Master research Internship

Internship report

High-level synthesis and arithmetic optimization applied to reductions

Domain: Computer Arithmetic - Hardware Architecture

Author:
Yohann Uguen

Supervisor:
Florent de Dinechin (Socrate)
Steven Derrien (Cairn)

Abstract: The use of high-level synthesis tools (HLS) - use of the `C` language to generate architectures - is a big step forward in terms of productivity, especially for programming FPGAs. However, it introduces the restriction to only use the data-types and operators given by the `C` language. It has been shown many times that application-specific arithmetic provides large benefits. We want to bring such optimization to HLS. In this work, we propose a source-to-source compiler can transform a `C` source code to a functionally equivalent code, but optimized by using non-standard/application specific operators. Our case study is targeting summations that we retrieve in a source code using a `pragma`, and then replace with the specialized operators. Using these transformations, the quality of the circuits produced by HLS tools is improved in speed, resource consumption and accuracy.

Contents

1 Introduction

When one wants to perform a numerical computation on a general purpose processor, a trade-off has to be made between computation speed and resource consumption (memory usage, silicon needed, energy consumption, etc.). For example, to increase computation speed by exposing more parallelism, the resource usage will greatly increase by using more cores. On the other hand, to reduce resource consumption one may have to use a processor with a lower frequency a fewer number of cores.

To increase computation speed while limiting resource usage, one may chose to use a hardware accelerator. One of the most popular options to improve computation speed while being energy-efficient is to use a Graphics Processor Unit (GPU). Other hardware accelerators are emerging due to the recent focus on energy saving, such as co-processors like the the Intel Xeon Phi [19].

Accuracy in the performance/cost trade-off

Numerical computations generally operates on real numbers. These are often represented as floating-point numbers due to the ease of use of such a format. However floating-point units cost a lot of silicon area and energy consumption. Another variable to take into consideration in this trade-off is the accuracy of the computation. Managing the accuracy can result in altering speed or resource consumption. To clarify what we mean by accuracy, we recall that the *accuracy* of a value is the number of correct bits of that value, and *precision* of a value is the number of bits in which it is represented. For example, a number that requires to be accurate to 10^{-3} (which translates to 10 bits to be represented) can be encoded in a simple precision floating point format. In that case, the floating point format offers 24 bits of *precision* but the data it contains only has an *accuracy* of 10 bits.

Here are a few examples of how managing accuracy can act in this trade-off by using an intermediate representation that has increased accuracy:

- One could use double precision instead of simple precision during the computation, at the cost of performance.
- There exists some dedicated libraries to boost accuracy such as MPFR [12] that allows the user to have an arbitrary number of bits for every variable, at the cost of performance.
- One can also chose to alter the accuracy of the computation to increase computation speed [3].
- One could use a fixed-point representation to improve computation speed and reduce resource usage. However, using a fixed-point format requires much more programming work to achieve the desired accuracy. There exists some software to automatically transform a program into its fixed-point equivalent, such as ID.Fix [26].

These tools require some fixed-point format knowledge as the program needs to be annotated with *pragmas* containing the desired formats.

Limitations of general purpose hardware

General purpose processors and accelerators do not offer the possibility to use a custom precision. In some cases, reducing the precision can result in the same accuracy and could be exploited to improve computation speed and reduce resource consumption. Using a floating-point format gives the user the ease of having a single format for all his real values. But this format, defined by the IEEE-754 standard [1] has flaws that we will be discussing in Section 2.

As the floating-point standard is used in every general purpose processors, new formats were proposed to avoid the current limitations. Kulisch et al. [22] described a large floating-point format that uses 4288 bits to cover to entire range of double precision of the current IEEE-754 floating-point values. Gustafson [15] proposed a new way of computing floating point numbers with a new format called the unums. It is based on dynamic adjustment of the number of bits required to store the floating-point number. It is also capable of annotate a number to store the fact that it might not be exact and lies in an interval to be able to provide a more accurate final result. Both these methods require processors manufacturers to completely change their standards which is not likely to happen.

Using custom application specific hardware

In order to achieve both performance and accuracy, one may then chose to design one's own hardware accelerator. The user is then able to adjust the custom circuit to the application requirements and use a custom format. Such a design could be implemented on a Field Programmable Gate Array (FPGA), which is a memory-based integrated circuit whose functionality can be modified after manufacturing. This hardware can increase the program performance and is very energy-efficient compared to a software implementation.

Originally FPGAs capacity (size of the design they can load) was limited and only made possible small designs (in terms of logic gates). Most early works on FPGAs targeted integer-based programs. Indeed, floating-point hardware operators are up to a factor of 30 larger that integer operators. However, the capacity of FPGAs has now reached a sufficient size to consider them as an alternative for floating-point computations. Moreover, they offer much more freedom in implementing floating-point computation than the previously mentionned (fixed hardware) accelerators. A purpose of the present work is to exploit this freedom.

Hardware design flow and High-level synthesis

A problem when designing an application-specific accelerator, is that it requires deep knowledge in circuit design. Programmers tends to be more comfortable with software languages such as `C` or `Java` than Hardware Description Languages (HDL). Using a HDL, such as VHDL, is very error prone and often results in a very long design time. In order to make hardware design more accessible, a new range of tools were created, called High-Level Synthesis (HLS) tools (VivadoHLS [18], GAUT [5], LegUp [4], Catapult C [14] among others). The goal of such tools is to allow the programmer provide a behavioral description (e.g `C` source code) of a component, and transform it to a circuit description. Typical compiler syntactic transformations cannot be applied because several constraints are not taken into consideration with high-level languages (e.g clock frequency, delay of a memory access). Therefore, these tools perform many analyses before returning a hardware description. They were not used until recently as they were not efficient, often resulting in bloated designs. Recent improvements on the internal heuristics ended up in making them effective. In this work, we only consider HLS tools for targeting FPGAs.

C standard driven hardware generation

A problem with HLS tools is that they are made for hardware conception but takes a `C` program as an input. However, the `C` language offers very limited number of data types with different sizes. This does not match with hardware design where the users want to have some custom variable width. Therefore, HLS tools introduced some dedicated libraries to support interger values with a custom number of bits. Such a support does not exists for floating point numbers that will always be 32 or 64 bits long, even if it is not needed in the context of the application. Furthermore, most compilers such as GCC follows the C99 or C11 standard for floating-point operations. Therefore, the transformations that they can apply are limited to improving computation speed without modifying the accuracy of the result. This tends to greatly reduce the number of transformations available. Plus they cannot perform any optimization that improves the accuracy of the result. For floating points programs, VivadoHLS also follows the C99 standard [17] as the resulting design will produce the exact same result as if it was computed using GCC for example. This is a limitation that should not exists as it is in contradiction with the flexible hardware it is aiming.

At the cost of silicon area, one could perform better computation speed while being consistent with the C99 standard. Kapre and DeHon [20] implemented a accumulator that increases performance speed while following the C99 standard for a large overhead in terms of silicon. In some cases, the C99 standard can be followed but optimised in an application specific way. For example, De Dinechin and Didier [6] proposed a very efficient (C99 compliant) division of a floating point number by a small integer. It uses very little resource but is highly specific to applications that need such a division.

The other way around is to relax the semantic of `C` and the C99/IEEE-754 standard. This is what we do in this work, we consider the `C` language to express mathematical formulas. Indeed, a program performing `a+b+c+d` would be interpreted as $((a+b)+c)+d$ using the C99 standard. We just consider it as a sum of four variables that can be performed in any order, just as the real numbers specification.

Application specific operators we focus on

In this work, we focus on floating-point reductions (accumulations, sums of product). A reduction is an associative and commutative computation which reduces a set of input values into a reduction location. *Listing* 1 provides the simplest example of reduction, where *sum* is the reduction location. Mathematically, such a reduction can be executed in any order, as the addition is associative and commutative. On the contrary, floating-point summations are not commutative, due to floating-point representation. Such a transformation on a floating-point reduction might introduce a large distortion of the result. Therefore, it cannot be executed in parallel (e.g tree-based addition), and must be serialized to ensure the semantic of the program [20].

Listing 1: Simple reduction

```
float sum = 0;
for (i=1; i<5; i++){
  sum+=A[i];
}
```

Hardware accelerators generation can be application specific. Therefore, beyond performance increase, the accuracy of floating-point computations can be tuned. Indeed, there is no need in following the IEEE-754 standard floating-point representation used in microprocessors, one could use custom format. Improving FPGA hardware accelerators for floating-point computations, through both performance and accuracy is a current research topic [8, 23], as well as optimizing HLS tools [13, 2]. This work aims at merging both worlds.

Internship motivation

Our work aims at bringing together HLS benefits, such as the abstraction given to the programmer, and low-level FPGA-specific optimization. As HLS tools lack optimization for floating-point operations, we want to provide a source-to-source tool that transforms a reduction loop-nest into a code that the HLS tools will synthesize efficiently. Our compilation flow is given in Figure 1.

As our approach lies on two tools - FloPoCo and GeCoS - we will first present them in Section 2. We will then describe in details the operators that we implemented, with the experimental results in Section 3. In Section 4, we describe our source-to-source transformations. Finally, we show a case study on convolutional neural networks (Section 5), and then conclude.

