Application-specific arithmetic in high-level synthesis tools

Yohann Uguen, Florent de Dinechin, Victor Lezaud, Steven Derrien

Application-specific arithmetic in high-level synthesis tools

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

Victor Lezaud
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

December 27, 2019

Abstract

This work studies hardware-specific optimization opportunities currently unexploited by HLS compilers. Some of these optimizations are specializations of floating-point operations that respect the usual semantics of the input program, without changing the numerical result. Some other optimizations, locally triggered by the programmer thanks to a pragma, assume a different semantics, where floating-point code is interpreted as the specification of computation with real numbers. The compiler is then in charge to ensure an application-level accuracy constraint expressed in the pragma, and has the freedom to use non-standard arithmetic hardware when more efficient. These two classes of optimizations are prototyped in the GeCoS source-to-source compiler, and evaluated on the Polybench and EEMBC benchmark suites. Latency is reduced by up to 93%, and resource usage is reduced by up to 58%.

1 Introduction

Many case studies have demonstrated the potential of Field-Programmable Gate Arrays (FPGAs) as accelerators for a wide range of applications, from scientific and financial computing to signal and data processing, bioinformatics, molecular dynamics, stencil computations and cryptography [46]. FPGAs offer massive parallelism and programmability at the bit level. These characteristics enable programmers to exploit a range of techniques that avoid many bottlenecks of classical von Neumann computing: data-flow 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 development costs are the two faces of the

[0]Extension of Conference Paper. This work extends [43] by proposing other floating-point arithmetic optimizations for high-level synthesis tools. The previous optimizations were modifying the semantic of the original program to boost performance and accuracy. This work adds semantic preserving optimizations that should be applied in every scenario.

same coin. One reason for the high development costs on FPGAs is that they inherited their programming model from digital circuit design. The hardware description languages (HDLs) used have a very different programming paradigm than the languages used for software design. When accelerating software code with FPGAs, software-oriented languages such as `C/C++` are increasingly being considered as hardware description languages. This has many advantages. Experimenting with different options of FPGA acceleration is simpler since no rewriting of software into HDL is needed. The language itself is more widely known than any HDL. The sequential execution model makes designing and debugging much easier. One can use software execution on a processor for simulation. All this drastically reduces development time.

Compiling a software description (such as a C program) into hardware is called High-Level Synthesis (HLS). Examples of HLS tools include Vivado HLS [3], Intel HLS[2] or Catapult C [1] among others [35]. Turning a C description into a circuit 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 good at this task, and can automatically synthesize highly efficient pipelined data-flow architectures.

Early HLS tools had a bottom-up approach, inherited from hardware design, of assembling optimized hardware IP components into more complex designs. This approach failed to scale to large and irregular applications. Conversely, modern HLS tools are built upon mainstream compiler projects such as GCC or Clang/LLVM. This top-down approach, from the language down to the hardware, ensures comprehensive language support and a large number of state-of-the-art optimization techniques [33, 32, 5]. However, most of these optimizations were designed for standard CPUs. It is therefore relevant to question if they make sense in an FPGA context [25]. It is also relevant to attempt to identify new optimizations that did not make sense in a software context but make sense in the FPGA context. Even hardware-specific optimizations that were already performed by some of the early HLS tools can be reconsidered and improved in the new context of modern optimizing compiler frameworks.

This is the main objective of the present work, with a focus on arithmetic-related optimizations.

Consider for example the integer multiplication by a constant. Optimization of hardware constant multiplication has been the subject of much research [14, 23, 47, 4, 41, 30], some of which is specific to FPGAs [8, 49, 48, 11]. Some of the early HLS tools could exploit this literature and generate an optimized architecture. However, it was not optimized for the big picture: this is what modern compiler frameworks provide, with global program transformations such as (interprocedural) constant propagation and folding, strength reduction, etc [33]. Unfortunately, some optimizations of the leaf hardware operators got lost in this transition to modern compiler frameworks, as we now show.

Listing 1 implements a simple integer multiplication by 7. Listing 2 shows the assembly code of Listing 1, when compiled with gcc 7.4.0 without any particular optimization flag, targeting an x86 microprocessor. One can see that the multiplication by 7 has been transformed by the compiler into a *shift-and-add* algorithm (here actually a shift-and-sub, hardware addition and subtraction being equivalent in this context): $7x = 8x - x = 2^3x - x$, where the multiplication by 2^3 is a simple shift left by 3 bits. Both on GCC or Clang/LLVM, if the code is compiled using -O2 optimization flag, the algorithm is the same but the multiplication by 8 is implemented by the `lea` instruction in a slightly less obvious way.

As a consequence, the architecture produced by an HLS tool based on GCC or Clang/LLVM will use a shift-and-add algorithm. It makes even more sense in HLS, since the constant shifts reduce to wires and therefore cost very little hardware. Indeed, the synthesis of Listing 1 in VivadoHLS reports 32 LUTs (on a Kintex7 FPGA), the cost of one 32-bit adder. Experiments with Vivado HLS (based

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

```
int mul7(int x){
  return x*7;
}
```

Listing 1: C code.

```
(...)
        a: 89 d0        mov  %edx,%eax
        c: c1 e0 03     shl  $0x3,%eax
        f: 29 d0        sub  %edx,%eax
(...)
```

Listing 2: Objdump of Listing 1 when compiled with gcc (x86).

```
int mul2228241 ( int x ){
  return x*2228241;
}
```

Listing 3: C code.

```
(...)
        10: ... imul $0x220011,%edi,%eax
        16: ... retq
(...)
```

Listing 4: Objdump of Listing 3 compiled with Clang/LLVM -O2 (x86).

on Clang/LLVM) and Intel HLS (based on GCC) show that for all the constant multiplications that can be implemented as an addition, these tools instantiate an adder instead of a multiplier.

Now consider the multiplication by another constant in Listing 3. On this example, the Clang/LLVM x86 backend keeps the operation as a multiplication.

Indeed, the synthesis of this operator by VivadoHLS on a Kintex reports 2 LUTs and 2 DSPs, which are the resources needed to implement a 32-bit multiplier.

However, although the constant looks more complex, it barely is: the multiplication by 2228241 can be implemented in two additions as $2228241 = 2^{17} \cdot 17 + 17$: first compute $t = 17x = 2^4x + x$ (one addition), then compute $2228241x = 2^{17}t + t$ (another addition). Still, neither the x86 backend of Clang/LLVM nor GCC use a shift-and-add in this case. The rationale could be the following: the cost of one addition will always be lower than or equal to the cost of a multiplication, whatever the processor, so replacing one multiplication with one addition is always a win. Conversely, it may happen on some (if not most) processors that the cost of two additions and two shifts is higher than the cost of one multiplication.

Is this true in an HLS context? The best architecture for this multiplication, achieved by the C program of Listing 5, consists of two adders: one that computes the 32 lower bits of $t = 17x = 2^4x + x$ (and should cost only 28 LUTs, since the lower 4 bits are those of x); one that computes the 32 lower bits of $2^{17}t + t$, and should cost 32-17=15 LUTs, for the same reason (the 17 lower bits are those of t). The total cost should be 43 LUTs.

For Listing 5, VivadoHLS indeed reports 46 LUTs, very close to the predicted 43 (and not much higher than the cost of the multiplication by 7).

In summary, the observation is that the arithmetic optimization has been completely delegated to the underlying compiler x86 backend, and that we have a case here for enabling further optimizations when targetting hardware or FPGAs.

The broader objective of the present work is to list similar opportunities of hardware-specific

```
int mul2228241 ( int x ){
  int t = (x<<4) + x;
  return  (t<<17) + t;
}
```

Listing 5: C code using a shift-and-add algorithm.

arithmetic optimizations that are currently unexploited, and demonstrate their effectiveness. We classify these optimizations in two broad classes:

- *Optimizations that strictly respect the semantic of the original program* are presented in Section 2. The previous constant multiplication examples belong to this class, we also discuss division by a constant, and we add in this section a few floating-point optimizations that make sense only in a hardware context. This section should be perfectly uncontroversial: all optimizations in this class should be available in an HLS flow as soon as they improve some metric of performance. The only reason why it is not yet the case is that commercial HLS tools are relatively young and don't include the portfolio of HLS optimizations that have been researched.

