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

Yohann Uguen, Florent de Dinechin, Steven Derrien

Arithmetic Optimizations for High-Level Synthesis

Yohann Uguen
Univ Lyon, INSA Lyon, Inria, CITI
F-69621 Villeurbanne, France
Yo.Uguen@gmail.com

Florent de Dinechin
Univ Lyon, INSA Lyon, Inria, CITI
F-69621 Villeurbanne, France
Florent.de-Dinechin@insa-lyon.fr

Steven Derrien
University Rennes 1, IRISA
Rennes, France
Steven.Derrien@univ-rennes1.fr

***Abstract*—High-level synthesis (HLS), which enables the synthesis of custom hardware from C/C++ specification, is a big step forward in terms of design productivity, especially for FPGAs. In HLS, data-types and operators are restricted to those available in the `C` language supported by the compiler. This hinders the use of application-specific arithmetic, which has been proven to improve performance and resource usage in FPGA designs. In this work, we propose to study how HLS tools can use application specific knowledge to take advantage of application-specific arithmetics. Our approach is implemented within a source-to-source compiler that optimizes `C` source code to use non-standard, application-specific operators. Our study focuses on widely used summation-reduction patterns, which we optimize by hoisting out floating-point management out the reduction/accumulation loop. As a consequence, the original floating-point addition is replaced by a much simpler fixed-point adder, while enforcing a user specified accuracy constraint for the result. Our results demonstrates significant improvements in terms speed, resource usage and accuracy, which rival with that of RTL level implementations.**

I. Introduction

FPGAs for application-specific accelerators: Many case studies have demonstrated the potential of Field-Programmable Gate Arrays (FPGAs) as accelerators for a wide range of applications, from scientific or financial computing to signal processing and cryptography. FPGAs offer massive parallelism and programmability a the bit level. This enables programmers to exploit a range of techniques that avoid many bottlenecks of classical von Neumann computing:

1) dataflow operation without the need of instruction decoding;
2) massive register and memory bandwidth, without contention on a register file and single memory bus;
3) operators and storage elements tailored to the application in nature, number and size.

However, to unleash this potential, development costs for FPGAs are orders of magnitude higher than classical programming. First, mainstream FPGA design flows use Hardware Description Languages (HDLs) such as VHDL or Verilog. HDLs are much more challenging than traditional programming languages since they have to expose the full freedom of circuit design. Second, circuit simulation can be very slow, making debug and performance evaluation more difficult. Finally, a programmer's constraint becomes a circuit designer's degree of freedom: finding the optimal implementation is more time-consuming in a larger parameter space. High performance and high design costs are therefore the two faces of the same coin.

Hardware design flow and High-level synthesis: As most programmers are more comfortable with languages such as `C` or `Java`, these are increasingly being considered as hardware description languages. Indeed, describing a circuit in C has many advantages: The language itself is more widely known than HDL, and the sequential execution model makes designing and debugging much easier. One can even use software execution on a processor for simulation. All this drastically reduces development time.

The process of compiling a software program into hardware is called High-Level Synthesis (HLS), with tools such as VivadoHLS [12], GAUT [4], LegUp [3] or Catapult C [10] among others. These tools are in charge of turning a C description into a circuit. This task requires to extract parallelism from sequential programs constructs (e.g. loops) and expose this parallelism in the target design. Today's HLS tools are reasonably efficient at this task, and can automatically synthesize highly efficient pipelined dataflow architectures. They however miss one important feature: they are not able to tailor operators to the application in size and even less in nature. This limitation comes from the C language itself: its high-level datatypes and operators are limited to a small number (more or less matching the hardware operators present in mainstream processors). Any high-level C description must therefore live with this constraint. The broader objective of this work is to address this limitation.

Arithmetic in HLS: To better exploit the freedom offered by hardware and FPGAs, HLS vendors have enriched the C language with integer and fixed-point types of arbitrary size[1]. However the operations on these types remain limited to the basic arithmetic and logic ones.

