HAL
open science

A high-level synthesis approach optimizing accumulations in floating-point programs using custom formats and operators

Yohann Uguen, Florent de Dinechin, Steven Derrien

To cite this version:

Yohann Uguen, Florent de Dinechin, Steven Derrien. A high-level synthesis approach optimizing accumulations in floating-point programs using custom formats and operators. 2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), Apr 2017, Napa, United States. pp.80-80, 10.1109/FCCM.2017.41 . hal-01498357v2

HAL Id: hal-01498357

https://hal.science/hal-01498357v2

Submitted on 24 Feb 2017

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

A high-level synthesis approach optimizing accumulations in floating-point programs using custom formats and operators

Yohann Uguen
Univ Lyon, INSA Lyon, Inria, CITI
F-69621 Villeurbanne, France
Yohann.Uguen@insa-lyon.fr

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) is a big step forward in terms of design productivity. However, it restricts data-types and operators to those available in the `C` language supported by the compiler. The present work lifts this restriction: it is a case study of enhancing an HLS design flow with non-standard operators, which can then be automatically optimized for their application context. The focus here is on widely used summation-reduction patterns. A source-to-source compiler rewrites, inside critical loop nests of the input `C` code, selected floating-point additions into sequences of simpler operator using non-standard arithmetic formats. This enables hoisting floating-point management out the loop. What remains inside the loop is a sequence of fixed-point additions whose size is computed to enforce a user-specified, application-specific accuracy constraint on the result. Evaluation of this method demonstrates significant improvements in the speed/resource usage/accuracy trade-off.**

I. Introduction

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 at the bit level. This enables programmers to exploit a range of techniques that avoid many bottlenecks of classical von Neumann computing: dataflow operation without the need of instruction decoding; massive register and memory bandwidth, without contention on a register file and single memory bus; 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. High performance and high design costs are the two faces of the same coin.

Hardware design flow and High-level synthesis: To address this, languages such as `C` or `Java` are increasingly being considered as hardware description languages. This has many advantages. The language itself is more widely known than any HDL. 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 [11] or Catapult C [1] among others [16]. 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 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[2]. However the operations on these types remain limited to the basic arithmetic and logic ones. Exotic or complex operators (for instance for finite-field or floating-point arithmetic) may be 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 parameterized library or operator generators [5]. The latter enables many opportunities of compiler optimizations to the operators (not all of which are currently exploited):

- Constant propagation is well understood in compilers, and allows to use hardware multipliers and dividers 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 [14].

[1]Catapult C Synthesis, Mentor Graphics, 2011, http://calypto.com/en/products/catapult/overview/

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

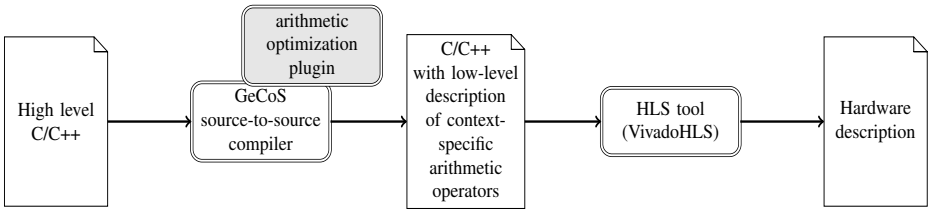


Figure 1: The proposed compilation flow

In these examples, the compiler directs the construction of the arithmetic, but the opposite can also be true. For instance, the pipeline levels necessary in large floating-point operators can be exploited as tiling memory [2]. In this case, it is the arithmetic datapath inside the loop that defines the parameters of a loop transformation.

The case study in this work 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 an operator generator, since it involves global loop information.

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.

HLS faithful to the floats: Most recent compilers, including the HLS ones [10], 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 parallelization of loops implementing reductions. A reduction is an associative computation which reduces a set of input values into a reduction location. Listing 1 provides the simplest example of reduction, where `acc` is the reduction location.

The first column of Table I shows how VivadoHLS synthesizes Listing 1 on Kintex7. The floating-point addition takes 7 cycles, and the adder is only active one cycle out of 7 due to the loop-carried dependency. Listing 2 shows a manually unrolled version of Listing 1. VivadoHLS will not transform Listing 1 into Listing 2, because they are 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 (second column of Table I). The main adder is now active at each cycle on a different sub-sum. Note that a parallel execution with the sequential semantics is also possible, but very expensive [12].