- *Optimizations that relax (and we argue, only for the better) the constraint of preserving the program semantics* are presented in Section 3. In this more controversial and forward-looking Section, we assume that programmers who used floating-point datatypes in their programs intended to compute with real numbers, and we consider optimizations that lead to cheaper and faster, but also more accurate hardware. This approach is demonstrated in depth on examples involving floating-point summations and sums of products.

In each case, we use a compilation flow illustrated by Figure 1 that involves one or several source-to-source transformations using the GeCoS framework [19] to improve the generated designs. Source-to-source compilers are very convenient in an HLS context, since they can be used as optimization front-ends on top of closed-source commercial tools. This approach is not new as source-to-source compilers are already used in an HLS context for dataflow optimization [9].

Finally, Section 4 concludes and discusses what we believe HLS tools should evolve to.

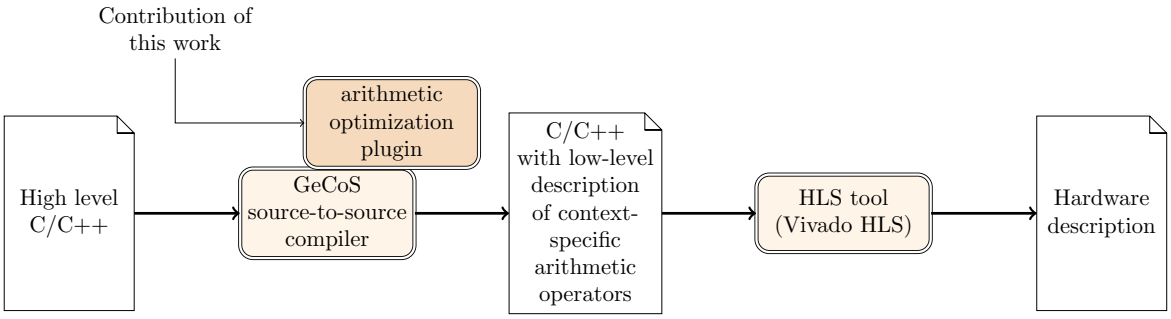


Figure 1: The proposed compilation flow.

2 Optimization examples that do not change the program semantic

The arithmetic optimizations that fit in this section go well beyond the constant multiplications studied in introduction. In particular, there are opportunities of floating-point optimizations in FPGAs that are more subtle than operator specialization.

2.1 Floating-point corner-case optimization

Computing systems follow the IEEE-754 standard [1] on floating-point arithmetic, which was introduced to normalize computations across different CPUs. Based on this standard, the C standard prevents compilers from performing some floating-point optimizations. Here are a some examples that can be found in the C11 standard [26]:

- x/x and 1.0 are not equivalent if x is zero, infinite, or NaN (in which case the value of x/x is NaN).
- $x - y$ and $-(y - x)$ are not equivalent because $1.0 - 1.0$ is $+0$ but $-(1.0 - 1.0)$ is -0 (in the default rounding direction).
- $x - x$ and 0 are not equivalent if x is a NaN or infinite.
- $0 \times x$ and 0 are not equivalent if x is a NaN, infinite, or -0.
- $x + 0$ and x are not equivalent if x is -0, because (-0) + (+0), in the default rounding mode (to the nearest), yields +0, not -0.
- $0 - x$ and $-x$ are not equivalent if x is +0, because -(+0) yields -0, but 0 - (+0) yields +0.

Of course, programmers usually do not write x/x or $x+0$ in their code. However, other optimization steps, such as code hoisting, or procedure specialization and cloning, may lead to such situations: their optimization is therefore relevant in the context of a global optimizing compiler [33].

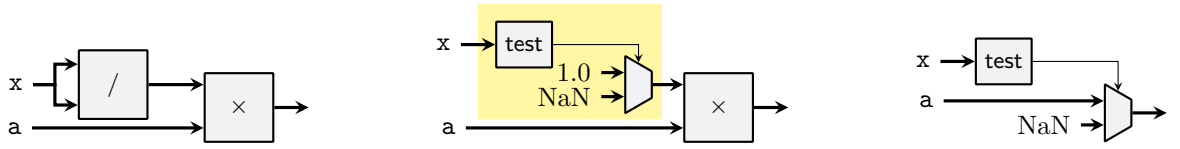
Let us consider the first example (the others are similar): A compiler is not allowed to replace x/x with 1.0 unless it is able to prove that x will never be zero, infinity or NaN. This is true for HLS as well as for a standard compiler. However, it could replace `x/x` with something like `(is_zero(x)||is_infty(x)||is_nan(x))?NaN:1.0;`. This is, to our knowledge, not implemented. The reason is again probably that in software, the test on `x` becomes more expensive than the division.

However, if implemented in hardware, this test is quite cheap: it consists in detecting if the exponent bits are all zeroes (which capture the 0 case) or all ones (which captures both infinity and NaN cases). The exponent is only 8 bits for single precision and 11 bits for double-precision.

In an FPGA context, it therefore makes perfect sense to replace x/x (Figure 2a) with an extremely specialized divider depicted in Figure 2b. Furthermore, the two possible values are interesting to propagate further (1.0 because it is absorbed by multiplication, NaN because it is extremely contagious). Therefore, this optimization step enables further ones, where the multiplexer is pushed down the computation, as illustrated by Figure 2c.

Note that figure 2c replaces `1.0*a` by `a`: this is a valid floating-point optimization, in the sense that it is valid even if a is a signed zero, an infinity or a NaN.

Occurrences of $x - x$, $0 \times x$, $x + 0$, $0 - x$ can similarly be replaced with a multiplexer and very little logic, and may similarly enable further optimizations.



(a) Original circuit (b) Divider specialization (c) Multiplier specialization

Figure 2: Optimization opportunities for floating-point $x/x * a$.

Since these arithmetic optimizations are expected to be triggered by optimizations (procedure specialization) and trigger further optimizations (conditional constant propagation), they need to be implemented and evaluated within an optimizing compiler. The source-to-source flow depicted on Figure 1 is ill-suited to studying such cascaded optimizations. Furthermore, the multiple conditional constant propagation that transforms Figure 2b into Figure 2c is probably not implemented yet, since it doesn't make much sense in software. This evaluation is therefore left out of the scope of the present article.

In the following, we focus on FPGA-specific semantic-preserving optimizations which do not trigger further optimizations.

2.2 Integer multiplication by a constant

Multiplication by a constant has already been mentioned in introduction. We just refer to the rich existing literature on the subject [8, 14, 49, 23, 47, 4, 41, 48, 30, 11]. These are mostly academic works, but back-end tools already embed some of it, so this optimization could be the first to be implemented. An issue is that its relevance, in the big picture of a complete application, is not trivial: Replacing DSP resources with logic resources is an optimization only in a design that is more DSP-intensive than logic-intensive. Besides, as soon as a logic-based constant multiplier requires more than a handful of additions, it may entail more pressure on the routing resources as well. Discussing this trade-off in detail in the context of an application is out of scope of the present article.

2.3 Integer division by a small constant

Integer division by a constant adds one more layer of optimization opportunities: In some cases, as illustrated by Listing 6 and Listing 7, a compiler is able to transform this division into a multiplication by a (suitably rounded) reciprocal. This then triggers the previous optimization of a constant multiplier. Actually, one may observe that on this example that the constant $1/7$ has the periodic pattern $100100100100100100100100_2$ (hidden in the hexadecimal pattern 924_{16} in Listing 7). This enables a specific optimization of the shift-and add constant multiplication algorithm [10].

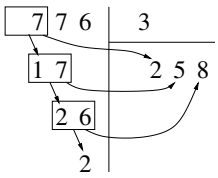
Table 1 shows synthesis results on the two FPGA mainstream HLS tools at the time of writing (Vivado HLS and Intel HLS). As a reminder of the resource names used by these tools: LUTs and ALMs are look-up tables, Regs. and FFs are registers, DSPs are dedicated hardware blocks (essentially for multipliers), SRLs are shift registers and MLABs are aggregated LUTs to emulate small RAM blocks. The timing constraint was set to 100 MHz, however this factor is not important here as it does not change the structure of the generated operators. The goal here is to observe the optimizations performed (or not) by the tools. Here is what we can infer from this table:

```
int div7(int in){
  return in/7;
}
```

Listing 6: C code.

```
(...)
        0: ... movslq %edi,%rax
        3: ... imul $0xffffffff92492493,%rax,%rcx
        a: ... shr $0x20,%rcx
        e: ... add %ecx,%eax
       10: ... mov %eax,%ecx
       12: ... shr $0x1f,%ecx
       15: ... sar $0x2,%eax
       18: ... add %ecx,%eax
       1a: ... retq
```

Listing 7: Objdump of Listing 6 when compiled with Clang -O2 (x86).



