot x/2)$
- An accuracy constraint ϵ
- ϵ of the form 2^p



Mathematical function evaluators

Focus of the HEART ’22 presentation: accuracy-constrained arbitrary fixed-point unary function evaluators



Given

- An arbitrary fixed-point domain $\mathcal{D}$,
  e.g. $\{k \cdot 2^{-2} \mid \forall k \in [0 \mathinner{..} 15]\}$
- A unary function $f : \mathcal{D} \to \mathbb{R}$,
  e.g. $f : x \mapsto \sin(\pi \cdot x/2)$
- An accuracy constraint ϵ
- ϵ of the form 2^p

Mathematical function evaluators

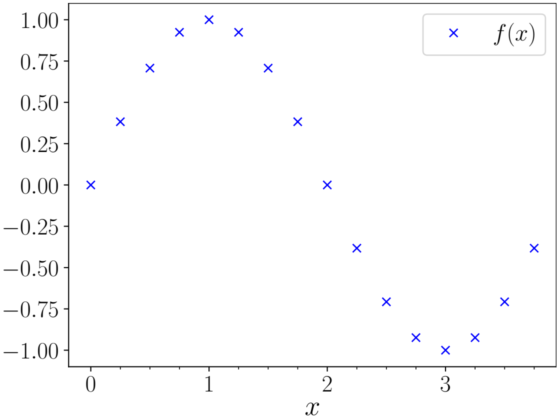
Focus of the HEART '22 presentation: accuracy-constrained arbitrary fixed-point unary function evaluators

Given

- An arbitrary fixed-point domain $\mathcal{D}$, *e.g.* $\{k \cdot 2^{-2} \mid \forall k \in [0 \,..\, 15]\}$
- A unary function $f : \mathcal{D} \to \mathbb{R}$, *e.g.* $f : x \mapsto \sin(\pi \cdot x/2)$
- An accuracy constraint ϵ

Generate HW computing $\tilde{f}_\epsilon$ such that

$$\forall x \in \mathcal{D}, \left| f(x) - \tilde{f}_\epsilon(x) \right| < \epsilon$$



- ϵ of the form 2^p

AMD

Mathematical function evaluators

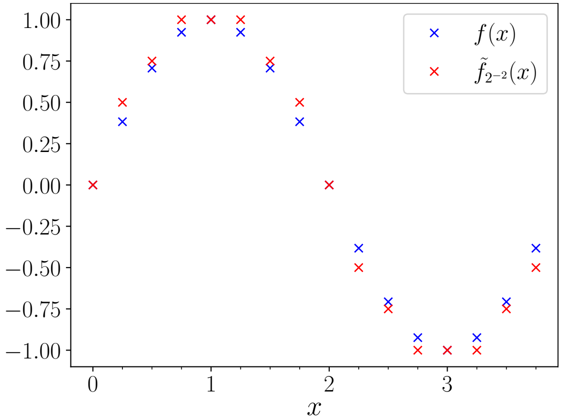
Focus of the HEART '22 presentation: accuracy-constrained arbitrary fixed-point unary function evaluators

Given

- An arbitrary fixed-point domain $\mathcal{D}$, *e.g.* $\{k \cdot 2^{-2} \mid \forall k \in [0 \,..\, 15]\}$
- A unary function $f : \mathcal{D} \to \mathbb{R}$, *e.g.* $f : x \mapsto \sin(\pi \cdot x/2)$
- An accuracy constraint ϵ

Generate HW computing $\tilde{f}_\epsilon$ such that

$$\forall x \in \mathcal{D}, \left| f(x) - \tilde{f}_\epsilon(x) \right| < \epsilon$$



- ϵ of the form 2^p

AMD

Mathematical function evaluators

Focus of the HEART '22 presentation: accuracy-constrained arbitrary fixed-point unary function evaluators

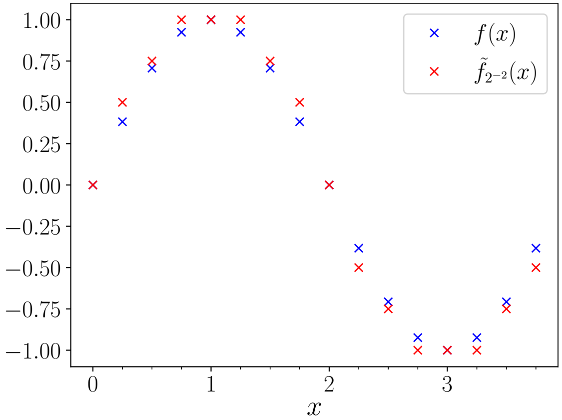
Given

- An arbitrary fixed-point domain $\mathcal{D}$, *e.g.* $\{k \cdot 2^{-2} \mid \forall k \in [0 \,..\, 15]\}$
- A unary function $f : \mathcal{D} \to \mathbb{R}$, *e.g.* $f : x \mapsto \sin(\pi \cdot x/2)$
- An accuracy constraint ϵ

Generate HW computing $\tilde{f}_\epsilon$ such that

$$\forall x \in \mathcal{D}, \left| f(x) - \tilde{f}_\epsilon(x) \right| < \epsilon$$

- ϵ of the form 2^p



AMD

C++ Library for fixed-point function evaluation

User types related to fixed-point:

- `FixedFormat<H, L, signedness>` template type to represent fixed-point formats
- `FixedNumber<Format>` to represent a value of a given format

C++ Library for fixed-point function evaluation

User types related to fixed-point:

- `FixedFormat<H, L, signedness>` template type to represent fixed-point formats
- `FixedNumber<Format>` to represent a value of a given format

C++ Library for fixed-point function evaluation

User types related to fixed-point:

- `FixedFormat<H, L, signedness>` template type to represent fixed-point formats

2^H					2^L
b_H	b_{H-1}	$\cdots$	b_{L+2}	b_{L+1}	b_L

```
using format_t = FixedFormat<2,-3, unsigned>;
```

`format_t` represents the fixed-point set

$$\mathcal{D} = \left\{ k \cdot 2^{-3} \mid \forall k \in \left[0 \mathinner{..} 2^6\right) \right\}$$

- `FixedNumber<Format>` to represent a value of a given format

C++ Library for fixed-point function evaluation

User types related to fixed-point:

- `FixedFormat<H, L, signedness>` template type to represent fixed-point formats

  ```
  using format_t = FixedFormat<2,-3, unsigned>;
  ```

  `format_t` represents the fixed-point set

$$\mathcal{D} = \left\{ k \cdot 2^{-3} \mid \forall k \in \left[0 \mathrel{.\,.} 2^6\right) \right\}$$

- `FixedNumber<Format>` to represent a value of a given format

AMD

C++ Library for fixed-point function evaluation

User types related to fixed-point:

- FixedFormat<H, L, signedness> template type to represent fixed-point formats

  ```
  using format_t = FixedFormat<2,-3, unsigned>;
  ```

  format_t represents the fixed-point set

  $$\mathcal{D} = \left\{k \cdot 2^{-3} \mid \forall k \in \left[0 \mathinner{.\,.} 2^{6}\right)\right\}$$

- FixedNumber<Format> to represent a value of a given format

  ```
  FixedNumber<format_t> value{0b010110};
  ```

