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

Luc Forget, Gauthier Harnisch, Ronan Keryell, Florent de Dinechin

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

LUC FORGET, GAUTHIER HARNISCH, and RONAN KERYELL, AMD Adaptive and Embedded Computing Group, USA

FLORENT DE DINECHIN, Univ Lyon, INSA Lyon, Inria, CITI, France

This paper presents a framework to reuse the intelligence of RTL generators in a single-source HLS setting. This framework is illustrated by a C++ fixed-point library to generate mathematical function evaluator. A compiler flow from C++20 to Vivado IPs has been developed to make the library usable with Vitis HLS. This flow is demonstrated on two applications: an adder for the logarithmic number system, and additive sound synthesis. These experiments show that the approach allows to easily tune the precision of the types used in the application. They also demonstrate the ability to generate arbitrary function evaluator at the required precision.

ACM Reference Format:
Luc Forget, Gauthier Harnisch, Ronan Keryell, and Florent de Dinechin. 2022. A single-source C++20 HLS flow for function evaluation on FPGA and beyond.. In *International Symposium on Highly-Efficient Accelerators and Reconfigurable Technologies (HEART2022), June 9–10, 2022, Tsukuba, Japan.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3535044.3535051

1 INTRODUCTION

High Level Synthesis (HLS) has been able to improve FPGA programming productivity compared to traditional register transfer level (RTL) flows. This higher productivity is obtained by describing hardware algorithm with software programming languages that offers high level abstractions. Compiling as-is a program targeting CPU with an HLS compiler can however lead to suboptimal results. Indeed, FPGAs expose much more parallelism and flexibility than CPUs, and a CPU program must often be adapted to exploit this potential. In order to use the parallelism, the algorithm implementation can be pipelined, loops can be unrolled, and buffer elements can be streamed in a dataflow fashion. Most of the recent HLS tools offer language extensions to this purpose.

A second way to exploit the flexibility of FPGAs is to tailor the numeric formats used in the computation to fit the application accuracy requirement. Not every application has its optimal numeric formats covered by the few standard formats supported by usual software programming languages. Recent work on machine learning or signal processing applications have for instance highlighted the interest for reduced-precision floating point formats [1, 4, 6, 12, 19], or even radically different numerical representations [2, 14, 18]. To address this, most vendor HLS tools provides support for custom-sized fixed point formats, and external HLS libraries offer basic support for custom-sized floating points or even more exotic formats [22, 25]. Such HLS libraries can benefit from all the optimizations performed by the HLS tools [22, 24]. However, these libraries only provide support for basic arithmetic operations. They usually miss, for instance, the elementary functions (exponential, logarithm, trigonometric) available only for floating-point formats.

Indeed, a third way to exploit the FPGA flexibility is to build ad-hoc architectures for numerical functions appearing in application code, at the precision required by the application. These functions may be elementary functions, but also composite ones such as $\log_2(1 + 2^x)$ or $\sin(\omega t + \phi)$.

Manuscript submitted to ACM

Tools exist [8, 13, 16, 21] that can generate an evaluator for an arbitrary function, usually provided as an arbitrary mathematical expression. These tools are complex program that analyze the function in order to use various approximation techniques before producing RTL code. This generated RTL IP may be used in an HLS tool, but it comes with some drawbacks:

- There is a loss in productivity and portability of the solution, since inclusion of RTL IPs in an HLS design is a tool-specific process and requires extra plumbing;
- HLS designs are pipelined at the HLS stage, which manipulates high level representations. HLS tools are not able to blend the RTL IPs in this pipeline process. Due to this, the RTL IPs behaves like optimization barriers for the HLS stage. Instead of optimizing the whole component, HLS tools will only be able to optimize separately the sub-components above and below the black box, which may lead to suboptimal synthesized designs.

Filling this gap requires to fully integrate in the HLS flow the complex process of generating an optimized architecture from an arbitrary function specification. This paper presents an HLS C++ library that is a first step towards this goal. It provides a single-source way to generate an evaluator for a function given as a mathematical expression, on custom-sized fixed-point inputs. This library is described in section 2.

Section 3 presents the compiler flow, which involves two compilation stages, and its interaction with external tools that perform the mathematical approximation of the function. In its current state, the library offers plain tabulation and bipartite function approximation [7, 20] introduced in more details in Section 5.

In order to use modern C++ constructs on AMD FPGAs, an experimental compiler targeting Vitis IP has been developed from `clang++`. This is necessary because AMD Vitis toolchain only supports C++14. This compiler is described in section 4.

A few applications have been developed in order to demonstrate the library capabilities. These applications are described in section 6.

Finally, improvement possibilities are discussed in section 7.

2 LIBRARY PRESENTATION

The library[1] allows the user to define custom fixed point formats and computations on value of these formats, as well as conversions between these formats and standard C++ types.

