RAIM 2024 Perpignan

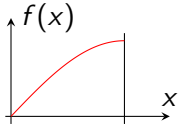
Lossless Differential Table Compression for Hardware Function Evaluation

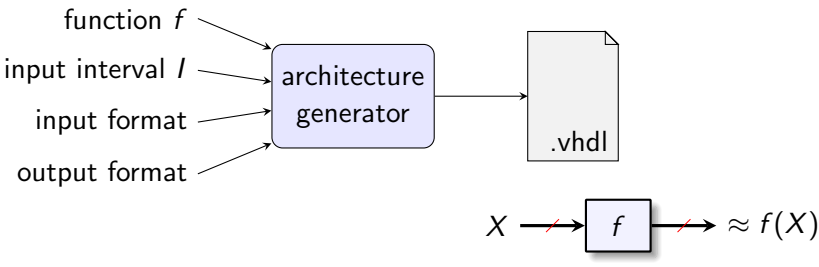
Maxime Christ, Luc Forget, Florent de Dinechin

Outline

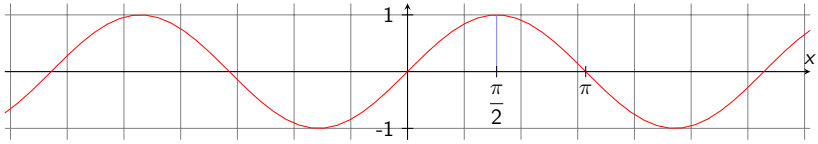
Generalities on fixed-point functions

Generators of hardware approximators for generic fixed-point functions

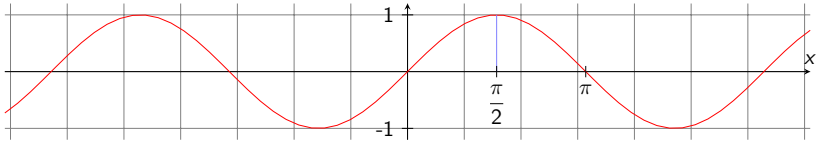




Function and input interval, illustrated by the sine function

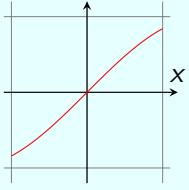


Function and input interval, illustrated by the sine function

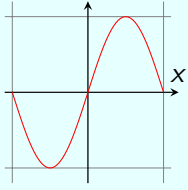


Simplifying choice in FloPoCo: input interval is $[0, 1)$ or $[-1, 1)$

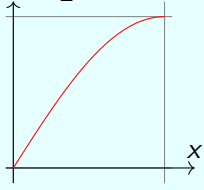
$\sin(x)$ on $[-1, 1)$



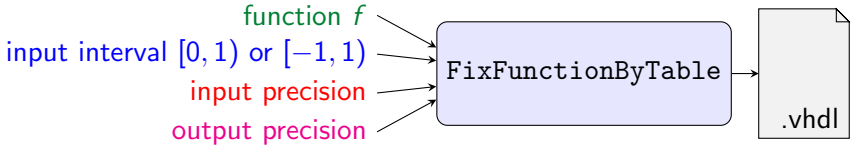
$\sin(\pi x)$ on $[-1, 1)$



$\sin(\frac{\pi}{2}x)$ on $[0, 1)$

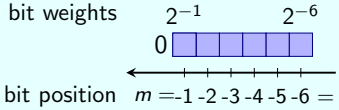


FixFunctionByTable



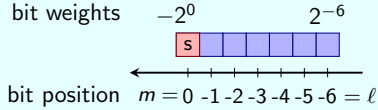

```
flopoco FixFunctionByTable f="sin(pi/2*x)" signedIn=false lsbIn=-6 lsbOut=-6
```

Input fixed-point format in $[0,1)$



Only the LSB parameter ℓ needs to be specified

Fixed-point format in [-1,1)

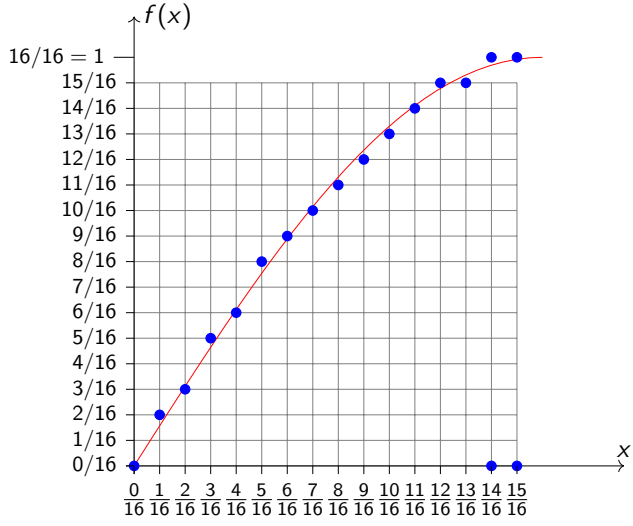


No output MSB nor output signedness: they can be computed by the tool.

Let's look at the VHDL...

