Application-specific arithmetic in high-level synthesis tools
Yohann Uguen, Florent de Dinechin, Victor Lezaud, Steven Derrien

To cite this version:
Yohann Uguen, Florent de Dinechin, Victor Lezaud, Steven Derrien. Application-specific arithmetic in high-level synthesis tools. 2019. hal-02178465

HAL Id: hal-02178465
https://hal.archives-ouvertes.fr/hal-02178465
Submitted on 9 Jul 2019

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.
Application-specific arithmetic in high-level synthesis tools

YOHANN UGUEN, FLORENT DE DINECHIN, VICTOR LEZAUD, STEVEN DERRIEN

This work studies hardware-specific optimization opportunities currently unexploited by HLS compilers. Some of these optimizations are specializations of floating-point operations. They respect the usual semantics of the input program. Other optimizations do not, assuming instead that a floating-point computation is actually intended to compute with real numbers. What matters then is to respect application-level accuracy constraints, expressed as pragmas in the source code. This provides the compiler with freedom to use non-standard arithmetic when more efficient. A source-to-source compiler is used to prototype the proposed optimizations and evaluate them on relevant benchmarks.

ACM Reference Format:

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 or financial computing to signal processing and cryptography. 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 design costs are the two faces of the same coin.

To address this, languages such as C/C++ 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 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 [3], Intel HLS[2] or Catapult C 1 among others [29]. 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 good at this task, and can automatically synthesize highly efficient pipelined data-flow architectures.

0Extension of Conference Paper. This work extends [37] 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.


Author’s address: Yohann Uguen, Florent de Dinechin, Victor Lezaud, Steven Derrien.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2019 Association for Computing Machinery.
XXXX-XXXX/2019/7-ART $15.00
https://doi.org/10.1145/nnnnnn.nnnnnnn
HLS tools rely heavily on compiler optimizations [5, 26, 27]. As most of these optimizations were designed for standard CPUs, it is relevant to question if they make sense in an FPGA context. It is also relevant to attempt to identify new optimizations that were not investigated previously because they make sense only in the FPGA context. 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. 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. We can see that the multiplication by 7 has been transformed by the compiler into a shift and add algorithm:

\[ 7 \times x = 8 \times x - x = 2^3 - x \]

where the multiplication by 2\(^3\) is a simple shift left by 3 bits (this multiplication by 8 may also be implemented by the lea instruction in a slightly less obvious way, and this is what happens, both on GCC or Clang/LLVM, when using -O2 optimization).

As a consequence, the architecture produced by a HLS tool based on GCC or Clang/LLVM will implement this algorithm. This optimization makes even more sense in HLS, since the constant shifts reduce to wires and therefore cost nothing. Indeed, the synthesis of Listing 1 in VivadoHLS reports 32 LUTs, the cost of one addition. Experiments with Vivado HLS (based 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, we observe that 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 only if one remarks that 2228241 = 17 \cdot 2^{17} + 17: first compute \( t = 17x = x \cdot 2^{4} + x \) (one addition), then compute 2228241x = t \cdot 2^{17} + 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 \( t = 17x = x \cdot 2^{4} + x \) (and should cost only 28 LUTs, since the lower 4 bits are those of \( x \)); one that computes the 32
lower bits of $t \cdot 2^{17} + t$, 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.

```
Listing 5. C code
int mul2228241 ( int x ){
  int t = (x<<4) + x;
  return (t<<17) + t;
}
```

For this program, 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, what we observe here is that the arithmetic optimization has been completely delegated to the underlying compiler x86 backend, and we have a case here for enabling further optimizations. Indeed, hardware constant multiplication has been the subject of much research [4, 13, 18, 24, 35, 39], some of which is specific to FPGAs [8, 10, 40, 41].

The broader objective of the present work is to list similar opportunities of hardware-specific arithmetic optimizations that are currently unexploited, and demonstrate their effectiveness. We classify these optimizations in two broad classes.

In Section 2, we discuss optimization opportunities that strictly respect the semantic of the original program. 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 it is not yet the case is that the field of HLS is still relatively young.

The second class, discussed in Section 3 is more controversial and forward-looking. It includes optimizations that relax (and we argue, only for the better) the constraint of preserving the program semantics. In this Section, we assume that the programmer who used floating-point data in their programs intended to compute with real numbers, and we consider optimizations that 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 [15] to improve the generated designs.