In all this paper, a fixed point format is defined by the weights `msb_weight` and `lsb_weight` associated respectively to its most-significant and least-significant bits, and its signedness `signedness`. Signed values are represented in 2's complement. The width of a format (in bits) is `width` = `msb_weight` − `lsb_weight` + 1. The representable values in such a format are scaled integers of the form $k \times 2^{\texttt{lsb\_weight}}$, where k is an integer such that

$$k \in \left[0, 2^{\texttt{width}} - 1\right] \quad \text{if the format is unsigned, or}$$

$$k \in \left[-2^{\texttt{width}-1}, 2^{\texttt{width}-1} - 1\right] \quad \text{if the format is signed.}$$

2.1 C++ types for fixed-point number

In the library, a format is represented by the template `FixedFormat`. For instance, the type `FixedFormat<31, 0, signed>` represents a fixed-point format which is equivalent to a 32 bit signed integer.

A `FixedFormat` is used to parameterize the template class `FixedNumber` which represents a value of a given format.

[1]The library is available on github at this address https://github.com/lforg37/marto, as subpart of the Marto library.

```
using format = FixedFormat<4, -3, unsigned>;
FixedNumber<format> X{0b00000010};
```

Listing 1. Construction of a `FixedNumber` from its representation.

A `FixedNumber` can be constructed from the representation of its value. For instance, in Listing 1, the variable `X` is initialized with the value 2^{-2} because only its second bit (of weight -2) is set.

A `FixedNumber` may also be constructed from the value of a standard arithmetic type, thanks to the static method `get_from_value`. This conversion returns the value of the target format closest to the source value. In case of a tie, the greatest of the two possible values is returned.

In addition, a user-defined literal (UDL) allows to create a `FixedNumber` from a value without having to specify the format. The narrower format that can represent exactly the expression will be automatically deduced from the operator. As decimal floating-point literals may have non-finite representation (for instance $0.1_{10} = 0.000110011001100\ldots_2$), this UDL only accepts hexadecimal floating-point literals.

Finally, conversion from `FixedNumber` to standard arithmetic types are expressed with the `get_as` template method from `FixedNumber`. This method takes a target arithmetic type as template parameter and returns the value of this type which is the closest to the represented value (with the same tie-breaking rule as the other conversions).

Listing 2 presents an example of computation performed with the library demonstrating many of its capabilities.

2.2 Classical arithmetic computations

The proposed library defines standard arithmetic operations between `FixedNumber` of different `FixedFormat`, in a way very similar to mainstream HLS fixed point libraries such as `ap_fixed` or `ac_fixed`.

Addition, subtraction and multiplication are computed exactly, the `FixedFormat` of the result being deduced by the library to be wide enough to hold the result. Rounding, overflow or saturation must be managed explicitely: `FixedNumber` provides additional methods to extract a part of it, round to a specific precision, or expand the number to a wider format.

```
using radius_fmt = FixedFormat<3, -4, unsigned>;
auto circumference(FixedNumber<radius_format> radius) {
  // For simplicity, FixedNumber<m, l, s> replaces FixedNumber<FixedFormat<m,l,s>>
  // Generate a constant of value two
  constexpr auto two = 2._fixed; // two is the same as FixedNumber<1, 1, unsigned>{1};
  // Generate an approximation of pi rounded to 2^(-4)
  constexpr auto pi = FixedNumber<1, -4, unsigned>::get_from_value(PI);
  // Diameter is a FixedNumber<4, -3, unsigned>;
  auto diameter = two * radius;
  // circumference is a FixedNumber<6, -7, unsigned>
  auto circumference = diameter * pi;
  // Eventually we get rid of low bits
  // Return type is FixedNumber<6, -4, unsigned>
  return circumference.round_to<-4>();
}
```

Listing 2. Computing the circumference of a circle from its radius

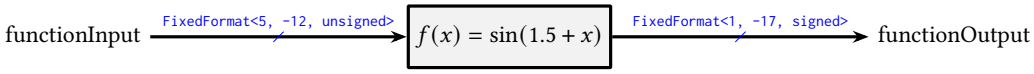


Fig. 1. Example of function evaluator (here correspondig to listing 3).

The proper way to handle division and remainder is still under study.

The main novelty introduced by the proposed library is the presence of an evaluator generator for more sophisticated mathematical expression. The mechanism behind this is introduced in the next section.

2.3 Defining and evaluating a mathematical function of a `FixedNumber` argument

The library provides a standard set of operators and elementary functions. As this set is limited, applications often require the evaluation of mathematical functions which are themselves defined as compositions of operations and functions from this standard library, for instance $f(x) = \log(1 + e^x)$. Such a function can be evaluated at run-time by a composition of library components (here an exponential, an addition and a logarithm). It would however often be more efficient to directly compile this whole expression into a single component, as illustrated by Figure 1. For this purpose, the proposed library also provides a mechanism to describe a function of one variable, then generate an accuracy-constrained fixed-point evaluator for this function.