Still, using these, exotic or complex operators, for instance for finite-field or floating-point arithmetic, may be described (still in C) and then encapsulated in a C function that is called to instantiate the operator.

This is actually what happens when ones processes floating-point code through HLS. Such low-level descriptions are currently taken from a fixed library with very little customization. However, there are several opportunities of exploiting compiler information to improve operators:

[1]Arbitrary-size floating-point should follow some day, it is well supported by mature libraries and tools

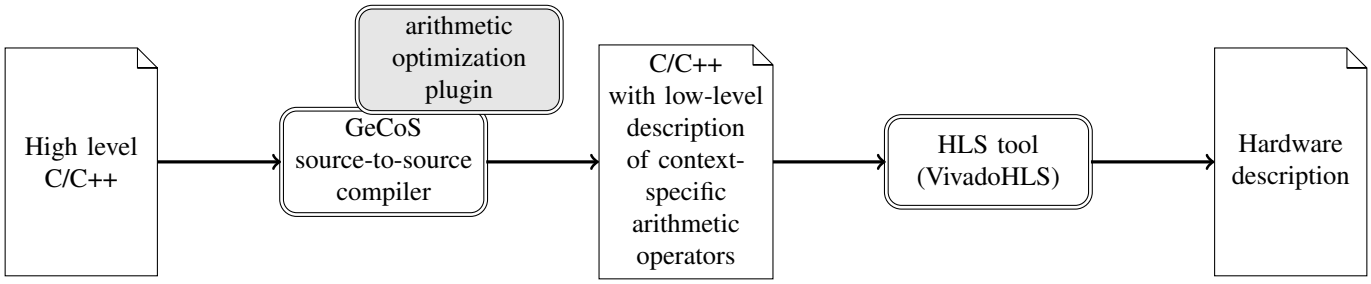


Fig. 1: The proposed compilation flow

- Constant propagation is well understood in compilers: multiplication and division can be specialized for a constant argument.
- Other algebraic transformations may be detected using classical instruction selection techniques, and exploited. For instance, in hardware, a squarer or a cuber may cost much less than a multiplier [5].
- Operators may be shared and fused [15].

The compiler directs the arithmetic, but the opposite can also be true. For instance, the pipeline levels necessary in complex floating-point operators can be exploited as tiling memory [1]. In this case, it is the operator that controls a loop transformation in the compiler.

Most of the previous approaches require heavily parameterized libraries or operator generators, which are already available [5]. In this work, we attempt to exploit further the power of the compiler.

Our case study is a program transformation that applies to floating-point additions on a loop's critical path.

It decomposes them into elementary steps, resizes the corresponding subcomponents to guarantee some user-specified accuracy, and merges and reorders these components to improve performance. The result of this complex sequence of optimizations could not be obtained from a library.

For this purpose, we envision a compilation flow involving one or several source-to-source transformations, as illustrated by Figure 1. Before detailing it, we must digress a little on the subtleties of the management of floating-point arithmetics by compilers.

Listing 1: Simple reduction

```
#define N 100000
  float acc = 0;
  for(int i=0; i<N; i++){
    acc+=in[i];
  }
```

Faithful to the floats, or faithful to the reals: Most recent compilers, including the HLS ones [11], attempt to follow established standards, in particular C11 and, for floating-point arithmetic, IEEE-754. This brings the huge advantage of almost bit-exact reproducibility – the hardware will compute exactly the same results as the software. However, it also greatly reduces the freedom of optimization by the compiler. For instance, as floating point addition is not associative, C11 mandates that code written `a+b+c+d` should be executed as `(((a+b)+c)+d`, although `(a+b)+(c+d)` would have shorter latency. This also prevents the transformations/parallelization of loops implementing reductions. A reduction is an associative computation which reduces a set of input values into a reduction location. Listing1 provides the simplest example of reduction, where sum is the reduction location.