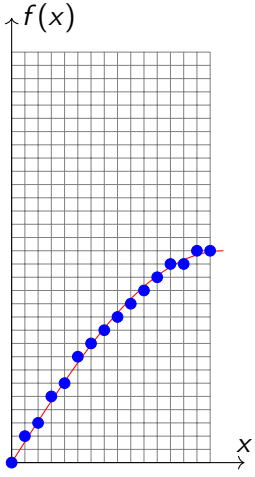
Discretization issues

Inputs and outputs in $[0, 1)$ with 4-bit fixed-point:



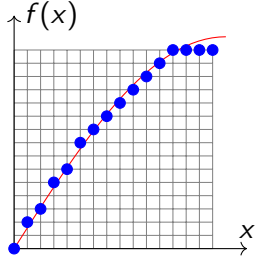
Possible fixes for discretization issues

Using 1 bit more



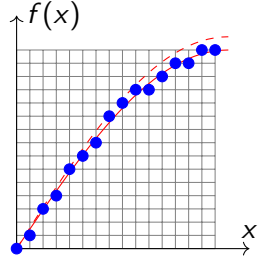
expensive (1 bit lost)

saturating



expensive (1 mux)

Scaling by $\frac{15}{16}$



free

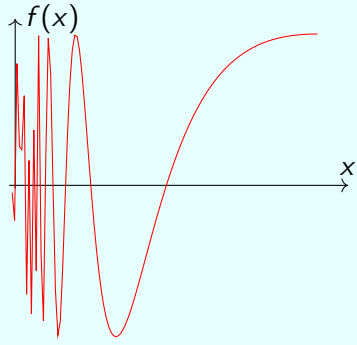
FixFunctionByTable, fixed

```
flopoco FixFunctionByTable f="63/64*sin(pi/2*x)" signedIn=0 lsbIn=-6 lsbOut=-6
```

Go check the VHDL...

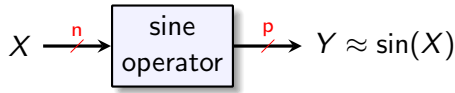
Tables can hold functions that are arbitrarily ugly

$\sin(\frac{\pi}{2x})$ on $[0, 1)$




```
flopoco FixFunctionByTable f="(1 - 1b-16) * sin(pi/2/x)" signedIn=0 lsbIn=-16
    lsbOut=-16
```

Hardware cost of plain tables



The simplest solution: plain tabulation

2^n entries of p bits each, so $2^n \times p$ bits

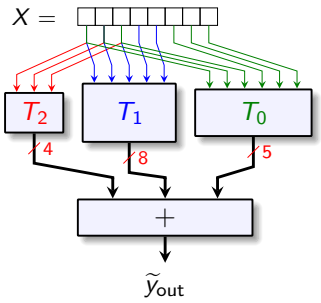
- very good for really small precisions
  - using 6-input LUTs: $n = p = 6$ costs 6 LUTs
  - using BlockRAMs of e.g. 10Kbits ($2^{10} \times 10$): $n = p = 10$ costs 1 BlockRAM
- for larger precisions, cost grows exponentially in n

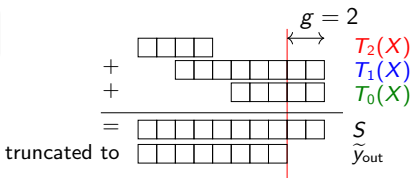
address	content
00000	00000
00001	00011
00010	00011
00011	00101
...	...
11101	11111
11110	11111
11111	11111

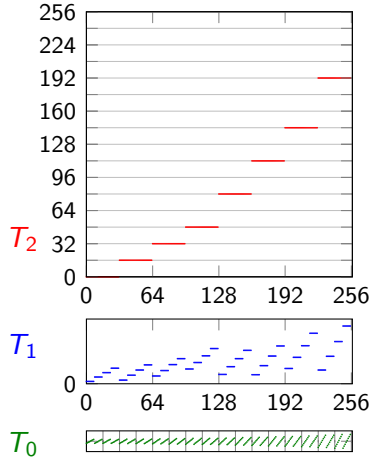
2^n (address rows) — p (content)

When simple solutions don't scale, use clever solutions

Multipartite table architectures





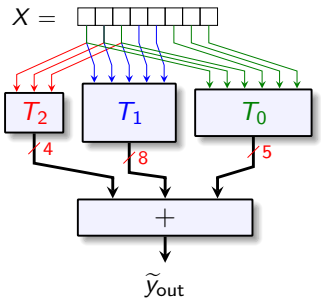


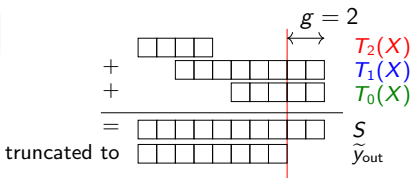
When simple solutions don't scale, use clever solutions

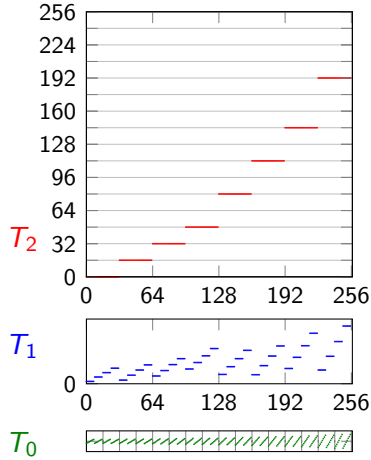
Multipartite table architectures

For this talk, please accept the magic:

let us just wonder that it works, here for $n = p = 8$.