In the proposed approach, such a composite function may be defined as a C++ expression with a specific property: it has a free variable defined by the `FreeVariable` constructor. More precisely, this expresssion must be a tree of mathematical operations or elementary functions, and the leaves are either free variables or constants. A valid expression, for the purpose of building an evaluator, should have at most one free variable. Then the expression tree defines a function, and the `FixedFormat` associated to the free variable defines the domain of this function.

Listing 3 shows how a simple expression tree can be constructed. At line 3 of the listing, a `FreeVariable` node is created from a fixed-point variable. The `FreeVariable` gets associated to the format of the `FixedNumber`, so the input domain of the resulting function consists of the set of values representable on `FixedFormat<5, -12, unsigned>`, or $[0, 2^5 - 2^{-12}]$. At line 4, a `Constant` leaf is created using a user-defined literal[2]. Similarly to what happen with the `_fixed` literal, the `FixedFormat` to associate to the constant will be deduced to fit exactly the constant storage requirement, but it is less relevant here as the constant will usually not directly appear in the final architecture. The library also has special constants to represent numbers like π, which are kept symbolic (infinitely accurate) in the expression tree defining a function. The `f` variable at the end of the listing represents the function $f : x \longrightarrow \sin(x + 1.5)$ over the input domain $[0, 2^5 - 2^{-12}]$.

```
1 FixedNumber<FixedFormat<5, -12, unsigned>> functionInput;
2 functionInput = (...) // HLS code computing functionInput
3 FreeVariable x{functionInput}; // defining a free variable
4 auto c = 1.5_cst;
5 auto f = sin(x + c);
6 auto functionOutput = f.evaluate<-17>();
```

Listing 3. Construction of the expression tree corresponding to $f(x) = \sin(x + 1.5)$ (lines 3-5), and construction of a fixed-point evaluator for this function (line 6).

[2]It is also possible to create it by specifying the exact type.

Once the function is properly defined, it is possible to generate an accuracy constrained evaluator for it. For a given accuracy constraint $\epsilon > 0$, an accuracy-constrained evaluator $\tilde{f_\epsilon}$ for a given function f over a domain I is a function such that

$$\forall x \in I, \left| f(x) - \tilde{f_\epsilon}(x) \right| < \epsilon$$

This is done using the `evaluate` library function. This function is templated by an integer `lsb_out`. The result to this call is a fixed-point number having `lsb_out`for lsb. The distance between the real value of the function and the `evaluate` function result is strictly lower than $2^{\texttt{lsb\_out}}$. This accuracy constraint is often called "faithful rounding" in the literature.

For instance, in listing 3, the variable functionOutput will contain an approximation of sin(1.5 + `functionInput`) faithful to 2^{-17}. The output domain of the evaluator is `FixedFormat<1, -17, signed>`. This format is deduced by the library from the input domain and expression tree.

This evaluator can resort to plain tabulation for small precisions, as described in the next section, or to more sophisticated approximation methods including arbitrarily complex optimization techniques, an example of which is described in Section 5.

3 INTERACTION WITH EXTERNAL TOOLS IN A SINGLE-SOURCE SETTING

While building the evaluators in a completely metaprogrammed way would have been ideal to reach the goal of the library, it has two important drawbacks. First, the evaluation of C++ functions at compilation-time is way slower than running their compiled equivalent, which can be problematic as the exploration space for evaluator architectures is important. Secondly, depending on the evaluation technique used, it requires either to be able to compute accurately the function on all the inputs or to do some analysis of the expression to evaluate. This requires specific tools, that to the authors knowledge are not provided by any existing C++ header-only library. The development of such a library would be a huge task, therefore the presented library opted for a hybrid approach. The complete flow for using the library is described in figure 2.

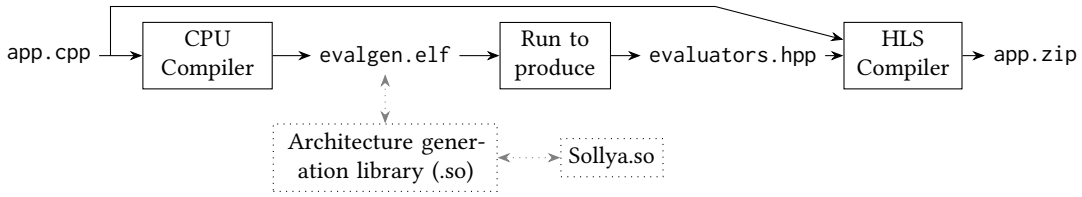


Fig. 2. The library usage flow. File extensions describe the type of each state outcome.

The source file, containing all the expression tree of the function and the associated calls to `evaluate`, has first to be compiled for CPU.

The produced CPU program links at runtime to an architecture generation library, which is responsible to determine an efficient hardware implementation for the problem that cannot be solved by metaprogrammatically. In the case of evaluator generation, it first builds a runtime representation of each expression tree for which an evaluation was required. The resulting runtime expression description is used to build a description understandable by the `Sollya` library [5]. `Sollya` is used to build the highly accurate reference table for the function. This table is then processed to

find a good bi-partite decomposition. If no good decomposition is found, then the tool falls back to a plain tabulation. The chosen algorithm and associated tables are then saved in the specialization header.