Listing 1: Naive reduction

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

Listing 2: Unrolled reduction

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

Towards HLS faithful to the reals: Another point of view, chosen in this work, is to assume that the floating-point C program is intended to describe a computation on real numbers. In other words, the floats are interpreted as *real numbers* in the initial C, thus recovering the freedom of associativity (among other). Indeed, most programmers will perform the kind of non-bit-exact optimizations illustrated by Listing 2 (sometimes assisted by source-to-source compilers or "unsafe" compiler optimizations). In a hardware context, we may 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], [3]. In this case, a pragma should specify the accuracy of the computation with respect to the exact result. A high-level compiler is then in charge of determining the best way to ensure the prescribed accuracy.

The proposed approach uses number formats that are larger or smaller than the standard ones. These, and the corresponding operators, are presented in Section II. Then Section III presents and evaluates the compiler side of the proposed technique, and section IV evaluates it on the FPMark benchmark suite.

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

Kulisch advocated a very large floating-point accumulator [13] whose 4288 bits would cover the entire range of double precision floating-point. Such an accumulator would remove

	Listing 1 (float)	Listing 2 (float)	Listing 1 (double)	Listing 2 (double)	Listing 3 (71 bits)	FloPoCo VHDL (71 bits)
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	24 bits

Table I: Different ways of implementing a simple accumulation.

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 and in Altera DSP Builder Advanced. 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 sub-blocks visible on this figure (shifter, adder, and leading zero counter) are essentially the building blocks of a classical 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 Listing 1. 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.

A. The parameters of a large 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 $\text{MSB}_A = 20$, 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 $\text{LSB}_A = -50$, the accumulator can hold data accurate to $2^{-50} \approx 10^{-15}$.

The accumulator width w_a is then computed as $\text{MSB}_A - \text{LSB}_A + 1$, 71 bits in the previous example. 71 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 7 to 10 cycles, depending on the target.

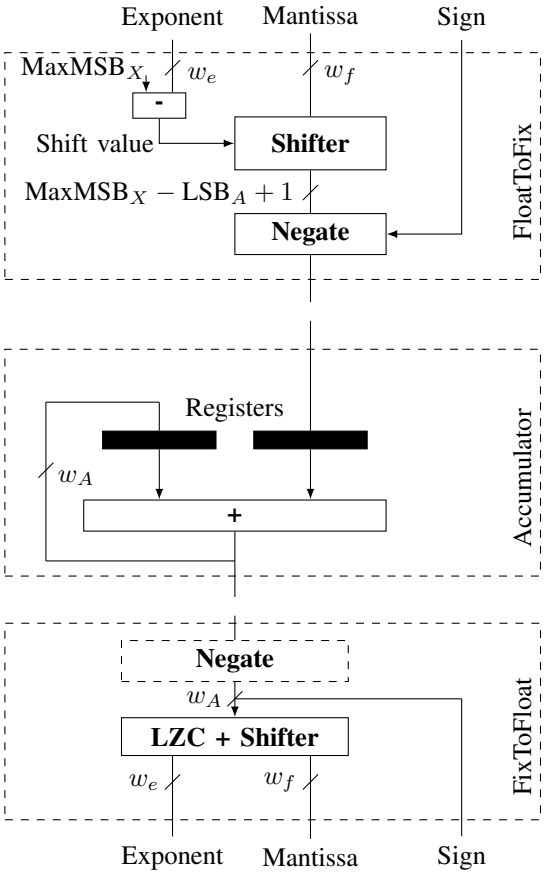


Figure 2: The conversion from float to fixed-point (top), the fixed-point accumulation (middle) and the conversion from the fixed-point format to a float (bottom).

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, bit range selections and other operations are implemented using VivadoHLS built-in functions. A few VivadoHLS compiler directives are also used to get the best performance from the design. Altogether these components are written as two C functions of 28 lines of code for the `FloatToFix`, 22 lines for the `FixToFloat`.

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. [15] the input values `c[i]=(float)cos (i)`, where i is the input array's index. Therefore the accumulation computes $\sum_i$ `c[i]`.

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 as the maximum input value is 1.
- LSBA = -50: the accumulator itself will be accurate to the 50th fractional bit. Note that 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 are reported in Table I for simple and double precision. The Accuracy line of the table reports the number of correct bits of each implementation, after the result has been rounded to a `float`. 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.