Its synthesis on Kintex7 using VivadoHLS shows that the floating-point addition used there takes 7 cycles (see Table I). The adder is actually active only one cycle out of 7 due to the loop carried dependency. Listing 2 shows an unrolled version of Listing 1. This had to be written by hand: VivadoHLS will not transform Listing 1 into Listing 2, because it is not semantically equivalent (the floating-point additions are reordered as if they were associative). However Listing 2 expresses more parallelism, which VivadoHLS is able to exploit. The main adder is now active at each cycle on a different sub-sum. From Table I, VivadoHLS instantiates a second adder to add these sub-sums at the end, which can be optimized using a few pragmas. Note that a parallel execution with the sequential semantics is also possible, but very expensive [13].

In the following, we assume that the floating-point C program is intended to describe a computation on reals numbers rather than floating point numbers.. Therefore, most programmers will perform the kind of non-bit-exact optimizations illustrated by Listing 2 (sometimes assisted by source-to-source compilers). In a hardware context, we also assume they wish they could tailor the precision (hence the cost) to the accuracy requirements of the application – a classical concern in HLS [9], [2].

The solution we suggest is therefore to interpret the floats as *real numbers* in the initial C++, thus recovering the freedom of associativity (among other).

However, this freedom has to be controlled: In our approach, a pragma specifies the accuracy of the computation with respect to the exact result of the computation. This is as high-level as it gets.

Listing 2: Unrolled reduction

```
#define N 100000
  float acc = 0;
  float tmp1=0, ... , tmp10=0;
  for(int i=0; i<N; i+=10){
    tmp1+=in[i];
    ...
    tmp10+=in[i+9];
  }
  acc=tmp1+...+tmp10;
```

This approach is very local. We let the user chose to optimize those floating-point operators that are used multiple times within a loop, leaving the rest of the code unchanged. This is easier, more general and less invasive than approaches that attempt to convert a whole floating-point program into fixed-point [18].

A high-level compiler is in charge of determining the best way to ensure the prescribed accuracy. This may use number formats that are larger or smaller than the standard ones. These, and the corresponding operators, are presented in Section II. Then Section II presents and evaluates the compiler side of the proposed technique.

II. The arithmetic side: an application-specific accumulator in VivadoHLS

Kulisch advocated a very large floating-point accumulator [14] wose 4288 bits would cover the entire range of double precision floating-point. Such an accumulator would remove rounding errors from all the possible floating-point additions and sums of products, with the added bonus that addition would become associative.

So far, Kulisch's full accumulator has proven too costly to appear in mainstream processors. However, in the context of application acceleration with FPGAs, it can be tailored to the accuracy requirements of applications. Its cost then becomes comparable to classical floating point operators, although it vastly improves accuracy [6]. This operator can be found in the FloPoCo [5] generator. Its core idea, illustrated on Figure 2, is to use a large fixed-point register into which the mantissas of incoming floating-point summands are shifted (top) then accumulated (middle). A third component (bottom) converts the content of the accumulator back to the floating-point format. The blocks visible on this figure (shifter, adder, and leading zero counter) are essentially the building blocks of a floating-point adder.

The accumulator described in this section presents two improvements over the one offered in FloPoCo [6]:

- In FloPoCo, Float-to-Fix and Accumulator form a single component, which restricts its application to simple accumulations similar to Listing1. The two components of Figure 2 enable a generalization to arbitrary summations within a loop, as Section III will show.
- Our implementation supports subnormal numbers.