2^2					2^{-3}
0	1	0	1	1	0

$$10110_2 \cdot 2^{-3} = 22 \cdot 2^{-3} = 2.75$$

C++ Library for fixed-point function evaluation

User types related to fixed-point:

- `FixedFormat<H, L, signedness>` template type to represent fixed-point formats
- `FixedNumber<Format>` to represent a value of a given format

User types related to function description:

- `FreeVariable` represents a free variable of an expression
- `Constant`

C++ Library for fixed-point function evaluation

User types related to fixed-point:

- `FixedFormat<H, L, signedness>` template type to represent fixed-point formats
- `FixedNumber<Format>` to represent a value of a given format

User types related to function description:

- `FreeVariable` represents a free variable of an expression
- `Constant`

C++ Library for fixed-point function evaluation

User types related to fixed-point:

- `FixedFormat<H, L, signedness>` template type to represent fixed-point formats
- `FixedNumber<Format>` to represent a value of a given format

User types related to function description:

- `FreeVariable` represents a free variable of an expression
- `Constant`

AMD

C++ Library for fixed-point function evaluation

User types related to fixed-point:

- `FixedFormat<H, L, signedness>` template type to represent fixed-point formats
- `FixedNumber<Format>` to represent a value of a given format

User types related to function description:

- `FreeVariable` represents a free variable of an expression
- `Constant`

```
FixedNumber<format_t> value; // coming from elsewhere
FreeVariable x{value};
auto f = sin(x * cst::pi + 1.5_cst);
```

- `f` represents the function $x \mapsto \sin(x \cdot \pi + 1.5)$
- `f` can be *evaluated* at an arbitrary precision

C++ Library for fixed-point function evaluation

User types related to fixed-point:

- `FixedFormat<H, L, signedness>` template type to represent fixed-point formats
- `FixedNumber<Format>` to represent a value of a given format

User types related to function description:

- `FreeVariable` represents a free variable of an expression
- `Constant`

```
FixedNumber<format_t> value; // coming from elsewhere
FreeVariable x{value};
auto f = sin(x * cst::pi + 1.5_cst);
```

- `f` represents the function $x \mapsto \sin(x \cdot \pi + 1.5)$
- `f` can be *evaluated* at an arbitrary precision

AMD

C++ Library for fixed-point function evaluation

User types related to fixed-point:

- `FixedFormat<H, L, signedness>` template type to represent fixed-point formats
- `FixedNumber<Format>` to represent a value of a given format

User types related to function description:

- `FreeVariable` represents a free variable of an expression
- `Constant`

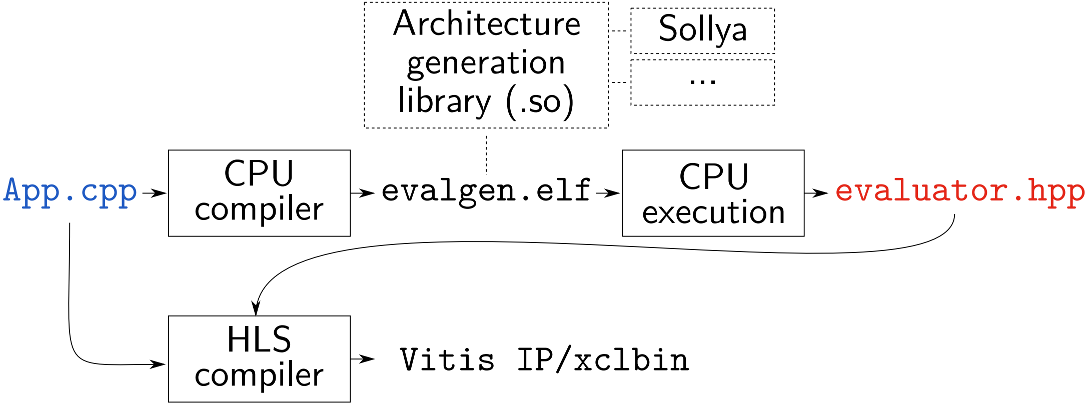
```
FixedNumber<format_t> value; // coming from elsewhere
FreeVariable x{value};
auto f = sin(x * cst::pi + 1.5_cst);
```

- `f` represents the function $x \mapsto \sin\left(x \cdot \pi + 1.5\right)$
- `f` can be *evaluated* at an arbitrary precision

Evaluation of $x \mapsto \sin(x \cdot \pi + 1.5)$ faithful at 2^{-9}

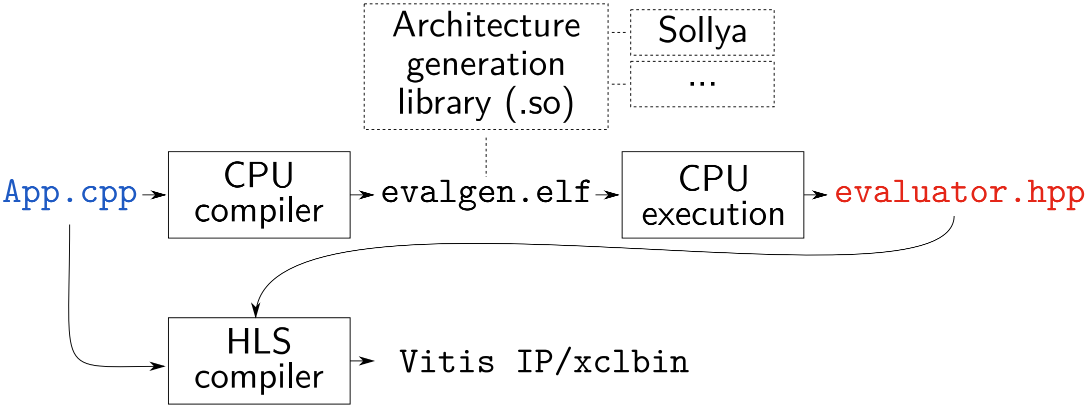
```
using format_t = FixedFormat<2, -3, unsigned>;
auto shifted_sin_pi(FixedNumber<format_t> value) {
  auto x = FreeVariable{value};
  auto f = sin(x * cst::pi + 1.5_cst);
  return f.evaluate<-9>();
}
```

System architecture



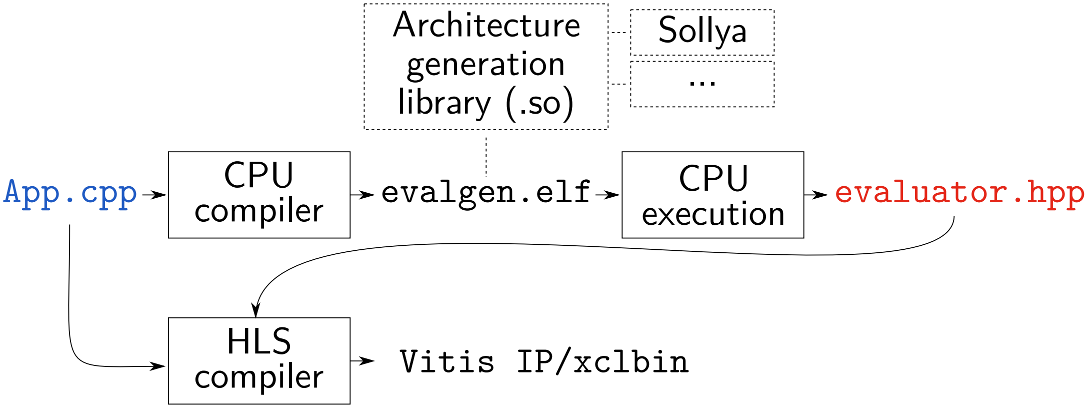
- App.cpp is user-written code
- evaluator.hpp is library-generated specialization header with specialized function evaluation algorithm