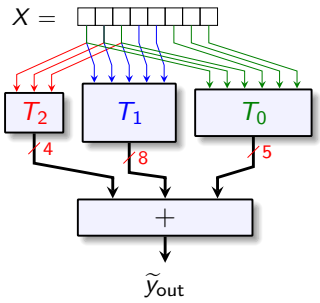


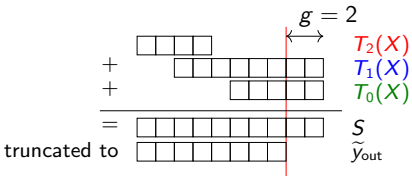
When simple solutions don't scale, use clever solutions

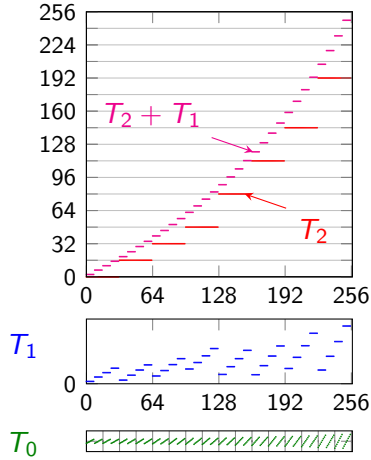
Multipartite table architectures

For this talk, please accept the magic:

let us just wonder that it works, here for $n = p = 8$.





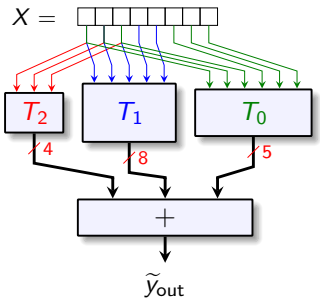


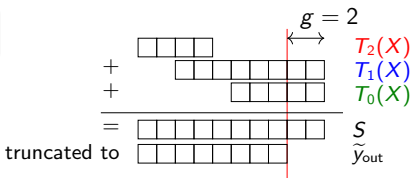
When simple solutions don't scale, use clever solutions

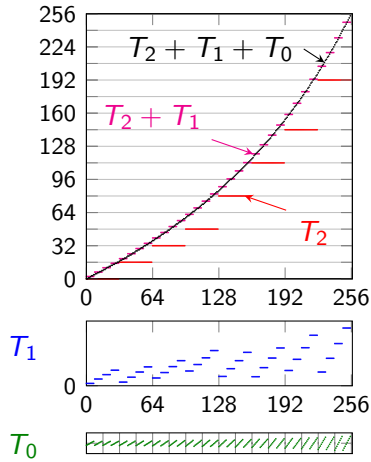
Multipartite table architectures

For this talk, please accept the magic:

let us just wonder that it works, here for $n = p = 8$.





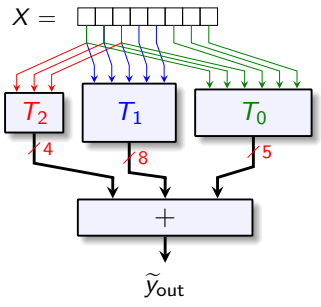


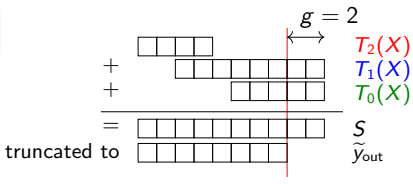
When simple solutions don't scale, use clever solutions

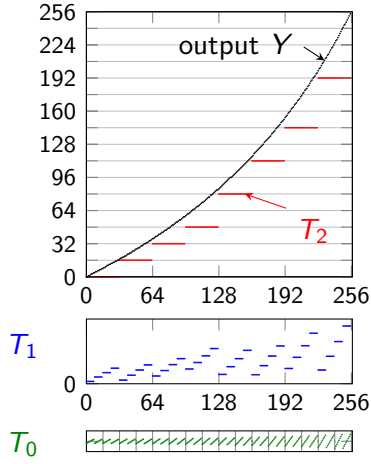
Multipartite table architectures

For this talk, please accept the magic:

let us just wonder that it works, here for $n = p = 8$.