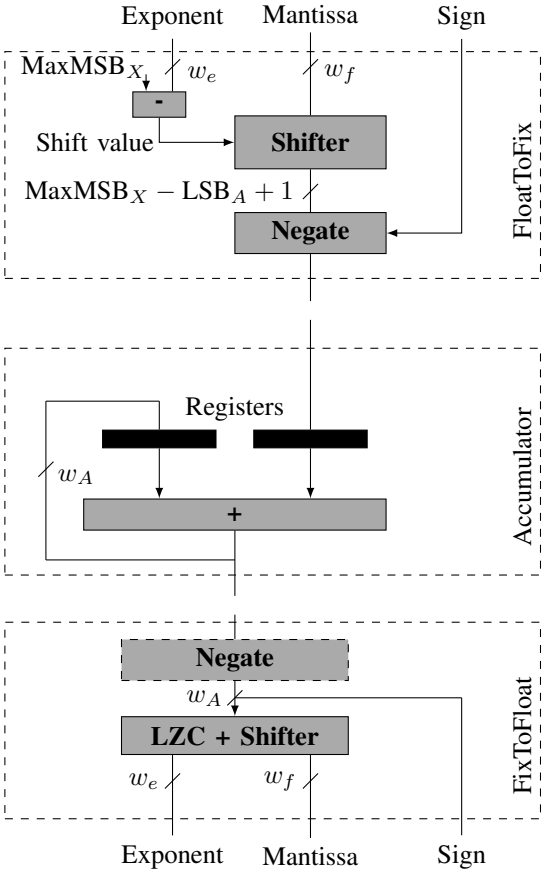


Fig. 2: The conversion from float to a fixed point format (top), the fixed-point accumulation (middle) and the conversion from the fixed-point format to a float (bottom).

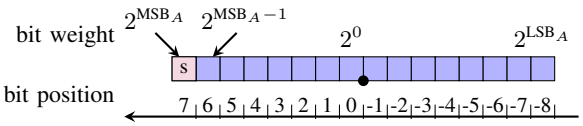


Fig. 3: The bits of a fixed-point format, here $(\text{MSB}_A, \text{LSB}_A) = (7, -8)$.

A. The parameters of FloPoCo's accumulator

The main feature of this approach is that the internal fixed-point representation is configurable in order to control accuracy. It has two parameters:

- MSB$_A$ is the weight of the most significant bit of the accumulator. For example, if MSB$_A = 20$, the accumulator can accommodate values up to a magnitude of $2^{20} \approx 10^6$.
- LSB$_A$ is the weight of the least significant bit of the accumulator. For example, if LSB$_A = -50$, the accumulator can hold data accurate to $2^{-50} \approx 10^{-15}$.

The accumulator width w_a is then computed as MSB$_A$ − LSB$_A$+1, 71 bits in the previous example. Figure 3 shows the bits of a fixed-point format for MSB$_A = 7$ and LSB$_A = -8$. The "virtual" point is placed between the bits of weight 0 and -1.

70 bits represents a wide range and high accuracy, and still additions on this format will have one-cycle latency for practical frequencies on recent FPGAs. If this is not enough the frequency can be improved thanks to partial carry save [6] but this was not useful in the present work. For comparison, for the same frequency a floating-point adder has a latency of about 7 to 10 cycles depending on the target.

B. Implementation details

This accumulator has been implemented in C, using arbitrary-precision fixed point types (`ap_int`). The addition is then written +, the shift is written using the C operator <<. The leading zero count, the bit range selections and the bits modifications are implemented using a VivadoHLS built-in functions. We also used VivadoHLS compiler directives in order to make it try to get the best performance from the design. Indeed, the transformations always include a *pragma* for reducing the iteration interval of the loop to 1. Altogether the three components are written as three C functions of 28 lines of code for the `FloatToFix`, 22 lines for the `FixToFloat`, 44 lines for the `ExactProduct` and 21 lines for the `ExactProductFloatToFix`.

C. Validation

To evaluate and refine this implementation, we used Listing 3, which we compared to Listings 1 and 2. In the latter, the loop was unrolled by a factor 7, as it is the latency of a floating-point adder on our target FPGA (Kintex-7).

For test data, we use as in Muller et al. [16] the input values `(float)cos (i)`, where i is the input array's index. Therefore the accumulation performs the computation of $\sum_i \cos(i)$. We computed this sum in multiple precision, and the line Accuracy of the table reports the number of correct bits of each implementation, after the result has been rounded to a `float`.