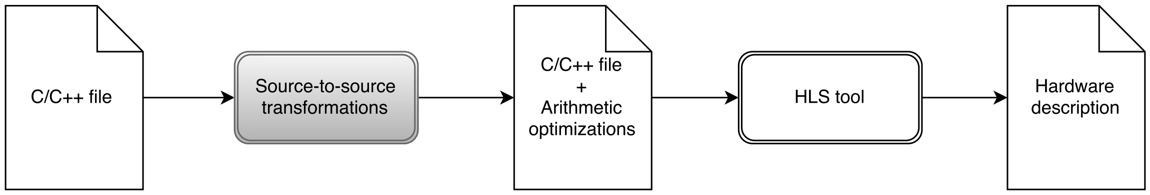

C/C++ file
Source-to-source transformations
C/C++ file + Arithmetic optimizations
HLS tool
Hardware description

Figure 1: Compilation flow

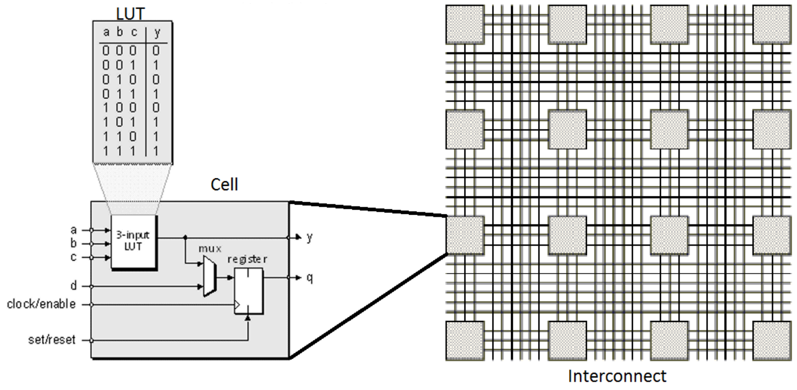


Figure 2: Internal structure of an FPGA

2 Context

As FPGAs architecture are different from other platforms, we first describe the structure of FPGAs. We then go in more details about the floating-point format before describing the tools we used.

2.1 Field Programmable Gate Arrays

An FPGA is made of Look-Up Tables (LUTs), which compute logic operations. The LUTs have α inputs and one output, on most recent FPGAs $\alpha = 6$ ($\alpha = 3$ in Figure 2). The designer then needs to keep in mind that algorithms relying on the 2^α values have efficient implementations in FPGAs. Indeed, values are scattered in chunks of α bits or less to fit in LUTs.

Depending on the vendor, one or several LUTs are gathered into cells. These cells are all connected through a programmable interconnect. This architecture is depicted in *Figure* 2.

Two neighbor cells benefit from a fast connection dedicated to carry propagation. This connection is faster than communication through the FPGA interconnect. It is shown in Figure 3 (here an Altera chip, Xilinx ones are equivalent) where we can see a `carry_in` signal coming from another cell, and a `carry_out` signal that goes toward next cell. This

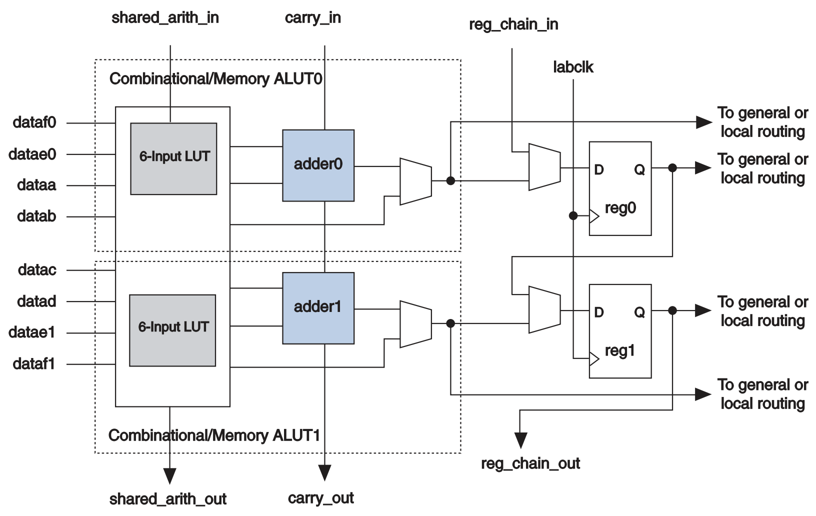


Figure 3: Cell of Altera's Stratix IV, Source : Stratix IV Device Handbook, Volume 1, Altera

particularity makes some classical optimization irrelevant for FPGAs. For example, this fast carry propagation makes a simple carry-ripple addition to be faster than most fast adders used for VLSI oriented architectures.

FPGAs also contains Digital Signal Processing (DSP) blocks. These embed for example fixed-point multipliers. Most recent FPGAs also embed floating point multipliers, typically 18×18 bits multipliers.

In order to program an FPGA, one have to design it's custom circuit through the use of a HDL. This HDL is then synthesized using dedicated softwares such as Vivado or Quartus II. Once the HDL is registered as syntactically correct, it is transformed to a logic gates representation (RTL). The RTL is optimized to remove unused hardware. It then has to be placed on the FPGA. Therefore a place and route phase solves an Integer Linear Programming (ILP) problem to try to put every operators as close as possible to the others. Once the ILP solved, a bitstream is created and given to the FPGA so that it configures itself respecting the original VHDL specification.

As the structure of FPGAs is radically different from a processor architecture, many optimization can be made for taking benefit from it.

2.2 Floating-point specification and limitations

The value of a floating-point number is then $(-1)^s \times 1.m \times 2^{(e-b)}$ where b is a constant called the floating point bias. Every operation over floating-point values has a specification that IEEE-754 compilers and processors has to follow. This standard makes floating-point computation consistent but has flaws. For example, the sum operation is non-associative, even if the real sum is. Indeed, if we consider the computation of $(small + big) - big$, the result is going to be 0. This, because the result of $small + big$ returns big as the mantissa cannot contain enough bits for containing the exact result of $small + big$. On the other side, the result of $small + (big - big)$ is going to be $small$.

We recall that a floating-point number is represented by three fields as depicted in *Figure* 6, a sign, an exponent and a mantissa (also called significand). The number of bits of the exponent and mantissa changes whether it is a 32 bit or 64 bit floating point representation. This is set by the IEEE-754 standard [1].

The mantissa gives the fractional part of the final number. An implicit 1 has to be added in front of the mantissa. This because the standard requires the numbers to be normalized with a leading 1. This leading one is thus removed and remains implicit.

In order to recover the floating-point number, one has to retreive $1.m$, where m is the mantissa. This number will have to be signed and shifted. The sign is given by computing $(-1)^s$, where s is the sign. A bias was added to the exponent, so the retrieved exponent need to be removed that bias. The bias is a constant that has a different value for simple and double precision. The final number is then returned as $(-1)^s \times 1.m \times 2^{(e-b)}$, where e is the exponent and b is the floating point bias.

There are some exceptions where the floating point number is not computed as above:

- When the exponent reaches the maximum value and the mantissa is filled with zeros: The floating-point value represents the infinity
- When the exponent reaches the maximum value and the mantissa contains bits other than zeros: This stands that the value hold is Not A Number (NaN)
- When the exponent reaches the minimum value and the mantissa is filled with zeros: The flaoting point value is 0.0
- When the exponent reached the minimum value and the mantissa contains bits other than zeros: The number is a subnormal

Subnormals are special floating-point numbers that have been introduced to fill the gap between the minimum positive floating-point value and the maximum negative floating-point value. Indeed, the format was not very accurate for numbers around zero. Figure 4 shows the representable floating-point values around zero (with a shorter mantissa and exponent for the sake of clarity). We can clearly see these gaps on both sides of 0.0. When a floating-point number has an exponent that reaches the minimum value, and the the mantissa is not filled with zeros, the former implicit 1 is replaced by an implicit 0. The subnormal number is then computed as $(-1)^s \times 0.m$. Figure 5 now shows the floating-point numbers around zero when extended with subnormals.

A slight change in the operations order might change the result accuracy. For the simplicity of the explanation, let us consider decimals with 7-digit significand numbers.

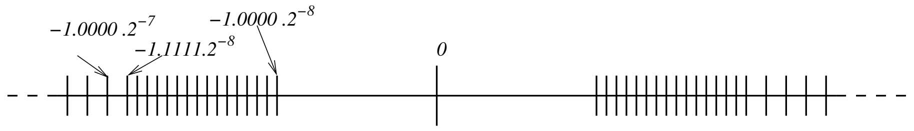


Figure 4: Floating-point values around zero without subnormals

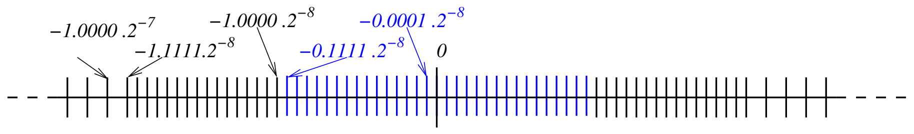


Figure 5: Floating-point values around zero with subnormals

We don't necessarily have $(a + b) + c = a + (b + c)$, for example, with $a = 1234.567$, $b = 45.67834$ and $c = 0.0004$, then:

```
(a+b)+c=
        ( 1234.567
        +   45.67834) + c
       =  1280.24534 + c // but 1280.24534 is rounded to 1280.245 (7 digits)
       =  1280.245
        +    0.0004
       =  1280.2454 // rounded to 1280.245
       =  1280.245
```

Whereas:

```
a+(b+c)=
         a + (45.67834
             + 0.0004)
       = a +  45.67874
       =  1234.567
        +   45.67874
       =  1280.24574 // rounded to 1280.246
       =  1280.246  // not equal to (a+b)+c
```

This format need special care if we want to achieve accurate floating point computing.

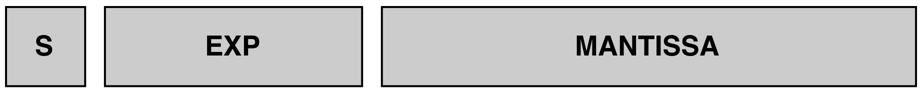


Figure 6: Representation of a floating-point number

2.3 Application specific arithmetic in FloPoCo

When using an HLS tool, one can generate an architecture for a specific program/operation. There is no need in following the IEEE-754 standard for floating point computation. Therefore, the internal format can be tuned to the program's need.

FloPoCo [7] is a generator of arithmetic cores for FPGAs. It generates operators that are radically different that what can be found in general purpose processors. These operators are parametrized in precision so as to fit the application needs, but no more. They have an internal representation that is generated given some program specification.

Complex operators tuned for the target

FloPoCo uses all the FPGAs optimization in order to benefit the most from its architecture. It can generate operators such as constant multipliers that removes the use of a large floating point multiplier. It can also generate divisors by small constants that are table based instead of using a lot of resource to create a floating-point divisor. It uses operator fusion to generate more complex operators. FloPoCo allows the users to manage complex operators that are parametrized in exponent and significand size, always last bit accurate, automatically optimized for Altera's and Xilix'stargets. The operators are also pipelined to frequencies close to the maximum practical on these FPGAs.