System architecture



- App.cpp is user-written code
- evaluator.hpp is library-generated specialization header with specialized function evaluation algorithm

System architecture



- `App.cpp` is user-written code
- `evaluator.hpp` is library-generated specialization header with specialized function evaluation algorithm

AMD

1. Call to `evaluate<...>` delegates to a templated global variable member function
2. The constructor of this variable builds a runtime representation of the function
3. The library interacts with external tools (e.g. Sollya, FloPoCo...) to get an evaluation plan for the function
4. The evaluation plan is serialized in the specialization header `evaluator.hpp`

System architecture

Runtime library internals

1. Call to `evaluate<...>` delegates to a templated global variable member function
2. The constructor of this variable builds a runtime representation of the function
3. The library interacts with external tools (e.g. Sollya, FloPoCo. . .) to get an evaluation plan for the function
4. The evaluation plan is serialized in the specialization header `evaluator.hpp`

1. Call to `evaluate<...>` delegates to a templated global variable member function
2. The constructor of this variable builds a runtime representation of the function
3. The library interacts with external tools (e.g. Sollya, FloPoCo. . .) to get an evaluation plan for the function
4. The evaluation plan is serialized in the specialization header `evaluator.hpp`

System architecture

Runtime library internals

1. Call to `evaluate<...>` delegates to a templated global variable member function
2. The constructor of this variable builds a runtime representation of the function
3. The library interacts with external tools (e.g. Sollya, FloPoCo...) to get an evaluation plan for the function
4. The evaluation plan is serialized in the specialization header `evaluator.hpp`

System architecture

Runtime library internals – computing $x \cdot \pi$

User written code

```
using number_format = FixedFormat<2, -1, unsigned>;
auto mul_by_pi(FixedNumber<number_format> x) {
  return evaluate<-2>(FreeVariable{x} * cst::pi);
}
```

Generated specialization header – specialized evaluator type

```
template<>
struct Evaluator<
BinaryOp<
  NullaryOp<OperationType<OperationKind::PI>>,
  FreeVariable<FixedNumber<FixedFormat<2, -1, unsigned>>>,
  OperationType<OperationKind::MUL>
>, -2> {
  auto evaluate(/*...*/) {/*...*/}
}
```

System architecture

Runtime library internals – computing $x \cdot \pi$

Generated specialization header – chunk of the specialization

```
auto initial_value = [](FixedNumber<FixedFormat<2, 0, unsigned>> p) {
  constexpr auto values = Table<3, FixedFormat<4, -4, unsigned>>{{
    0b000001101_ubi, // Values specific to the function to evaluate
    0b000111111_ubi, // Need external tools to compute them with an
    0b001110001_ubi, // accuracy warranty.
    0b010100011_ubi,
    0b011010110_ubi,
    0b100001000_ubi,
    0b100111010_ubi,
    0b101101100_ubi}};
  auto to_num_key = p.value();
  return values[to_num_key];
  };
```


AMD

Method used in the current library state

- Requires a reference function of high precision (provided by `sollya`)
- Sub-sampling to produce table of initial values (TIV)
- Differences between TIV and values stored in Offset Table (TO)
- Offset sharing reduces the number of TO elements
- Iterating to find smallest table under accuracy constraint

Function approximation methods

Bipartite approximation

Method used in the current library state

- Requires a reference function of high precision (provided by `sollya`)
- Sub-sampling to produce table of initial values (TIV)
- Differences between TIV and values stored in Offset Table (TO)
- Offset sharing reduces the number of TO elements
- Iterating to find smallest table under accuracy constraint

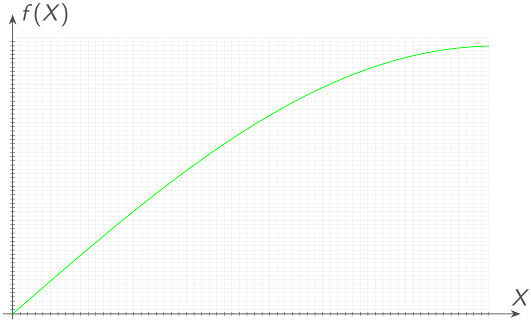


Figure: The function to approximate

AMD

Method used in the current library state

- Requires a reference function of high precision (provided by `sollya`)
- Sub-sampling to produce table of initial values (TIV)
- Differences between TIV and values stored in Offset Table (TO)
- Offset sharing reduces the number of TO elements
- Iterating to find smallest table under accuracy constraint

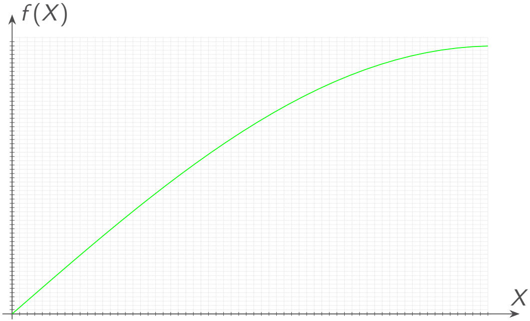


Figure: The function to approximate

Method used in the current library state

- Requires a reference function of high precision (provided by `sollya`)
- Sub-sampling to produce table of initial values (TIV)
- Differences between TIV and values stored in Offset Table (TO)
- Offset sharing reduces the number of TO elements
- Iterating to find smallest table under accuracy constraint

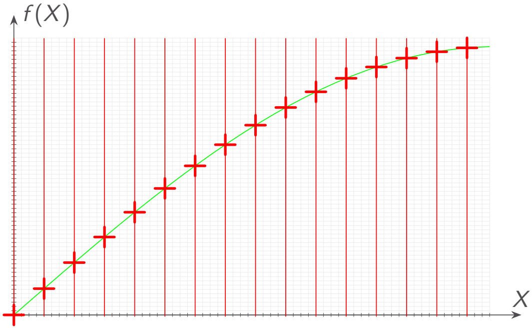


Figure: Materializing the sampling

Method used in the current library state

- Requires a reference function of high precision (provided by `sollya`)
- Sub-sampling to produce table of initial values (TIV)
- Differences between TIV and values stored in Offset Table (TO)
- Offset sharing reduces the number of TO elements
- Iterating to find smallest table under accuracy constraint