We first compute the Euclidean division of 7 by 3. This gives the first digit of the quotient, here 2, and the remainder is 1. In other words $7 = 3 \times 2 + 1$. The second step divides 77 by 3 by first rewriting $77 = 70 + 7 = 3 \times 20 + 10 + 7$: dividing 17 by 3 gives $17 = 3 \times 5 + 2$. The third steps rewrites $776 = 770 + 6 = 250 + 20 + 6$ where $26 = 3 \times 8 + 2$, hence $776 = 3 \times 258 + 2$.
The only computation in each step is the Euclidean division by 3 of of a number between 0 and 29: it can be pre-computed for these 30 cases and stored in a look-up table (LUT).

Figure 3: Illustrative example: division by 3 in decimal.

- The generic divider (Divisor=x) is based on Xilinx on a shift-and-add algorithm, while on Intel a polynomial approach is used [36] that consumes multipliers and DSP resources.
- Both tools correctly optimize the division by a power of two, converting it into a shift.
- Division by non-power of two integers is implemented by a multiplication by the inverse on Xilinx (it consumes DSP blocks). On Intel, this multiplication is further optimized as a logic-only operation.

For the division of an integer by a very small constant, the best alternative is the algorithm described by Ugurdag et al. [45]. It is based on the decimal *paper-and-pencil* algorithm illustrated in Figure 3. Figure 4 describes an unrolled architecture for a binary-friendly variant of this algorithm. There, the input X is written in hexadecimal (each 4-bit word X_i is an hexadecimal digit). The quotient bits come out in hexadecimal. The remainder of the division by 3 is always between 0 and 2, therefore fits on 2 bits. Each look-up table (LUT) on the figure therefore stores the quotient Q_i and the remainder R_i of the division by 3 of a number $R_{i+1}X_i$. This number is between 00_h and $2F_h$. On a recent LUT-based FPGA, each 6-input, 6-output LUT of Figure 4 consumes exactly 6 FPGA LUTs: This architecture is very well suited to FPGAs.

Table 2 compares the performance on Xilinx of the division of a 64-bit integer by a small constant, when left to the Vivado HLS tool (left part), and when first replaced by an HLS description of the architecture of Figure 4 by a source-to-source transformation (right part of the table). For constants smaller than 9, all the metrics (logic resources, DSP, latency and frequency) are improved by this

Table 1: Synthesis results for the division y/x, where y is a 32-bit integer variable and x is either a 32-bit integer variable, or an integer constant between 1 and 9. Synthesis is performed using Vivado HLS 2019.1 for Kintex 7, and Intel HLS 19.2 for Arria 10, targeting 100MHz.

(a) Vivado HLS

Divisor	LUTs	Regs.	DSPs
x	*217*	*294*	*0*
1	0	0	0
2	78	0	0
3	162	67	4
4	77	0	0
5	162	67	4
6	162	67	4
7	163	68	4
8	75	0	0
9	160	65	4

(b) Intel HLS

ALMs	FFs	RAMs	DSPs	MLABs
668	*707*	*4*	*10*	*9*
26.5	45	0	0	0
31.5	41	0	0	0
126.5	102	0	0	2
30.5	40	0	0	0
132.5	110	0	0	2
123.5	99	0	0	2
136	106	0	0	2
30.5	40	0	0	0
165.5	103	0	0	2

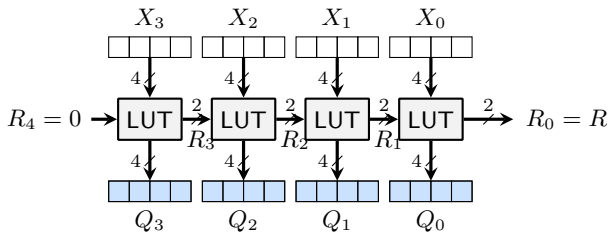


Figure 4: Architecture for division by 3 of a 16-bit number written in hexadecimal, using LUTs with 6 input bits.

Table 2: Synthesis results for the division y/x, where y is a 64-bit integer variable and x is either a 64-bit integer variable, or a constant between 1 and 9. Synthesis is performed using Vivado HLS 2019.1 for Kintex 7, and Intel HLS 19.2 for Arria 10, targeting 100MHz.

(a) C division

Divisor	LUTs	reg.	DSPs	Cycles @ Freq	WCT
x	*8704*	*8588*	*0*	*67 @ 291MHz*	*230.24*
2	162	193	0	2 @ 464MHz	4.31
3	550	728	16	22 @ 343MHz	64.13
4	161	190	0	2 @ 453MHz	4.41
5	548	726	16	22 @ 348MHz	63.21
6	548	726	16	22 @ 348MHz	63.21
7	548	726	16	22 @ 348MHz	63.21
8	159	187	0	2 @ 451MHz	4.43
9	550	726	16	22 @ 351MHz	62.67

(b) [45] division in HLS

LUTs	reg.	Cycles @ Freq	WCT
NA	*NA*	*NA*	*NA*
163	193	2 @ 396MHz	5.05
301	230	16 @ 364MHz	43.95
162	190	2 @ 406MHz	4.92
391	272	22 @ 316MHz	69.62
222	208	16 @ 362MHz	44.19
414	469	22 @ 316MHz	69.62
160	187	2 @ 390MHz	5.12
371	354	32 @ 294MHz	108.84

transformation. The wall clock time (WCT) is also reported. Throughout the rest of this paper, the WCT is given in ns, unless stated otherwise. As the constant grows larger, the latency degrades and the resource consumption increases: for division by 9 we already have a worst latency and frequency than the default multiplication-based implementation, but still with much less resources. For smaller constants, the transformed operators reduce the resource usage by at least a factor 2 while still requiring a smaller or equal WCT.

2.4 Floating-point multiplications and division by small constants

As illustrated by Table 3, even fewer optimizations are implemented for floating-point multiplications and divisions by a constant.

- Both Vivado HLS and Intel HLS are able to remove the constant multiplication and division by 1.0 (unsurprisingly, since it is a valid simplification in software compilers).

Table 3: Synthesis results for the multiplication $y*x$ and the division y/x, where y is a single precision floating-point variable and x is either a single precision floating-point variable, or an integer constant between 1 and 5. Synthesis is performed using Vivado HLS 2019.1 for Kintex 7, and Intel HLS 19.2 for Arria 10, targeting 100MHz.

(a) Vivado HLS

(b) Intel HLS

	Value	LUTs	Regs.	DSPs	SRLs	ALMs	FFs	RAMs	DSPs	MLABs
Mult	*x*	*74*	*99*	*3*	*0*	*76.5*	*104*	*0*	*1*	*2*
	1.0	0	0	0	0	26.5	45	0	0	0
	2.0	53	67	3	0	85.5	63	0	0	2
	3.0	53	67	3	0	111	62	0	0	2
	4.0	59	67	3	0	86	60	0	0	2
	5.0	59	67	3	0	124	61	0	0	2
Div	*x*	*754*	*391*	*0*	*24*	*343.5*	*706*	*3*	*4*	*7*
	1.0	0	0	0	0	26.5	45	0	0	0
	2.0	58	67	3	0	85	60	0	0	2
	3.0	686	244	0	23	341	547	3	4	9
	4.0	58	67	3	0	86.5	62	0	0	2
	5.0	688	244	0	24	331	540	3	4	9

- Intel HLS seems to optimize constant multiplications (it never requires a DSP). Vivado HLS, on the other hand, doesn't even optimize floating-point multiplications by 2.0 or a power of 2. This class of operations should resume to an addition on the exponents, and specific overflow/underflow logic.
- Both tools use a specific optimization when dividing by a power of two. This can easily be explained by looking at the assembly code generated by GCC or Clang/LLVM in such cases: both compilers will transform a division by 4.0 into a multiplication by 0.25, which is bit-for-bit equivalent, and much faster on most processors.
- Both tools use a standard divider for constants that are not a power of 2, with minor resource reductions thanks to the logic optimizer.

