



High-Level Synthesis Using Application-Specific Arithmetic: A Case Study

Yohann Uguen, Florent de Dinechin, Steven Derrien

► To cite this version:

Yohann Uguen, Florent de Dinechin, Steven Derrien. High-Level Synthesis Using Application-Specific Arithmetic: A Case Study. 2017. hal-01502644

HAL Id: hal-01502644

<https://hal.science/hal-01502644>

Preprint submitted on 5 Apr 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.

High-Level Synthesis

Using Application-Specific Arithmetic:

A Case Study

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—On the one hand, a strength of FPGAs is their ability to perform non-standard computations not supported by classical microprocessors. Many libraries of highly customizable application-specific IPs have been developed to exploit this strength.

On the other hand, HLS tools, which allow to program an FPGA using a dialect of the C language, are gaining traction. However, the ease of use of the C language becomes a hindrance when one wants to express non-standard computations. Indeed, the C language was designed for programming microprocessors and carries with it many restrictions of the microprocessor paradigm. This is especially true when computing with floating-point, whose data-types and evaluation semantics are defined by the IEEE-754 and C11 standards. If the high-level specification was a computation on the reals, HLS imposes a very restricted implementation space.

This work attempts to bridge FPGA application-specific efficiency and HLS ease of use. This is illustrated on the ubiquitous floating-point summation-reduction pattern. A source-to-source compiler rewrites, inside critical loop nests of the input C code, selected floating-point additions into sequences of simpler operators using non-standard arithmetic formats.

Evaluation of this method demonstrates that the benefits of application-specific operators (better performance and better accuracy) can be brought to HLS workflows while keeping their ease of use.

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 Vivado HLS [11] or Catapult C¹ among others [18]. 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). Indeed, such a high-level language have been designed to be compiled and run on hardware and not to describe hardware. Therefore, **HLS tools performs better for general purpose computing whereas FPGAs performs better for application-specific computing.**

The broader objective of this work is to show that HLS tools can produce application-specific hardware and therefore, unleash FPGAs potential again.

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 size². 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

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

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

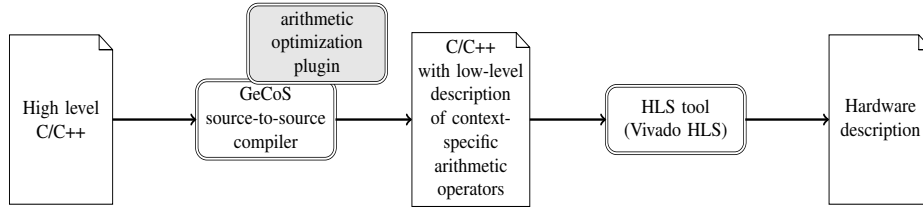


Figure 1: The proposed compilation flow

be encapsulated in a C function that is called to instantiate the operator.

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 sub-components 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 arithmetic 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 Vivado HLS 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 different version of Listing 1 that we coded such that Vivado HLS expresses more parallelism. Vivado HLS 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 as Listing 2 expresses more parallelism, Vivado HLS 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 [13].

Note that Listing 2 is only here as an example and might need more logic if N was not a multiple of 10.