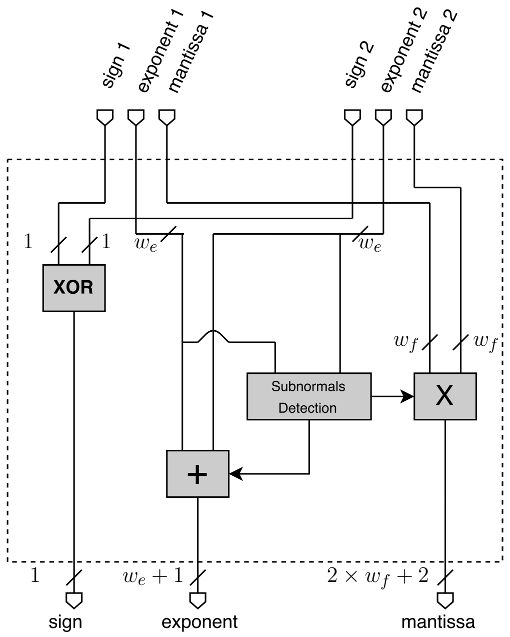
FloPoCo generates these operators as `VHDL` blocks. The user then needs to integrate it to their own `VHDL` design. This involves manipulating low level code and can be tedious.

As we want the users not to manipulate such low level code, we want to make FloPoCo's knowledge available to HLS users. Therefore we need to transform their high-level `C` code so that the HLS tool generates a `VHDL` description that has the same behavior as FloPoCo's operators.

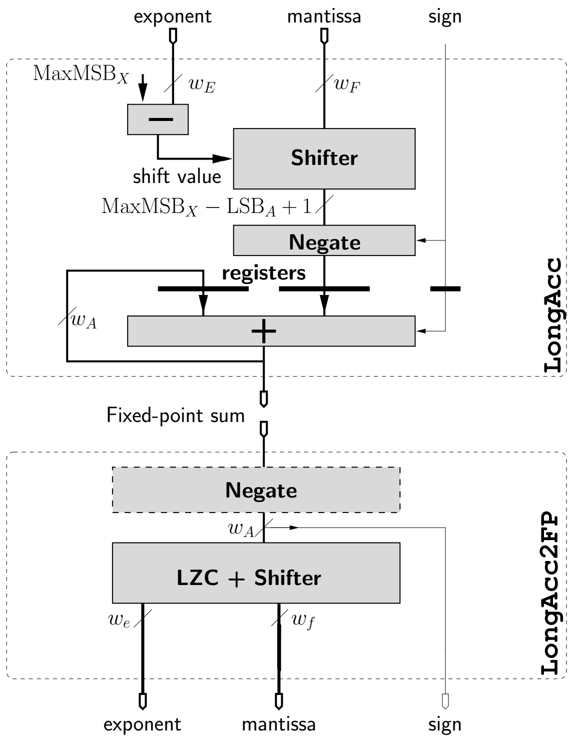
2.4 GeCoS, a source-to-source framework

Among all the source-to-source compiler frameworks available we chose GeCoS [11]. It is able to take a `C/C++` code as an input and translate it to its own intermediate representation. It can then generate the corresponding `C/C++` code. Many features allow the user to navigate through the intermediate representation in order to analyze or modify it. It also contains transformations to unroll a loop when the corresponding `pragma` is inserted. GeCoS is a compiler infrastructure entirely written in `Java`. It follows the model-driven engineering design principles. It is then easily extendable. One can create a `Java` plug-in to modify either the front-end, the intermediate representation or the back end of the compiler.

We can then create a GeCoS plug-in in order to apply our transformations on the intermediate representation. In the next section, we will describe some application specific operators for FPGAs and their corresponding VivadoHLS implementations. We will also



(a) Accurate floating-point multiplier



(b) The accumulator (top) and post-normalization unit (bottom).

Figure 7: The blocs used for implementing a sum-of-product

compare their performance with FloPoCo's operators in terms of latency, LUT usage and accuracy.

3 VivadoHLS implementation of application specific arithmetic operators

As we are implementing some FloPoCo operators into their corresponding `C` code, we first describe these operators before showing our `C` implementation. We then compare our results with FloPoCo and more standard implementations.

3.1 Application specific version of Kulisch's accumulator

We will describe the FloPoCo's specialized accumulator that performs an accurate computation for a low LUT usage and little latency. We will then show how to get a similar

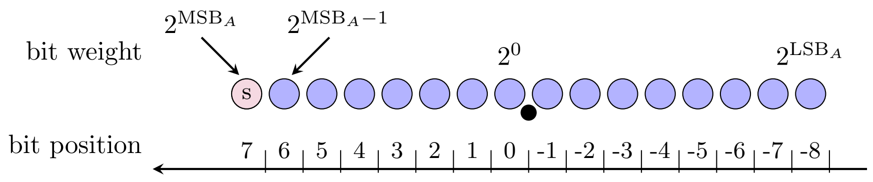


Figure 8: The bits of a fixed-point format, here $(\mathrm{MSB}_A, \mathrm{LSB}_A) = (7, -8)$.

operator from VivadoHLS, while described in a `C` source code. Finally, we will compare our operator generated by VivadoHLS to more naive implementations of accumulators.

3.1.1 FloPoCo's accumulator

The accumulator described in this section is an extension of the accumulator proposed by de Dinechin et al. [9]. Its purpose of it is to have a configurable internal representation in order to control accuracy. The support of denormals have been added to the original version. Such an accumulator is described in Figure 7b (top).

The internal representation of the accumulator is using a fixed-point format. The fixed-point format is represented by two values:

- MSB_A which is the weight of the most significant bit of the accumulator. For example, if the accumulator maximum value is set to $1E6$, then $\mathrm{MSB}_A = \lceil log_2(1E6) \rceil = 20$.
- LSB_A which is the weight of the least significant bit of the accumulator. For example, if the accumulator accuracy is set to $1E - 15$, then $\mathrm{LSB}_A = \lfloor log_2(1E - 15) \rfloor = -50$.

The accumulator width w_a is then computed as $\mathrm{MSB}_A - \mathrm{LSB}_A + 1 = 71$ bits in this axample. This kind of notation for fixed-point format is shown in Figure 8. For the sake of simplicity, we used $\mathrm{MSB}_A = 7$ and $\mathrm{LSB}_A = -8$ on this example. The circles represents bits of the number. The "virtual" point is placed between the bits with weight 0 and -1.

Float-to-fixed conversion

The accumulator takes a floating-point value as input. This number has to be converted into the fixed-point representation of the accumulator. It is decomposed into 1 bit of sign, w_E bits of exponent and w_F bits of mantissa. The leading 1 or 0 is then added to the mantissa. The decimal and fraction parts of the mantissa and the accumulator has to be aligned so that the "points" match.

To do so, we first place the extended mantissa at the left of the accumulator, so that their most significant bits match. This is shown in figure 9 where *imp* represents the implicit bit and the | is the "point". As the user provided MSB_A, we know that the mantissa cannot be be shifted to the left, otherwise it would mean that the maximum value was reached. In order to align the "points", the extended mantissa has to be shifted by $\mathrm{MSB}_A - 1 - exp_s$

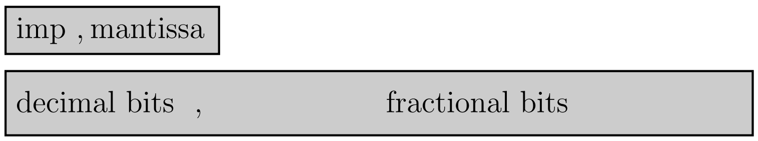


Figure 9: Mantissa placement for shifting to fit accumulator format

where exp_s is the exponent from which we removed the floating-point bias. The resulting number (using $\text{MSB}_A - \text{LSB}_A$ bits) is then XORed to match the fixed-point representation of the accumulator. It is then ready to be accumulated.

Note that, as shown in Figure 7b (top), the user can provide another variable called MaxMSB_X which represents the maximum value of the inputs. Whereas MSB_A represents the maximum value of the all accumulation. This reduces the above mentioned shifter by a few bits so that the number to add to the accumulator is only $\text{MaxMSB}_X - \text{LSB}_A$ bits instead of $\text{MSB}_A - \text{LSB}_A$ bits.

Fixed-point summation

As the input number has been converted to its fixed-point representation, the sum has to be a fixed-point summation. Therefore, it does not require the logic overhead of a floating-point sum. A simple bit-to-bit adder is used. The fixed-point adder has a 1-cycle latency whereas the floating-point adder requires 7 cycles.

Fixed-to-float conversion

The value of the accumulator can be retrieved in its fixed-point format and be converted to a floating-point value using a software conversion. As the goal of our transformations is to remain invisible to the user, we perform a hardware conversion instead. The hardware bloc that we used is depicted in figure 7b (bottom).

The value of the accumulator is first XORed to fit with the floating-point representation. We then perform a leading zero count (LZC) to find the most significant bit (MSB) of the accumulation. This bit will be the implicit bit of the resulting mantissa. The result is then shifted by $\text{LSB}_A + \text{MSB}_A - LZC - 1 - w_f$ into a w_f bit wide variable which is the mantissa. This way, the w_f bits remaining are exactly the w_f bits that are right after the MSB. The exponent is computed as $\text{MSB}_A - LZC + FPBIAS$ where $FPBIAS$ represents the floating-point bias. The sign is directly extracted from the negated accumulator value.

Note that no rounding is done, the mantissa is truncated. Several rounding could be applied here to get a IEEE-754 rounding such as *round to the nearest, tie to even.* As we went away from the IEEE-754 standard during the computation and improved accuracy, there is no need to introduce more logic to do this rounding. We could apply a simpler rounding through, where we add 1 to the right bit of the least significant bit before truncating. This would give a more accurate result in most cases, but it not not implemented for the moment.

This accumulator takes a float as an input and provides a float as an output. The rest of the computation can remain in the floating-point format whereas ID.Fix for example would make the whole program to be in fixed-point arithmetic. Thus, it would lose the ease of use of the floating-point format and might not be as accurate.