The parameters chosen for the accumulator are:

- MSBA = 17. Indeed, as we are adding $\cos(i)$ 100K times, an upper bound is 100K, which can be encoded in 17 bits.
- MAXMSBx = 1. For the same reason as for MSBA, at every step, the maximum input will be 1, which can be encoded in 1 bit.
- LSBA = -50. The accumulator itself will be accurate to the 50th fractional bit. Note a `float` input will see its mantissa rounded by FloatToFix only if its exponent is smaller than 2^{-25}, which is very rare. In other words, this accumulator is much more accurate than the data that is thrown to it.

The results is reported in Table I for simple and double precision. All the data in this table was obtained by generating `VHDL` from `C` synthesis using VivadoHLS followed by place and route from Vivado v2015.4, build 1412921. This table also reports synthesis results for the corresponding FloPoCo-generated VHDL (which doesn't include the array management).

	Naive (float)	Unrolled (float)	Naive (double)	Unrolled (double)	Our code	FloPoCo VHDL
LUTs	266	907	801	2193	736	719
DSPs	2	4	3	6	0	0
Latency	700K	142K	700K	142K	100K	100K
Accuracy	17 bits	17 bits	24 bits	24 bits	24 bits	

TABLE I: Comparison between the different accumulators. The Naive version corresponds to Listing 1, the unrolled version corresponds to Listing 2 and our code is given Listing 3.

Listing 3: Sum of `floats` using the large fixed-point accumulator

```
#define N 100000
float acc = 0;
ap_int<68> long_accumulator = 0;
for(int i = 0; i < N; i++) {
  long_accumulator += FloatToFix(in[i]);
}
acc = FixToFloat(long_accumulator);
```

VivadoHLS uses DSPs to implement the shifts in its floating-point adders. Even if the shifts were implemented in LUTs, the first column would remain well below 500 LUTs: it has the best resource usage. However the latency of one iteration is 7 cycles, hence 100K iterations takes 700K cycles. When unrolling the loop, VivadoHLS is using almost 4 times more LUTs for floats, and 3 times more for doubles. The unrolled versions improves latency over naive versions. Nevertheless, our approach gets even betters latencies for a reasonable LUT usage. Also, we achieve maximum accuracy for the `float` format which caps at 24 bits. The internal representation of the *double*, unrolled *double* and our approach have a higher accuracy than 24 bits but are casted to the float format.

Finally, our results are very close to FloPoCo ones, both in terms of LUTs usage, DPSs and latency.

We have shown through this example that VivadoHLS is able to generate a design comparable to FloPoCo's operators.

D. The exact product

Following Kulisch, we also implemented an exact multiplier. Due to lack of space we do not present it in detail. It is simply a floating-point multiplier with the final mantissa rounding removed. The result mantissa is twice as large as the input mantissas (48 bits in single precision). This poses no particular problem to add it to the large accumulator, however the Float-to-Fix block has to be adapted: in the sequel, it is called Exact Product FloatToFix. This component is depicted in Figure 4.

III. The compiler side: GeCoS source-to-source transformations

Now that we showed that VivadoHLS is able to generate specialized operators of similar quality to FloPoCo's ones, we want to provide a tool that not only transforms Listing 1 into

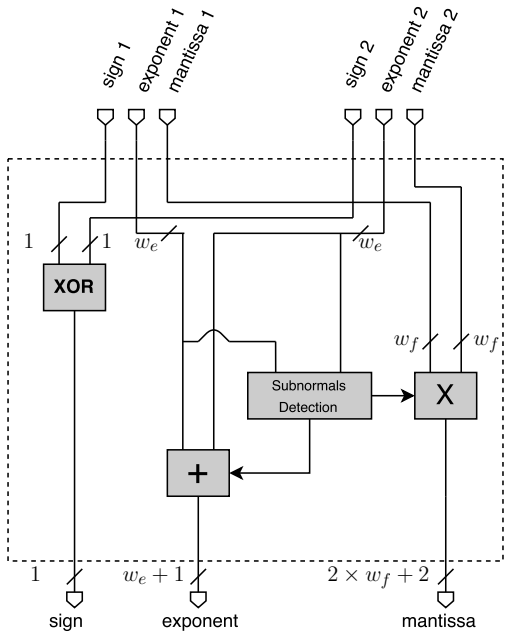


Fig. 4: Exact floating-point multiplier

Listing 2, but also generalizes this transformation to many more situations.