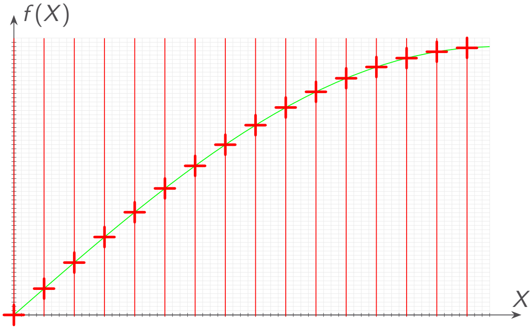


Figure: Materializing the sampling

Method used in the current library state

- Requires a reference function of high precision (provided by `sollya`)
- Sub-sampling to produce table of initial values (TIV)
- Differences between TIV and values stored in Offset Table (TO)
- Offset sharing reduces the number of TO elements
- Iterating to find smallest table under accuracy constraint

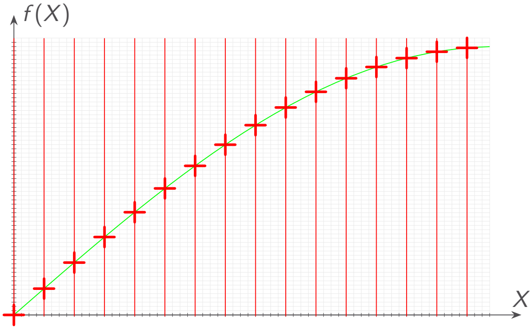


Figure: Materializing the sampling

Method used in the current library state

- Requires a reference function of high precision (provided by `sollya`)
- Sub-sampling to produce table of initial values (TIV)
- Differences between TIV and values stored in Offset Table (TO)
- Offset sharing reduces the number of TO elements
- Iterating to find smallest table under accuracy constraint

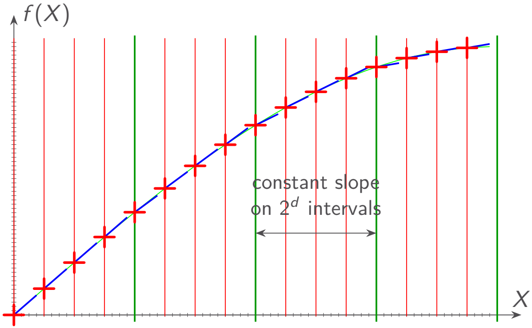


Figure: The bipartite approximation

AMD

Function approximation methods

Bipartite approximation

Method used in the current library state

- Requires a reference function of high precision (provided by `sollya`)
- Sub-sampling to produce table of initial values (TIV)
- Differences between TIV and values stored in Offset Table (TO)
- Offset sharing reduces the number of TO elements
- Iterating to find smallest table under accuracy constraint

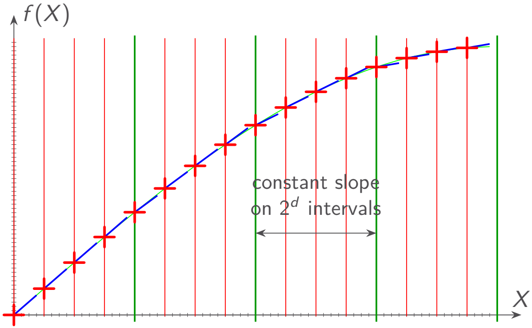


Figure: The bipartite approximation

Function approximation methods

Bipartite approximation

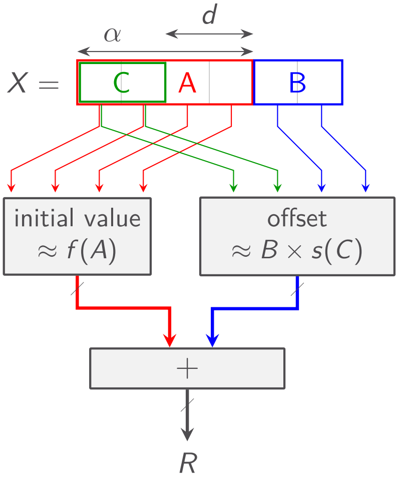


Figure: Bipartite architecture

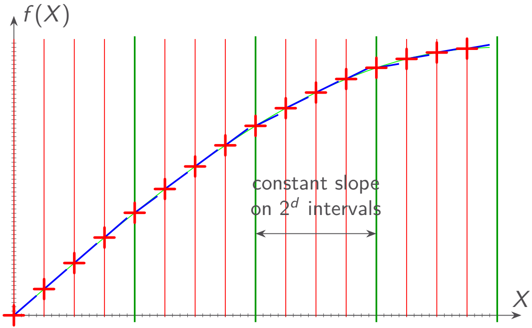


Figure: The bipartite approximation

Function approximation
Other methods

- Bipartite method not state of the art
- Multipartite methods, post processing of TIV
- Attempts of ILP models for simultaneous compression, decomposition and correct rounding under development
- For bigger functions, range reduction, polynomial evaluation
- Specific optimized operators for specific functions

- Bipartite method not state of the art
- Multipartite methods, post processing of TIV
- Attempts of ILP models for simultaneous compression, decomposition and correct rounding under development
- For bigger functions, range reduction, polynomial evaluation
- Specific optimized operators for specific functions

Function approximation
Other methods

- Bipartite method not state of the art
- Multipartite methods, post processing of TIV
- Attempts of ILP models for simultaneous compression, decomposition and correct rounding under development
- For bigger functions, range reduction, polynomial evaluation
- Specific optimized operators for specific functions

- Bipartite method not state of the art
- Multipartite methods, post processing of TIV
- Attempts of ILP models for simultaneous compression, decomposition and correct rounding under development
- For bigger functions, range reduction, polynomial evaluation
- Specific optimized operators for specific functions

Function approximation

Other methods

- Bipartite method not state of the art
- Multipartite methods, post processing of TIV
- Attempts of ILP models for simultaneous compression, decomposition and correct rounding under development
- For bigger functions, range reduction, polynomial evaluation
- Specific optimized operators for specific functions

LNS Adder

Simplified LNS adder code

```
// lns_t is a specialization of FixedNumber
lns_t LNSAdder(lns_t op1, lns_t op2) {
  auto min_op = min(op1, op2);
  auto max_op = max(op1, op2);
  lns_t diff_op = min_op - max_op;
  auto diff_fv = FreeVariable { diff_op };
  auto f = log2(1._cst + pow(2._cst, diff_fv));
  auto rounded_f = f + lns_t::rounding_constant;
  auto result_op_diff = evaluate<lns_t::add_prec>(rounded_f);
  return max_op + result_op_diff;
}
```

Toy applications

Additive audio synthesis

Toy applications
Additive audio synthesis

Synthesized signal y

$$y(t) = \sum_{k=1}^{K} r_k(t) \sin(2\pi f_k t + \phi_k)$$

- $\phi_k = 0$ and $K = 256$, $f_k = 1000 \cdot k/256$
- Reference experiment with all computations in `float`
- Fixed-point experiments:
  - r_k on 8 bits
  - evaluation of the sinus faithful at 2^{-6}, 2^{-8}, 2^{-16} and 2^{-22}

Synthesized signal y

$$y(t) = \sum_{k=1}^{K} r_k(t) \sin(2\pi f_k t + \phi_k)$$

- $\phi_k = 0$ and $K = 256$, $f_k = 1000 \cdot k/256$
- Reference experiment with all computations in `float`
- Fixed-point experiments:
  - r_k on 8 bits
  - evaluation of the sinus faithful at 2^{-6}, 2^{-8}, 2^{-16} and 2^{-22}