3.1.2 Implementation of the accumulator for VivadoHLS

As VivadoHLS generates a floating-point adder when a sum of floating-point values is encountered, we need to specify this operator at bit level in `C`. The accumulator can be split into three part:

- A float-to-fixed conversion
- A fixed point sum
- A fixed-to-float conversion

Listing 2: Float to accumulable conversion

```
typedef union {
  uint32_t i;
  float f;
 } bitwise_cast;
ap_int<MAXMSBx-LSBA+1> FP_to_accumulable(float a) {
  bitwise_cast var;
  var.f = a;
  ap_uint<32> in = var.ap;
  ap_uint<MANTISSA+1> mantissa = in.range(22, 0);
  mantissa[MANTISSA] = 1;
  ap_uint<EXP> exp_u = in.range(30, 23);
  ap_uint<1> sign = in[31];
  ap_int<EXP> exp_s = (ap_int<8>)exp_u - FP_BIAS;

  ap_uint<MAXMSBx-LSBA-MANTISSA> zeros = 0;
  ap_uint<MAXMSBx-LSBA+1> shift_buffer = mantissa.concat(zeros);
  uint32_t shift_val = MAXSMBx - 1 - exp_s;
  ap_int<MAXMSBx-LSBA+1> current_shifted_value = shift_buffer >> shift_val;
  if (sign == 1)
        return ~current_shifted_value + 1;
  else
        return current_shifted_value;
}
```

The float-to-fixed conversion code for VivadoHLS is given in Listing 2. We used a type *union*, denoted by `bitwise_cast` to swap between the floating-point and integer formats. The `i` field of this *union* contains the unsigned integer value of the bits of field `f`.

For the sake of explanation and modularity, we used integer variables instead of hard-coding bit width of every wire. As we are using VivadoHLS, we are taking advantage of Xilinx's *ap_int* library which allows to manipulate data at bit level. Here is a list of the variables used:

- `MAXMSBx`: Weight of the most significant bit of every input.
- `LSBA`: Weight of the least significant bit of the accumulator. This is the accuracy of the output fixed point format. This value is negative if there are fractional bits. We consider that for the fractional part of a fixed point number, the bits are denoted with a negative sign.
- `MANTISSA`: Number of bits of the floating-point mantissa (23).
- `EXP`: Number of bits of the floating-point exponent (8).
- `FP_BIAS`: The floating-point exponent bias.

We used a few operations of this library, which are:

- Bit field declarations. A `ap_int<N>` variable is a field of `N` bits. The format specifies it is signed in case of an extension to a larger format. In order to get the same unsigned field of bits one would have to declare it as a `ap_uint<N>` variable.
- The method `range(uint32_t start, uint32_t end)`. It allows to retrieve bits from index `start` to bit index `end` from the *ap_int* variable it is applied to.
- The method `concat(ap_int variable)`. It allows to concatenate the `variable` at the end of the variable the method is applied to.
- A syntax shortcut to retrieve or set a single bit. The library allows to access *ap_int* variables the same way one would do with arrays to retrieve a single bit.

An accumulable value is a fixed point value that has the same amount of fractional bits as the accumulator does. Also it has to have a lower or equal number of bits than the accumulator. That way when aligning the bits on the right side, as depicted in Figure 10, the sum can be computed with aligned points.

The output format of our function `FP_to_accumulable` is then a field of bits of size `MAXMSBx-LSBA+1` bits. The format is chosen signed in order to preserve the sign of the result when extending the format to be able to perform the sum with the accumulator. It has `LSBA` bits for the fractional part (same as the accumulator), and `MAXMSBx+1` bits for

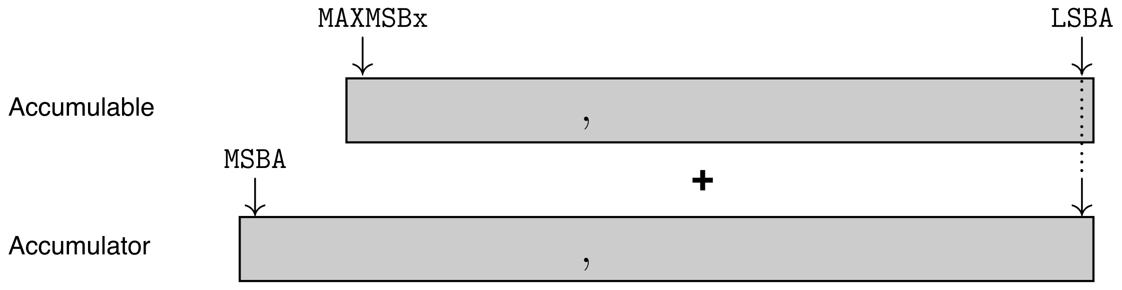


Figure 10: Fixed point accumulation

the decimal part. The decimal part has one more bit than `MAXMSBx` so that $2^{\texttt{MAXMSBx}}$ can be reached.

The first thing we do is to store the `float` input into our union type in order to retrieve the bits into a `ap_uint` bit field. We then retrieve the sign, exponent and mantissa using the range methods. We also set the implicit 1 for the mantissa. In case of a 0, it doesn't matter to add this implicit one, because the mantissa is going to be fully flushed during the shift operation because of a very little exponent. A future optimization would be not to flush it to zero if the accumulator accuracy is larger than the floating-point exponent range.

We then instantiate a variable that will contain the shifted value that we call `shift_buffer`. To prepare the mantissa shifting in order to align "points", we place the mantissa to the left of the `shift_buffer` and fill the rest of the bits with zeros as shown in Figure 9.

The `shift_buffer` is then shifted by a value depending on the float exponent. The final check is to verify if the float number was positive or negative. If it was negative, it has to be negated to fit the fixed point representation.

The fixed-point sum can be performed directly with a + operator between two fixed point numbers.

Listing 3: Fixed-point to floating-point conversion

```
float fixed-to-float(ap_int<MSBA+1-LSBA> acc)
  ap_int<MSBA+1-LSBA> tmp;
  ap_uint<1> s;
  if (acc[MSBA-LSBA] == 1) {
    tmp = -acc;
    s = 1;
  } else{
    s = 0;
    tmp = acc;
  }
  uint32_t lzc = tmp.countLeadingZeros();
  ap_uint<8> exp = MSBA - 1 + FP_BIAS - lzc;
  ap_uint<23> mant = tmp >> MSBA - LSBA - MANTISSA - lzc;
  ap_uint<31> ret_without_sign = exp.concat(mant);
  ap_uint<32> ret = s.concat(ret_without_sign);
  bitwise_cast bits_to_fp;
  bits_to_fp.ap = ret;
  return bits_to_fp.f;
}
```

Finally, the fixed-to-float operation must be performed to retrieve the floating-point value of the accumulator. The implementation of this operation is given in Listing 3. The accumulator is given as an input, and the sign of it is retrieved. If case of a negative value, the accumulator is negated to fit in the floating-point representation. The leading zero count (`lzc`) is performed using the `countLeadingZeros` method from the `ap_int` library. Using the `lzc`, the exponent and mantissa are computed. The sign, exponent and mantissa are then concatenated using the `concat` method also given by the `ap_int` library. Using the same *union* as in Listing 9, the bit field is returned as as float.

3.1.3 Experimentation

Objective

Besides checking the correctness of our implementation, this experimentation aimed at verifying that the generated hardware was indeed what we expected. We want our implementation to fulfill two main objectives, other than having improved accuracy:

- We want it to have a latency close to the one from FloPoCo
- We want LUT usage to be as close as FloPoCo's

In order to verify the above mentioned points, we used a simple floating point accumulation. Such an accumulation is depicted in Listing 4. As this accumulation is manipulating floating-points values, VivadoHLS is not able to reorder the operations and will wait for an iteration to end before computing the next one. This introduces delays between iterations. As our implementation is not working on floating-point values anymore, VivadoHLS is going to be able to perform more aggressive optimization. Thus we avoid waiting between iterations.

Experimental setup

We decide to compare our results with an equivalent to the naive float accumulations. If the loop is unrolled, VivadoHLS is going to achieve more parallelism and reduce overall latency. We unrolled the loop by a factor 7, as it is the latency of a floating point adder on our target FPGA (Kintex 7). This code is shown in Listing 5. This is not C99 equivalent, but is a trade-off to get a better computation speed at the cost of resource consumption. This code cannot be generated from a tool as it requires data knowledge. In order to improve accuracy, one may chose to perform the calculation using the `double` format for the inner computation. Thus we compared our results with a naive double accumulation, and it's unrolled equivalent.

The accumulation operates during 99995 iterations. This number was chosen in order to be a large value, close to 100000, while being a multiple of 7 (the unroll factor). This is not in our advantage but to get the best out of the unrolled version. It provides the unrolled code not to use control over the iteration domain boundaries, and remove some control.

Listing 4: Naive float accumulation

```
#define N 99995
float accumulation(float in[N]) {
  float acc = 0;
  for(int i=0; i<N; i++){
    acc+=in[i];
  }
  return acc;
}
```

Listing 5: Unrolled float accumulation

```
#define N 99995
float accumulation(float in[N]){
 float acc= 0;
 float p0=0,p1=0,p2=0,p3=0,
       p4=0,p5=0,p6=0;
 for(int i=0; i<N; i+=7){
   p0+=in[i];
   p1+=in[i+1];
   p2+=in[i+2];
   p3+=in[i+3];
   p4+=in[i+4];
   p5+=in[i+5];
   p6+=in[i+6];
 }
 acc = p0+p1+p2+p3+p4+p5+p6;
 return acc;
}
```

Note that for the addition on Kintex 7, the latency of a double adder is also of 7 cycles, so the unrolled `double` accumulation is unrolled by a factor 7. Muller et al. [24] proposed to use cos values to perform an error evaluation for the accumulation. Therefore, our input values are $(float)\cos(i)$ where i is the input array's index so that the accumulation performs the computation of $\sum_i cos(i)$. Every LUT usage number and latency of the design are given by generating `VHDL` from `C` synthesis using VivadoHLS followed by place and route from Vivado v2015.4, build 1412921.

To compare the quality of the result, we compute the relative difference between the result $\widetilde{R}$ computed for all the different programs, and the exact result R computed using MPFR [12]. To measure the number of correct bits of the result we compute $(-\log_2 \frac{\widetilde{R}-R}{R})$. In the case of this study case, the returned result is a `float`. The maximum accuracy is then of 24 bits, the size of the IEEE-754 `float` mantissa.