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 representations of the *double*, unrolled *double* and our approach have a higher accuracy than 24 bits, but are then casted to the 24 bits of the `float` format). Finally, our results are very close to FloPoCo ones, both in terms of LUTs usage, DPSs and latency.

Using this implementation method, we also created an exact floating-point multiplier with the final rounding removed as in [6]. This function is called `ExactProductFloatToFix`. Due to lack of space we do not present it in detail. As the output of this multiplier is not standard, we also created an adapted Float-to-fix block called `ExactProductFloatToFix`. These functions represent 44 lines of code for `ExactProduct` and 21 lines of code for `ExactProductFloatToFix`.

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 Listing 3, 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. Indeed, all the present work is already freely available as a GeCoS plugin.

This work 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 variable through a `pragma`, and provide additional information such as the dynamic range of the accumulated data along with the target accuracy.

This local approach is easier, more general and less invasive than approaches that attempt to convert a whole floating-point program into fixed-point [18].

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 appear inside such a construct. Listing 4 illustrates its usage on the code of Listing 1.

The `pragma` must contain the following information:

- The keyword `FPacc`, which triggers the 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 MSB_A.
- The desired accuracy of the accumulator using the `epsilon` keyword. This value is used to determine the weight LSB_A.
- Optional: The maximum value of the inputs of the accumulator in the `MaxInput` field. This value is used to determine the weight 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. Also note that the user can quietly overestimate the maximum value of it's accumulator without major impact on area. For instance, overestimating `MaxAcc` by a factor 10 only adds 3 bits to the accumulator width.

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 3a. The keywords *FPMul* and *FPAdd* represent the use of floating-point multipliers and adders respectively. The dotted arrows represent the data dependency between two consecutive iterations of the loop. With a floating-point adder needing 7 cycles, the next iteration will be stalled during this operation. As the proposed algorithm pushes the floating-point normalization out of the loop, the adder only have a 1-cycle delay. Therefore we are able to have an iteration iterval of 1 as shown in Figure 4a.