We chose to develop this work in GeCoS [8], an open-source, extensible source-to-source compiler framework built upon model-driven engineering. We hope that this work will eventually be distributed as a GeCoS plugin.

Our approach focuses on two computational patterns, namely the accumulation and the sum of product. Both are specific instances of the reduction pattern, which can be optimized by many compilers or parallel run-time environments. Such patterns are therefore exposed to the compiler/runtime either by the user through directives, or automatically identified using static analysis techniques [17], [7].

Since our focus is not on the detection of reductions, our tool follows a simple approach based on a combination of user directive and (simple) program analysis. More specifically, the user has to identify target accumulation operation through a `pragma`, and provide additional information such as the dynamic range of the accumulated data along with the target accuracy.

Listing 4: Illustration of the use of a `pragma` for the naive accumulation

```
#define N 100000
float accumulation(float in[N]) {
  float acc = 0;
  #pragma FPacc VAR=acc MaxAcc=100000
   epsilon=1E-15 MaxInput=1
  for(int i=0; i<N; i++){
    acc+=in[i];
  }
  return acc;
}
```

A. Compiler directive

In imperative languages such as `C`, reduction are implemented using `for` or `while` constructs. Our compiler directive must therefore be attached to such a construct to be considered as valid. Listing 4 illustrates its usage on the code of Listing 1.

The `pragma` must contain the following informations:

- The keyword `FPacc` that will trigger our transformations.
- The name of the variable in which the accumulation is performed, preceded with the keyword `VAR`. In the example, the accumulation variable is `acc`.
- The maximum value that can be reached by the accumulator through the use of the `MaxAcc` keyword. This value is used to determine the weight of MSB_A.
- The desired accuracy of the accumulator using the `epsilon` keyword. This value is used to determine the weight of LSB_A.
- Optional: The maximum value of the inputs of the accumulator in the `MaxInput` field. This value is used to determine the weight of MaxMSB_X. If this information is not provided, then MaxMSB_X is set to MSB_A.

In the case when no size parameters are given, a full Kulisch accumulator is produced.

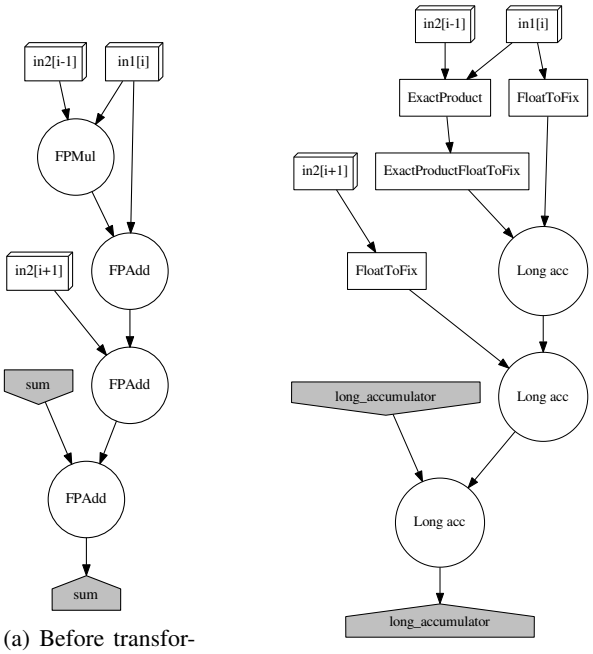
Listing 5: Simple reduction with multiple accumulation statements

```
#define N 100000
float computeSum(float in1[N], float in2[N]){
  float sum = 0;
  #pragma FPacc VAR=sum MaxAcc=300000
   epsilon=1e-15 MaxInput=3
  for (int i=1; i<N-1; i++){
    sum+=in1[i]*in2[i-1];
    sum+=in1[i];
    sum+=in2[i+1];
  }
  return sum;
}
```