Finally, the parameters that we chose for our accumulator are:

- MSBA = 17. Indeed, as we are adding $\cos(i)$ 99995 times, an upper bound is 99995, which can be encoded in 17 bits.

- MAXMSBx = 1. For the same reason as for MSBA, at every step, the maximum input will be 1, which can be encoded in 1 bit.

- LSBA = -50. This is an arbitrary precision that we chose. The accumulator will then be accurate until the 50th fractional bit.

The corresponding implementation is shown in Listing 6. It uses the previously presented blocks.

Listing 6: Our implementation of the accumulation for VivadoHLS

```
#define N 99995
float Accumulation(float in[N]) {
  float acc = 0;
  ap_int<68> long_accumulator = 0;

  for(int i=0; i<N; i++) {
    long_accumulator += FP_to_accumulable(in[i]);
  }

  acc = fixed-to-float(long_accumulator);
  return acc;
}
```

Performance and accuracy results

The results we obtained are described in Table 1. As expected, VivadoHLS is not performing any optimization on the naive examples, and the latency of one iteration is 7 cycles. When unrolling the loop, VivadoHLS is using almost 4 times more LUTs for floats, and 3 times more for doubles. VivadoHLS uses DSPs for both `float` and `double` format whereas

	Naive Acc (float)	Unrolled Acc (float)	Naive Acc (double)	Unrolled Acc (double)	Our Code	FloPoCo's Operator
LUTs	266	907	801	2193	736	719
DSPs	2	4	3	6	0	0
Latency	699 966	142 880	699 966	142 880	100 000	100 001
Accuracy	17 bits	17 bits	24 bits	24 bits	24 bits	

Table 1: Comparison between the different accumulators

it does not for our approach, just as FloPoCo's operators. The unrolled versions improves latency over naive versions. Nervertheless, our approach gets even betters latencies for a reasonable LUT usage. Also, we achieve maximum accuracy for the `float` format. The internal representation of the *double*, unrolled *double* and our approach have a higher accuracy than 24 bits. However, the *float* format caps the accuracy to 24 bits. Our results are very close to FloPoCo's ones, both in terms of LUTs usage, DPSs and latency.

We have shown through this example that VivadoHLS is able to generate a design comparable to FloPoCo's operators. In the next section, we will describe how we implemented another FloPoCo operator, the multiplication, and how we combined it with the accumulator to get a sum of product operator.

3.2 Sum-of-product

We will describe the FloPoCo's specialized sum-of-product operator that performs an accurate computation for a low LUT usage and little latency. We will then show how to get a similar operator from VivadoHLS, while described in a `C` source code just as in the previous section. Finally, we will compare our generated operator with more naive implementations of sums of products.

3.2.1 Floating-point multiplier and its combination with the accumulator

The floating-point multiplier described in this section is a part of the FloPoCo framework [9]. We detail here a variation of it as shown in Figure 7a. It computes the product of two numbers in a floating-point format with w_e bits of exponent and w_f bits of mantissa. The exponents are scanned to detect if a 0 or a subnormal is detected. In both cases the multiplier returns 0. Otherwise, the exponents are added and the mantissas multiplied. The result's sign is simply a XOR operation between the two input signs.

The output of this multiplier is not a standard float as it is made of $w_e + 1$ bits of exponent and $2 \times w_f + 2$ bits of mantissa. We do not round the result as we want improved accuracy. Note that this multiplier returns the exact result of a floating-point product, except when one entry is a subnormal. A further optimization would be to handle subnormals without flushing them to 0.

In order to perform a sum-of-product, we need to make the multiplier's output fit into the accumulator's input. As the accumulator can be tuned for any size of exponent and mantissa, we just have to change its input size to fit the large exponent and mantissa size of the multiplier.

The combination of these two operators gives us a sum-of-product operator that accumulates exact results from the products. We will now see how to write it using the `C` language in order for VivadoHLS to generate the expected design.

3.2.2 Implementation of the sum-of-product for VivadoHLS

The implementation of the multiplier in `C` for VivadoHLS in given in Listing 7. It takes two floating-point inputs and convert them into bits field using the *union* defined in Listing 9. The signs, exponents and mantissas are retrieved using the `range` method from the `ap_int` library. The implicit bits of the mantissas are set to one, because in case of a subnormal or a zero, the result is going to be 0 anyway because of the shifting. In the case of no zeros nor subnormals, the mantissas are multiplied and the exponents computed. Finally, all parts are concatenated using the `concat` method. The result holds in `2*MANTISSA+2+EXP+1+1`, indeed:

- The product of two `MANTISSA+1` bits variables holds in `2*MANTISSA+2` bits
- The sum of two `EXP` bits variables holds in `EXP+1` bits
- The XOR of two `1` bit variables holds in `1` bit

Listing 7: Accurate floating-point product for VivadoHLS

```
ap_uint<2*MANTISSA+2+EXP+1+1> product(float in1, float in2){
  bitwise_cast var1,var2;
  var1.f=in1;
  var2.f=in2;
  ap_uint<32> a=var1.i;
  ap_uint<32> b=var2.i;

  exponent_u exp_a = a.range(MANTISSA+EXP-1,MANTISSA);
  exponent_u exp_b = b.range(MANTISSA+EXP-1,MANTISSA);
  mantissa m_a = a.range(MANTISSA-1, 0);
  mantissa m_b = b.range(MANTISSA-1, 0);
  m_a[MANTISSA]=1;
  m_b[MANTISSA]=1;
  sign s_a = a[MANTISSA+EXP];
  sign s_b = b[MANTISSA+EXP];

  sign ret_s = s_a^s_b;

  ap_int<EXP+1> ret_exp;
  ap_uint<2*MANTISSA+2> m_product;
  if(exp_a == 0 || exp_b == 0){
    ret_exp = 0;
    m_product = 0;
  }
  else{
    m_product = ((ap_uint<2*MANTISSA+2>) m_a)*((ap_uint<2*MANTISSA+2>) m_b);
    ret_exp = exp_a + exp_b - 2*FP_BIAS;
  }

  ap_uint<2*MANTISSA+2+EXP+1> ret_without_sign = ret_exp.concat(m_product);
  product_output ret = ret_s.concat(ret_without_sign);

  return ret;
}
```

As the floating point to accumulable operator used to take a float as an entry and now needs to take a `2*MANTISSA+2+EXP+1+1` bits fields variable, we created a new one which is made to be placed after a product. It is shown in Listing 8. The only changes made to the `FP_to_accumulable` from Listing 9 is the input format and the bits selections using the `range` methods.

Listing 8: Floating point to accumulable after product

```
ap_int<MAXMSBx-LSBA+1> Mult_to_accumulable(ap_uint<2*MANTISSA+2+EXP+1+1>in){
  ap_int<EXP+1> exp_s;
  ap_uint<2*MANTISSA+2> m;
  ap_uint<1> s;
  ap_int<MAXMSBx-LSBA+1> current_shifted_value;

  m = in.range(MANTISSA+MANTISSA+1, 0);
  exp_s = in.range(2*MANTISSA+2+EXP,2*MANTISSA+2);

  s = in[2*MANTISSA+2+EXP+1];

  ap_uint<MAXMSBx-LSBA+1> shift_buffer;
  ap_uint<MAXMSBx-LSBA-(2*MANTISSA+1)> zeros = 0;
  shift_buffer = m.concat(zeros);

  uint32_t shift_val = MAXMSBx-1 - exp_s-1;
  current_shifted_value = shift_buffer>>shift_val;

  if(s==1){
    return (~current_shifted_value)+1;
  }
  else{
    return current_shifted_value;
  }
}
```

3.2.3 Experimentation

Objective

As the experiment made in Section 3.1.3, this experiment was done to check the correctness of our implementation and to verify that the generated hardware is indeed what we expected. Our goal is to obtain a latency close to what FloPoCo can offer for an equivalent LUT usage.

Experimental setup

The naive implementation of a sum of product is depicted in Listing 9. When using this source code, VivadoHLS is not able to perform any arithmetic optimization as it needs to respect the `C` semantic and the IEEE-754 standard. Therefore a loop iteration needs to be completed before the next one can start in order to respect inter-iteration RAW dependencies. This implementation can also make several rounding errors:

- When computing the product of `in1[i]*in2[i]` the result is rounded and stored into a `float`

- When computing the sum, the result is rounded again

In order to increase computation speed and reduce the latency of the design, one may chose to unroll the loop. As the latency of a product followed by a sum is 10 cycles on Kintex 7, we decided to implement another version of the naive program by unrolling the loop by a factor 10. This is not equivalent to the original `C` semantic of the naive program but it allows us to greatly reduce the latency of the design. Such a program is shown in Listing 10

Listing 9: Naive float accumulation

```c
#define N 130000
float sumOfProduct(float in1[N],
                   float in2[N]){
  float sum = 0;
  for (int i=0; i<N; i++){
    sum+=in1[i]*in2[i];
  }
  return sum;
}
```

Listing 10: Unrolled float sum of product

```c
#define N 130000
float sumOfProduct(float in1[N],
                   float in2[N]){
 float sum = 0;
 float p0=0, p1=0,p2=0,p3=0,p4=0,
       p5=0, p6=0,p7=0,p8=0,p9=0;
 for(int i = 0; i<N; i+=10){
   p0+=in1[i]*in2[i];
   p1+=in1[i+1]*in2[i+1];
   p2+=in1[i+2]*in2[i+2];
   p3+=in1[i+3]*in2[i+3];
   p4+=in1[i+4]*in2[i+4];
   p5+=in1[i+5]*in2[i+5];
   p6+=in1[i+6]*in2[i+6];
   p7+=in1[i+7]*in2[i+7];
   p8+=in1[i+8]*in2[i+8];
   p9+=in1[i+9]*in2[i+9];
 }
 sum = p0+p1+p2+p3+p4+
       p5+p6+p7+p8+p9;
 return sum;
}
```

For the same reason as in Section 3.1.3, we also compare our implementation to these two programs when using a `double` internal format for increased accuracy. Note that the latency of a product followed by a sum is 13 cycles on a Kintex 7 when using the `double` format. Therefor, the unrolled implementation of the sum of product using a `double` internal format is unrolled by a factor of 13. The number of iterations performed is set to 130000 as it is a large number that is a multiple of 10 and 13. This way, we don't have to add any control flow in the loop body to prevent from getting out of the bounds of the array.