This header contains template specialization for the required `Evaluator`, which is the object produced under the hood by the calls to `evaluate`. Listing 4 shows the outline of the generic Evaluator class, while listing 5 illustrates what a specialization looks like. This listing is a simplification, and of course there is no need to use the evaluator mechanism for a simple addition.

```
template<ExpressionType ET, prec_t OutputPrec>
struct Evaluator {
  Evaluator(){ /* Find evaluation plan and dump it to specialization header */ }
  auto evaluate(ET expr) {/* Return marker type indicating unevaluated context */}
};
```

Listing 4. Outline of the generic `Evaluator` class.

```
    template<>
    struct Evaluator<SumExpr<FreeVariable<...>, Constant<1, ...>, /*OutputPrec=*/-17> {, ..
      auto evaluate(SumExpr<FreeVariable<...>, Constant<1, ...>> & expr) {
        return expr.variable + 1;
      }
    };
```

Listing 5. One `Evaluator` specialization for $x + 1$ at with a target accuracy of 2^{-17}.

With this specialization header, it is now possible to compile the component with an HLS compiler. However the library types and the specialization mechanism make use of multiple C++20 constructs. To our knowledge no vendor compiler supports C++20, therefore we had to design a custom compiler. This custom compiler interact with the Vitis toolchain to build Vitis IP. It is described in the next section.

4 COMPILER FOR RECENT C++ STANDARD HLS ON AMD FPGAS

This section describes the custom C++ compiler that has been developed for enabling C++20 constructs on HLS targeting AMD FPGAs. The Vitis frontend, which has recently been open sourced, consists in a derivative from Clang/LLVM [17], with added support for custom pragmas and some specific optimization pass. The LLVM version at which the tool is frozen is LLVM 7. Moving the Vitis frontend back from the past to align it with the current LLVM 15 seems to represent an enormous amount of work. Instead of doing so, the strategy has been to start from a modern Clang++ 15 compiler, and use its output to feed the Vitis frontend with LLVM IR. This modern `clang++` is able to support C++ 20 and some of C23 constructs that the generator needs, such as concepts. However, the LLVM IR version produced by the recent Clang/LLVM is not directly compatible with Vitis for two reasons:

- first, Vitis understands only LLVM 7 IR, and LLVM 15 IR has some constructs invalid in LLVM 7 IR;
- secondly, Vitis HLS expects some IR patterns that are no longer emitted by `clang++`. Without them, the quality of result can degrade abruptly. For instance, the canonical way to move objects in recent LLVM is via a `memcpy` instructions, while Vitis works better with loads and stores.

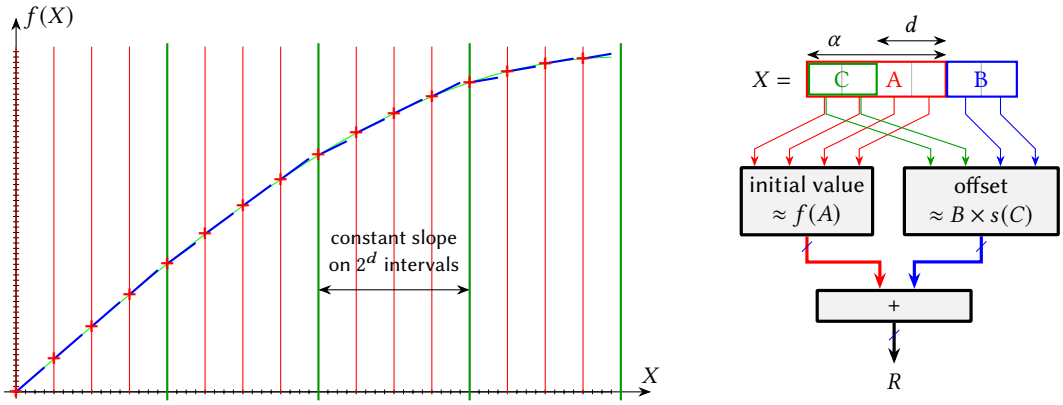
So a few LLVM passes that downgrade modern LLVM IR to Vitis understandable IR has been developed. This compiler also offers a few custom C++ decorators that replace Vitis HLS pragmas. Most of this work is shared with the experimental porting of SYCL[3] on AMD FPGAs [23]. To allow the usage of the tool outside the SYCL environment, new Vitis-IP target has been developed and added too this compiler. With this target, the downgrading passes are run and `vitis_hls` is invoked on the result, to produce an IP package that can be imported in a Vivado block design.