Listing 1: Naive reduction

```

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

Listing 2: Parallel 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 when the user specifies it. 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], [2]. 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. The contribution of this paper, which are compiler transformations to generate C description of these operators in a HLS workflow, is presented in Section III. Section IV evaluates our approach on the FPMARK benchmark suite.

II. THE ARITHMETIC SIDE: AN APPLICATION-SPECIFIC ACCUMULATOR IN VIVADO HLS

The accumulator that we used for this paper is based on a more general idea developed by Kulisch. He advocated a very large floating-point accumulator [14] whose 4288 bits would

| | 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.

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 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 is the one offered in FloPoCo [6], although it is not the contribution of this paper, we included two improvements to it:

- 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 (special floating-point numbers with leading zeros to their significand to fill the underflow gap around 0).

Note that we could have implemented any other non-standard operator performing a reduction such as [16], [12].

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 $MSB_A = 20$, 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 $2^{-50} \approx 10^{-15}$.

Such a fixed-point format is illustrated in Figure 3.

The accumulator width w_a is then computed as $MSB_A - LSB_A + 1$, 71 bits in the previous example. 71 bits represents a wide range and high accuracy, and still additions on this format

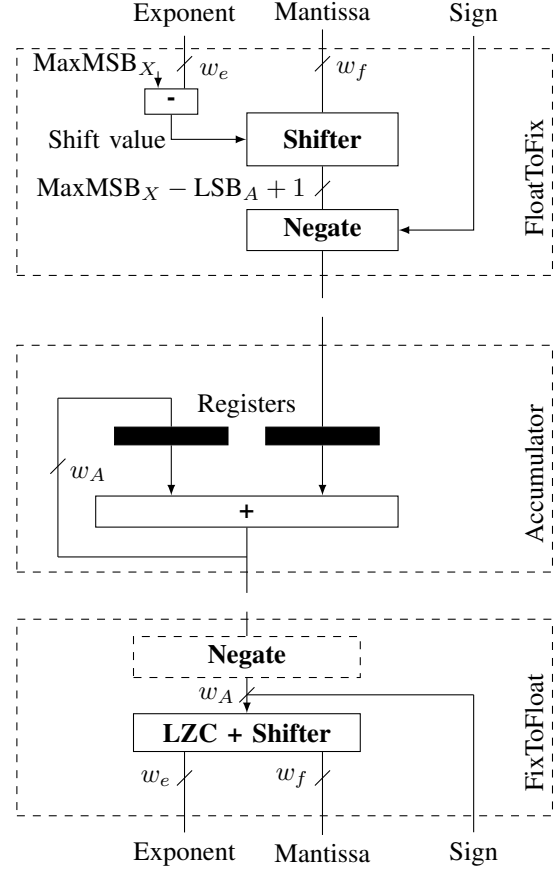


Figure 2: The conversion from float to fixed-point (top), the fixed-point accumulation (middle) and the conversion from the fixed-point format to a float (bottom).

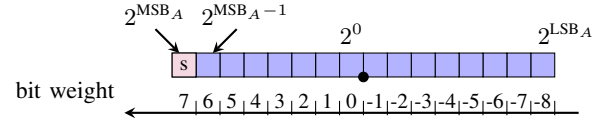


Figure 3: The bits of a fixed-point format, here $(MSB_A, LSB_A) = (7, -8)$.

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.

In the following of this paper, we refer as *latency* of a

circuit the number of cycles needed for the entire application to complete.

B. Implementation within a HLS tool

This accumulator has been implemented in C, using arbitrary-precision fixed point types (`ap_int`). The leading zero count, bit range selections and other operations are implemented using Vivado HLS built-in functions. For modularity purposes, the `FloatToFix` and `FixToFloat` are wrapped into C functions (28 LoC for the `FloatToFix`, 22 LoC for `FixToFloat`) whose calls are inlined to enable HLS optimizations.

Because the internal accumulation is performed on a fixed-point integer representation, the combinational delay between two accumulations is lower compared to a direct floating point addition. We expect HLS tools to benefit from this delay reduction by taking advantage of more aggressive loop pipelining (with shorter Initiation Interval pipeline) resulting in a design with a shorter overall latency.

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. [17] the input values $c[i] = (\text{float}) \cos(i)$, 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 Vivado HLS 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.

Vivado HLS 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, Vivado HLS 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 better 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.

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);
```

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

We have shown in previous Section that Vivado HLS can be used to synthesize very efficient specialized floating point operators which rival in quality with those generated by FloPoCo. Our goal is now to study how such optimization can benefit from automation. More precisely, we aim at being able to automatically optimize Listing 1 into Listing 3, and generalizes this transformation to many more situations.