The input values that we used were $\cos(i)$ and $\cos(i+0.5)$ were i is the input array's index so that the sum of product computes $\sum_i \cos(i) \times \cos(i+0.5)$. This example is a variation of what was proposed in [24] to test the accuracy of an accumulator. The parameters that we used for our accumulator are then chosen according to the input values as follows:

- MSBA $= 17$. Indeed we are multiplying $\cos(i)$ by $\cos(i+0.5)$ which is bounded by 1. And we add then 130000 times so the maximum value of the accumulator is bounded by 130000 which can be encoded in 17 bits.
- MAXMSBx $= 1$. Indeed, each input is bounded by 1 (product of two cosines).
- LSBA=-50. This is an arbitrary precision that we chose, just as in Section 3.1.3.

The resulting code that we propose is shown in Listing 11. It uses the above mentioned `C` for VivadoHLS implementations.

Listing 11: Our implementation of the sum of product for VivadoHLS

```
#define N 130000
float sumOfProduct(float in1[N], float in2[N]) {
  float sum = 0;
  ap_int<68> long_accumulator = 0;

  for(int i=0; i<N; i++) {
    long_accumulator += Mult_to_accumulable(product(in1[i], in2[i]));
  }

  acc = fixed-to-float(long_accumulator);
  return acc;
}
```

Performance and accuracy results

The results we obtained are described in Table 2. The latency of the non unrolled versions are as expected very high because of VivadoHLS not being able to perform any optimization. The unrolled versions performs lower latencies at a cost of a way higher LUT usage. We can see that our approach offers a very little latency, such as FloPoCo's one. We are very close to FloPoCo's operators in terms of LUT usage.

We have shown through two examples with the accumulation and the sum of product that VivadoHLS is able to generate specialized floating-point operators. The cost for being able to obtain such operators is to rewrite the program in a more detailed way, by changing the internal representation. The generated designs have improved the accuracy when specialized for a specific application. Even if we improved accuracy and latency, we did not followed the `C` semantic nor the IEEE-754 floating-point standard. This will be discussed in next section.

4 GeCoS source-to-source transformations

Now that we showed that VivadoHLS is able to generate specialized operators of similar quality to FloPoCo's ones, we want to provide a tool that transforms the naive imple-

	Naive Code (float)	Unrolled Code (float)	Naive Code (double)	Unrolled Code (double)	Our Code	FloPoCo's Operator
LUTs	313	1552	881	3727	868	693
DSPs	5	5	14	14	2	2
Latency	1 300 001	182 045	1 430 001	195 046	130 006	130 006
Accuracy	16 bits	21 bits	24 bits	24 bits	24 bits	

Table 2: Comparison between the different sums of products

mentations into the code that we wrote. We focused on two operators, the accumulation and the sum of product. These operators applies to reductions, which can be automatically detected within a source code in many cases [25, 10]. We chose not to detect these automatically to let to user decide which part of his code is to be optimized.

Compiler directive

We ask the user to use a compiler directive such as a `pragma` to specify the accumulation targeted. Plus, this gives us the possibility to ask the user to insert some application specific information such as the range of the manipulated data. The `C` implementation of reductions comes with the use of a `for` loop (or a `while` loop), therefore, our `pragma` must be used on a loop. To illustrate our `pragma`, we wrote it in for the naive accumulation of Listing 4. The resulting code is depicted in Listing 12. The `pragma` must contain the following informations:

- The keyword `FPacc` so that we can detect that this pragma is for using our transformations.
- The name of the variable in which the accumulation is performed. This is done using the keyword `VAR` followed by the name of the variable. In this case, the accumulation variable is `acc`.
- The maximum value that can be reached by the accumulator through the use of the `MaxAcc` keyword. This value is used to determine the weight of MSB_A.
- The desired accuracy of the accumulator using the `epsilon` keyword. This value is used to determine the weight of LSB_A.
- Optional: The maximum value of the inputs of the accumulator in the `MaxInput` field. This value is used to determine the weight of MaxMSB_X.

Listing 12: Illustration of the use of a `pragma` within the naive accumulation

```
#define N 99995
float accumulation(float in[N]) {
  float acc = 0;
  #pragma FPacc VAR=acc MaxAcc=99995 epsilon=1E-15 MaxInput=1
  for(int i=0; i<N; i++){
    acc+=in[i];
  }
  return acc;
}
```

The values that we gave in this `pragma` are the ones that we would have used for the example of Section 3.1.3. Indeed, if the desired accuracy is up to $1E-15$, then the weight of the LSB_A is $\lfloor \log_2(1E-15) \rfloor = -50$. The values of `MaxAcc` and `MaxInput` are explained in Section 3.1.3.

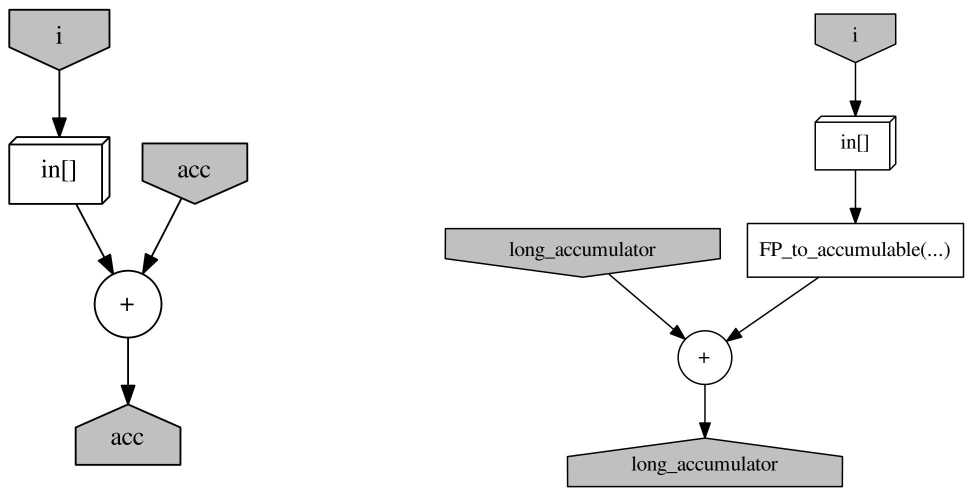
GeCos extension

In order to parse the source code and to generate our modified version of it, we used the GeCoS [11] source-to-source compiler. As GeCoS is built upon model driven engineering, it is easily extensible. Indeed one can modify the front-end to extend the `C` language or even to create his own language. One can also apply transformations to the Internal Representation (IR) of GeCoS in order to have a generated code that is different from the input. Also, any back-end can be written to extend the supported list of languages. GeCoS can be extended as above by using `Java` plug-ins.

What we want to achieve is to apply transformations on the IR so that the core of the loop marked with a `pragma` is going to be modified with our implementation. In a first part we present the basics transformations that we performed before explaining more general transformations.

4.1 Basic pattern recognition

To illustrate our transformations, Figure 11a shows the Direct Acyclic Graph (DAG) of the body of the loop from Listing 12. The DAG shows that the value of `in[i]` is added to the value of `acc` and stored into `acc`. This is the pattern that we want to transform for the accumulation. If such a pattern is detected in the DAG of the body of the loop, then the `FP_to_accumulable` and `fixed-to-float` functions are instantiated. Within the DAG the variable `acc` is replaced with the variable `long_accumulator`, which is instantiated in a fixed-point format. A call to the function `FP_to_accumulable` is inserted between the value to be accumulated and the `+` node. The new DAG associated to the for loop is shown in Figure 11b.



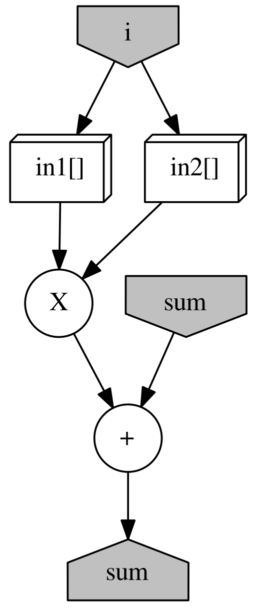
(a) Before transformations (b) After transformations

Figure 11: DAG of the loop body from Listing 12

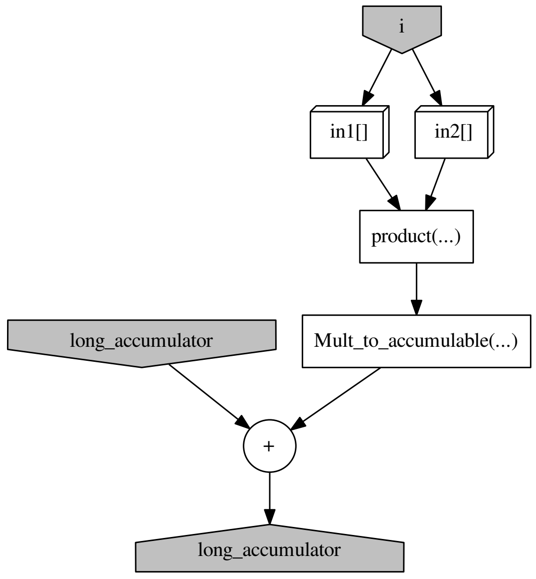
Just after the `for` block, the variable `acc` is assigned the value of the function `fixed-to-float()` applied to `long_accumulator`. The code generated from this IR is the one from Listing 6.

Similarly, we want to apply transformations for sums of products. The only difference in the pattern we look for is the presence of a $\times$ node in place of the value to be accumulated. The DAG corresponding to the body of the loop from Listing 9 is given in Figure 12a. We can see on the graph that the product of `in1[i]` and `in2[i]` is accumulated to the variable `sum`. When we find this pattern, we instantiate the function `product` from Listing 7 and `Mult_to_accumulable` from Listing 8. We then replace the $\times$ node by a call to `product` which is then given to `Mult_to_accumulable`. The resulting DAG is shown in Figure 12b. The conversion of the accumulator from fixed to floating-point is also done outside the for loop by calling the `fixed-to-float` function.