A command to launch an IP synthesis is then similar to a `clang++` invocation such as

```
clang++ --target=vitis_ip-xilinx vitis_ip.cpp -std=c++20 -o ip.zip
↪ --vitis-ip-part=xc7vx330t-ffg1157-1
```

5 A NON-TRIVAL FUNCTION APPROXIMATION METHOD : THE BIPARTITE APPROXIMATION

The flow described in figure 2 introduces the opportunity to run arbitrarily complex programs to optimize the implementation of the function. To illustrate this opportunity, we implemented a classical but non-trivial function approximation technique: the bipartite table method [7, 20]. As illustrated by Figure 3, this method is based on a piecewise linear approximation. The input domain is partitionned in 2^α sub-intervals by splitting the input as $X = A + 2^{-\alpha}B$, where A (the interval index) consists of the α leading bits of X and B (the index within one interval) consists of the least significant bits. The linear approximation on each subdomain indexed by A is $f(X) \approx f(A) + 2^{-\alpha}B \times s(A)$ where $s(A)$ is the slope of the approximation segment. The idea of the bipartite method is to tabulate the multiplications $B \times s(A)$. To reduce the cost of this tabulation, the slopes are shared between 2^d consecutive intervals, so instead of $B \times s(A)$ it is possible to tabulate $B \times s(C)$ where C consists of the $\alpha - d$ leading bits of A.



(a) The bipartite approximation (before discretization)

(b) Bipartite architecture

Fig. 3. Example bipartite approximation architecture, here replacing a table of 2^6 entries with two tables of 2^4 entries.

Many evolutions to this method have been developed (see [13] and references therein). Due to lack of time, the current tool flow only provides a basic bipartite approximation: its purpose is to demonstrate that the approach is not restricted to simple tabulation. The current implementation computes the parameters α and d, and fills the corresponding tables for a faithful approximator. Plain tabulation is used when this bipartite approximation fails (for instance when the function does not offer sufficient linearity).

[3]SYCL is a programming standard from the Khronos Group [10] to program applications using heterogeneous accelerators with a modern single-source C++ framework

```
// lns_t is a specialization of FixedNumber
lns_t LNSAdder(lns_t op1, lns_t op2) {
  auto min_exp = min(op1, op2);
  auto max_exp = max(op1, op2);
  min_exp -= max_exp;
  auto diff_fv = FreeVariable{min_exp};
  auto f = log2(0x1.p0_cst + pow(0x2.p0_cst, diff_fv));
  auto rounded_f = f + lns_t::rounding_constant;
  auto result_exp_diff = evaluate<lns_t::add_eval_precision>(expr_rounded);
  max_exp += result_exp_diff;
  return max_exp;
}
```

Listing 6. A simplified LNS adder

6 APPLICATION STUDY

This section presents applications that illustrates the usage of the library. The first application, a LNS adder, consists mostly in one function evaluation call. The next one, an additive synthesizer, illustrates the usage of the the evaluator integrated in a more complex computation. It also demonstrate the compromises on QOR that are made possible by the combination of variable precision fixed-point and HLS. Finally, the LNS adder component is used to build a SYCL kernel, to illustrate the advantages of a single-source system.

6.1 Narrow LNS Adder

The Logarithm Number System (or LNS) is an alternative to floating-point. Instead of representing a number by an exponent and a fraction, only the exponent is kept, but it is kept with a fraction part. In other words, a number is represented by a sign bit s and a fixed-point 2's complement base b logarithm L, coded in some `FixedFormat` <m, l, signed> format with $l < 0$. In other words the real value represented by the s and L fields is

$$X = (-1)^s \times b^L. \tag{1}$$

The main advantage of LNS is the simplicity of multiplication, division and square root, implemented respectively by addition, subtraction and right shift of the logarithms of the operands. Besides, multiplication and division are exact (they incur no rounding error).

The main drawback of LNS is that addition and subtraction are much more complicated than in floating-point. In the case of the sum, if X and Y are two positive numbers such that $X > Y$, we have

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

The function $f_\oplus(r)$ is a good example of composite functions that can be implemented using the techniques described in the present article, for a simple implementation of LNS. With this approach the constant parameter b can be changed easily [2].

In order to demonstrate the capabilities of the library, a simple LNS adder has been developed. The simplified adder is presented in listing 6.

The LNS Adder has been synthesized for an `lns_t` having an exponent of `FixedFormat<5,-6, unsigned>`. This requires to evaluate $f_\oplus$ with an accuracy constraint of 2^{-8}. The report after synthesis and place-and-route shows that the design uses two BRAM, that matches what was expected.

The next section describes a more complex application that was developed after having checked that this simple experiment gives a design with sound resource consumption.

6.2 Additive sound synthesis

Additive synthesis is a sound synthesis technique typically used in electronic music to create timbres by adding sine waves together [9] as in the following computation :

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

It allows very rich sounds when a lot of sines are used, but then it is very compute-intensive. When used in an interactive configuration with hardware in the loop to simulate real instruments interacting in real-time with the real world physics, it requires very low latency.