For convenience, we chose to develop our optimization as a source-to-source transformation implemented within the open source GeCoS compiler framework [8], and aim at making our tool publicly available. Source-to-source compiler are very convenient in a HLS context since they can be used as optimization front-ends on top of closed-source commercial tools.

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. Reduction pattern are exposed to the compiler/runtime either through user directives (e.g `#pragma reduce` in openMP), or automatically inferred using static analysis techniques [19], [7].

As the problem of detecting reductions is not the main focus on this work, our tool uses a straightforward solution to the problem using a combination of user directive and (simple) program analysis. More specifically, the user must identify a target accumulation variable through a `pragma`, and provide additional information such as the dynamic range

of the accumulated data along with the target accuracy (in the future, we expect to automate our flow such that the two later parameter could systematically be omitted).

We found this approach easier, more general and less invasive than those attempting to convert a whole floating-point program into a fixed-point implementation [20].

A. Proposed compiler directive

In imperative languages such as C, reductions are implemented using `for` or `while` loop constructs. Our compiler directive must therefore appear right outside 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 among all 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 .

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=100000.0
        epsilon=1E-15 MaxInput=1.0
    for(int i=0; i<N; i++){
        acc+=in[i];
    }
    return acc;
}
```

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. Proposed code transformation

The proposed transformation operates on the compiler program intermediate representation (IR), and rely on the ability to identify loops constructs and expose def/use relations between instructions of a same basic block in the form of an operation dataflow graph (DFG).

To illustrate our transformation, consider the sample program shown in Listing 5. This program performs a reduction into the variable `sum`, involving both sums and sums of product operations. The operation dataflow graph associated to the basic block forming the loop body in this program is

depicted in Figure 4a. In this Figure, dotted arrows represent loop carried dependencies between operations belonging to distinct loop iterations. Such loop carried dependencies have a very negative impact on the kernel latency as they prevent loop pipelining. For example, when using a pipelined floating-point adder with a seven cycle latency, the HLS tool will schedule a new iteration of the loop at best every seven cycle.

As illustrated in Figure 5a, our proposed optimization hoists the floating-point normalization step out of the loop, and performs the accumulation using fixed point arithmetic. Since integer add operations can generally be implemented with a 1-cycle delay, the HLS tool may now be able to initiate a new iteration every cycle, improving the overall latency by a factor of 7.

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=300000.0
        epsilon=1e-15 MaxInput=3.0