As these patterns relies on very specific conditions, such as a single accumulation/sum-of-product statement within a for loop, we extended our transformations to a more general context.



(a) Before transformations



(b) After transformations

Figure 12: DAG of the loop body from Listing 9

4.2 DAG exploration

In order to support a larger class of programs, we implemented a DAG exploration method. This program extends the pattern recognition detailed above. To illustrate what the algorithm does, we take a sample program shown in Listing 13. This programs performs a reduction into the variable `sum`. It does both sums and a sum of product. When retrieving the DAG associated to the loop body, we obtain the DAG from figure 14a. Even if the program performs an accumulation, the corresponding pattern is not present in the graph. Indeed, the GeCoS IR puts the `sum` variable to accumulate upper in the DAG than just above the very bottom node.

Listing 13: Simple reduction with multiple accumulation statements

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

To address this problem, we perform a first pass on the DAG. If the DAG represents an accumulation, the `sum` variable that needs to be accumulated is switched with the input from the next node. This next node is checked to be a `+` node as otherwise the DAG would not represent an accumulation. To illustrate this transformation, the pattern from Figure 13a is modified to the pattern from Figure 13b. The transformation is performed until we obtain a DAG that looks like the one from Figure 12a. Indeed, we want the result of `sum` to be the sum of `sum` and something else. We call this pass the normalization pass.

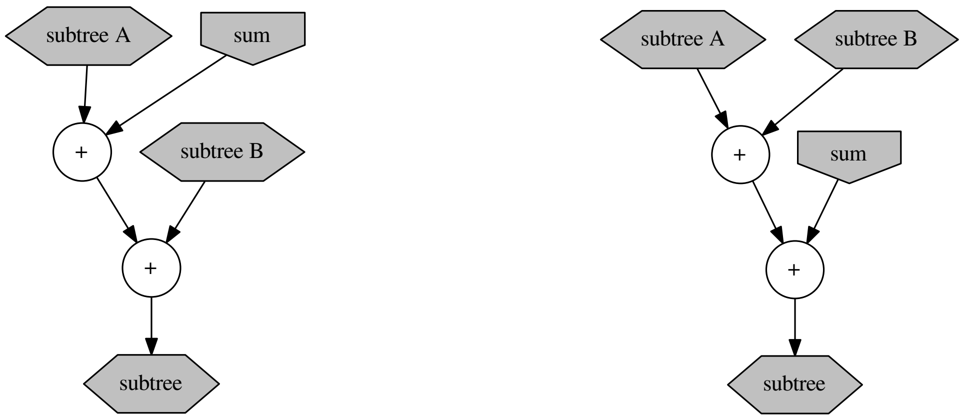
It is important to note that by doing these transformations, we did not followed the IEEE-754 standard for floating-points. Indeed $(a + b) + c$ is not equivalent to $a + (b + c)$ for floating-point values. We used the DAG of the `C` program to represent a mathematical computation and did not stick to the `C` semantic. As we used a fixed-point representation instead, the `+` operator is associative.

After performing this normalization pass on the DAG from Figure 14a, we obtained the new DAG showed in Figure 14b. This new DAG is ready to be applied our transformations.

Our analysis is a bottom-up transformation. First of all, we check that the structure of the bottom of the DAG is in the shape of an accumulation, like what we obtain from our normalization pass. We then run from the bottom `+` node to higher source nodes. The source nodes from a + node can be from different types:

- The `sum` node. This node is ignored and the analysis continues.
- A `+` node. The analysis is recursively launched on that node.
- A `X` node. The node is replaced by a call to the `product` function that is given to a call to `Mult-to-accumulable`.
- Any other node. A call to `FP_to_accumulable` is inserted.

Performing this algorithm on the DAG from Figure 14b we obtain the new DAG shown in Figure 15. This algorithm is performed for every bloc within the `for` bloc annotated



(a) Pattern to transform (b) Pattern transformed

Figure 13: DAG transformation to make our analysis (from 13a to 13b)

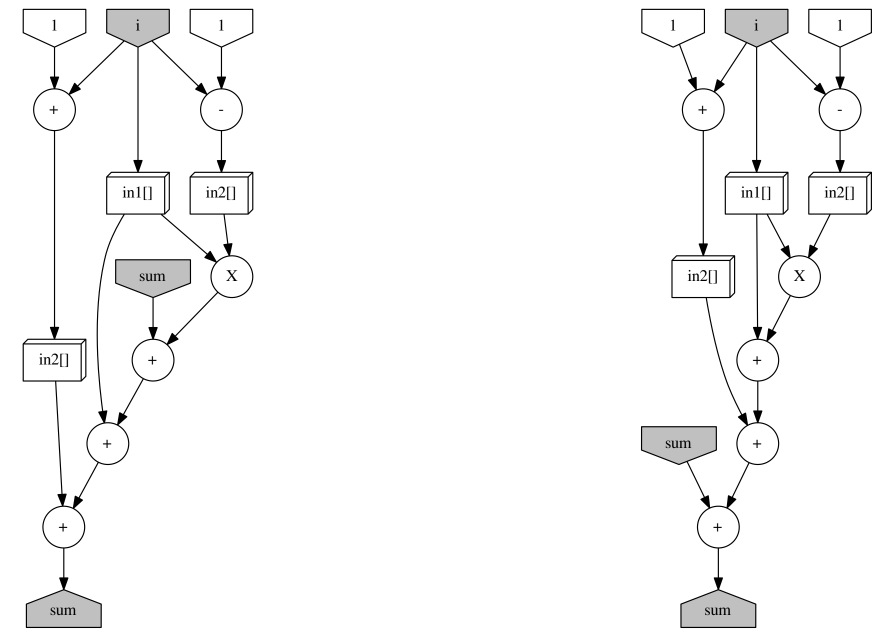
with our `pragma`. When every DAG as been transformed, the corresponding `C` code is generated.

We have been able to provide a tool to generate automatically our transformations. Through the use of a `pragma`, every user is able to obtain FPGA specific floating-point operators, without any knowledge on using fixed-point arithmetic. The use of the `pragma` allowed us to recover some program specific informations that are necessary to tune our generated operators. The transformed code can be given to VivadoHLS without any change and will hopefully result in lower latency, area usage with higher accuracy.

5 A case study : Deep convolutional neural networks

In order to get a real-life example to test our transformations, we performed a case study on the matrix convolution. It is a big part of the computation for image recognition algorithms. The matrix convolution is widely used in convolutional neural networks (CNN) applied to image recognition. CNNs are made of two major steps:

- The forward propagation. It consists in applying transformations (e.g convolution) to the input image a given number of times.
- The backward propagation. This is the learning step were the algorithm goes back through all the layers of the forward propagation to estimate the error and update



(a) Before transformations (b) After normalization

Figure 14: DAG of the loop body from Listing 9

its values (e.g values of the convolution matrix).

As CNNs tends to have more and more computation layers [16, 21], one may want to create a hardware accelerator. Therefore we used our tool on the matrix convolution.

The matrix convolution algorithm consists in computing points of a matrix condidering their neighbours. The algorithm takes a matrix as an entry wich is the image to transform and a convolution matrix, wich is a way smaller matrix (usualy 3×3 or 5×5) that gives weight to the neighbours cells. Figure 16 provides an example of a matrix convolution. We can see that to compute a point in the new matrix, we take the corresponding point in the original matrix as well as its neighbours. We take as many neighbours as the size of the convolution matrix. The value of the resulting point is computed as follows: $(0 \times 4) + (0 \times 0) + (0 \times 0) + (0 \times 0) + (1 \times 0) + (1 \times 0) + (0 \times 0) + (1 \times 0) + (2 \times -4) = -8$.

These weights can be bounded as they are constants of an application. Plus, the values of the cells of the matrix are normalized before the begining of the convolition to a value between -1 and 1.

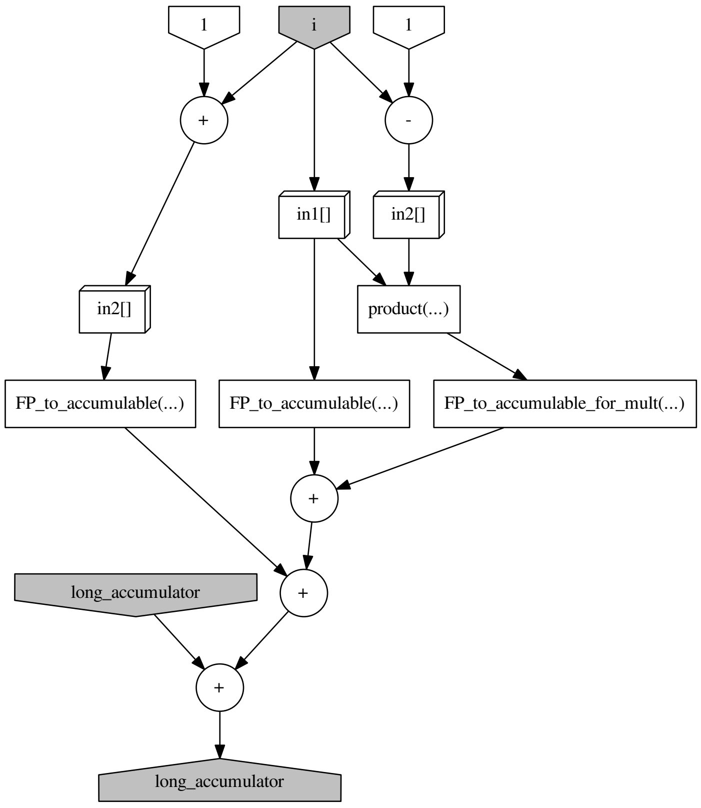


Figure 15: DAG of the loop body from Listing 9 after transformations

The `C` code of such a program is depicted in Listing 14 for an image of size 150×150 pixels. It performs 150×150 accumulations. Each accumulation contains 5×5 products as this is the filter size. The filter we used for our experiment is the following:

$$\begin{pmatrix} 0 & 0 & -1 & 0 & 0 \\ 0 & 0 & -1 & 0 & 0 \\ 0 & 0 & 4 & 0 & 0 \\ 0 & 0 & -1 & 0 & 0 \\ 0 & 0 & -1 & 0 & 0 \end{pmatrix}$$