Synthesized signal y

$$y(t) = \sum_{k=1}^{K} r_k(t) \sin(2\pi f_k t + \phi_k)$$

- $\phi_k = 0$ and $K = 256$, $f_k = 1000 \cdot k/256$
- Reference experiment with all computations in `float`
- Fixed-point experiments:
  - r_k on 8 bits
  - evaluation of the sinus faithful at 2^{-6}, 2^{-8}, 2^{-16} and 2^{-22}

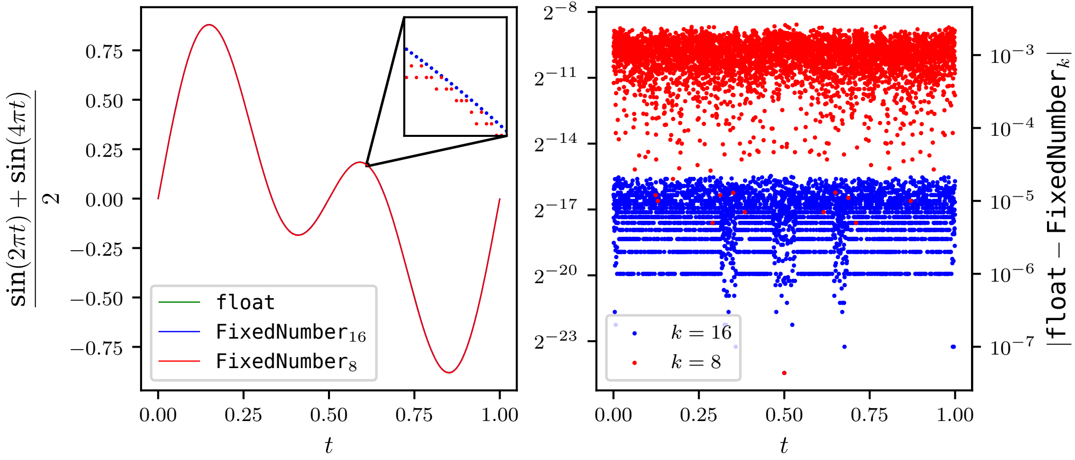
Synthesized signal y

$$y(t) = \sum_{k=1}^{K} r_k(t) \sin(2\pi f_k t + \phi_k)$$

- $\phi_k = 0$ and $K = 256$, $f_k = 1000 \cdot k/256$
- Reference experiment with all computations in `float`
- Fixed-point experiments:
  - r_k on 8 bits
  - evaluation of the sinus faithful at 2^{-6}, 2^{-8}, 2^{-16} and 2^{-22}

Toy applications

Additive audio synthesis



Toy applications
Additive audio synthesis

Experiment		Area				Timing		
Pipeline	Format	LUT	Flip-flop	DSP	BRAM	Latency	Critical path (ns)	II
Unpipelined	`float`	12389	15222	175	0	653	2.787	-
	$\texttt{FixedNumber}_6$	7932	6985	256	129	279	2.907	-
	$\texttt{FixedNumber}_8$	8882	7284	256	257	297	2.901	-
	$\texttt{FixedNumber}_{16}$	5777	6917	256	513	284	2.995	-
	$\texttt{FixedNumber}_{22}$	6551	7304	256	769	270	2.945	-
Pipelined	`float`	141491	166723	1304	0	226	2.995	128
	$\texttt{FixedNumber}_6$	7442	4977	256	128	5	2.803	1
	$\texttt{FixedNumber}_8$	7568	6500	256	256	5	2.701	1
	$\texttt{FixedNumber}_{16}$	4513	8806	256	512	8	2.806	1
	$\texttt{FixedNumber}_{22}$	5627	9385	256	768	8	3.074	1

Table: Area and timing metrics comparison between `float` and various `FixedNumber` for an additive synthesizer of 256 oscillators (target `xcvu13p-fhga2104-3-e`, clock period 3 ns)

AMD

Takeaways on this library

- Relatively simple flow for arbitrary function evaluation in HLS
- Many more custom arithmetic blocks could be integrated
  - Tiled/truncated multipliers
  - Bitheaps
  - Better function evaluator strategies
  - Filters. . .
- Possible usability improvement by hiding the tool interaction inside the compiler itself
- Might be interesting to use MLIR to go from high level function representation to implementation

Takeaways on this library

- Relatively simple flow for arbitrary function evaluation in HLS
- Many more custom arithmetic blocks could be integrated
  - Tiled/truncated multipliers
  - Bitheaps
  - Better function evaluator strategies
  - Filters...
- Possible usability improvement by hiding the tool interaction inside the compiler itself
- Might be interesting to use MLIR to go from high level function representation to implementation

Takeaways on this library

- Relatively simple flow for arbitrary function evaluation in HLS
- Many more custom arithmetic blocks could be integrated
  - Tiled/truncated multipliers
  - Bitheaps
  - Better function evaluator strategies
  - Filters...
- Possible usability improvement by hiding the tool interaction inside the compiler itself
- Might be interesting to use MLIR to go from high level function representation to implementation

Takeaways on this library

- Relatively simple flow for arbitrary function evaluation in HLS
- Many more custom arithmetic blocks could be integrated
  - Tiled/truncated multipliers
  - Bitheaps
  - Better function evaluator strategies
  - Filters...
- Possible usability improvement by hiding the tool interaction inside the compiler itself
- Might be interesting to use MLIR to go from high level function representation to implementation

Takeaways on this library

- Relatively simple flow for arbitrary function evaluation in HLS
- Many more custom arithmetic blocks could be integrated
  - Tiled/truncated multipliers
  - Bitheaps
  - Better function evaluator strategies
  - Filters...
- Possible usability improvement by hiding the tool interaction inside the compiler itself
- Might be interesting to use MLIR to go from high level function representation to implementation

The terrible library's secret

The library uses many modern C/C++ constructs!

- Concepts
- extended `constexpr` possibilities
- `_BitInt()`
- ...

How to compile it with our HLS tools?

The terrible library's secret

The library uses many modern C/C++ constructs!

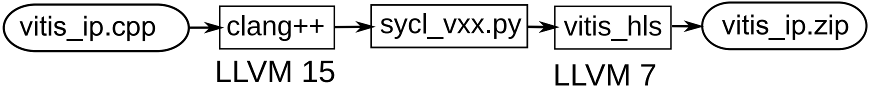
- Concepts
- extended `constexpr` possibilities
- `_BitInt()`
- ...

How to compile it with our HLS tools?