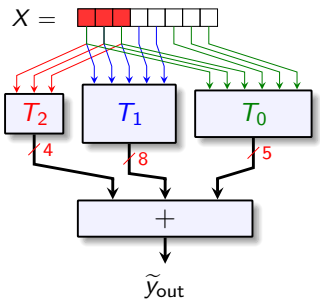


When simple solutions don't scale, use clever solutions

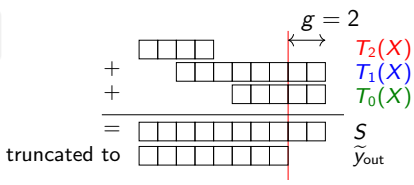
Multipartite table architectures

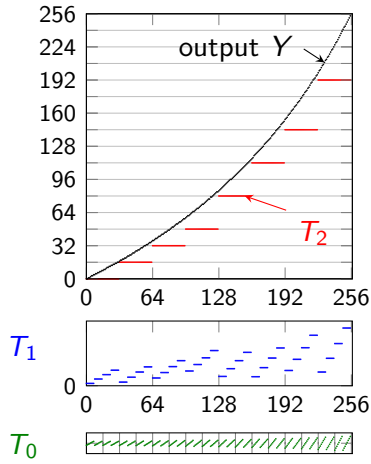
For this talk, please accept the magic:

let us just wonder that it works, here for $n = p = 8$.



$2^3 \cdot 4$



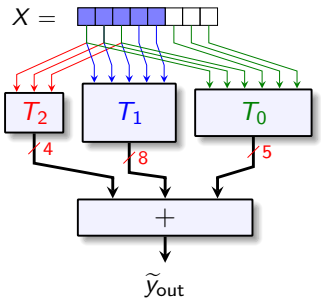


When simple solutions don't scale, use clever solutions

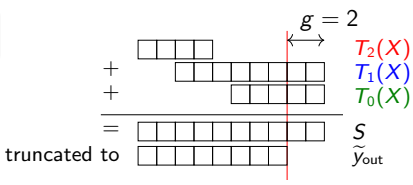
Multipartite table architectures

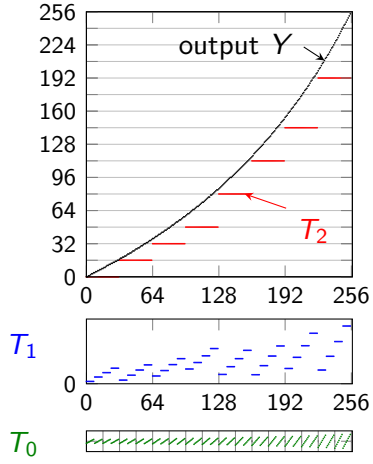
For this talk, please accept the magic:

let us just wonder that it works, here for $n = p = 8$.



$2^3 \cdot 4 + 2^5 \cdot 8$



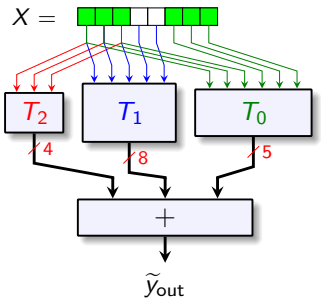


When simple solutions don't scale, use clever solutions

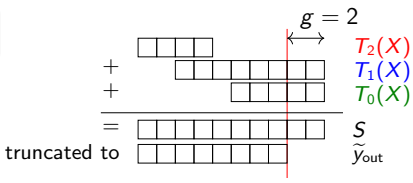
Multipartite table architectures

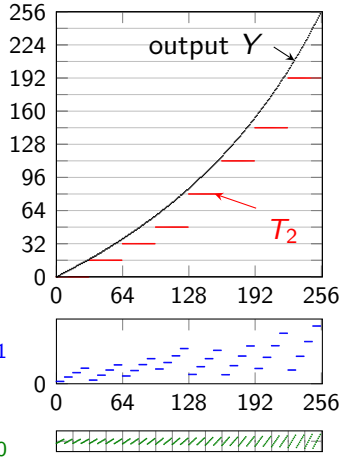
For this talk, please accept the magic:

let us just wonder that it works, here for $n = p = 8$.



$2^3 \cdot 4 + 2^5 \cdot 8 + 2^6 \cdot 5$



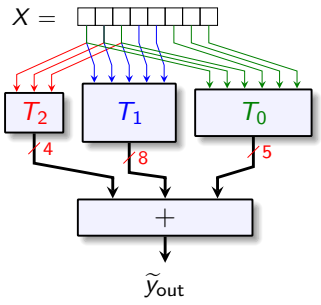


When simple solutions don't scale, use clever solutions

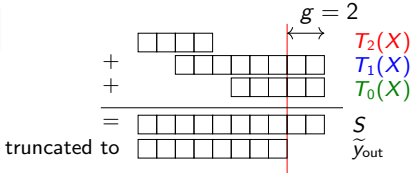
Multipartite table architectures

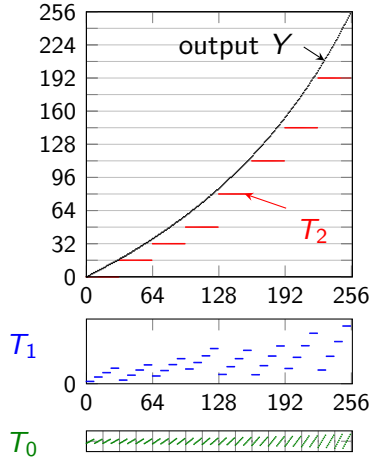
For this talk, please accept the magic:

let us just wonder that it works, here for $n = p = 8$.