This filter is widely used as it allows to perform an edge detection of images. Many optimisations can be applied to such a filter as it contains many zeros. However, this

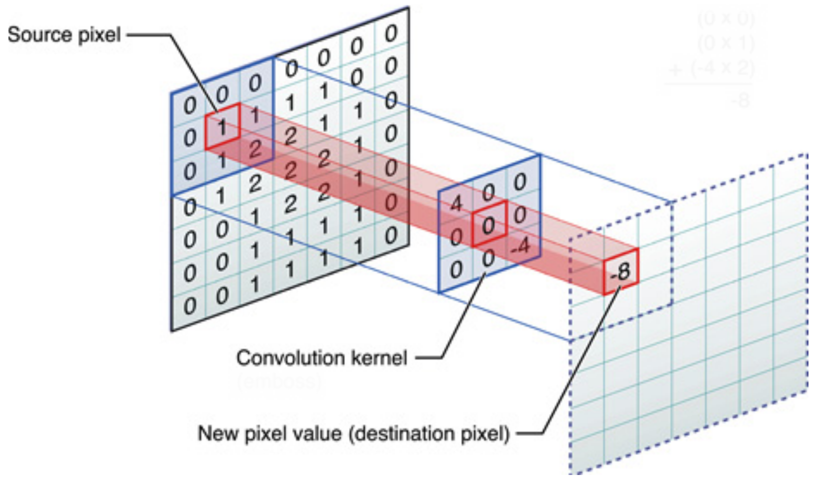


Figure 16: Matrix convolution algorithm

filter is going to be modified during the learning stages, making these zeros very small values next to zero. We do not show such a matrix has it depends on the algorithm, the input image, etc. Given all these informations, we can build our *pragma* to tune our operator in an application specific way. The maximum value that an accumulator can take is $(-1 \times -1) + (-1 \times -1) + (4 \times 1) + (-1 \times -1) + (-1 \times -1) = 8$. The maximum input at each iteration is then $4 \times 1 = 4$. We can set the parameters of our pragma to:

- `MaxAcc` $= 8$
- `MaxInput` $= 4$
- `epsilon` $=$ 1E-15 (arbitrary precision)

We compared the results of VivadoHLS when using the code of Listing 14 with the code we generated from that same code. The values of the input matrix are defined as $\cos(i+j)$ where i is the line index and j is the column index of the matrix. These values are chosen to be reproductible *random* values between -1 and 1.

	Naive Implementation	Our Code
LUTs	465	711
DSPs	5	2
Latency	4 802 102	2 559 752
Accuracy	18 bits	23 bits

Table 3: Comparison between the naive matrix convolution and our generated code

Listing 14: Matrix convolution

```
#define FILTER_SIZE 5
#define WIDTH 150
#define HEIGHT 150

void image_convolution(float filter[FILTER_SIZE][FILTER_SIZE],
                       float image[WIDTH][HEIGHT],
                       float result[WIDTH][HEIGHT]){
  for (int i = 0; i < HEIGHT; i++) {
    for (int j = 0; j <= WIDTH; j++) {
      float sum = 0;
      #pragma FPacc VAR=sum MaxAcc=8 MaxInput=4 epsilon=1E-15
      for (int w = 0; w < FILTER_SIZE; w++) {
        if(HEIGHT - i < FILTER_SIZE) break;
        for (int z = 0; z < FILTER_SIZE; z++) {
          if(WIDTH - j < FILTER_SIZE) break;
          sum += image[w + i][z + j] * filter[w][z];
        }
      }
      if(i < HEIGHT && j < WIDTH)
        result[i][j] = sum;
    }
  }
}
```

Table 3 shows the LUT usage, DPS usage, Latency and accuracy of the baseline code (Listing 14) and our generated code. Our LUT usage is a bit higher for a lower DSP usage. However the latency we acheive is almost divided by 2. The accuracy comparison is based on the lowest accuracy achieved given all the values of the output matrix. The exact result was obtained by computing the result using the MPFR library with 10000 bits of precision.

We have shown through this experiment that our tool provides good results in terms of area needed on chip, latency and accuracy for very little work. The use of a *pragma* provides a code that can be directly given to VivadoHLS for such improvements.

6 Conclusion

We have shown that HLS tools are able to generate efficient designs for handling floating point computations. They can achieve such results by using an application specific intermediate format. Hence, not having to follow the IEEE-754 standard and its limitations. We have been able to identify which `C` code is giving the best results in terms of latency, area needed on chip and accuracy. This proof of concept made us believe that merging application specific arithmetic with HLS was possible.

We also provided a tool that generates these operators in a given `C` program through the use of a *pragma* and some application specific informations. This information is domain knowledge such as the interval in which the variable lies. The main goal was to make the users access such optimization without writing VHDL nor having a large knowledge over the fixed point format. Our tool then transforms loop nests of summations to highly tune the floating point operators. The resulting code is an extension of the `C` language that is compatible with VivadoHLS.

This work focused on one single operation (the reduction), but there are a large number of specialized floating-point operators available in the literature. Just as compilers performs optimisations such as removing a multiplication by 1, we could be able to produce a multiplier by a constant when it is detected a compile time. This an example of operator *specialisation* that is not useful in a software compiler context. Therefore, it is specific to compiling to hardware. Other examples includes constant divisors/multipliers, squarers, etc. [8]. Also, software compilers tries to match patterns (that corresponds to operators) to a program DAG whereas we could create the operators given the patterns that we find. This an example of operator *fusion* that is not applicable in a software compiler context. The long term objective of this work is to explore these new opportunities.

There are a few short term objectives such as:

- We only provide support for VivadoHLS. Many HLS tools are available and a support to the most used one can also be a possible future work. As we want such arithmetic optimization to be used by the largest number of users, targeting a largest amount of platforms might help.

- Our accuracy comparisons were made using hand-written `MPFR` equivalent code. The results from MPFR where exact and it was to our mind, the best way to compare the quality of our results.

  However, this code is handwritten and we could implement a source-to-source GeCoS plug-in that transforms any HLS friendly `C` code to its `MPFR` equivalent. This would provide the user a full software that can compare the original accuracy of his program to both the transformed program and the exact result.

- In the spirit of ease of use offered to the programmer, further user interface optimization could be applied. In many cases, we might be able to know in advance the number of iterations of a loop nest. This could help us find a boundary of the accumulator in the case that the programmer only provides the maximum value of every entry. However, as this technique is not applicable in all cases, we could just notify through the use of a warning that the accumulator value might exceed the maximum value provided by the user.

Our approach has the advantage of being used in a very local way. We only optimize a floating-point operator that is used multiple times within a loop. The ID.Fix approach transforms the whole floating-point variables to a fixed-point format. Being able to transform a such a little part of the program and letting the user the ease of use of the floating-point format elsewhere seems to be a plus to us.

One of the most questionable side of our approach is that we used the `C` language to depict mathematical formulas. This makes us not following the C99 standard and thus it might not be what the programmer had in mind. The architecture of tools such as GeCoS might give us the opportunity to create our own domain-specific programming language. This language would evolve with the supported mathematical operators. It would ensure the programmer that his mathematical formula would be translated to hardware using state-of-the-art HLS tool and their optimization while embedding highly optimized operators.

We truly believe that high-level synthesis and arithmetic optimization can interleave. Hence give the programmer the ease of use and the application specific optimization of his operators.

References

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

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

[3] Simone Campanoni, Glenn Holloway, Gu-Yeon Wei, and David Brooks. Helix-up: Relaxing program semantics to unleash parallelization. In *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '15, pages 235–245, Washington, DC, USA, 2015. IEEE Computer Society.

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

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

[6] Florent De Dinechin and Laurent-Stéphane Didier. Table-based division by small integer constants. In *Applied Reconfigurable Computing*, pages 53–63, Hong Kong, Hong Kong SAR China, March 2012.

[7] Florent de Dinechin and Bogdan Pasca. Designing custom arithmetic data paths with FloPoCo. *Design Test of Computers, IEEE*, 28(4):18–27, July 2011.

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

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

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

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

[12] Laurent Fousse, Guillaume Hanrot, Vincent Lefèvre, Patrick Pélissier, and Paul Zimmermann. MPFR: A multiple-precision binary floating-point library with correct rounding. *ACM Transactions on Mathematical Software*, 33(2):13:1–13:15, June 2007.

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

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

[15] John Gustafson. The end of numerical error. In *22nd IEEE Symposium on Computer Arithmetic, ARITH 2015, Lyon, France, June 22-24, 2015*, page 74, 2015.

[16] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015.

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

[18] Xilinx Inc. Vivado design suite user guide high-level synthesis. 2015.

[19] James Jeffers and James Reinders. *Intel Xeon Phi Coprocessor High Performance Programming.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2013.

[20] Nachiket Kapre and Andre DeHon. Optimistic parallelization of floating-point accumulation. In *Proceedings of the 18th IEEE Symposium on Computer Arithmetic*, ARITH '07, pages 205–216, Washington, DC, USA, 2007. IEEE Computer Society.

[21] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems 25*, pages 1106–1114. 2012.

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

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

[24] Jean-Michel Muller, Nicolas Brisebarre, Florent de Dinechin, Claude-Pierre Jeannerod, Vincent Lefèvre, Guillaume Melquiond, Nathalie Revol, Damien Stehlé, and Serge Torres. *Handbook of Floating-Point Arithmetic*. Birkhäuser Boston, 2010. ACM G.1.0; G.1.2; G.4; B.2.0; B.2.4; F.2.1., ISBN 978-0-8176-4704-9.

[25] Xavier Redon and Paul Feautrier. Detection of scans in the polytope model. *Parallel Algorithms Appl.*, 15(3-4):229–263, 2000.

[26] Olivier Sentieys, Daniel Menard, David Novo, and Karthick Parashar. Automatic Fixed-Point Conversion: a Gateway to High-Level Power Optimization. Tutorial at IEEE/ACM Design Automation and Test in Europe (DATE), March 2014.