    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 first identifies all relevant basic blocks (i.e those associated to the pragma directive). It then performs a backward traversal of the dataflow graph, starting from a `Float Add` node that writes to the accumulation variable identified by the `#pragma`.

During this traversal, the following actions are performed depending on the visited nodes:

- A node with the summation variable is ignored.
- A `Float Add` node is transformed to an accurate fixed-point adder. The analysis is then recursively launched on that node.
- A `Float Mul` 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 DFG from Figure 4a into the new DFG shown on Figure 5a. In addition, a new basic block containing a call to `FixToFloat` is inserted immediately after the transformed loop, in order to expose the floating point representation of the results to the remainder of the program.

From there, it is then possible to regenerate the corresponding C code. As an illustration of the whole process, the synthesized codes from before and after the transformations result in the architectures from Figure 4b and Figure 5b respectively.

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

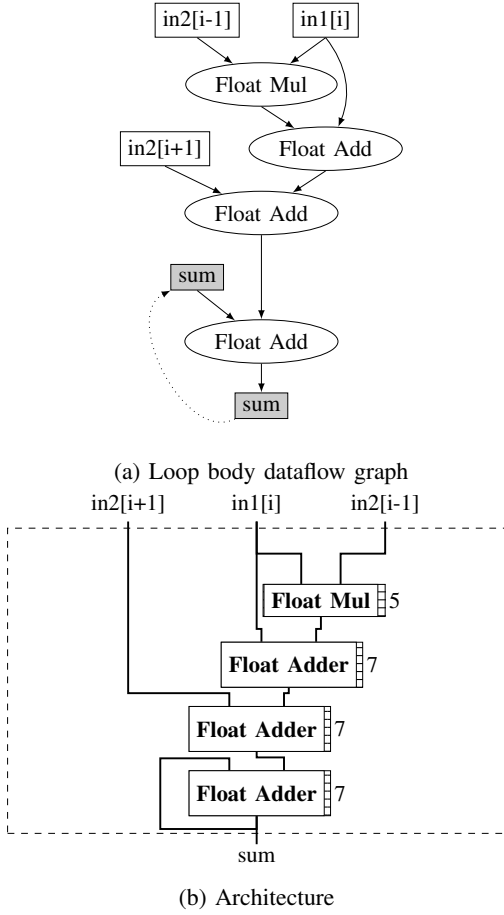


Figure 4: DFG of the loop body from Listing 5 (top) and its corresponding architecture (bottom). Keywords *float mul* and *float add* correspond to floating-point multipliers and adders respectively

depends on `epsilon`, all the other parameters being those given in the `pragma` of Listing 5. All these versions were synthesised for 100 MHz. Indeed, some circuits might need more work than others to increase their frequency and would not give a fair comparison.

Our transformed code makes Vivado HLS use more LUTs for less DSPs compared to the classical IEEE-754 implementation. This is due to the smaller shifter and multiplier of our implementation. In all cases, on this example, the transformed code has its latency reduced by a factor 20. This is due to the fact that Vivado HLS is able to perform the `Float Mul` and the `Float Adder` in a single block with a latency of 10 cycles for the naive version. The two following `Float Adder` are accelerated from 14 cycles to 10 cycles due to a very short pipeline that follows the IEEE-754 standard. Comparatively, the transformed code has 1 cycle latency operators that can be pipelined.

IV. EVALUATION ON FPMARK BENCHMARKS

In order to evaluate the relevance of the proposed transformations on real-life programs, we used the EEMBC FPMark

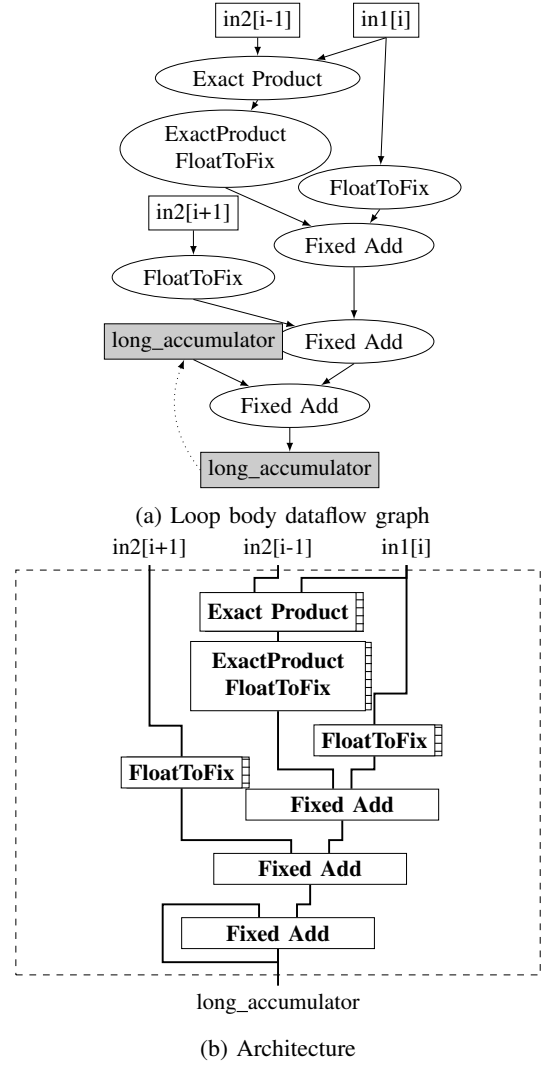


Figure 5: DFG of the loop body from Listing 5 (top) and its corresponding architecture (bottom) after transformations

| | 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.

benchmark suite [1].

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. This way, the `epsilon` of the following results is obtained through profiling. The accuracy of the obtained results are computed as the number of correct bits of the result.

We first present the benchmarks that are improved by our approach before discussing the reasons why we can’t prove that the others are.

B. Benchmarks improved by the proposed transformation

Enhanced Livermore Loops: This program contains 16 kernels of loops that compute numerical 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 pre-computed. This is a perfect candidate for the proposed transformations.

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