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;
}
```

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` node that writes to the accumulation variable. During this walk, the following actions are performed depending on the visited nodes:

- 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.

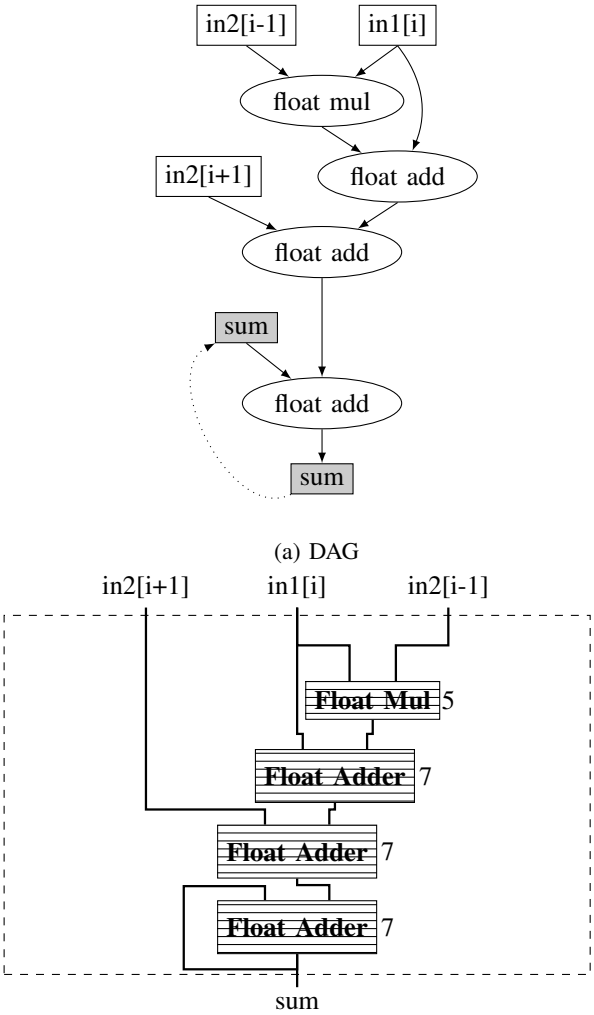


Figure 3: DAG of the loop body from Listing 5 (top) and it's corresponding architecture (bottom)

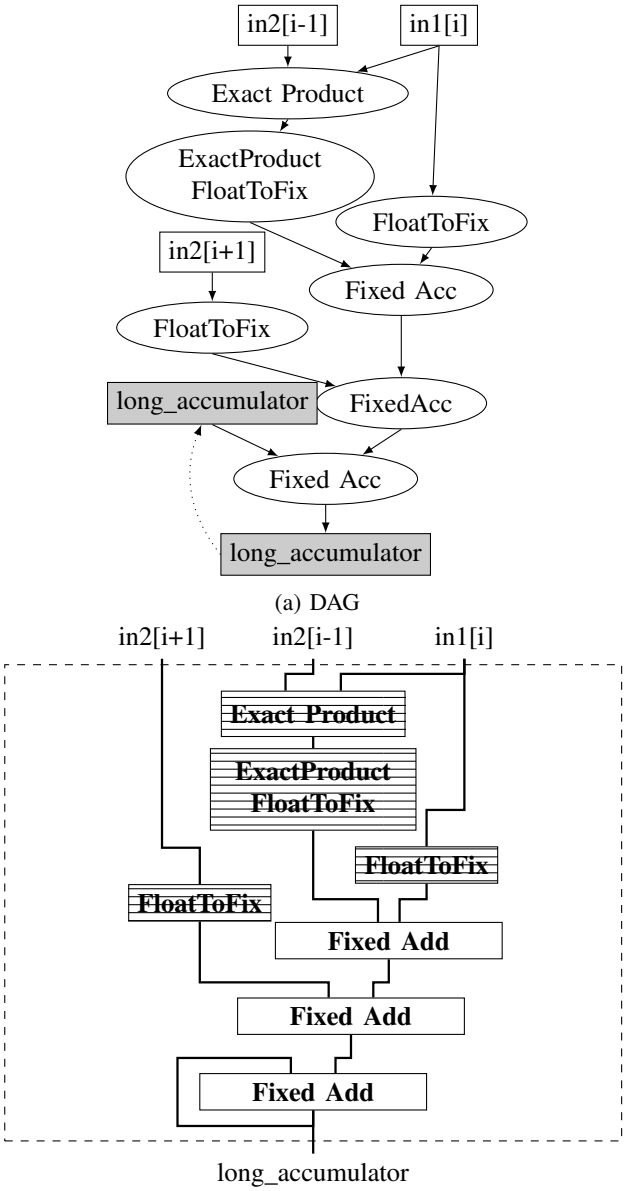
- A `FPMul` node is replaced with a call to the `ExactProduct` function followed by a call to `ExactProdFloatToFix`.
- Any other node has a call to `FloatToFix` inserted.

This algorithm rewrites the DAG from Figure 3a into the new DAG shown on Figure 4a.

Then the corresponding `C` code is generated. In addintion, we provide a statement on the accumualation loop that tries to reduce the iteration interval to 1. The synthesized codes from before and after the transformations result in the architectures from Figure 3b and Figure 4b respectively. 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 of the toy example of Listing 5

The proposed transformations work on non-trivial examples such as Listing 5. Table II shows how resource consumption



(a) DAG

(b) Architecture

Figure 4: DAG of the loop body from Listing 5 (top) and it's corresponding architecture (bottom) after transformations

depends on `epsilon`, all the other parameters being those given in the `pragma` of Listing 5. All these versions where synthesised for 100MHz, but note that they all can run at 400MHZ.

Our transformed code makes VivadoHLS use more LUTs for less DSPs. Again this is due to Vivado's use of DSP for the shifters. In all cases, on this example, the transformed code has its latency reduced by a factor 20.

IV. Evaluation on FPMark benchmarks

In order to evaluate the relevance of the proposed transformations on real-life programs, we used the EEMBC FPMark benchmark suite [1].

	Naive	Transformed $LSB_A = -14$	Transformed $LSB_A = -20$	Transformed $LSB_A = -50$
LUTs	538	693	824	1400
DSPs	5	2	2	2
Latency	2000K	100 K	100K	100K

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

This suite consists of 10 programs. A first result is that half of these programs contain visible accumulations:

- Enhanced Livermore Loops (1/16 kernels contains one accumulation)
- LU Decomposition (multiple accumulations)
- Neural Net (multiple accumulations)
- Fourier Coefficients (one accumulation)
- Black Scholes (one accumulation)

The following focuses on these, and ignores the other half (Fast Fourier Transform, Horner's method, Linpack, ArcTan, Ray Tracer).

Most benchmarks come in single-precision and double-precision versions, and we focus here on the single-precision ones.

A. Benchmarks and accuracy: methodology

Each benchmark comes with a golden reference against which the computed results are compared. As the proposed transformations are controlled by the accuracy, it may happen that the transformed benchmark is less accurate than the original. In this case, it will not pass the benchmark verification test, and rightly so.

A problem is that the transformed code will also fail the test if it is *more* accurate than the original. Indeed, the golden reference is the result of a certain combination of rounding errors using the standard FP formats, which we do not attempt to replicate.

To work around this problem, each benchmark was first transformed into a high-precision version where the accumulation variable is a 10,000-bit floating-point numbers using the MPFR library. We used the result of this highly-accurate version as a "platinum" reference, against which we could measure the accuracy of the benchmark's golden reference. This allowed us to choose our `epsilon` parameter such that the transformed code would be at least as accurate as the golden reference.

B. Benchmarks improved by the proposed transformation

Enhanced Livermore Loops: This program contains 16 kernels of loops that compute physics equations. Among these kernels, there is one that performs a sum-of-product (banded linear equations). This kernel computes 20000 sums-of-products. The values accumulated are taken from an array. This is a perfect candidate for the proposed transformations.

For this benchmark, the optimal accumulation parameters were found as:

```
MaxAcc=50000 epsilon=1e-5 MaxInput=22000
```

Synthesis results of both codes (before and after transformation) are given in Table III. As in the previous toy examples, latency and accuracy are vastly improved for comparable area.

	Latency	LUTs	DSPs	Accuracy
Original Code	80K	384	5	11 bits
Transformed Code	20K	576	2	13 bits

Table III: Synthesis results of one Livermore Loops kernel before and after transformations.

LU Decomposition and Neural Net: Both the LU decomposition and the neural net programs contain multiple nested small accumulations. In the LU decomposition program, an inner loop accumulates between 8 and 45 values. Such accumulations are performed more than 7M times. In the neural net program, inner loops accumulate between 8 and 35 values, and such accumulations are performed more than 5K times.

Both of these programs accumulate values from registers or memory that are already computed. It makes these programs good candidates for the proposed transformations.

Due to lack of time, and also because VivadoHLS is unable to predict a latency for these implemented designs due to their variable loop sizes, we do not present complete results for these two benchmarks. Still, in order to show that the approach works on these examples, the LU inner loops were transformed and synthesized. Table IV shows the results obtained for the smallest (8 terms) and the largest (45 terms) sums-of-products: the latency is vastly improved even for the smallest one.

		Latency	LUTs	DSPs
8 terms	Original Code	82	809	5
	Transformed Code	17	1007	2
45 terms	Original Code	452	819	5
	Transformed Code	54	1034	2

Table IV: Synthesis results of the loops from LU benchmark.

C. Benchmarks that suggest future work

Fourier Coefficients: The Fourier coefficients program contains an accumulation which is performed in simple precision. This program comes in three different configurations: small, medium and big. Each of them computes the same algorithm but with a different amount of iterations. The "big" version is supposed to compute the most accurate answer. We get similar results for the three versions of this program, so we only present the "big" version here. In this version, 2K terms are accumulated multiple times during the computation. The accumulator is reset at every call.

The parameters determined for this benchmark were the following:
`MaxAcc=6000 epsilon=1e-7 MaxInput=10`

This results in an accumulator using 14 bits for the integer part and 24 bits for the fractional part. The synthesis results obtained for the original and transformed codes are given in Table V.

	Latency	LUTs	DSPs	Accuracy
Original Code	8K	34596	64	6 bits
Transformed Code	8K	34681	59	11 bits

Table V: Synthesis results of the Fourier coefficients program before and after transformations.

Here, area is again comparable, accuracy is improved by one order of magnitude, but latency is not improved. To understand why, Listing 6 provides an example illustrating the problem. The variable `x` from statement 1 (S1) is used as a function parameter in S2. This introduces an intra-iteration dependency between these statements. As we transform S2 to be an application-specific accumulator with a 1 cycle latency, we still have to wait for `x` to be ready at each iteration. This makes the overall latency of one iteration to be limited by the speed at which `x` can be computed.

As `x` is itself an accumulation, an idea is to apply the transformation to `x` as well. The problem is that S2 needs its value as a float at each iteration. If `x` is computed as a non-standard fixed point number, it will need to be converted to a float at each iteration, which takes several cycles.

In principle there is a solution, which is to decouple the two accumulation loops and insert a fix-to-float pipeline between them. However this transformation is well beyond the scope of this article and is left for future work.

Listing 6: Illustration of a loop intra-iteration dependency preventing a latency reduction