Again we may question the relevance of these choices on FPGAs. It is indeed possible to design floating-point versions of both constant multiplications [6] and constant divisions [45] that are bit-for-bit compatible with IEEE correctly rounded ones. For instance, in the case of division, the remainder R that is output by Figure 4 can be used to determine the proper rounding of the significand quotient (for the full details, see [45]).

As we expect constant multiplications to be properly supported soon (it seems to be already the case on Intel HLS), we focus our evaluation on constant division. Table 4 provides synthesis results of Vivado HLS C++ generated operators for floating-point divisions by small constants. The standard floating-point division is also given for comparison purposes, since Table 3 shows that it is the default architecture. All these operators can be more/less deeply pipelined to achieve higher/lower frequencies

Table 4: Synthesis results of the division of y/x, where y is a floating-point numbers and x is either a floating-point variable, or an integer constant between 1 and 5. In the constant case the architecture from [45] is used. Synthesis is performed using Vivado HLS 2019.1 for Kintex 7, and Intel HLS 19.2 for Arria 10, targeting 100MHz.

	(a) float				(b) double			
Divisor	LUTs	reg.	Cycles @ Freq.	WCT(ns)	LUTs	reg.	Cycles @ Freq.	WCT(ns)
x	*808*	*1444*	*29 @ 451MHz*	64.30	*3266*	*3177*	*30 @ 183MHz*	163.93
2.0	63	61	2 @ 619MHz	3.23	126	227	3 @ 574MHz	5.22
3.0	266	177	10 @ 359MHz	27.85	588	459	17 @ 187MHz	90.90
4.0	89	62	2 @ 608MHz	3.28	176	281	3 @ 452MHz	6.63
5.0	292	199	12 @ 356MHz	33.70	687	488	22 @ 204MHz	107.84
6.0	276	178	10 @ 361MHz	27.70	596	438	17 @ 189MHz	89.94
7.0	301	193	17 @ 360MHz	47.22	685	540	22 @ 185MHz	118.91
8.0	85	61	2 @ 579MHz	3.45	176	228	3 @ 577MHz	5.19
9.0	329	258	17 @ 242MHz	70.24	698	590	32 @ 209MHz	153.11
10.0	301	198	13 @ 325MHz	40.00	696	472	22 @ 190MHz	115.78
11.0	328	269	17 @ 229MHz	74.23	697	572	32 @ 189MHz	169.31

at the expense of latency and registers: we attempt to achieve a frequency comparable to that of the standard divider.

Each optimized constant divider uses fewer resources (up to 5.55 times) and has a lower latency (up to 2.3 times) for a comparable frequency. When dividing by a power of two, the cost of the custom divider is virtually nothing (again it resumes to an operation on the exponents).

2.5 Evaluation in context

We implemented a C-to-C source-to-source transformation that detects floating-point multiplications and divisions by constants in the source code, and replaces it by a custom operator that is bit-for-bit equivalent. This transformation was implemented as a plug-in within the open source source-to-source GeCoS compiler framework [19], as per Figure 1.

This work was then evaluated on the Polybench benchmark suite [37]. It contains several C programs that fit the polyhedral model. The focus here is on the stencil codes of this benchmark suite. Most of them contains a division by a small constant. Indeed, out of the 6 stencil codes, 5 were well suited for our transformations. The *Jacobi-1d* benchmark contains two divisions by 3; *Jacobi-2d* contains two divisions by 5; *Seidel-2d* contains a division by 9; *Fdtd-2d* contains two divisions by 2 and a multiplication by 0.7; finally, *Heat-3d* contains six divisions by 8 and six multiplications by 2.

Table 5 compares the synthesis results obtained

- using the original C code, targeting the maximum frequency achievable, and
- using the code after transformation by our GeCoS plug-in.

Table 5: Benchmark synthesis results using Vivado HLS 2019.1 for Kintex 7, before and after constant multplications and divisions transformations.

Benchmark	Type	LUTs	regs.	DSPs	Cycles @ Freq.	WCT(ms)
Fdtd-2d	Original	4780	6162	17	153G @ 349MHz	438.39
	Transformed	3326	4386	17	11G @ 349MHz	31.51
Heat-3d	Original	3842	6081	31	193G @ 349MHz	553.00
	Transformed	5737	6629	17	141G @ 348MHz	405.17
Jacobi-1d	Original	4274	4964	3	185M @ 348MHz	0.53
	Transformed	2167	2792	3	129M @ 358MHz	0.36
Seidel-2d	Original	4419	5126	6	373G @ 351MHz	1062.67
	Transformed	1878	2341	6	321G @ 304MHz	1053.52
Jacobi-2d	Original	4588	5462	9	213G @ 347MHz	613.83
	Transformed	2548	3294	9	176G @ 335MHz	525.37

Each benchmark benefits from the transformations. Wall-clock time is always improved, with latency improved up to 13.9 times for similar frequencies. The benefits of the transformations in terms of timing differ from one benchmark to another, the best improvements being achieved when the transformed operator is in the critical path of an inner loop.

Concerning resource usage, only the *Heat-3d* benchmark has more LUTs and registers in the transformed version. However, it has fewer DSPs. This is explained by the transformation of 6 dividers and 2 multipliers from DSP-based architectures to LUT-based ones. In all other cases, LUT and register usage is reduced for the same amount of DSPs.

3 Optimization examples that change the program semantic

From a compiler point of view, the previous transformations were straightforward and semantic preserving. Conversely, the case study in this section replaces the standard semantics of a floating-point code fragment with a new one, based on a user-specified accuracy with respect to the exact computation on the reals. This enables deeper program transformations, such as a change of the internal number representation that enables both improved accuracy and tightened loop-carried dependencies.

Before detailing it, we must digress a little on the subtleties of the management of floating-point arithmetic by compilers.

3.1 HLS faithful to the floats

Most recent compilers, including the HLS ones [24], 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

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

Listing 8: Usual reduction.

```
#define N 100000
float acc=0, t1=0, t2=0, t3=0, t4=0;
for(int i=0; i<N; i+=4){
  t1+=in[i]; t2+=in[i+1]; t3+=in[i+2]; t4+=in[i+3];
}
acc=t1+t2+t3+t4;
```

Listing 9: Unrolled reduction to express parallelism.

addition is not associative, C11 mandates that code written `a+b+c+d` is executed as `((a+b)+c)+d`, although `(a+b)+(c+d)` would have a shorter latency.

This also prevents exploiting parallelism in floating-point reductions. A reduction is a computation which reduces a set of input values into a reduction location using an associative operation. Examples of reductions include summations, products, wide logical AND, etc.

Reductions are typically expressed as loops. If floating-point addition was associative, Listing 8 would be an example of reduction, where `acc` is the reduction location. The first column of Table 6 gives the synthesis results of Listing 8 using Vivado HLS for Kintex7. The latency of a floating-point addition is 7 cycles in this case. However, using a feedback loop that injects the current denormalized sum as a new input to the adder [15, 18], the Vivado HLS IP is able to achieve an initiation interval of 4 cycles. Still, the adder is only active one cycle out of 4 due to the loop-carried dependency.

Listing 9 shows a different version of Listing 8 that we manually unrolled in such a way to express more parallelism. However, this transformation assumes that floating-point addition is associative, which it is not. Indeed, Vivado HLS is not allowed to transform Listing 8 into Listing 9. According to C or C++ semantics, Listing 9 and 8 are not equivalent. Remark that a parallel execution with the sequential semantics is possible, but very expensive [28].

Vivado HLS is able to exploit the parallelism in Listing 9 (second column of Table 6). It reports using two floating-point adder. The one inside the loop is now active at each cycle on a different subsum, hence a factor 4 gain on latency. The second floating-point adder (whose shifts are implemented using DSP blocks) is used for the final additions. Note that Listing 9 represents the easy case when N is a multiple of the initialization interval of the addition: it would become more complex otherwise.