$$2^3 \cdot 4 + 2^5 \cdot 8 + 2^6 \cdot 5 < 2^8 \cdot 8$$



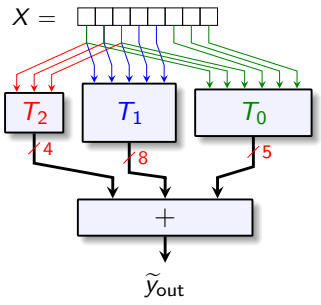


When simple solutions don't scale, use clever solutions

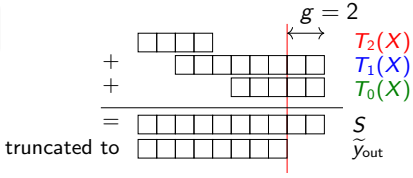
Multipartite table architectures

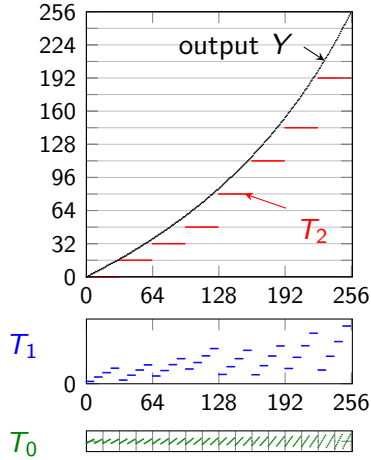
For this talk, please accept the magic:

let us just wonder that it works, here for $n = p = 8$.



$$2^3 \cdot 4 + 2^5 \cdot 8 + 2^6 \cdot 5 < 2^8 \cdot 8$$

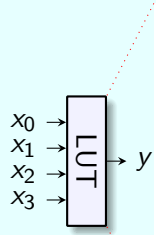




- grows (roughly) as $2^{n/2} \times p$ instead of $2^n \times p$ (good for 8 to 24 bits)
- (there are other methods for even larger precisions, but they need multipliers)

Hardware cost of plain tables on FPGAs

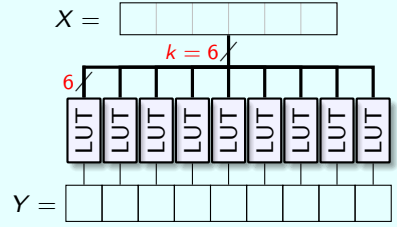
FPGAs are LUT-based



x_0	x_1	x_2	x_3	y
0	0	0	0	0
0	0	0	1	1
0	0	1	0	1
0	0	1	1	1
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	1
1	0	0	0	1
1	0	0	1	1
1	0	1	0	0
1	0	1	1	1
1	1	0	0	1
1	1	0	1	0
1	1	1	0	0
1	1	1	1	1

Practical sizes on FPGAs with k-input LUTs

- A table of $2^k \times m$ bits costs exactly m LUTs.

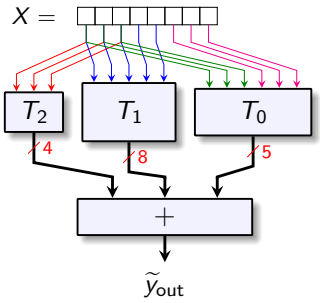


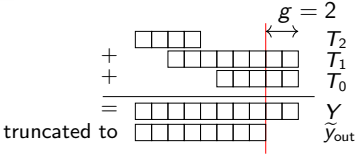
- $2^{-\texttt{lsbIn}}$ rows of `lsbOut` bits each
- In general: LUT cost: $2^{-\texttt{lsbIn}-k} \times \texttt{lsbOut}$

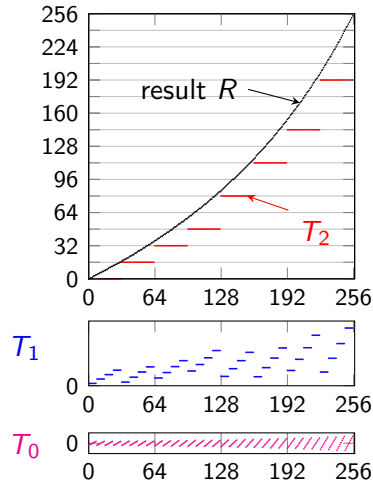
- A 20 Kb dual-port BlockRAM can hold two tables of $2^{10} \times 10$ bits.

A bestiary of function approximation methods

Multipartite Tables for 12 to 24 bits

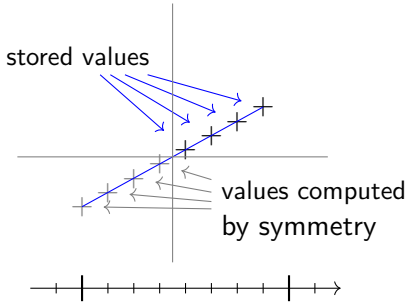


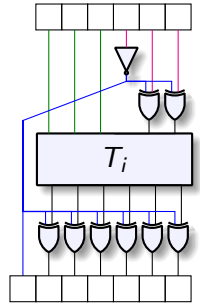




- rule of thumb: cost grows as $2^{p/2} \times p$ instead of $2^p \times p$
- but requires the function to be **continuous**, **derivable**, and even **monotonic**