Finally, we discuss in Section 4 what we believe HLS tools should evolve to.
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 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 [20]:

- \( x/x \) and 1.0 are not equivalent if \( x \) can be zero, infinite, or NaN (in which case the value of \( x/x \) is NaN).
- \( x - y \) and \(- (y - x)\) are not equivalent because \( 1 - 1 \) is +0 but \(- (1 - 1) \) is \(- 0\) (in the default rounding direction).
- \( x - x \) and 0, 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 \times x \) and \(- x\) are not equivalent if \( x \) is +0, because \(- (0) \) yields \(- 0\), but 0 - \(+ 0\) yields +0.

Of course, programmers usually don’t 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 [27].

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 standard compiler. However, it could replace \( x/x \) with something like \((\text{is}_0(x) || \text{is}_\infty(x) || \text{is}_\text{NaN}(x)) \?\text{NaN} : 1.0\). 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 on 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 will be pushed down the computation, as illustrated by Figure 2c.

![Figure 2. Optimization opportunities for floating point x/x * a.](image)

Note that this figure replaces \( 1.0 \times x \) by \( x \): this is a valid floating-point optimization, in the sense that it is valid even if \( x \) is a signed zero, an infinity or a NaN.

, Vol. 1, No. 1, Article . Publication date: July 2019.
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.

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 will 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 [4, 8, 10, 13, 18, 24, 35, 39–41]. These are mostly academic works, but back-end tools already embed some of it, so this optimization could be the first to arrive. 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 $\frac{1}{7}$ has the periodic pattern $100100100100100100100100100\ldots$ (hidden in the hexadecimal pattern $924_{16}$ in Listing 7). This enables a specific optimization of the shift-and-add constant multiplication algorithm [9].

Listing 6. C code

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

Listing 7. Objdump of listing 6 when compiled with Clang -O2

```
... 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
```

Table 1 shows synthesis results on the two FPGA mainstream HLS flows. 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:

- The generic divider (Value=x) is based on Xilinx on a shift-and-add algorithm, while on Intel a polynomial approach is used [30] that consumes multiplier and DSP resources.
Table 1. Synthesis results of 32-bit integer dividers with Vivado HLS for Kintex 7, and Intel HLS for Arria 10.

<table>
<thead>
<tr>
<th>Value</th>
<th>Vivado HLS</th>
<th>Intel HLS</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>LUTs</td>
<td>Regs.</td>
</tr>
<tr>
<td>x</td>
<td>235</td>
<td>295</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>94</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>142</td>
<td>113</td>
</tr>
<tr>
<td>4</td>
<td>94</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>142</td>
<td>113</td>
</tr>
<tr>
<td>6</td>
<td>163</td>
<td>103</td>
</tr>
<tr>
<td>7</td>
<td>142</td>
<td>111</td>
</tr>
<tr>
<td>8</td>
<td>92</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>142</td>
<td>114</td>
</tr>
</tbody>
</table>

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 rewriting $776 = 700 + 70 + 6 = 250 + 20 + 6$ 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 a number between 0 and 29: it can be pre-computed for these 30 cases and stored in a look-up table (LUT).

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

- 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 in [38]. 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). The results were obtained using Vivado HLS 2016.3 targeting a Kintex 7 (part xc7k160tfbg484-1) at 330MHz. For constants smaller than 9, all the metrics (logic resources, DSP, latency and frequency) are improved by this transformation. 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.
Fig. 4. Architecture for division by 3 of a 16-bit number written in hexadecimal, using LUTs with 6 input bits.

Table 2. Synthesis results of 64-bit integer constant divisors using Vivado HLS for Kintex 7.