Compiler overview

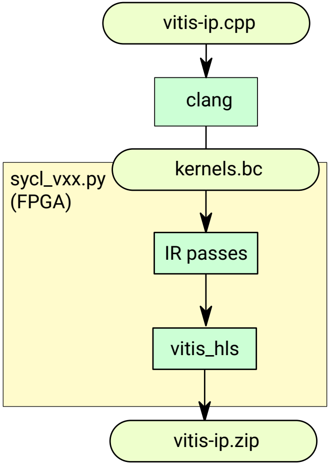
- The library uses C++20
- C++20 compiler for Vitis IPs integrable in Vivado block design
- One command line:

```
clang++ -target=vitis_ip-xilinx vitis_ip.cpp -vitis-ip-part=part-id -o vitis_ip.zip
```

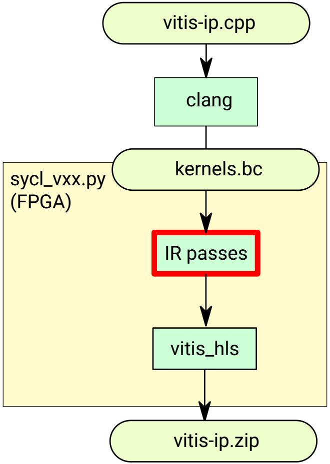


- clang++ is a fork based on the latest upstream
- clang++ 15 has _BitInt for free!

Compiler for Vitis IP



LLVM IR passes used in `sycl_vxx.py`



- Basic optimization
- Promote memory to register
  - LLVM 15 version is better than HLS LLVM 7 one!
- Lower `memcpy` to load & store
- Extension lowering
- Decoration property generation: pipes, interface...
- Downgrading LLVM IR 15→7

Downgrading LLVM IR

- LLVM IR is kept upward-compatible but not downward-compatible ☹
- So any version of LLVM can consume previous versions easily

How is downgrading done:

- Run a pass to remove or rewrite constructs that don't exist in the LLVM IR 7
  - remove or rewrite Attributes, Intrinsics and Instructions
- Run modified version of the IR printer
- Assemble with Vitis's `llvm-as`

Limitation:

- Not all semantic in LLVM 15 can be expressed in LLVM 7
- Some transformations that might cause miscompiles
- The gap keeps growing

Downgrading IR is not sustainable ☹

Unexpected users of `sycl_vxx.py`

- Many advanced users seem to try feeding latest LLVM IR (15) to HLS
- We have users of `sycl_vxx.py` at PNNL

- And SYCL...

Pacific Northwest
NATIONAL LABORATORY

fyquah commented on Mar 18, 2021

I am thinking of working on a tool that generate a custom HLS front-end which generates HLS-friendly LLVM-bitcode, and leverage plugins + vitis_hls backend for optimization passes and RTL generation.

How can I pass the generated bit files into the HLS compilation flow?

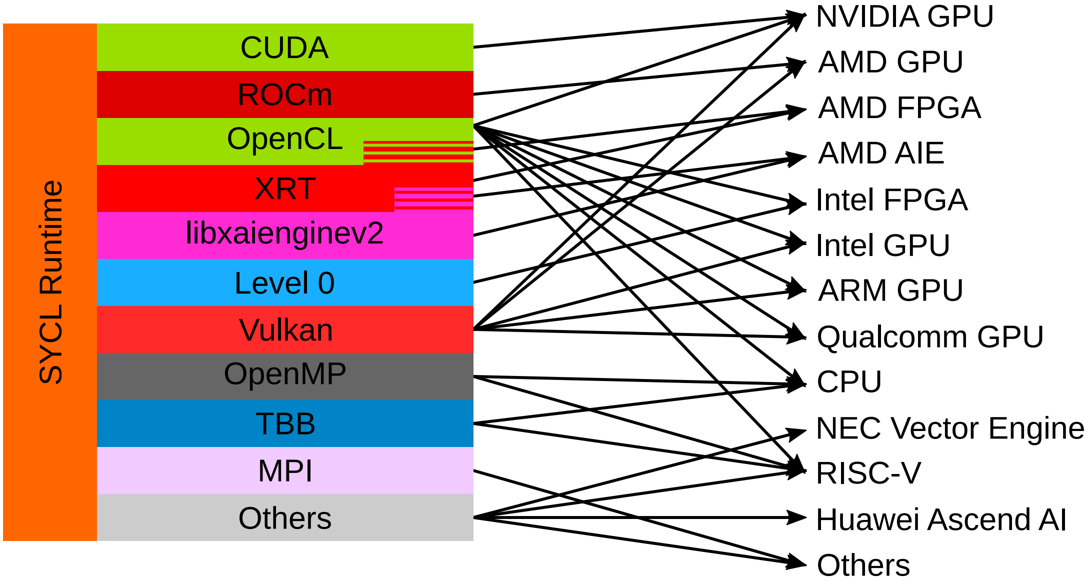
Mario Ruiz Noguera 1 day ago

Thank you all for the info. Now customer wants to know the road map to use LLVM 12 or newer.

SYCL

- Khronos Group standard using modern C++ for heterogeneous programming
  - Single-source for simplicity and safety
  - CPU emulation & debug for free since it is pure C++
  - Support any back-end of any vendor at the same time: CUDA, OpenCL, HIP, Level0...
  - Bidirectional interoperability mode: CUDA or OpenCL can use SYCL, SYCL can use OpenMP, HIP or XRT code...
- A dozen of implementations, with 3 more serious ones
  - Codeplay ComputeCpp, mainly targeting embedded systems nowadays
  - hipSYCL (open-source github.com/illuhad/hipSYCL) for any GPU/CPU
  - Intel oneAPI DPC++ (open-source github.com/intel/llvm) for almost anything

Typical SYCL stack



SYCL for FPGA

- Intel oneAPI SYCL DPC++ for Intel FPGA (open-source product)
- Codeplay ComputeCpp for AMD FPGA (not announced yet)
- AMD SYCL for Vitis for AMD FPGA & AIE (open-source research prototype github.com/triSYCL/sycl)

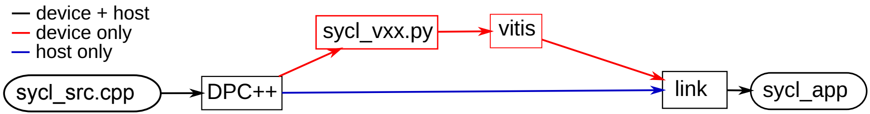
Specificities of a SYCL compiler

- Our SYCL Compiler for AMD/Xilinx FPGAs
- Fork of DPC++ from Intel
- Single-source and single-binary
  - Feels like normal CPU programming
  - Simpler to use
  - Safer
  - Single source of truth
- Direct programming
  - Fine grain control
  - Performance

Specificities of a SYCL compiler

- Our SYCL Compiler for AMD/Xilinx FPGAs
- Fork of DPC++ from Intel
- Single-source and single-binary
  - Feels like normal CPU programming
  - Simpler to use
  - Safer
  - Single source of truth
- Direct programming
  - Fine grain control
  - Performance