One more trick with linear approximations: symmetry



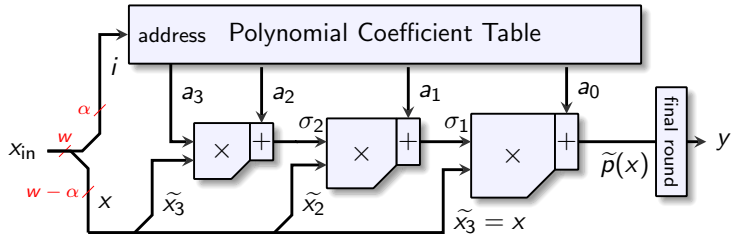


We exploit symmetry to trade one table input bit for two rows of XOR gates...

And above 16 bits...

A generic piecewise polynomial approximation method: `FixFunctionByPiecewisePoly`

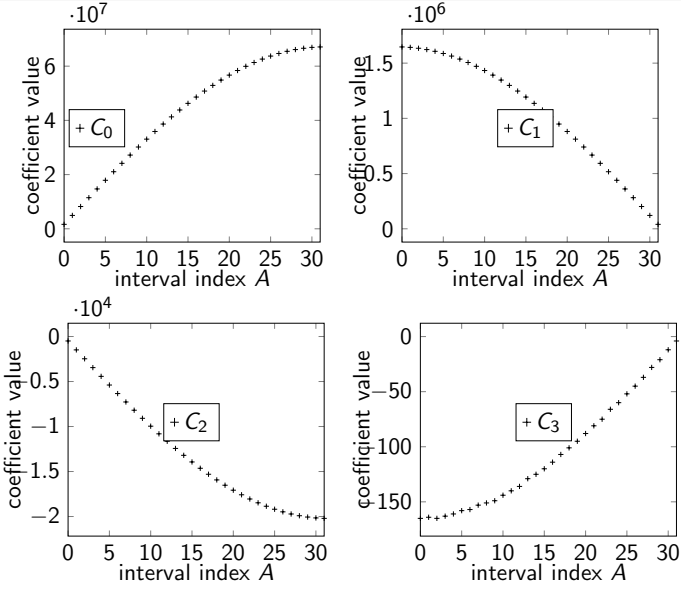
- requires higher-order derivability, but scales to 64 bits.
- One more parameter: the *degree* of the polynomials, trades-off **memory** and **multipliers**



Conclusion so far

- Whatever the method, there are tables in it
- These tables have some regularity. Can it be exploited ?

Coefficients for a 24-bit, degree-3 approximation to $\sin(\frac{\pi}{2}x)$ $\longrightarrow$

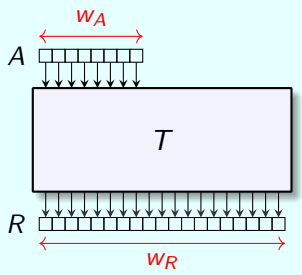


Lossless Differential Table Compression

Lossless Differential Table Compression

A good idea by Hsiao, generalized in FloPoCo to all sorts of tables.

A table...

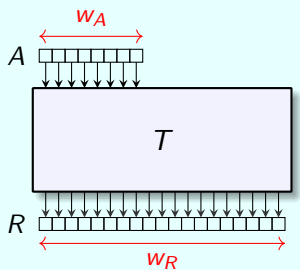


Size $2^{w_A} \times w_R$ bits

Lossless Differential Table Compression

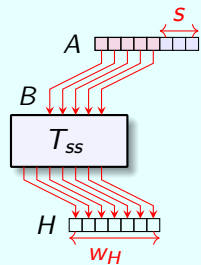
A good idea by Hsiao, generalized in FloPoCo to all sorts of tables.

A table...



Size $2^{w_A} \times w_R$ bits

... can sometimes be compressed

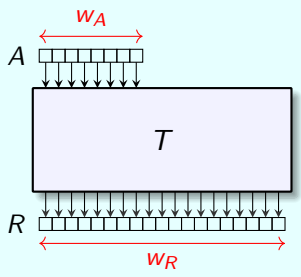


Size $2^{w_A - s} \times w_H$ bits

Lossless Differential Table Compression

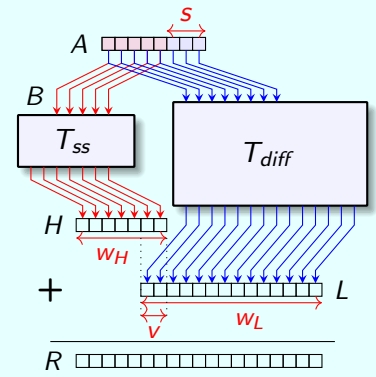
A good idea by Hsiao, generalized in FloPoCo to all sorts of tables.

A table...