<table>
<thead>
<tr>
<th>Value</th>
<th>LUTs</th>
<th>reg.</th>
<th>DSPs</th>
<th>Cycles @ Freq</th>
<th>LUTs</th>
<th>reg.</th>
<th>Cycles @ Freq</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>8831</td>
<td>8606</td>
<td>0</td>
<td>68 @ 293MHz</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>2</td>
<td>194</td>
<td>193</td>
<td>0</td>
<td>3@467MHz</td>
<td>0</td>
<td>0</td>
<td>1 @ 1488MHz</td>
</tr>
<tr>
<td>3</td>
<td>931</td>
<td>966</td>
<td>16</td>
<td>23@373MHz</td>
<td>62</td>
<td>127</td>
<td>17 @ 403MHz</td>
</tr>
<tr>
<td>4</td>
<td>191</td>
<td>190</td>
<td>0</td>
<td>3@444MHz</td>
<td>0</td>
<td>0</td>
<td>1 @ 1488MHz</td>
</tr>
<tr>
<td>5</td>
<td>925</td>
<td>962</td>
<td>16</td>
<td>23@364MHz</td>
<td>107</td>
<td>152</td>
<td>23 @ 377MHz</td>
</tr>
<tr>
<td>6</td>
<td>927</td>
<td>964</td>
<td>16</td>
<td>23@363MHz</td>
<td>72</td>
<td>156</td>
<td>17 @ 380MHz</td>
</tr>
<tr>
<td>7</td>
<td>923</td>
<td>956</td>
<td>16</td>
<td>23@355MHz</td>
<td>107</td>
<td>152</td>
<td>23 @ 377MHz</td>
</tr>
<tr>
<td>8</td>
<td>189</td>
<td>187</td>
<td>0</td>
<td>3@449MHz</td>
<td>0</td>
<td>0</td>
<td>1 @ 1488MHz</td>
</tr>
<tr>
<td>9</td>
<td>929</td>
<td>961</td>
<td>16</td>
<td>23@356MHz</td>
<td>100</td>
<td>193</td>
<td>33 @ 351MHz</td>
</tr>
</tbody>
</table>

Table 3. Synthesis results of single-precision floating-point multipliers/dividers using Vivado HLS (Kintex 7) and Intel HLS (Arria 10) targeting 100 MHz.

<table>
<thead>
<tr>
<th>Value</th>
<th>LUTs</th>
<th>Regs.</th>
<th>DSPs</th>
<th>SRLs</th>
<th>ALMs</th>
<th>FFs</th>
<th>RAMs</th>
<th>DSPs</th>
<th>MLABs</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>86</td>
<td>99</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1.0</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>70</td>
<td>67</td>
<td>3</td>
<td>0</td>
<td>2.0</td>
<td>69.5</td>
<td>21</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>67</td>
<td>70</td>
<td>3</td>
<td>0</td>
<td>3.0</td>
<td>102</td>
<td>20</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>71</td>
<td>67</td>
<td>3</td>
<td>0</td>
<td>4.0</td>
<td>69</td>
<td>20</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>71</td>
<td>67</td>
<td>3</td>
<td>0</td>
<td>5.0</td>
<td>108</td>
<td>21</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

| x     | 780  | 392  | 0    | 25   | 1.0  | 2   | 3    | 0    | 0     |
|       | 311.5| 634  | 3    | 500  | 2.0  | 72  | 23   | 0    | 0     |
| 3     | 311.5| 500  | 3    | 4    | 4.0  | 71.5| 22   | 0    | 0     |
| 4     | 322.5| 504  | 3    | 4    | 5.0  | 322.5| 504  | 3    | 4     |
| 5     | 311.5| 634  | 3    | 4    |       |       |       |       |       |
2.4 Floating-point multiplications and division by small constants

As illustrated by Table 3, there are even fewer optimizations for floating-point multiplication and division 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).
- 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 compiler 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 [38] 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 [38]).

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 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 12 times) and has a lower latency (up to 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 [15], as per Figure 1.

This work was then evaluated on the Polybench benchmark suite [31]. 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 \textit{Jacobi-1d} benchmark contains two divisions by 3; \textit{Jacobi-2d} contains two divisions by 5; \textit{Seidel-2d} contains a division by 9; \textit{Fdtd-2d} contains two divisions by 2 and a multiplication by 0.7; finally, \textit{Heat-3d} contains six divisions by 8 and six multiplications by 2. Note that the division by 0.7 can be transformed to a multiplication by 7 and a division by 10.