In order to check the impact of the library, an additive synthesis kernel was developed both with `FixedNumber` and `float` computation. Both version have an underlying collection of 256 oscillators, having a frequency $f_k = \frac{k*1000}{256}$ and phases $\phi_k = 0$. The amplitudes r_k is an input of the kernel. In both versions, the oscillators are instantiated using template unrolling, ensuring that they are inlined.

In the `FixedNumber` implementation, the r_k are fixed-point number of format `FixedFormat<-1, -8, unsigned>`. The sine is computed from a wave table of 2^{12} entries generated by the library. The function thus evaluated is $f(x) = sin(2\pi x)$ for x in $[0; 1)$. Two experiments, $\texttt{FixedNumber}_8$ and $\texttt{FixedNumber}_{16}$ have been performed with an evaluator accuracy constraint of 2^{-8} and 2^{-16} respectively.

In the `float` implementation, the `std::sin` function from the mathematical library is used. All the computations are done in `float`.

Function value over one period for all the formats and difference with `float` for $\texttt{FixedNumber}_8$ and $\texttt{FixedNumber}_{16}$ is plotted on Figure 4. The graphs for all the format coincide perfectly, and as expected, the absolute difference to `float` is smaller for $\texttt{FixedNumber}_{16}$ than for $\texttt{FixedNumber}_8$.

The evaluation is performed using our compiler and targeting a Virtex ultrascale plus FPGA with part number `xcvu13p-fhga2104-3-e`. Both pipelined and unpipelined solution have been generated for the three formats. Results are given after place and route. The design metrics are are reported in table 1.

These results shows that fixed-point is definitely a good option for this application. Three factors can explain this better quality of results. The first one is a question of bitwidth of the argument that have to be moved in the design. Indeed, the 256 8 bits amplitude coefficients input of the `FixedNumber` case can be implemented as a very wide input register while it is not possible for the 256 32bits coefficients of the input. Secondly, the intrinsic arithmetic of floating point is more complex than fixed-point on the operation involved in this application. Lastly, the fact that floating-point arithmetic is based on external IP integrated by the design can hinder some optimization opportunities.

6.3 Single-source end-to-end LNS with SYCL

As our compiler also offers an experimental support for compiling SYCL programs targeting AMD FPGAs, we tried to use the library in a full end-to-end single-source application. In the SYCL compilation flow, the computation kernels are

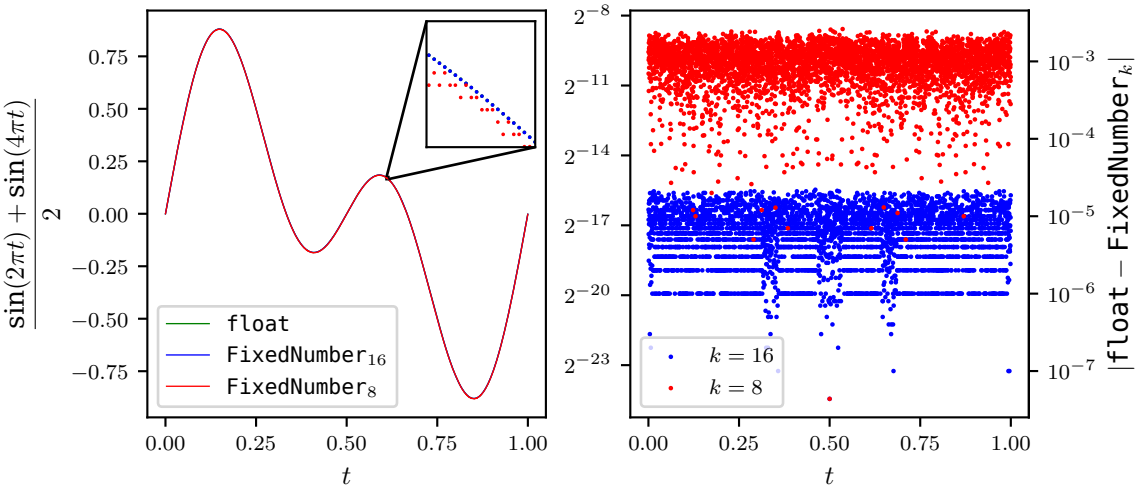


Fig. 4. Output of an additive synthesis with two frequencies of equal amplitudes and the computational error (as absolute difference on a logarithmic scale).

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

Table 1. Area and timing metrics comparison between `float` and various `FixedNumber` for an additive synthesizer of 256 oscillators. II stands for Initiation Interval, the number of clock cycles that pass before the pipeline is ready to be fed a new input.

outlined by the compiler, and compiled separately for the hardware targets. Instead of producing a Vitis IP, it generates a x86 fat binary that embeds the hardware kernel binary, and contains all the glue to invoke the kernel execution on the FPGA. Using the previously described LNS adder and the expression evaluator generator framework, we built a small SYCL application containing a addition between two `lns_t` having an exponent of format `FixedFormat<7,-8>` running on FPGA. While being able to run it and launch it on an alveo u200 board, the process was far from straightforward. Due to the shell of the board that is used and the `M_AXI` controller synthesized to move inputs and outputs between the host and the kernel, it is difficult to comment on the QOR of the design. It however validates that it is possible to integrate our expression evaluator generator framework into a different compilation flow.