```
MaxAcc=50000.0 epsilon=1e-5
MaxInput=22000.0
```

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.

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.

| Benchmark | Type | LUTs | DSPs | Latency | Accuracy |
|-----------|-------------|-------|------|---------|-----------|
| Livermore | Original | 384 | 5 | 80K | 11 bits |
| | Transformed | 576 | 2 | 20K | 13 bits |
| LU-8 | Original | 809 | 5 | 82 | 8-23 bits |
| | Transformed | 1007 | 2 | 17 | 23 bits |
| LU-45 | Original | 819 | 5 | 452 | 8-23 bits |
| | Transformed | 1034 | 2 | 54 | 23 bits |
| Scholes | Original | 15640 | 175 | N/A | 19 bits |
| | Transformed | 15923 | 175 | N/A | 23 bits |
| Fourier | Original | 34596 | 64 | N/A | 6 bits |
| | Transformed | 34681 | 59 | N/A | 11 bits |

Table III: Synthesis results of benchmarks before and after transformations

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

Vivado HLS is unable to predict a latency for these implemented designs due to their non-constant loop counts, therefore 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 III shows the results obtained for the smallest (8 terms) and the largest (45 terms) sums-of-products in lines LU-8 and LU-45 respectively. The latency is vastly improved even for the smallest one. The accuracy results of the original code here varies from 8 to 23 bits between different instances of the loops. To have a fair comparison, we generated a conservative design that performs 23 bits accuracy on all loops, using a sub-optimal amount of resources.

C. Benchmarks that exposed the limitations of HLS tools

Black Scholes: This program contains an accumulation that sums 200 terms. The result of this computation is divided by a constant (that could be optimized by using transformations based on [3]). This process is performed 5000 times.

Here the optimal accumulator parameters are the following:
MaxAcc=245000.0 epsilon=1e-4
MaxInput=278.0

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 III.

For comparable area, accuracy is vastly improved but latency could not be obtained from Vivado HLS. Indeed, the Black Scholes algorithm uses the mathematical function *power*. Such a function is not implemented in Vivado HLS, therefore we coded it using a loop with a non-constant count. As the latency of this operator depends on the input data, the latency of the all circuit cannot be statically recovered.

Fourier Coefficients: The Fourier coefficients program, which computes the coefficients of a Fourier series, contains an accumulation which is performed in single 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 obtain similar results for the three versions of this program, as a consequence

we only present the "big" version here. In this version, there are multiple instances of 2K terms accumulations. The accumulator is reset at every call.

The parameters determined for this benchmark were the following:

MaxAcc=6000.0 epsilon=1e-7 MaxInput=10.0

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 III.

Here, area is again comparable, accuracy is improved by 5 bits (which represents one order of magnitude), but latency could not be obtained. Vivado HLS faces the same problem as in Black Scholes, it cannot compute the latency of the *power* function.

Note that our operators have a shorter latency by nature. Therefore, even if Vivado HLS is not able to provide latency results, the circuits indeed have a shorter latency.

The *pow* operator could be implemented such as in [4] in the near future. This would allow us to obtain the latency of the two last benchmarks.

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 instantiate 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 Vivado HLS.

The proposed transformation already works very well on all the FPMarks that contains a reduction where it improves both latency and accuracy by an order of magnitude for comparable area.