Table 5 compares the synthesis results obtained from Vivado HLS for Xilinx Kintex 7 FPGAs (xc7k160tfbg484-1)

- using the original C code, targeting the maximum frequency achievable, and
- using the code after transformation by our GeCoS plug-in.
Table 4. Synthesis results of floating-point constant divisors for single and double precision that implements [38] using Vivado HLS for Kintex 7.

<table>
<thead>
<tr>
<th>Value</th>
<th>LUTs</th>
<th>reg.</th>
<th>Cycles @ Freq</th>
<th>Value</th>
<th>LUTs</th>
<th>reg.</th>
<th>Cycles @ Freq</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>784</td>
<td>1446</td>
<td>30 @ 330MHz</td>
<td>x</td>
<td>3244</td>
<td>3178</td>
<td>31 @ 188MHz</td>
</tr>
<tr>
<td>2.0</td>
<td>34</td>
<td>0</td>
<td>1 @ 458MHz</td>
<td>2.0</td>
<td>77</td>
<td>68</td>
<td>2 @ 539MHz</td>
</tr>
<tr>
<td>3.0</td>
<td>152</td>
<td>130</td>
<td>10 @ 314MHz</td>
<td>3.0</td>
<td>608</td>
<td>310</td>
<td>17 @ 182MHz</td>
</tr>
<tr>
<td>4.0</td>
<td>35</td>
<td>0</td>
<td>1 @ 467MHz</td>
<td>4.0</td>
<td>179</td>
<td>70</td>
<td>2 @ 422MHz</td>
</tr>
<tr>
<td>5.0</td>
<td>149</td>
<td>151</td>
<td>12 @ 307MHz</td>
<td>5.0</td>
<td>606</td>
<td>319</td>
<td>22 @ 182MHz</td>
</tr>
<tr>
<td>6.0</td>
<td>126</td>
<td>126</td>
<td>10 @ 325MHz</td>
<td>6.0</td>
<td>604</td>
<td>311</td>
<td>17 @ 177MHz</td>
</tr>
<tr>
<td>7.0</td>
<td>151</td>
<td>151</td>
<td>12 @ 270MHz</td>
<td>7.0</td>
<td>624</td>
<td>319</td>
<td>22 @ 177MHz</td>
</tr>
<tr>
<td>8.0</td>
<td>55</td>
<td>0</td>
<td>1 @ 397MHz</td>
<td>8.0</td>
<td>208</td>
<td>68</td>
<td>2 @ 453MHz</td>
</tr>
<tr>
<td>9.0</td>
<td>180</td>
<td>161</td>
<td>17 @ 278MHz</td>
<td>9.0</td>
<td>628</td>
<td>333</td>
<td>32 @ 194MHz</td>
</tr>
<tr>
<td>10.0</td>
<td>261</td>
<td>162</td>
<td>13 @ 206MHz</td>
<td>10.0</td>
<td>609</td>
<td>320</td>
<td>22 @ 180MHz</td>
</tr>
<tr>
<td>11.0</td>
<td>189</td>
<td>161</td>
<td>17 @ 276MHz</td>
<td>11.0</td>
<td>636</td>
<td>333</td>
<td>32 @ 189MHz</td>
</tr>
</tbody>
</table>

Table 5. Synthesis results of benchmarks before and after transformations.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Type</th>
<th>LUTs</th>
<th>regs.</th>
<th>DSPs</th>
<th>Cycles @ Freq.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fdtd-2d</td>
<td>Orig.</td>
<td>4741</td>
<td>6262</td>
<td>17</td>
<td>153G @ 320MHz</td>
</tr>
<tr>
<td></td>
<td>Trans.</td>
<td>2819</td>
<td>4628</td>
<td>17</td>
<td>11G @ 345MHz</td>
</tr>
<tr>
<td>heat-3d</td>
<td>Orig.</td>
<td>3744</td>
<td>6118</td>
<td>31</td>
<td>193G @ 341MHz</td>
</tr>
<tr>
<td></td>
<td>Trans.</td>
<td>4886</td>
<td>6984</td>
<td>17</td>
<td>147G @ 331MHz</td>
</tr>
<tr>
<td>Jacobi-1d</td>
<td>Orig.</td>
<td>4221</td>
<td>4985</td>
<td>3</td>
<td>185M @ 354MHz</td>
</tr>
<tr>
<td></td>
<td>Trans.</td>
<td>2006</td>
<td>2971</td>
<td>3</td>
<td>137M @ 348MHz</td>
</tr>
<tr>
<td>Seidel-2d</td>
<td>Orig.</td>
<td>4514</td>
<td>5481</td>
<td>9</td>
<td>213G @ 358MHz</td>
</tr>
<tr>
<td></td>
<td>Trans.</td>
<td>2328</td>
<td>3491</td>
<td>9</td>
<td>183G @ 337MHz</td>
</tr>
<tr>
<td>Jacobi-2d</td>
<td>Orig.</td>
<td>4335</td>
<td>5157</td>
<td>6</td>
<td>373G @ 355MHz</td>
</tr>
<tr>
<td></td>
<td>Trans.</td>
<td>1806</td>
<td>2861</td>
<td>6</td>
<td>357G @ 336MHz</td>
</tr>
</tbody>
</table>