7 FUTURE WORK

A limitation of the current evaluator architecture is its restriction on function of only one free variable. Accuracy constrained efficient evaluation of function of many variables is in the general case is an open problem. However, for some specific functions like composition of sums and products, it is possible to build evaluators that have better performance than the execution of all the individual operations. Specialized efficient evaluators include for instance truncated tiled multipliers or bit heaps [3]. An other issue is the lack of mechanism to represent multiple-output functions. While these functions can be computed with one sub-function for each input, this can shadow some sharing optimization. An example of such sharing opportunities can be seen with multiple constant multipliers[11, 26].

Representing expressions as trees can also hinder optimization. While this is not an issue for the currently supported class of functions, an expression would be more adequately represented as a directed acyclic graph. Indeed, the same variable can be reused twice. For instance, the expression `x + x` can be optimized as `2 * x` if we are able to understand that the two `x` variables represent the same value. This problem is more fundamental with the current architecture. As the expression representation is ultimately depending on the type of the nodes, being able to identify the reuse of a variable would require each variable to have a unique type. There is no mechanism that allows to do this automatically within the current C++ standard, so the user would need to add manually a unique ID in the declaration of each variable. This way of doing would be far from convenient and error-prone.

An alternate approach that can lift this limitation would be to move inside the compiler all the work that is currently done when running the specialization header generation executable. The compiler can identify the shared leaves from the invocation context.

If going in this direction, it can also be beneficial to move toward using the MLIR framework [15] to describe and optimize expression DAG instead of relying on a custom expression tree runtime class.

Finally, some improvement could be done on `FixedNumber`. One of them would be to parameterize the rounding and overflow behavior of the class, to give more freedom of usage. The second one would be to associate bounds with the types. This would allow better automatic format deduction for operand result. As of now for instance, adding together 3 `FixedNumber<0,0, ...>` will produce an output of the deduced type `FixedNumber<2, 0, ...>`, while `FixedNumber<1, 0, ...>` would be sufficient.

8 CONCLUSION

This paper presents a framework to perform high-level optimization in a single source manner at HLS stage. This framework usage is illustrated by a generator of accuracy-constrained evaluator for mathematical functions. In order to evaluate this generator, a fixed-point library and a custom C++ compiler have been developed. This compiler allows to run Vitis HLS on C++20 programs. The system is shown to be able to build efficient (while simple) evaluators for functions of one free variable. Limitations of the current infrastructure have been identified as well as directions to lift them.

Acknowledgements

This work was partly funded by the Imprenum project of Agence Nationale de la Recherche.

REFERENCES

[1] Ankur al, Silvia M Mueller, Bruce M Fleischer, Xiao Sun, Naigang Wang, Jungwook Choi, and Kailash Gopalakrishnan. 2019. Dlfloat: A 16-b floating point format designed for deep learning training and inference. In *26th Symposium on Computer Arithmetic* (Kyoto). IEEE, 92–95.

[2] Syed Asad Alam, James Garland, and David Gregg. 2021. Low-precision Logarithmic Number Systems: Beyond Base-2. *Transactions on Architecture and Code Optimization* 18, 4 (2021), 1–25.

[3] Nicolas Brunie, Florent de Dinechin, Matei Istoan, Guillaume Sergent, Kinga Illyes, and Bogdan Popa. 2013. Arithmetic core generation using bit heaps. In *23rd International Conference on Field programmable Logic and Applications* (Porto). IEEE, 1–8. https://doi.org/10.1109/FPL.2013.6645544

[4] Neil Burgess, Nigel Stephens, Jelena Milanovic, and Konstantinos Monachopolous. 2019. Bfloat16 processing for Neural Networks. In *26th Symposium on Computer Arithmetic* (Kyoto). IEEE, 88–91. https://doi.org/10.1109/ARITH.2019.00022

[5] S. Chevillard, M. Joldeş, and C. Lauter. 2010. Sollya: An Environment for the Development of Numerical Codes. In *Third International Congress on Mathematical Software* (Kobe) *(Lecture Notes in Computer Science, Vol. 6327)*, K. Fukuda, J. van der Hoeven, M. Joswig, and N. Takayama (Eds.). Springer, Heidelberg, Germany, 28–31. https://doi.org/10.1007/978-3-642-15582-6_5

[6] Bita Darvish Rouhani, Daniel Lo, Ritchie Zhao, Ming Liu, Jeremy Fowers, Kalin Ovtcharov, Anna Vinogradsky, Sarah Massengill, Lita Yang, Ray Bittner, Alessandro Forin, Haishan Zhu, Taesik Na, Prerak Patel, Shuai Che, Lok Chand Koppaka, XIA SONG, Subhojit Som, Kaustav Das, Saurabh Tiwary, Steve Reinhardt, Sitaram Lanka, Eric Chung, and Doug Burger. 2020. Pushing the Limits of Narrow Precision Inferencing at Cloud Scale with Microsoft Floating Point. In *Advances in Neural Information Processing Systems*. Curran Associates, Inc., 10271–10281.