B. Code transformation

The proposed transformation consists in an algorithm that navigates and modifies the intermediate representation (IR) of the program. To illustrate this algorithm, consider the sample program shown in Listing 5. It performs a reduction into the variable `sum`, involving both sums and sums of product. The IR associated to the loop body is the directed acyclic graph (DAG) of figure 5a. The keywords *FPMul* and *FPAdd* represent the use of floating-point multipliers and adders respectively. With a floating-point adder needing 7 cycles, this graph has a long latency. The proposed algorithm will push the mantissa shifts out of the loop's critical path, leaving a loop with a 1-cycle delay (Figure 5b).

The code transformation is applied to every block within the `for` block annotated with our `pragma`. It is a bottom-up walk of the program DAG, starting from a `FPAdd` nodes that write to the accumulation variable. During this walk, the following actions are performed depending on the visited nodes:



(a) Before transformations

(b) After transformations

Fig. 5: DAG of the loop body from Listing 5

- A node with the summation variable is ignored.
- A `FPAdd` node is transformed to an accurate fixed-point adder. The analysis is then recursively launched on that node.
- A `FPMul` node is replaced with a call to the `ExactProduct` function followed by a call to `ExactProdFloatToFix`.
- Any other node has a call to `FloatToFix` inserted.

This algorithm rewrites the DAG from Figure 5a into the new DAG shown on Figure 5b.

Then the corresponding `C` code is generated. Also, at the end of the transformed loop, a call to `FixToFloat` is inserted in order for the programmer to retrieve his value as a floating-point.

C. Evaluation

The transformed code reduces the latency by a factor 20 compared to VivadoHLS synthesis of Listing 5, as show in Table II. Our transformed code makes VivadoHLS use more LUTs for less DSPs. Again this is due to Vivado's use of DSP for the shifters.

	Naive Accumulation	Transformed code
LUTs	538	2200
DSPs	5	2
Latency	2000K	100 K

TABLE II: Comparison between the naive code from Listing 5 and its transformed equivalent. Both versions run at 100MHz.

IV. Conclusion

The main result of this work is that HLS tools have the potential to generate efficient designs for handling floating-point computations in a completely non-standard way. The use of application-specific intermediate formats can provide both performance and accuracy at a competitive cost. For this, we have to sacrifice the strict respect of the IEEE-754 and C11 standards. It is replaced by the strict respect of a high-level accuracy specification.

Classically, designers have to face a trade-off between performance and cost. This approach adds computation accuracy to this trade-off.

This work also provides a practical tool that improves a given `C` program. The input to the tool is application-specific informations representing high-level domain knowledge such as the range and desired accuracy of a variable. The resulting code is compatible with VivadoHLS.

There is much more to come. The arithmetic optimizations that a classical compiler can do are very limited by the fixed hardware of classical processors. With compilers of high-level software to hardware, there is much more freedom, hence many more opportunities to build application-specific arithmetic operators. Future work will attempt to explore this new realm, starting with operator specialisation, operator fusion, and compile-time generation of application-specific cores, building upon compiler progresses in program analysis.

References

[1] Christophe Alias, Bogdan Pasca, and Alexandru Plesco. FPGA-specific synthesis of loop-nests with pipelined computational cores. *Microprocessors and Microsystems*, 36(8), 2012.

[2] Gabriel Caffarena, Juan A. Lopez, Carreras Carreras, and Octavio Nieto-Taladriz. High-level synthesis of multiple word-length DSP algorithms using heterogeneous-resource FPGAs. In *2006 International Conference on Field Programmable Logic and Applications*, pages 1–4, Aug 2006.