Each benchmark benefits from the transformations. Latency is improved up to 12 times for similar frequencies. The Heat-3d benchmark trades a bit more LUTs and registers for a lot less DSPs. In all other cases, LUTs, registers and DSPs usage is reduced.

The benefit of the transformations in terms of cycles differ from one benchmark to another. The best improvements are achieved when the transformed operator is in the critical path of an inner loop.

3 OPTIMIZATION EXAMPLES THAT CHANGE THE PROGRAM SEMANTIC

From a compiler point of view, the previous transformations were straightforward and semantic preserving.

The case study in this section is a more complex program transformation that applies to floating-point reductions. The use of custom formats, driven by user-specified accuracy allows to tighten loop carried dependencies. The result of this complex sequence of optimizations cannot be obtained from an operator generator since it involves knowledge of the program behaviour in which the
Table 6. Synthesis results of different accumulators using Vivado HLS for Kintex 7.

<table>
<thead>
<tr>
<th></th>
<th>Listing 8 (float)</th>
<th>Listing 9 (float)</th>
<th>Listing 8 (double)</th>
<th>Listing 9 (double)</th>
<th>Listing 10 (71 bits)</th>
<th>FloPoCo VHDL (71 bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>LUTs</td>
<td>266</td>
<td>907</td>
<td>801</td>
<td>2193</td>
<td>736</td>
<td>719</td>
</tr>
<tr>
<td>DSPs</td>
<td>2</td>
<td>4</td>
<td>3</td>
<td>6</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Latency</td>
<td>700K</td>
<td>142K</td>
<td>700K</td>
<td>142K</td>
<td>100K</td>
<td>100K</td>
</tr>
<tr>
<td>Accuracy</td>
<td>17 bits</td>
<td>17 bits</td>
<td>24 bits</td>
<td>24 bits</td>
<td>24 bits</td>
<td>24 bits</td>
</tr>
</tbody>
</table>


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


```c
#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;
```

operator is to be instantiated. 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 [19], 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

```c
(a+b)+c+d
```

is executed as

```c
((a+b)+c)+d
```

although

```c
(a+b)+(c+d)
```

have a 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 8 provides the simplest example of reduction, where acc is the reduction location.

The first column of Table 6 shows how Vivado HLS synthesizes Listing 8 on a Kintex7 FPGA. 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 9 shows a different version of Listing 8 that we coded such that Vivado HLS expresses more parallelism. Vivado HLS will not transform Listing 8 into Listing 9, because they are not semantically equivalent\(^2\) (the floating-point additions are reordered as if they were associative). However, the tool is able to exploit the parallelism in Listing 9 (second column of Table 6): The main adder is now active at each cycle on a different sub-sum.

Note that Listing 9 is only here as an example and might need more logic if N was not a multiple of 10.

### 3.2 Towards HLS faithful to the reals

The point of view chosen in this work is to assume that the floating-point C/C++ program is intended to describe a computation on real numbers when the user specifies it. In other words,

\(^2\)A parallel execution with the sequential semantics is also possible, but very expensive [22].
the floats are interpreted as *real numbers* in the initial C/C++, thus recovering the freedom of associativity (among other). Indeed, most programmers will perform the kind of non-bit-exact optimizations illustrated by Listing 9 (sometimes assisted by source-to-source compilers or “unsafe” compiler optimizations). In a hardware context, we may also assume they wish they can tailor the precision (hence the cost) to the accuracy requirements of the application – a classical concern in HLS [7, 17]. 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.