[7] D. Das Sarma and D.W. Matula. 1995. Faithful bipartite ROM reciprocal tables. In *12th Symposium on Computer Arithmetic*. ACM. https://doi.org/10.1109/ARITH.1995.465381

[8] Florent de Dinechin, Mioara Joldeş, and Bogdan Pasca. 2010. Automatic generation of polynomial-based hardware architectures for function evaluation. In *Application-specific Systems, Architectures and Processors*. IEEE.

[9] Alan Lockhart Monteith Douglas. 1957. *The electrical production of music*. Philosophical Library, New York.

[10] The Khronos® SYCL™ Working Group. 2021. *SYCL™ 2020 Specification*. https://www.khronos.org/sycl

[11] Oscar Gustafsson. 2007. A Difference Based Adder Graph Heuristic for Multiple Constant Multiplication Problems. In *International Symposium on Circuits and Systems*. IEEE. https://doi.org/10.1109/ISCAS.2007.378201

[12] Greg Henry, Ping Tak Peter Tang, and Alexander Heinecke. 2019. Leveraging the Bfloat16 Artificial Intelligence Datatype For Higher-Precision Computations. In *26th Symposium on Computer Arithmetic* (Kyoto). IEEE, 97–98. https://doi.org/10.1109/ARITH.2019.00019

[13] Shen-Fu Hsiao, Po-Han Wu, Chia-Sheng Wen, and Pramod Kumar Meher. 2015. Table Size Reduction Methods for Faithfully Rounded Lookup-Table-Based Multiplierless Function Evaluation. *IEEE Transactions on Circuits and Systems II: Express Briefs* 62, 5 (2015). https://doi.org/10.1109/TCSII.2014.2386232

[14] Jeff Johnson. 2018. Rethinking floating point for deep learning. https://doi.org/10.48550/ARXIV.1811.01721

[15] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. 2021. MLIR: Scaling Compiler Infrastructure for Domain Specific Computation. In *International Symposium on Code Generation and Optimization*. IEEE, 2–14. https://doi.org/10.1109/CGO51591.2021.9370308

[16] Dong-U Lee, Peter Cheung, Wayne Luk, and John Villasenor. 2009. Hierarchical Segmentation Schemes for Function Evaluation. *IEEE Transactions on VLSI Systems* 17, 1 (2009).

[17] LLVM 2022. *The LLVM Compiler Infrastructure*. http://llvm.org

[18] Jinming Lu, Chao Fang, Mingyang Xu, Jun Lin, and Zhongfeng Wang. 2021. Evaluations on Deep Neural Networks Training Using Posit Number System. *IEEE Trans. Comput.* 70, 2 (2021), 174–187.

[19] NVIDIA. 2020. *NVIDIA A100 Tensor Core GPU Architecture*. Technical Report.

[20] D.A. Sunderland, R.A. Strauch, S.S. Wharfield, H.T. Peterson, and C.R. Cole. 1984. CMOS/SOS frequency synthesizer LSI circuit for spread spectrum communications. *IEEE Journal of Solid-State Circuits* 19, 4 (1984). https://doi.org/10.1109/JSSC.1984.1052173

[21] David B. Thomas. 2015. A general-purpose method for faithfully rounded floating-point function approximation in FPGAs. In *22d Symposium on Computer Arithmetic*. IEEE.

[22] David B. Thomas. 2019. Templatised Soft Floating-Point for High-Level Synthesis. In *27th Annual International Symposium on Field-Programmable Custom Computing Machines*. IEEE, 227–235. https://doi.org/10.1109/FCCM.2019.00038

[23] AMD 2022. *SYCL for Vitis: Experimental fusion of triSYCL with Intel SYCL oneAPI DPC++ up-streaming effort into Clang/LLVM*. AMD. https://github.com/triSYCL/sycl

[24] Yohann Uguen, Florent de Dinechin, Victor Lezaud, and Steven Derrien. 2020. Application-Specific Arithmetic in High-Level Synthesis Tools. *ACM Transactions on Architecture and Code Optimization* 17, 1 (2020). https://doi.org/10.1145/3377403

[25] Yohann Uguen, Luc Forget, and Florent de Dinechin. 2019. Evaluating the Hardware Cost of the Posit Number System. In *29th International Conference on Field Programmable Logic and Applications*. IEEE. https://doi.org/10.1109/fpl.2019.00026

[26] Yevgen Voronenko and Markus Püschel. 2007. Multiplierless Multiple Constant Multiplication. *ACM Transactions on Algorithms* 3, 2 (may 2007). https://doi.org/10.1145/1240233.1240234