As a final remark, it is important to point out that the accuracy of these solutions is far from perfect: the 100K rounding errors in the reduction sum up. For test data, we use as in Muller et al. [34] the input values `in[i]=(float)cos(i)`, where i is the input array's index. Therefore the accumulation computes $\sum_i$ `in`[i]. The golden value to compare with is obtained using the MPFR library [21]. We measure that only 20 bits of the result are correct (out of the 24 bits of a `float` significand). When computing the sum of the sines, the accuracy is reduced down to 17 bits. A solution to this is to perform the accumulation in double precision (changing the keyword `float` into `double` in the second lines of Listings 8 and 9). This is obviously more expensive, as the two middle columns of Table 6 show.

Table 6: Synthesis results of different accumulators using Vivado HLS for Kintex 7 at 100MHz

	`float` acc		`double` acc		68-bit fixed-point acc	
	Listing 8	Listing 9	Listing 8	Listing 9	Listing 10	FloPoCo VHDL
LUTs	387	835	784	1908	462	375*
DSPs	0	2	0	3	0	0
Latency	400K	100K	400K	100K	100K	100K
Accuracy	20 bits	20 bits	24 bits	24 bits	24 bits	24 bits

*: does not include the control part

3.2 Towards HLS faithful to the reals

Many floating-point programmers routinely assume associativity of floating-point addition and multiplication, tweaking their code in the same way as we transformed Listing 8 into Listing 9, sometimes assisted by source-to-source compilers or "unsafe" compiler optimizations.

The point of view chosen in this work is to go one step further, and allow the programmer to express that some of the floating-point C/C++ program is intended to describe a computation on *real numbers*. In other words, under programmer control, some of the floats in the C/C++ will be interpreted as real numbers.

This recovers the associativity of the addition, so Listing 8 is again a reduction and can validly be transformed into Listing 9. Besides, this new point of view brings in the picture a new degree of freedom: accuracy. In an hardware context, designers want to tailor the precision (hence the cost) to the accuracy requirements of the application – a classical concern in HLS [22, 7]. Now that the C/C++ describes a computation on the reals, the user should also be able to specify the accuracy of the computation with respect to this exact (real) result. It should then be the task of the compiler to determine the best way to ensure the prescribed accuracy. This is what we explore in the sequel.

3.3 The arithmetic side: application-specific accumulator support

Several approaches for performing a floating-point summation using non-standard hardware have been proposed [31, 27]. In this work, we chose an approach better suited to the HLS context: a generalization of an idea developed by Kulisch. He advocated to augment processors with a very large fixed-point accumulator [29] whose 4288 bits would cover the entire range of double precision floating-point, and then some more: Such an accumulator would remove rounding errors from all the possible floating-point additions and sums of products. The added bonus of an exact addition is that it becomes associative, since the loss of associativity in floating-point is due to rounding.

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 [13]. This operator can be found in the FloPoCo [12] generator and in Intel DSP Builder Advanced. Its core idea, illustrated on Figure 5, 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.

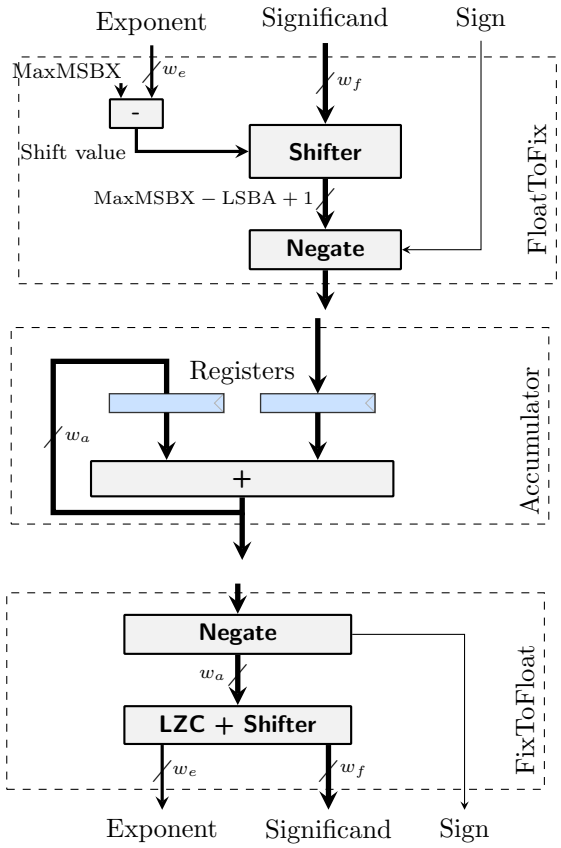


Figure 5: Conversion from float to fixed-point (top), fixed-point accumulation (middle), and conversion from fixed-point to float (bottom).

The accumulator used here slightly improves the one offered by FloPoCo [13]:

- It supports subnormal numbers [34].
- In FloPoCo, FloatToFix and Accumulator form a single component, which restricts its application to simple accumulations similar to Listing 8. The decomposition in two components of Figure 5 enable a generalization to arbitrary summations within a loop, as Section 3.4 shows.

3.3.1 The parameters of a large accumulator

The main feature of this approach is that the internal fixed-point format is configurable in order to control accuracy. For this purpose, this format (represented in Figure 6) has two parameters:

- MSBA is the weight of the most significant bit of the accumulator. For example, if MSBA = 17, the accumulator can accommodate values up to a magnitude of $2^{17} \approx 10^5$.

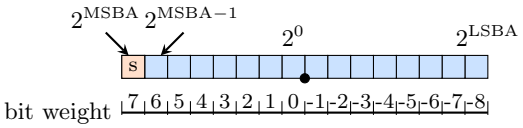


Figure 6: Fixed-point accumulator format, here $(\text{MSBA}, \text{LSBA}) = (7, -8)$.

- LSBA is the weight of the least significant bit of the accumulator. For example, if LSBA $= -50$, the accumulator can hold data accurate to $2^{-50} \approx 10^{-15}$.

The accumulator width w_a is then computed as MSBA $-$ LSBA $+ 1$, for instance 68 bits in the previous example. 68 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 [13] 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, with an initiation interval of about 4 cycles.

3.3.2 Implementation within a HLS tool

This accumulator has been implemented in C/C++, using arbitrary-precision fixed point types (`ap_int`) and the *Modern Arithmetic Tools for HLS* library [44].

For modularity purposes, `FloatToFix` and `FixToFloat` are wrapped into C/C++ functions (respectively 33 and 28 lines of code). Their calls are inlined to enable HLS optimizations. The implementation of the `FloatToFix` function has one more parameter, called MaxMSBX: it defines the largest possible exponent of the floating-point input. Its default value is equal to MSBA, but it may be smaller, when the application context dictates an upper bound on the magnitude of the input to `FloatToFix`. In this case, the size of the shifter (Figure 5) can be reduced.

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

3.3.3 Validation

To evaluate and refine this implementation, we used Listing 10, which we compared to Listings 8 and 9.

The parameters chosen for the accumulator in this experiment 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.

```
#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);
```

Listing 10: Sum of `floats` using the large fixed-point accumulator.

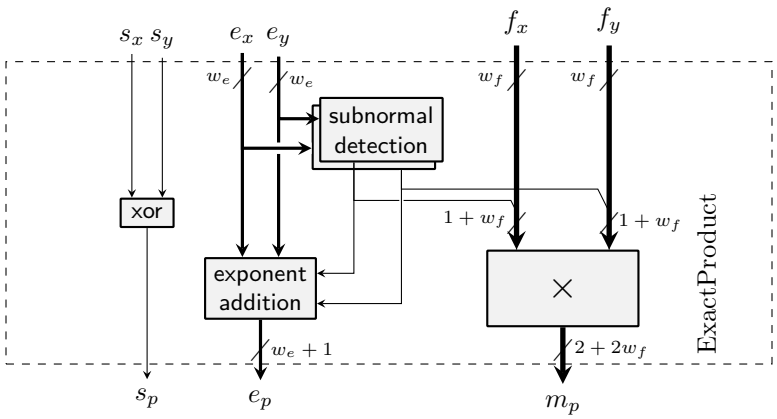


Figure 7: Exact floating-point multiplier.

The results are reported in the two rightmost columns of Table 6. The Accuracy line of the table reports the number of correct bits of each implementation, after the result has been converted to a `float`. The proposed fixed-point accumulator achieves minimal latency and maximum accuracy, for a LUT usage comparable to the usual `float` accumulation of Listing 8.