### 3.3 The arithmetic side: application-specific accumulator support

The architecture used for this work is based on a more general idea developed by Kulisch. He advocated to augment processors with a very large fixed-point accumulator [23] 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 [12]. This operator can be found in the FloPoCo [11] 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.

The accumulator used here slightly improves the one offered by FloPoCo [12]:

- It supports subnormal numbers [28].
- 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 will show.

Note that we could have implemented any other non-standard operator performing a reduction such as [21, 25].

#### 3.3.1 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:

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

Such a fixed-point format is illustrated in Figure 6.

The accumulator width $w_a$ is then computed as MSBA − LSBA + 1, for instance 71 bits in the previous example. 71 bits represents a wide range and high accuracy, and still additions on this format will have one-cycle latency for practical frequencies on recent FPGAs. If this is not enough the frequency can be improved thanks to partial carry save [12] 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.
3.3.2 Implementation within a HLS tool. This accumulator has been implemented in C/C++, using arbitrary-precision fixed point types (ap_int). The leading zero count, bit range selection and other operations are implemented using Vivado HLS built-in functions. For modularity purposes, the FloatToFix and FixToFloat are wrapped into C/C++ functions (respectively 28 and 22 lines of code). Their 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 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. 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. [28] the input values $c[i] = \text{float}(\cos(i))$, where $i$ is the input array’s index. Therefore the accumulation computes $\sum_i c[i]$.

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

```c
#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);
```

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 6 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, the proposed approach gets even better latencies for a reasonable LUT usage. It also achieves maximum accuracy for the float format, which caps at 24 bits (the internal representations of the double, unrolled double and proposed approach have a higher accuracy than 24 bits, but their result is then rounded to a float). Finally, our results are very close to FloPoCo ones, both in terms of LUTs usage, DSPs and latency.

Using this implementation method, we also created an exact floating-point multiplier with the final rounding removed as in [12]. This 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). This component is depicted in Figure 7.
The previous section, as well as previous works by various groups [34] 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. Our goal is now to study how such optimizations can be automated. More precisely, we aim at automatically optimize Listing 8 into Listing 10, and generalize 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). 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 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 [14, 32].

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 improve the program analysis, so that the two later parameters could be omitted in some situations. Recent studies evaluated different Kulisch accumulators in a FPGA context [36]. We use the optimal architecture as a fall-back strategy when no specification on the input data is given.

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 [33].

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`.
• 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 `MSBA`.
• The desired accuracy of the accumulator using the `epsilon` keyword. This value is used to determine the weight `LSBA`.
• Optional: The maximum value among all inputs of the accumulator in the `MaxInput` field. This value is used to determine the weight `MaxMSBX`. If this information is not provided, then `MaxMSBX` is set to `MSBA`.

```c
#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 naive accumulation.

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

3.4.2 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 data-flow graph (DFG).

```c
#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 toy but non-trivial program 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
(a) Loop body dataflow graph

(b) Architecture

Fig. 8. DFG of the loop body from Listing 12 (top) and its corresponding architecture (bottom). Keywords float mul and float add correspond to floating-point multipliers and adders respectively.

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 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 7.

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 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 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.

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 after the transformation.
Application-specific arithmetic in high-level synthesis tools

![Loop body data-flow graph](image1)

![Architecture](image2)

Fig. 9. DFG of the loop body from Listing 12 (top) and its corresponding architecture (bottom) after transformations.

Table 7. Comparison between the naive code from Listing 12 and its transformed equivalent. All these versions run at 100MHz.

<table>
<thead>
<tr>
<th></th>
<th>Naive</th>
<th>Transformed LSBA = −14</th>
<th>Transformed LSBA = −20</th>
<th>Transformed LSBA = −50</th>
</tr>
</thead>
<tbody>
<tr>
<td>LUTs</td>
<td>538</td>
<td>693</td>
<td>824</td>
<td>1400</td>
</tr>
<tr>
<td>DSPs</td>
<td>5</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Latency</td>
<td>2000K</td>
<td>100 K</td>
<td>100K</td>
<td>100K</td>
</tr>
</tbody>
</table>

3.4.3 Evaluation of the toy example 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. All these versions were synthesized for 100 MHz.