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 specialization; operator fusion such as in [21]

but at a more coarse grain allowing more aggressive fusion; compile-time generation of application-specific cores; error analysis such as in [15] benefiting from compilers static analysis and more generally building upon compiler progresses in program analysis.

REFERENCES

- [1] EEMBC - the embedded microprocessor benchmark consortium. <http://www.eembc.org/>.
- [2] G. Caffarena, J. A. Lopez, C. Carreras, and O. Nieto-Taladriz. High-level synthesis of multiple word-length DSP algorithms using heterogeneous-resource FPGAs. In *Field Programmable Logic and Applications*, pages 1–4, 2006.
- [3] F. de Dinechin and L-S. Didier. *Table-Based Division by Small Integer Constants*, pages 53–63. 2012.
- [4] F. de Dinechin, P. Echeverria, M. Lopez-Vallejo, and B. Pasca. Floating-Point Exponentiation Units for Reconfigurable Computing. *ACM Transactions on Reconfigurable Technology and Systems*, pages 4:1–4:15, 2013.
- [5] F. de Dinechin and B. Pasca. *High-Performance Computing Using FPGAs*, chapter Reconfigurable Arithmetic for High-Performance Computing, pages 631–663. 2013.
- [6] F. de Dinechin, B. Pasca, O. Creț, and R. Tudoran. An FPGA-specific approach to floating-point accumulation and sum-of-products. In *Field-Programmable Technologies*, pages 33–40. IEEE, 2008.
- [7] J. Doerfert, K. Streit, S. Hack, and Z. Benaissa. Polly's polyhedral scheduling in the presence of reductions. *International Workshop on Polyhedral Compilation Techniques*, 2015.
- [8] A. Floc'h, T. Yuki, A. El-Moussawi, A. Morvan, K. Martin, M. Naullet, M. Alle, L. L'Houss, N. Simon, S. Derrien, F. Charot, C. Wolinski, and O. Sentieys. GeCoS: A framework for prototyping custom hardware design flows. In *Source Code Analysis and Manipulation*, pages 100–105, 2013.
- [9] M. Gort and J. H. Anderson. Range and bitmask analysis for hardware optimization in high-level synthesis. In *Asia and South Pacific Design Automation Conference*, pages 773–779, 2013.
- [10] J. Hrica. Floating-point design with vivado HLS, 2012. Xilinx Application Note.
- [11] Xilinx Inc. Vivado Design Suite User Guide: High-Level Synthesis. 2015.
- [12] E. Kadric, P. Gurniak, and A. DeHon. Accurate parallel floating-point accumulation. *IEEE Transactions on Computers*, pages 3224–3238, 2016.
- [13] N. Kapre and A. DeHon. Optimistic parallelization of floating-point accumulation. In *Symposium on Computer Arithmetic*, pages 205–216, 2007.
- [14] U. Kulisch and V. Snyder. The exact dot product as basic tool for long interval arithmetic. *Computing*, pages 307–313, 2011.
- [15] M. Langhammer and T. VanCourt. FPGA floating point datapath compiler. In *Field Programmable Custom Computing Machines*, pages 259–262, 2009.
- [16] Z. Luo and M. Martonosi. Accelerating pipelined integer and floating-point accumulations in configurable hardware with delayed addition techniques. *IEEE Transactions on Computers*, pages 208–218, 2000.
- [17] J-M. Muller, N. Brisebarre, F. de Dinechin, C-P. Jeannerod, V. Lefèvre, G. Melquiond, N. Revol, D. Stehlé, and S. Torres. *Handbook of Floating-Point Arithmetic*. Birkhäuser Boston, 2010.
- [18] 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. *Computer-Aided Design of Integrated Circuits and Systems*, pages 1591–1604, 2016.
- [19] X. Redon and P. Feautrier. Detection of scans in the polytope model. *Parallel Algorithms and Applications*, pages 229–263, 2000.
- [20] O. Sentieys, D. Menard, D. Novo, and K. Parashar. Automatic Fixed-Point Conversion: a Gateway to High-Level Power Optimization. Design Automation and Test in Europe, 2014.
- [21] D. Ye and N. Kapre. MixFX-SCORE: Heterogeneous fixed-point compilation of dataflow computations. In *Field-Programmable Custom Computing Machines*, pages 206–209, 2014.