Removing the control part from Listing 10 reduces resource usage to 395 LUTs, very comparable to the 375 LUTs of the corresponding FloPoCo-generated VHDL (which doesn't include this control, nor subnormal support). This shows that there is no overhead due to the use of HLS.

Using this implementation method, we also created an exact floating-point multiplier with the final rounding removed [13]. This component is depicted in Figure 7. The corresponding function is called `ExactProduct` and represents 44 lines of code. The result mantissa is twice as large as the input mantissas (48 bits in single precision). To add it to the large accumulator, the Float-to-Fix block has to be adapted: in the sequel, it is called `ExactProductFloatToFix` (21 lines of code).

3.4 The compiler side: source-to-source transformation

The previous section, as well as previous work by various groups [40, 20] has shown that Vivado HLS can be used to synthesize very efficient specialized floating-point operators which rival in quality with

those generated by FloPoCo or vendor tools. Our goal is now to study how such optimizations can be automated. More precisely, we aim at automatically optimizing Listing 8 into Listing 10, and generalizing this transformation to many more situations.

For convenience, this optimization was also developed as a source-to-source transformation implemented within GeCoS and is publicly available (`https://gitlab.inria.fr/gecos/gecos-arith`).

This part 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 patterns are exposed to the compiler/runtime either though user directives (e.g `#pragma reduce` in openMP), or automatically inferred using static analysis techniques [38, 16].

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 a user-provided compiler directive (pragma) and some simple program analysis.

More specifically, the proposed code transformation are triggered by a pragma that defines a target accumulation variable, along with application-level information such as the dynamic range of the accumulated data or the overall target accuracy.

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 [39].

The pragma approach has another advantage: we advocate relaxing the standard C semantics, which is in principle dangerous. Therefore the user should explicitely ask for it. A pragma may allow such relaxation in a local, controlled way, without sacrificing standard compliance for the rest of the program.

3.4.1 Proposed compiler directive

In imperative languages such as `C`, reductions are implemented using `for` or `while` loop constructs. The proposed compiler directive must therefore appear right outside such a construct. Listing 11 illustrates its usage on the code of Listing 8.

The `pragma` must contain the following information:

- The keyword `FPacc`, which triggers the transformation.
- The name of the variable in which the accumulation is performed, preceded with the keyword `VAR`. In the example, the accumulation variable is `acc`.

In addition, it may provide the following information:

- The maximum value that can be reached by the accumulator through the use of the `MaxAcc` keyword. This value is used to determine MSBA.
- The desired accuracy of the accumulator using the `epsilon` keyword. This value is used to determine LSBA.
- The maximum value that the inputs can take, using the `MaxInput` keyword. This value is used to determine MaxMSBX. If this information is not provided, then MaxMSBX is set to MSBA.

Note that the user can quietly overestimate the maximum value of the accumulator without major impact on area. For instance, overestimating `MaxAcc` by a factor 10 only adds 3 bits to the accumulator width.

```
#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;
}
```

Listing 11: Illustration of the use of a `pragma` for the usual accumulation.

In cases when the user is unable to provide such information, a full-range Kulisch accumulator is used as a fall-back strategy – a recent study has evaluated different implementations of the full Kulisch accumulator in an FPGA context [42].

3.4.2 Proposed code transformation

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

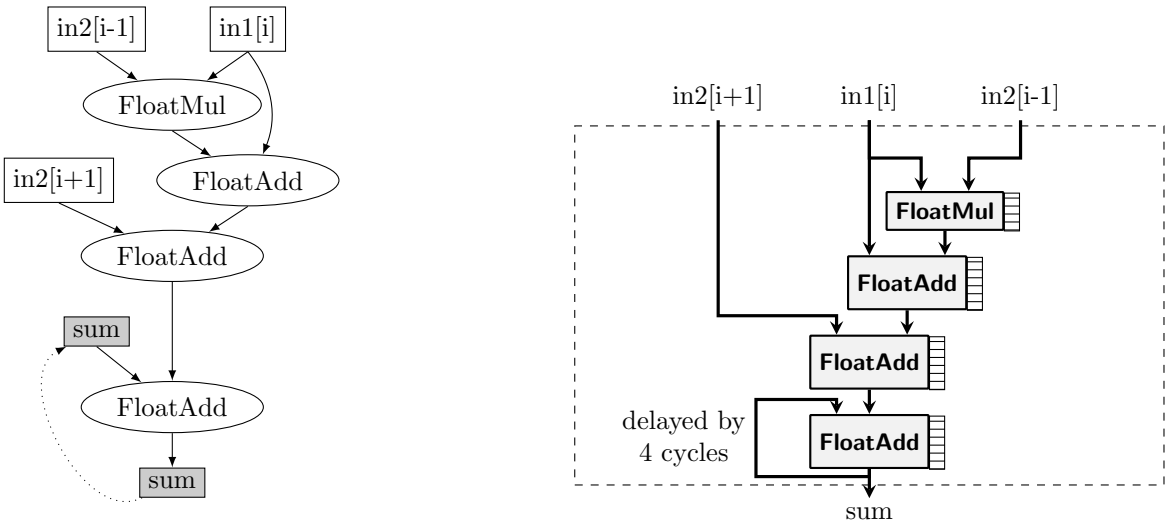
```
#define N 100000
float computeSum(float in1[N], float in2[N]){
  float sum = 0;
  #pragma FPacc VAR=sum MaxAcc=300000.0 epsilon=1e-15 MaxInput=3.0
  for (int i=1; i<N-1; i++){
    sum+=in1[i]*in2[i-1];
    sum+=in1[i];
    sum+=in2[i+1];
  }
  return sum;
}
```

Listing 12: Simple reduction with multiple accumulation statements.

To illustrate the transformation, consider the non-trivial code of Listing 12. This program performs a reduction into the variable `sum`, involving both sums and sums of product operations. Figure 8a shows the operation data-flow graph for the loop body of this program. 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 an inner latency of 4 cycles the HLS tool will schedule a new iteration of the loop at best every 4 cycles.

As illustrated in Figure 9a, the proposed transformation hoists the floating-point normalization step out of the loop, and performs the accumulation using fixed point arithmetic. Since integer add operations can be implemented with a 1-cycle delay at our target frequency, the HLS tool may now be able to initiate a new iteration every cycle, improving the overall latency by a factor of 4.



(a) Loop body dataflow graph

(b) Architecture

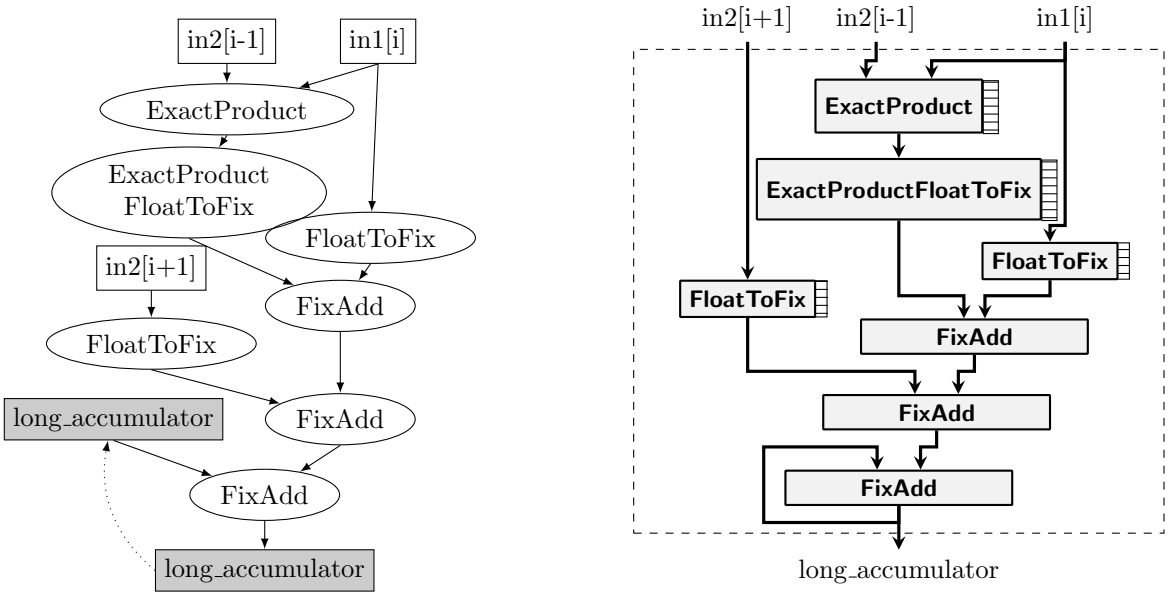
Figure 8: DFG of the loop body from Figure 12 (left) and its corresponding architecture (right). *FloatMul* and *FloatAdd* correspond to floating-point multipliers and adders respectively.