Compared to the classical IEEE-754 implementation, the transformed code uses more LUTs for less DSPs. This is due to Vivado implementing shifters using DSPs within the floating-point IP, but not in the transformed code. In all cases, on this example, the transformed code has its latency reduced by a factor 20.

3.5 Evaluation

In order to evaluate the relevance of the proposed transformations on real-life programs, we used the EEMBC FPMark 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. 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 [16]. 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.

3.5.2 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 8. As in the previous toy examples, latency and accuracy are vastly improved for comparable area.

Table 8. Synthesis results of benchmarks before and after transformations.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Type</th>
<th>LUTs</th>
<th>DSPs</th>
<th>Latency</th>
<th>Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Livermore</td>
<td>Orig.</td>
<td>384 576</td>
<td>5 2</td>
<td>80K 20K</td>
<td>11 bits 13 bits</td>
</tr>
<tr>
<td>LU-8</td>
<td>Orig.</td>
<td>809 1007</td>
<td>5 2</td>
<td>82 17</td>
<td>8-23 bits 23 bits</td>
</tr>
<tr>
<td>LU-45</td>
<td>Orig.</td>
<td>819 1034</td>
<td>5 2</td>
<td>452 54</td>
<td>8-23 bits 23 bits</td>
</tr>
<tr>
<td>Scholes</td>
<td>Orig.</td>
<td>15640 15923</td>
<td>175 175</td>
<td>N/A 23 bits</td>
<td>19 bits 23 bits</td>
</tr>
<tr>
<td>Fourier</td>
<td>Orig.</td>
<td>34596 34681</td>
<td>64 59</td>
<td>N/A N/A</td>
<td>6 bits 11 bits</td>
</tr>
</tbody>
</table>
LU Decomposition and Neural Net. Both the LU decomposition and the neural net programs contain multiple nested small accumulations. In the LU decomposition program, an inner loop accumulates between 8 and 45 values. Such accumulations are performed more than 7M times. In the neural net program, inner loops accumulate between 8 and 35 values, and such accumulations are performed more than 5K times.

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

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.

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 from Section 2). 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 8.

For comparable area, accuracy is vastly improved but latency could not be obtained statically from Vivado HLS. Indeed, the Black Scholes algorithm uses the mathematical function power. Such a function is not natively supported by Vivado HLS, and was therefore implemented by hand using a data dependent trip count loop. Because of this, the tool cannot statically derive the execution latency of the benchmark making the overall latency data dependent. One could use cosimulation to obtain the latency of a specific set of inputs.

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 8.

Here again, area cost is comparable, while accuracy is improved by 5 bits (which represents one order of magnitude). As for Black Scholes, Vivado HLS cannot compute the overall latency due to the power function. However, since our operators have a shorter latency by design, we expect the circuits to also have a shorter latency.

4 CONCLUSION
This study demonstrates how today’s HLS tools fail at exploiting full FPGAs potential when dealing with floating-point numbers. The historic nature of x86 backends compiler is embedded in these
hardware compilers. CPU specific optimizations are then followed in a custom hardware context. This choice is questionable knowing FPGAs best assets are custom data-path that differs from CPUs’.

Well known low-level arithmetic optimizations can still be applied to a high-level input source, as showcased in this study. The benefit in terms of resource usage and latency makes these optimizations a must do to close the gap between HLS and RTL design. Specially since the behavioural description of a program, as seen per the compilers, allows for further optimizations than what can be applied than RTL design. Indeed, the HLS compiler can extract information from the context in which the operator is used.

We provided a tool that automatically transforms a Vivado HLS compliant C/C++ code to a transformed equivalent. This transformed code got its floating-point accumulations; divisions and multiplications by small constants enhanced using application-specific arithmetic. The goal of this tool is not to be used before using a HLS tool but to show that HLS tools should implement these transformations.

A very little number of operators were studied in this work. These were examples to showcase that HLS tools are capable of highly effective arithmetic optimizations. The greater goal of this work is to gather two communities; the arithmeticians and compiler designers. Therefore, integrating and enhancing a lot more operators within HLS tools.

REFERENCES


Application-specific arithmetic in high-level synthesis tools