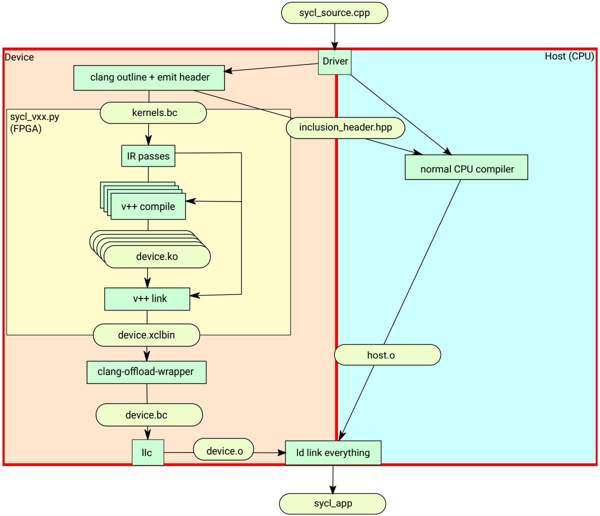
Overview



Same tools as for the Vitis IP compiler

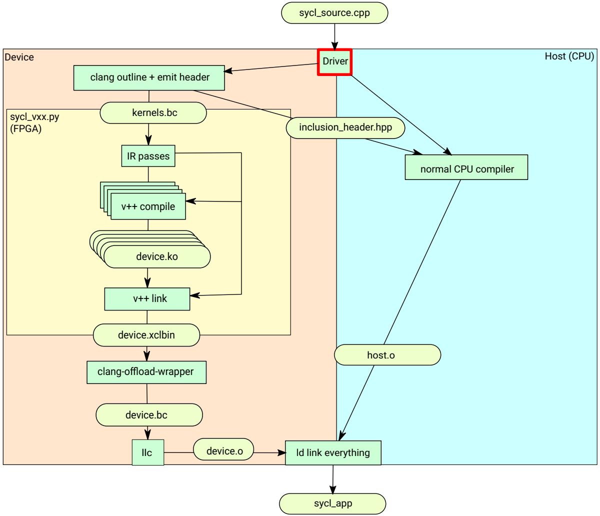
- The device path is shared with the Vitis IP compiler

Detailed view



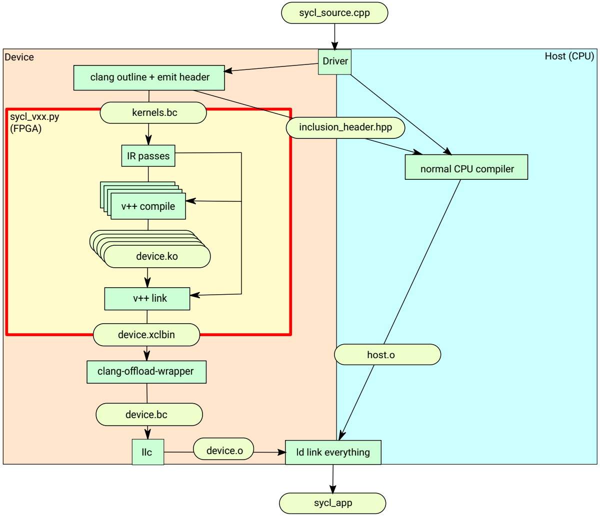
- Host + Device
- Driver hides the complexity
- Still uses sycl_vxx.py
- Still uses the same IR passes
- But uses v++

Detailed view



- Host + Device
- `Driver` hides the complexity
- Still uses `sycl_vxx.py`
- Still uses the same IR passes
- But uses `v++`

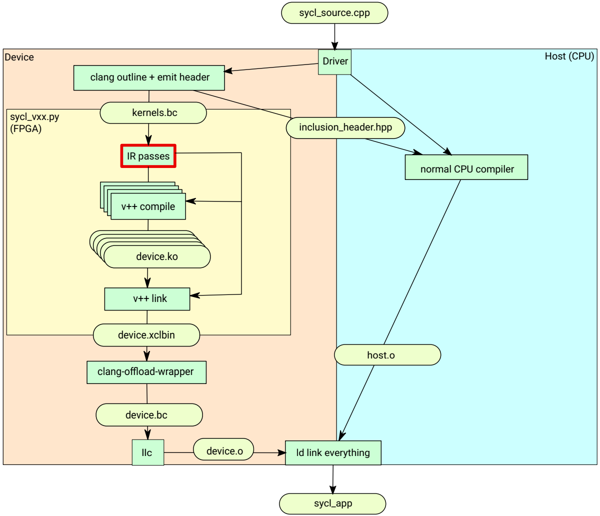
Detailed view



- Host + Device
- `Driver` hides the complexity
- Still uses `sycl_vxx.py`
- Still uses the same IR passes
- But uses `v++`

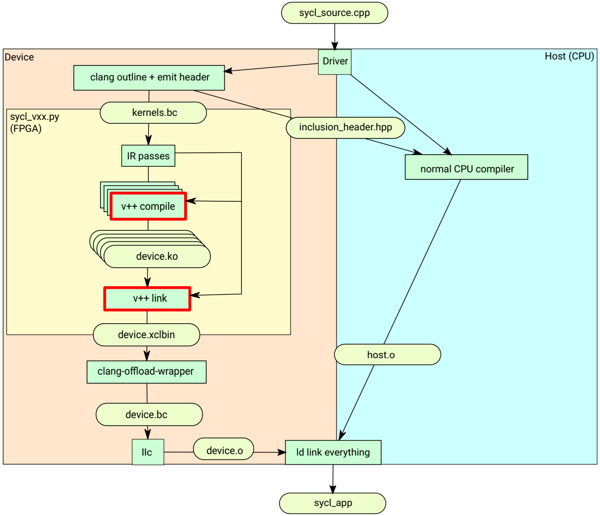
AMD

Detailed view



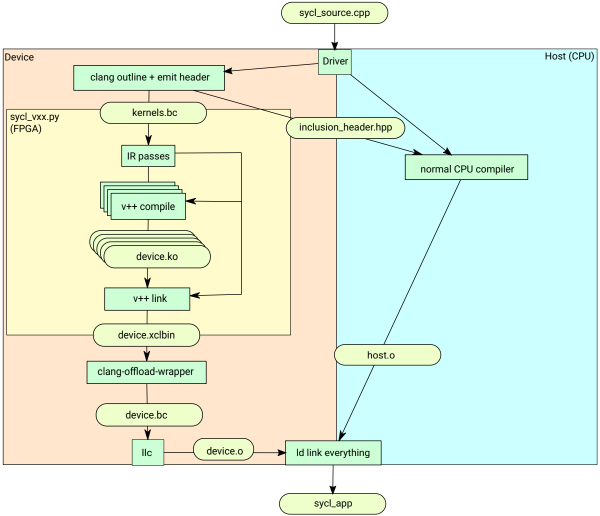
- Host + Device
- `Driver` hides the complexity
- Still uses `sycl_vxx.py`
- Still uses the same IR passes
- But uses `v++`

Detailed view



- Host + Device
- `Driver` hides the complexity
- Still uses `sycl_vxx.py`
- Still uses the same IR passes
- But uses `v++`

Single-source & single-binary

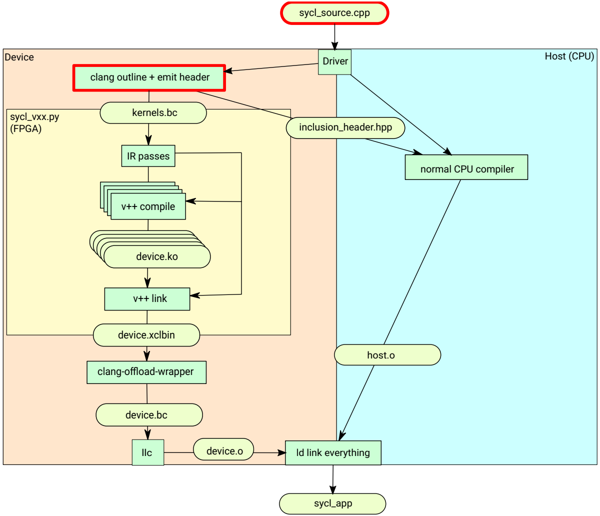



```
// Some host code here
cgh.single_task([=] {
  // Some device code here
});
// Back to host code
```