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 data-flow graph, starting from a `FloatAdd` 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 `FloatAdd` node is transformed to an accurate fixed-point adder. The analysis is then recursively launched on that node.
- A `FloatMul` 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 transforms the DFG from Figure 8a into the new DFG shown on Figure 9a. 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.



(a) Loop body data-flow graph

(b) Architecture

Figure 9: DFG of the loop body from Figure 12 (left) and its corresponding architecture (right) after transformations.

From there, it is then possible to regenerate the corresponding `C` code. As an illustration of the whole process, Figures 8b and 9b describe the architectures corresponding to the code before and and after the transformation.

3.4.3 Evaluation on the code of Listing 12

The proposed transformations work on non-trivial examples such as the one represented in Listing 12. Table 7 shows how resource consumption depends on `epsilon`, all the other parameters being those given in the `pragma` of Listing 12.

Compared to the classical IEEE-754 implementation, the transformed code uses similar or more LUTs depending on the precision of the accumulator. The original code requires 3 DSPs for the floating-point multiplier (the adder is implemented in LUTs). The transformed code only requires 2 DSPs as the floating-point multiplier IP is removed for the custom exact multiplier that removes the normalisation part (which required an extra DSP). In all cases, on this example, the transformed code has its latency reduced by a factor equal to 9.

3.5 Evaluation on the EEMBC FPMark benchmarks

In order to evaluate the relevance of the proposed transformations on real-life programs, we used the EEMBC FPMark benchmark suite [17]. The only transformations considered in this evaluation are the ones presented in Section 3.4. This suite consists of 10 programs. A first result is that half of these

Table 7: Comparison between the usual code from Listing 12 and its transformed equivalent. Results obtained with Vivado HLS 2019.1 targeting Kintex7. All these versions run at 100MHz.

	Usual	Transformed LSBA $= -14$	Transformed LSBA $= -20$	Transformed LSBA $= -50$
LUTs	534	501	587	1089
DSPs	3	2	2	2
Latency	900K	100K	100K	100K

programs contain floating-point accumulations:

- Enhanced Livermore Loops (1/16 kernels contains a sum of product).
- LU Decomposition (multiple accumulations).
- Neural Net (multiple sum-of-products).
- 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) as they do not benefit from the transformations (code is unchanged).

Most benchmarks come in single-precision and double-precision versions. We focus here on the single-precision. Double-precision benchmarks lead to the same conclusions.

3.5.1 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 [21]. 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.

Let us review in detail how each benchmark is improved by the proposed transformation. All the synthesis results (before and after transformation) are given in Table 8.

Table 8: Synthesis results of benchmarks before and after sum-of-product transformations. Results obtained with Vivado HLS 2019.1 targeting Kintex7. All these versions run at 100MHz.

Benchmark	Type	LUTs	DSPs	BRAM	Latency	Accuracy
Livermore	Original	398	5	0	54M	11 bits
	Transformed	608	4	0	6M	13 bits
LU-8	Original	435	3	0	38	8-23 bits
	Transformed	274	2	0	14	23 bits
LU-45	Original	472	4	0	186	8-23 bits
	Transformed	300	3	0	51	23 bits
Black Scholes	Original	14388	147	0	4M	19 bits
	Transformed II=4	13623	138	0	4M	23 bits
	Transformed II=1	22646	308	0	1M	23 bits
Fourier Coefficients	Original	22707	319	14	34K*	6 bits
	Transformed	13990	145	7	25K*	11 bits

*: obtained through cosimulation

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 the sum of 20000 products 300 times resulting in 6M accumulated terms. 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
```

As previously, latency is vastly improved while achieving improved accuracy. The area is comparable as the operator implemented in the transformation uses a bit more LUTs and fewer DSPs (as discussed in 3.4.3).

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 45 values, and such accumulations are performed more than 5K times.

Both 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 designs due to their non-constant loop trip counts. As a consequence, instead of presenting results for the complete benchmark, we restrict ourselves to the LU innermost loops. Table 8 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. Still, the transformed code requires fewer resources.

Black Scholes This program contains an accumulation that sums 200 terms. These terms are computed performing polynomial approximations and exponentiations from the input data. The result of the accumulation is divided by a constant (that could be further optimized by using transformations from Section 2). This process is performed 5000 times, hence it results in 1M accumulations.

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 original code is only able to achieve an initiation interval of 4 cycles (limited by the accumulation). Because of this 4 cycles latency between two iterations, Vivado HLS is able to reuse some of the hardware that computes the terms to accumulate in order to reduce the hardware consumption. For example, a *floating-point exponential* IP is reused for the computation of a single term to accumulate.

The transformed code is able to achieve an initiation interval of 1 cycle. Therefore, this exponential hardware can no longer be reused and must be replicated to feed the accumulator. This explains the increased resource usage presented as *Transformed II=1*.

As the transformed code can achieve a 1-cycle initiation interval, it is also able to achieve any higher initiation intervals (2, 3, or 4 cycles), maintaining the same accuracy. The transformation then offers a trade-off between latency and resource usage. The most conservative alternative (keeping the original latency, but for reduced resource usage and improved accuracy), is presented as *Transformed II=4*.

Fourier Coefficients This program computes the coefficients of a Fourier series, using an accumulation performed in single precision. It comes in three different configurations: small, medium and big. Each of them computes the same algorithm but with a different number of iterations. The big version is supposed to compute the most accurate answer. Therefore, the parameters of the custom accumulator are chosen according to this version:

```
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.

Table 8 shows that area is considerably reduced, while accuracy is improved by 5 bits (more than one order of magnitude). However, Vivado HLS cannot statically compute the overall latency due to the use of a floating-point *power* function. Therefore, the latencies reported here are obtained through cosimulation. In order to keep the cosimulation duration reasonable, only the small dataset is used. It computes 100 accumulations of 20 terms, followed by other computations. In that case, the latency is also reduced.

4 Conclusion

This study first demonstrates how today's commercial HLS tools such as Vivado HLS and Intel HLS fail at exploiting full FPGAs potential when dealing with numerical programs, in particular with floating-point numbers.

One reason is the historical heritage of processor-oriented compiler backends. We advocate the integration in HLS compilers of well-known hardware-oriented low-level arithmetic optimizations. Some of them can be applied at the level of the front-end, as showcased by the source-to-source approach used in this study. Others need to be integrated as optimization passes on a compiler intermediate representation. The benefits demonstrated by this study, both in terms of resource usage and latency, makes these optimizations a *must do*.

This study also looks a bit further, towards compiler directives that would instruct the compiler to treat floats as reals in a local, user-controlled way. The role of the compiler is then to achieve the prescribed accuracy (defined with respect to an ideal, exact computation on the reals) with the best performance at the minimal cost. In the context of hardware synthesis, this approach opens the opportunity of using non-standard internal formats and operators, as demonstrated on the the case study of floating-point sums and sums of products. With improvements in performance, cost, and accuracy, this is a promising research direction for higher-level synthesis tools.

References

[1] *IEEE standard for binary floating-point arithmetic*. Institute of Electrical and Electronics Engineers. Note: Standard 754–1985.

[2] Intel Corp., Intel High Level Synthesis Compiler: Best Practices Guide, 2019.

[3] Xilinx Inc., Vivado Design Suite User Guide: High-Level Synthesis, 2019.

[4] Levent Aksoy, Eduardo Costa, Paulo Flores, and Jose Monteiro. Optimization of area in digital FIR filters using gate-level metrics. In *Design Automation Conference*, pages 420–423. IEEE, 2007.