```
float x=0, acc=0;
for(int i=0; i<N; i++){
  S1:  x += dx;
  S2:  acc += f(...,x,...);
}
```

Black Scholes: This program contains an accumulation that sums 200 terms. The result of this computation is divided by a contant (that could be optimized by using transformations based on [4]). This process is performed 5000 times.

Here the optimal accumulator parameters are the following:
`MaxAcc=245000 epsilon=1e-4 MaxInput=278`
This gives us an accumulator that uses 19 bits for the integer part and 10 bits for the fractional part. The result of the synthesis are provided in Table VI.

	Latency	LUTs	DSPs	Accuracy
Original Code	3M	15640	175	19 bits
Transformed Code	3M	15923	175	23 bits

Table VI: Synthesis results of the Black Scholes program before and after transformations. The last column provides the results for a hand tuned version where we split the loop in two.

Again, for comparable area, accuracy is vastly improved by latency is not improved. This time it is for a different reason, illustrated by Listing 7. Let us consider the latency of the floating-point accumulation to be N, and the latency

of the function `f` to be M. The latency of one iteration is then going to be $\max(N, M)$. In the case where $M < N$ our transformations will improve the overall latency by reducing it to M.

Listing 7: Illustration of a computation latency preventing a latency reduction

```
float acc=0;
for(int i=0; i<N; i++){
  acc += f(...);
}
```

However, in the context of Black Sholes, M is greater than N. Therefore, reducing the latency of the accumulation has no impact on performance, as we have to wait for the result of the computation of f at each iteration. Again, there is an architectural solution to this, but VivadoHLS is currently unable to find it.

V. 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. Some designers may not like this. To convince them, consider that established performance benchmarks compute results which are accurate only to a few bits. If only a few bits are important, do we really need to instanciate 32-bit or 64-bit floating-point operators to compute them ? Isn't this accuracy information worth investigating and exploiting?

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

The proposed transformation already works very well on 3 of the 10 FPMarks where it improves both latency and accuracy by an order of magnitude for comparable area. For 2 more benchmarks, the latency is not improved (but not degraded either) due to current limitations of HLS tools. This defines short-term future work, which could be addressed either within the back-end HLS tool, or as more source-to-source transformations.

In the longer term, we believe 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, and building upon compiler progresses in program analysis.

References

[1] EEMBC - the embedded microprocessor benchmark consortium. http://www.eembc.org/.

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

[3] 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.

[4] Florent de Dinechin and Laurent-Stéphane Didier. *Table-Based Division by Small Integer Constants*, pages 53–63. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[5] Florent de Dinechin and Bogdan Pasca. *High-Performance Computing Using FPGAs*, chapter Reconfigurable Arithmetic for High-Performance Computing, pages 631–663. Springer, 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 *18th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 773–779, Jan 2013.

[10] James Hrica. Floating-point design with vivado HLS, 2012. Xilinx Application Note.

[11] Xilinx Inc. Vivado Design Suite User Guide: High-Level Synthesis. 2015.

[12] Nachiket Kapre and Andre DeHon. Optimistic parallelization of floating-point accumulation. In *18th IEEE Symposium on Computer Arithmetic*, pages 205–216. IEEE, 2007.

[13] Ulrich Kulisch and Van Snyder. The exact dot product as basic tool for long interval arithmetic. *Computing*, 91(3):307–313, March 2011.

[14] Martin Langhammer. Floating point datapath synthesis for FPGAs. In *2008 International Conference on Field Programmable Logic and Applications*, pages 355–360, Sept 2008.

[15] 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.

[16] R. Nane, V. M. Sima, C. Pilato, J. Choi, B. Fort, A. Canis, Y. T. Chen, H. Hsiao, S. Brown, F. Ferrandi, J. Anderson, and K. Bertels. A survey and evaluation of fpga high-level synthesis tools. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 35(10):1591–1604, Oct 2016.

[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.