- Single source
  - Internal attribute
  - Device frontend
  - Host frontend
- Single binary
  - `clang-offload-wrapper`
  - Inclusion header

AMD

Single-source & single-binary

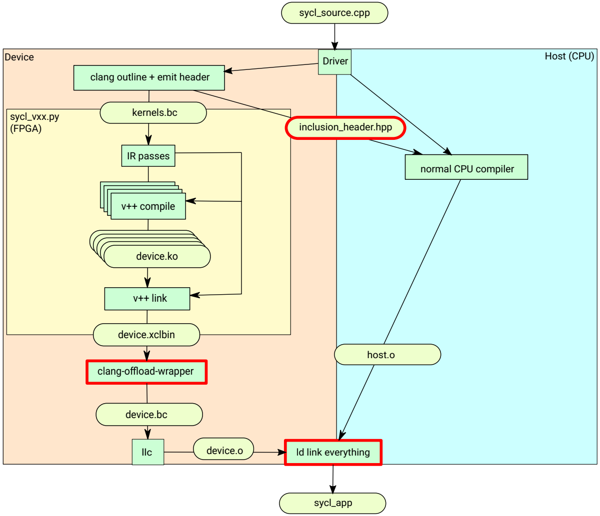



```
// Some host code here
cgh.single_task([=] {
  // Some device code here
});
// Back to host code
```

- Single source
  - Internal attribute
  - Device frontend
  - Host frontend
- Single binary
  - `clang-offload-wrapper`
  - Inclusion header

AMD

Single-source & single-binary




```
// Some host code here
cgh.single_task([=] {
  // Some device code here
});
// Back to host code
```

- Single source
  - Internal attribute
  - Device frontend
  - Host frontend
- Single binary
  - `clang-offload-wrapper`
  - Inclusion header

AMD

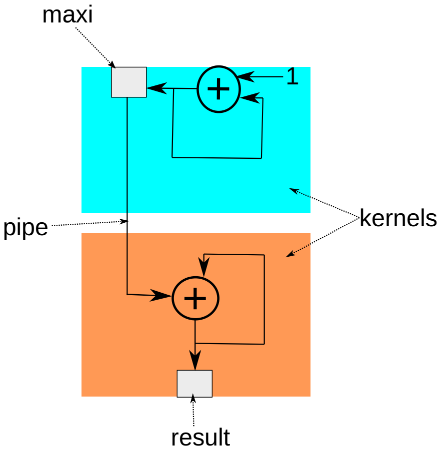
Pipeline and dataflow

```
h.single_task(pipeline_kernel([=]{/*Device code*/}));
/// like HLS's "#pragma HLS pipeline" in a kernel
h.single_task(pipeline_kernel<constrained_ii<4>>([=]{/*Device code*/}));
/// like HLS's "#pragma HLS pipeline II=4" in a kernel
h.single_task(dataflow_kernel([=]{/*Device code*/}));
/// like HLS's "#pragma HLS dataflow" in a kernel
for (...)
  pipeline<auto_ii, rewind_pipeline, flushable>([&]{/*loop body*/});
/// like HLS's "#pragma HLS pipeline enable_flush rewind" on a loop
for (...)
  dataflow([&] {/* Loop body executed in dataflow mode */});
/// like HLS's "#pragma HLS dataflow" on a loop
```

- Can express everything `#pragma HLS pipeline` or `#pragma HLS dataflow` can

Pipes

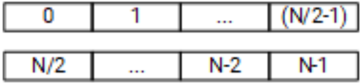
```
using MyHLSstream = pipe<class MyID, int>;
// declare a pipe of ints

cgh.single_task([=] {
  for (int i = 0; i < size; i++)
    MyHLSstream::write(acc_a[i]);
});
//...
cgh.single_task([=] {
  for (int i = 0; i < size; i++)
    result += MyHLSstream::read();
});
```



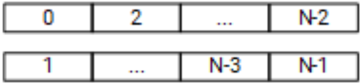
- Same API as Intel's SYCL static pipe extension
- No extra configuration, just C++

Partition Array

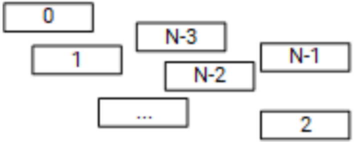
```cpp
/// equivalent to int array1[6]
partition_array<int, 6, block<2>> array1;
```




```cpp
/// equivalent to float array2[10]
partition_array<float, 10, cyclic<2>> array2;
```




```cpp
/// equivalent to long array3[4]
partition_array<long, 4, complete<>> array3;
```



- Multi-dimensional

Memory banks

```
sycl::buffer<int, 1> a;
sycl::buffer<int, 1> b;

q.submit([&](sycl::handler &cgh) {
  sycl::accessor acc_a(a, cgh, accessor_property_list{ddr_bank<1>});
  sycl::accessor acc_b(b, cgh, accessor_property_list{hbm_bank<1>});
  cgh.single_task([=]{
    for (int i = 0; i < size; i++)
      acc_b += acc_a;
  });
});
```

- Support HBM and DDR banks
- No extra configuration, just C++

How are extensions implemented

How are extensions processed:

- C++ wrapper with syntax
- Internal generic attribute
- For memory banks only: the frontend generate the generic attribute
- Emit generic information in IR
- IR passes create the correct IR for HLS and configuration for `v++`
- Reuses Vitis and HLS

Why:

- The frontend is rarely involved
- The syntax is handled is the SYCL C++ library
- Minimal conflicts with upstream

Tooling

- Users only write C++ (no compiler extensions)
- Internally compiler features are used but with fallbacks
- Users can use all their usual source tools
  - IDE & editors (auto-completion on FPGA properties!)
  - Static analysis
  - Refactoring tools
  - any other...
- Single-command to build an application:
  `clang++ -fsycl -fsycl-target=fpga64_hls_hw src.cpp -o app`
- Users can use all the usual build systems on top of it

Conclusion

- Successful PoC on using modern Clang/LLVM as HLS front-end
- *Pis aller* for fulfilling "strong" demand for HLS with modern LLVM IR (PNNL, . . .)
- Enable "HLS IP library" using modern C++, help bridging the gap between RTL and HLS
- Building on Intel oneAPI DPC++ to bring SYCL for AMD FPGA (+other devices)
  - Single-source
  - Standard programming model
  - Allow the same flexibility as HLS `#pragma`

One shared goal: make the programming of FPGAs easier!

Future work:

- Advance on SYCL for AIE, for full SYCL programmability on ACAP
- Investigate possible usage of MLIR
- Think about SYCL extensions to handle FPGAs I/O in embedded settings (Zynq MPSoC)
- Many possibilities of "HLS IP" libraries