[5] Randy Allen and Ken Kennedy. *Optimizing compilers for modern architectures*. Morgan Kaufmann, 2002.

[6] Nicolas Brisebarre, Florent de Dinechin, and Jean-Michel Muller. Integer and floating-point constant multipliers for FPGAs. In *Application-specific Systems, Architectures and Processors*, pages 239–244. IEEE, 2008.

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

[8] Ken Chapman. Fast integer multipliers fit in FPGAs (EDN 1993 design idea winner). *EDN magazine*, (10), May 1993.

[9] Jason Cong, Muhuan Huang, Peichen Pan, Yuxin Wang, and Peng Zhang. Source-to-source optimization for HLS. In *FPGAs for Software Programmers*, pages 137–163. Springer, 2016.

[10] Florent de Dinechin. Multiplication by rational constants. *Transactions on Circuits and Systems, II*, 52(2):98–102, 2012.

[11] Florent de Dinechin, Silviu-Ioan Filip, Luc Forget, and Martin Kumm. Table-based versus shift-and-add constant multipliers for FPGAs. In *Symposium of Computer Arithmetic*. IEEE, June 2019.

[12] Florent de Dinechin and Bogdan Pasca. *High-Performance Computing using FPGAs*, chapter Reconfigurable Arithmetic for High Performance Computing, pages 631–664. Springer, 2013.

[13] Florent de Dinechin, Bogdan Pasca, Octavian Cret, and Radu Tudoran. An FPGA-specific approach to floating-point accumulation and sum-of-products. In *International Conference on Field-Programmable Technology*, pages 33–40. IEEE, 2008.

[14] Andrew G Dempster and Malcolm D Macleod. Constant integer multiplication using minimum adders. *IEE Proceedings-Circuits, Devices and Systems*, 141(5):407–413, 1994.

[15] Michael Dibrino. Floating point multiplier/accumulator with reduced latency and method thereof, June 2005. US Patent 6,904,446.

[16] Johannes Doerfert, Kevin Streit, Sebastian Hack, and Zino Benaissa. Polly's polyhedral scheduling in the presence of reductions. *International Workshop on Polyhedral Compilation Techniques*, 2015.

[17] EEMBC, the Embedded Microprocessor Benchmark Consortium. FPMark floating-point benchmark suite. `http://www.eembc.org/fpmark`, 2013.

[18] Bruce M. Fleischer, Juergen Haess, Michael Kroener, Martin S. Schmookler, Eric M. Schwarz, and Son Dao-Trong. System and method for a floating point unit with feedback prior to normalization and rounding, June 2010. US Patent 7,730,117.

[19] Antoine Floc'h, Tomofumi Yuki, Ali El-Moussawi, Antoine Morvan, Kevin Martin, Maxime Naullet, Mythri Alle, Ludovic L'Hours, Nicolas Simon, Steven Derrien, et al. GeCoS: A framework for prototyping custom hardware design flows. In *International Working Conference on Source Code Analysis and Manipulation*, pages 100–105. IEEE, 2013.

[20] Luc Forget, Yohann Uguen, Florent de Dinechin, and David Thomas. A type-safe arbitrary precision arithmetic portability layer for HLS tools. In *HEART 2019 - International Symposium on Highly Efficient Accelerators and Reconfigurable Technologies*, June 2019.

[21] Laurent Fousse, Guillaume Hanrot, Vincent Lefèvre, Patrick Pélissier, and Paul Zimmermann. MPFR: A multiple-precision binary floating-point library with correct rounding. *Transactions on Mathematical Software*, 33(2), 2007.

[22] Marcel Gort and Jason H Anderson. Range and bitmask analysis for hardware optimization in high-level synthesis. In *Asia and South Pacific Design Automation Conference*, pages 773–779. IEEE, 2013.

[23] Oscar Gustafsson. Lower bounds for constant multiplication problems. *Transactions On Circuits And Systems II: Express Briefs*, 54(11):974 – 978, November 2007.

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

[25] Qijing Huang, Ruolong Lian, Andrew Canis, Jongsok Choi, Ryan Xi, Nazanin Calagar, Stephen Brown, and Jason Anderson. The effect of compiler optimizations on high-level synthesis-generated hardware. *Transactions on Reconfigurable Technology and Systems*, 8(3):14, 2015.

[26] ISO. *C11 Standard*, 2011. ISO/IEC 9899:2011.

[27] Edin Kadric, Paul Gurniak, and André DeHon. Accurate parallel floating-point accumulation. *Transactions on Computers*, 65(11):3224–3238, 2016.

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

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

[30] Martin Kumm, Oscar Gustafsson, Mario Garrido, and Peter Zipf. Optimal single constant multiplication using ternary adders. *Transactions on Circuits and Systems II*, 65(7):928–932, 2018.

[31] Zhen Luo and Margaret Martonosi. Accelerating pipelined integer and floating-point accumulations in configurable hardware with delayed addition techniques. *Transactions on Computers*, 49(3):208–218, 2000.

[32] Robert Morgan. *Building an optimizing compiler*. Digital Press, 1998.

[33] Steven Muchnick et al. *Advanced compiler design implementation*. Morgan kaufmann, 1997.

[34] Jean-Michel Muller, Nicolas Brunie, Florent de Dinechin, Claude-Pierre Jeannerod, Mioara Joldes, Vincent Lefèvre, Guillaume Melquiond, Nathalie Revol, and Serge Torres. *Handbook of Floating-Point Arithmetic, 2nd edition*. Birkhauser Boston, 2018.

[35] Razvan Nane, Vlad-Mihai Sima, Christian Pilato, Jongsok Choi, Blair Fort, Andrew Canis, Yu Ting Chen, Hsuan Hsiao, Stephen Brown, Fabrizio Ferrandi, et al. A survey and evaluation of FPGA high-level synthesis tools. *Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 35(10):1591–1604, 2015.

[36] Bogdan Pasca. Correctly rounded floating-point division for DSP-enabled FPGAs. In *International Conference on Field Programmable Logic and Applications*, pages 249–254. IEEE, 2012.

[37] Louis-Noël Pouchet. Polybench: The polyhedral benchmark suite. *http://www.cs.ucla.edu/pouchet/software/polybench*, 2012.

[38] Xavier Redon and Paul Feautrier. Detection of scans. *Parallel Algorithms and Applictations*, 15(3-4):229–263, 2000.

[39] Olivier Sentieys, Daniel Menard, David Novo, and Karthick Parashar. Automatic fixed-point conversion: a gateway to high-level power optimization. In *Tutorial at IEEE/ACM Design Automation and Test in Europe*, 2014.

[40] David Thomas. Templatised soft floating-point for high-level synthesis. In *27th International Symposium on Field-Programmable Custom Computing Machines*. IEEE, 2019.

[41] Jason Thong and Nicola Nicolici. An optimal and practical approach to single constant multiplication. *Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 30(9):1373–1386, 2011.

[42] Yohann Uguen and Florent de Dinechin. Design-space exploration for the kulisch accumulator (online). 2017.

[43] Yohann Uguen, Florent de Dinechin, and Steven Derrien. Bridging high-level synthesis and application-specific arithmetic: The case study of floating-point summations. In *Field Programmable Logic and Applications*. IEEE, 2017.

[44] Yohann Uguen, Luc Forget, and Florent de Dinechin. Evaluating the hardware cost of the posit number system. In *International Conference on Field-Programmable Logic and Applications*, 2019.

[45] H Fatih Ugurdag, Florent de Dinechin, Y Serhan Gener, Sezer Gören, and Laurent-Stéphane Didier. Hardware division by small integer constants. *Transactions on Computers*, 66(12):2097–2110, 2017.

[46] Wim Vanderbauwhede and Khaled Benkrid. *High-performance computing using FPGAs*, volume 3. Springer, 2013.

[47] Yevgen Voronenko and Markus Püschel. Multiplierless multiple constant multiplication. *Transactions on Algorithms*, 3(2), 2007.

[48] E. George Walters. Reduced-area constant-coefficient and multiple-constant multipliers for xilinx FPGAs with 6-input LUTs. *Electronics*, 6(4):101, 2017.

[49] Michael J Wirthlin. Constant coefficient multiplication using look-up tables. *Journal of VLSI signal processing systems for signal, image and video technology*, 36(1):7–15, 2004.