Size $2^{w_A} \times w_R$ bits

... can sometimes be compressed



Size $2^{w_A - s} \times w_H + 2^{w_A} \times w_L$ bits

Algorithm is plain brute force

Algorithm 1: Generic LDTC optimization

function `optimizeLDTC(`T, w_A, w_R`)`

$bestVector \leftarrow (0, w_R, 0)$; `// no compression`

forall (s, w_H, w_L); `// enumerate all the possible parameter values`
do

end forall

return *bestVector*

Algorithm is plain brute force

Algorithm 2: Generic LDTC optimization

function `optimizeLDTC(`T, w_A, w_R`)`

$bestVector \leftarrow (0, w_R, 0)$; `// no compression`

forall (s, w_H, w_L); `// enumerate all the possible parameter values`
do

end forall

return *bestVector*

Algorithm is plain brute force

Algorithm 3: Generic LDTC optimization

function `optimizeLDTC(`T, w_A, w_R`)`

$bestVector \leftarrow (0, w_R, 0)$; `// no compression`

$bestCost \leftarrow \text{cost}(bestVector)$;

forall (s, w_H, w_L); `// enumerate all the possible parameter values`

do

$cost \leftarrow \text{cost}(w_A, s, w_H, w_L)$; `// cost can be #bits or FPGA cost`

end forall

return *bestVector*

Algorithm is plain brute force

Algorithm 4: Generic LDTC optimization

```
function optimizeLDTC(T, w_A, w_R)
    bestVector ← (0, w_R, 0) ;                                  // no compression
    bestCost ← cost(bestVector) ;
    forall (s, w_H, w_L);          // enumerate all the possible parameter values
     do
        cost ← cost(w_A, s, w_H, w_L);            // cost can be #bits or FPGA cost
        if cost < bestCost then
            if isValid(T, w_A, w_R, s, w_H, w_L) then
                bestCost ← cost;
                bestVector ← (s, w_H, w_L);
            end if
        end if
    end forall
    return bestVector
```

The `isValid` function is also brute force

Algorithm 5: Is a parameter vector valid?

function `isValid`$(T, w_A, w_R, s, w_H, w_L)$

  for $B \in (0, 1, \ldots, 2^{w_A - s} - 1)$; `// loop on slices`

  do

    $S \leftarrow \{T[j]\}_{j \in \{B \cdot 2^s \ldots (B+1) \cdot 2^s - 1\}}$; `// slice`

    $M \leftarrow \max(S)$; `// max on slice`

    $m \leftarrow \min(S)$; `// min on slice`

    $mask \leftarrow 2^{w_R - w_H} - 1$;

    $H \leftarrow m - (m \,\&\, mask)$; `//` w_H `upper bits of` m

    $M_{\text{low}} \leftarrow M - H$; `// max diff value on this slice`

    if $M_{low} \geq 2^{w_L}$ **then**

      return *false* ; `// this slice won't fit: exit with false`

    end if

  end for

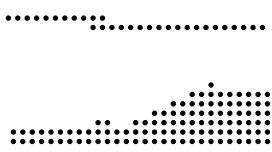
  return *true*

What about cost of addition?

Probably not worth the compression... except if the table is input to a bit heap:
Dot-diagram overhead of lossless table compression, here for the 24-bit multipartite implementation of $\sin(\frac{\pi}{4}x)$

uncompressed

compressed with LDTC



Some results

Compression of plain tables

PTsin, PTlog plain tabulation of $\sin(\frac{\pi}{4}x)$ (pt) or $\log(1+x)$ (ptl) on $[0,1)$

	plain table	LDTC (ratio)	T_{ss}	T_{diff}	v
PTsin8	2,048	992 (0.48)	224	768	
	$8 \cdot 2^8$		$7 \cdot 2^5$	$3 \cdot 2^8$	2
PTsin9	4,608	2,048 (0.44)	512	1,536	
	$9 \cdot 2^9$		$8 \cdot 2^6$	$3 \cdot 2^9$	2
PTsin10	10,240	4,224 (0.41)	1,152	3,072	
	$10 \cdot 2^{10}$		$9 \cdot 2^7$	$3 \cdot 2^{10}$	2
PTsin12	49,152	17,920 (0.36)	5,632	12,288	
	$12 \cdot 2^{12}$		$11 \cdot 2^9$	$3 \cdot 2^{12}$	2
PTlog:12	49,152	18,432 (0.38)	6,144	12,288	
	$12 \cdot 2^{12}$		$12 \cdot 2^9$	$3 \cdot 2^{12}$	3

mpti Table of Initial Values (TIV)
of multipartite approximation to $\sin(\frac{\pi}{4}x)$ on $[0, 1)$

	plain table	LDTC (ratio)	T_{ss}	T_{diff}	v
mpti:12	960	800 (0.83)	96	704	
	$15 \cdot 2^6$		$6 \cdot 2^4$	$11 \cdot 2^6$	2
mpti:14	2,048	1,632 (0.8)	224	1,408	
	$16 \cdot 2^7$		$7 \cdot 2^5$	$11 \cdot 2^7$	2
mpti:16	4,864	3,808 (0.78)	224	3,584	
	$19 \cdot 2^8$		$7 \cdot 2^5$	$14 \cdot 2^8$	2
mpti:20	24,576	18,560 (0.76)	1,152	17,408	
	$24 \cdot 2^{10}$		$9 \cdot 2^7$	$17 \cdot 2^{10}$	2
mpti:24	114,688	83,456 (0.73)	5,632	77,824	
	$28 \cdot 2^{12}$		$11 \cdot 2^9$	$19 \cdot 2^{12}$	2