[3] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Tomasz Czajkowski, Stephen D. Brown, and Jason H. Anderson. LegUp: An open-source high-level synthesis tool for FPGA-based processor/accelerator systems. *ACM Trans. Embed. Comput. Syst.*, 13(2):24:1–24:27, September 2013.

[4] Philippe Coussy, Cyrille Chavet, Pierre Bomel, Dominique Heller, Eric Senn, and Eric Martin. *High-Level Synthesis: From Algorithm to Digital Circuit*, chapter GAUT: A High-Level Synthesis Tool for DSP Applications, pages 147–169. Springer Netherlands, Dordrecht, 2008.

[5] Florent de Dinechin and Bogdan Pasca. *High-Performance Computing Using FPGAs*, chapter Reconfigurable Arithmetic for High-Performance Computing, pages 631–663. Springer New York, New York, NY, 2013.

[6] Florent de Dinechin, Bogdan Pasca, Octavian Creţ, and Radu Tudoran. An FPGA-specific approach to floating-point accumulation and sum-of-products. In *Field-Programmable Technologies*, pages 33–40. IEEE, 2008.

[7] Johannes Doerfert, Kevin Streit, Sebastian Hack, and Zino Benaissa. Polly's polyhedral scheduling in the presence of reductions. *CoRR*, abs/1505.07716, 2015.

[8] Antoine Floc'h, Tomofumi Yuki, Ali El-Moussawi, Antoine Morvan, Kevin Martin, Maxime Naullet, Mythri Alle, Ludovic L'Hours, Nicolas Simon, Steven Derrien, Francois Charot, Christophe Wolinski, and Olivier Sentieys. GeCoS: A framework for prototyping custom hardware design flows. In *Source Code Analysis and Manipulation (SCAM)*, pages 100–105. IEEE, September 2013.

[9] Marcel Gort and Jason H. Anderson. Range and bitmask analysis for hardware optimization in high-level synthesis. In *Design Automation Conference (ASP-DAC), 2013 18th Asia and South Pacific*, pages 773–779, Jan 2013.

[10] Mentor Graphics. Catapult C synthesis, 2011. http://calypto.com/en/products/catapult/overview/.
[11] James Hrica. Floating-point design with vivado HLS, 2012. Xilinx Application Note.
[12] Xilinx Inc. Vivado design suite user guide high-level synthesis. 2015.
[13] Nachiket Kapre and Andre DeHon. Optimistic parallelization of floating-point accumulation. In *Proceedings of the 18th IEEE Symposium on Computer Arithmetic*, ARITH '07, pages 205–216, Washington, DC, USA, 2007. IEEE Computer Society.
[14] Ulrich Kulisch and Van Snyder. The exact dot product as basic tool for long interval arithmetic. *Computing*, 91(3):307–313, March 2011.
[15] Martin Langhammer. Floating point datapath synthesis for FPGAs. In *2008 International Conference on Field Programmable Logic and Applications*, pages 355–360, Sept 2008.
[16] Jean-Michel Muller, Nicolas Brisebarre, Florent de Dinechin, Claude-Pierre Jeannerod, Vincent Lefèvre, Guillaume Melquiond, Nathalie Revol, Damien Stehlé, and Serge Torres. *Handbook of Floating-Point Arithmetic*. Birkhäuser Boston, 2010. ACM G.1.0; G.1.2; G.4; B.2.0; B.2.4; F.2.1., ISBN 978-0-8176-4704-9.
[17] Xavier Redon and Paul Feautrier. Detection of scans in the polytope model. *Parallel Algorithms Appl.*, 15(3-4):229–263, 2000.
[18] Olivier Sentieys, Daniel Menard, David Novo, and Karthick Parashar. Automatic Fixed-Point Conversion: a Gateway to High-Level Power Optimization. Tutorial at IEEE/ACM Design Automation and Test in Europe (DATE), March 2014.