ea e^A table in a FP Exp for single (sp) or double (dp) precision

	plain table	LDTC (ratio)	T_{ss}	T_{diff}	v
ea:dp:10	$58,368$	52,352 (0.90)	$1,152$	$51,200$	
	$57 \cdot 2^{10}$		$9 \cdot 2^7$	$50 \cdot 2^{10}$	2
ea:dp:12	$233,472$	202,240 (0.87)	$5,632$	$196,608$	
	$57 \cdot 2^{12}$		$11 \cdot 2^9$	$48 \cdot 2^{12}$	2
ea:dp:14	$933,888$	780,288 (0.84)	$26,624$	$753,664$	
	$57 \cdot 2^{14}$		$13 \cdot 2^{11}$	$46 \cdot 2^{14}$	2

C_i coefficients of a degree-2 uniform piecewise approximation to $sin(\frac{\pi}{4}x)$ on $[0, 1)$ for 32-bit accuracy ($w_A = 9$)

	plain table	LDTC (ratio)	T_{ss}	T_{diff}	v
C_0	17,920	15,360 (0.86)	512	14,848	
	$35 \cdot 2^9$		$8 \cdot 2^6$	$29 \cdot 2^9$	2
C_1	12,800	9,792 (0.76)	576	9,216	
	$25 \cdot 2^9$		$9 \cdot 2^6$	$18 \cdot 2^9$	2
C_2	6,656	4,160 (0.62)	576	3,584	
	$13 \cdot 2^9$		$9 \cdot 2^6$	$7 \cdot 2^9$	2
all C_i	$73 \cdot 2^9$	29,312 (0.78)	$26 \cdot 2^6$	$54 \cdot 2^9$	

ASIC implementation results for $\sin(\frac{\pi}{4}x)$ (addition included)

case	LDTC results (ratio WRT uncompressed table)		
PTsin8	130 μm^2 (0.82)	0.3 ns (1.44)	0.12 mW (0.91)
PTsin9	184 μm^2 (0.56)	0.3 ns (1.12)	0.16 mW (0.62)
PTsin10	298 μm^2 (0.54)	0.4 ns (0.82)	0.26 mW (0.58)
PTsin12	1,190 μm^2 (0.63)	0.6 ns (0.94)	1.06 mW (0.66)
mPTsin12	269 μm^2 (0.98)	0.6 ns (1.00)	0.34 mW (0.96)
mPTsin14	424 μm^2 (0.98)	0.7 ns (1.00)	0.50 mW (0.96)
mPTsin16	793 μm^2 (0.92)	0.8 ns (1.00)	0.88 mW (0.88)
mPTsin20	2,888 μm^2 (0.84)	1.2 ns (1.12)	2.76 mW (0.67)
mPTsin24	11,801 μm^2 (0.87)	1.8 ns (1.01)	11.13 mW (0.72)

Synthesis results obtained through Synopsys design compiler using a 28nm FDSOI standard cell library from STMicroelectronics. Timing includes an estimation of interconnect delays. The compressor tree used for mpt is FloPoCo's default.

FPGA implementation results for $\sin(\frac{\pi}{4}x)$ (addition included)

case	without LDTC		with LDTC (ratio)	
PTsin8	27 LUT	5.5 ns	26 LUT (0.96)	6.4 ns (1.17)
PTsin9	61 LUT	6.3 ns	46 LUT (0.75)	6.7 ns (1.08)
PTsin10	134 LUT	6.8 ns	83 LUT (0.62)	7.5 ns (1.1)
PTsin12	536 LUT	9.8 ns	287 LUT (0.54)	8.7 ns (0.89)
mPTsin12	76 LUT	7.6 ns	73 LUT (0.96)	7.7 ns (1.02)
mPTsin14	102 LUT	8.1 ns	98 LUT (0.96)	8.1 ns (1.0)
mPTsin16	184 LUT	8.9 ns	190 LUT (1.03)	9.3 ns (1.04)
mPTsin20	676 LUT	12.8 ns	637 LUT (0.94)	12.5 ns (0.98)
mPTsin24	2489 LUT	18.1 ns	2322 LUT (0.93)	16.1 ns (0.89)

Results after implementation on Kintex7 using Vivado 2020.2.
The compressor tree used for mpt is FloPoCo's default.

Conclusion

- Completely generic lossless compression of tables of numerical values
- Exhaustive brute-force optimization works well
  - (less than 1s for practical sizes)
- A definite win as soon as the table output is an input to a bit heap
  - Multipartite and other piecewise polynomial approximations using parallel evaluation
- In other cases... your mileage may vary
  - examples in next talk

Future work:

- Evaluate, on a case by case basis, in actual composite operators in FloPoCo
  - Currently the compression is systematically evaluated (to fill the previous tables) but only exploited in **FixFunctionByTable** and **FixFunctionByMultipartite**
- integrate this compression approach in the global optimization of an operator