Profile-Guided Floating- to Fixed-Point Conversion for Hybrid FPGA-Processor Applications

DORIS CHEN and DESHANAND SINGH, Altera Corporation

The key to enabling widespread use of FPGAs for algorithm acceleration is to allow programmers to create efficient designs without the time-consuming hardware design process. Programmers are used to developing scientific and mathematical algorithms in high-level languages (C/C++) using floating point data types. Although easy to implement, the dynamic range provided by floating point is not necessary in many applications; more efficient implementations can be realized using fixed point arithmetic. While this topic has been studied previously [Han et al. 2006; Olson et al. 1999; Gaffar et al. 2004; Aamodt and Chow 1999], the degree of full automation has always been lacking. We present a novel design flow for cases where FPGAs are used to offload computations from a microprocessor. Our LLVM-based algorithm inserts value profiling code into an unmodified C/C++ application to guide its automatic conversion to fixed point. This allows for fast and accurate design space exploration on a host microprocessor before any accelerators are mapped to the FPGA. Through experimental results, we demonstrate that fixed-point conversion can yield resource savings of up to 2x–3x reductions. Embedded RAM usage is minimized, and 13%–22% higher F_{max} than the original floating-point implementation is observed. In a case study, we show that 17% reduction in logic and 24% reduction in register usage can be realized by using our algorithm in conjunction with a High-Level Synthesis (HLS) tool.

Categories and Subject Descriptors: B.2.2 [Arithmetic and Logic Structures]: Performance Analysis and Design Aids

General Terms: Design, Performance

Additional Key Words and Phrases: FPGA, fixed-point conversion, LLVM

ACM Reference Format:

Chen, D. and Singh, D. 2013. Profile-guided floating- to fixed-point conversion for hybrid FPGA-processor applications. *ACM Trans. Architect. Code Optim.* 9, 4, Article 43 (January 2013), 25 pages.
DOI = 10.1145/2400682.2400702 <http://doi.acm.org/10.1145/2400682.2400702>

1. INTRODUCTION

The use of Field Programmable Gate Arrays (FPGAs) for general-purpose algorithm acceleration remains a promising field. Numerous studies have shown that FPGAs have better performance (in speed, resource usage, or power consumption) when implementing algorithms such as image processing [Cope et al. 2005], HD-video [Fowers et al. 2012], FFTs, and linear algebra routines [Strickland and Langhammer 2008; Zhang et al. 2009; Johnson et al. 2008]. Unfortunately, FPGAs have not gained widespread acceptance in algorithm acceleration when compared to technologies such as multicore CPUs and GPUs.

There are two major reasons for this seeming discontinuity. First, there is an inherent mismatch between the capabilities of the FPGA and the traditional application

Authors' address: D. Chen (corresponding author) and D. Singh, Altera Corporation; email: dochen@altera.com.

Permission to make digital or hard copies of part or all 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 show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.

© 2013 ACM 1544-3566/2013/01-ART43 \$15.00

DOI 10.1145/2400682.2400702 <http://doi.acm.org/10.1145/2400682.2400702>

Fig. 1. High-level block diagram of an FPGA + processor system [Balough 2011].

model. FPGAs are not well suited to implement entire high-level applications. The large amounts of fine-grained parallelism available on the FPGA are better suited for implementing small, highly data parallel, and performance-critical computational kernels. The remainder of the sequential control and setup code is better implemented on a microprocessor. Both Altera and Xilinx have announced FPGA products with integrated ARM CPU cores [Balough 2011; Xilinx 2011] to handle applications of this nature as shown in Figure 1. Using these devices, programmers will have access to traditional processors as well as highly parallel FPGA hardware for acceleration. In addition, design flows targeting hybrid processor-FPGA environments such as Altera's OpenCL Compiler [Singh 2011] have also been announced to allow for design of algorithms at a higher level of abstraction than HDL.

A second reason limiting the use of FPGAs is the dominant use of floating point in algorithm development. Floating-point numbers can represent a larger range of real values for a given number of bits than fixed-point equivalents. The number is represented using a sign bit, a fixed number of significant digits, and an exponent which scales the significand as shown by the following equation.

$$(-1)^{sign} * significand * 2^{exp} \quad (1)$$

Floating-point representations greatly simplify programming because the developer does not have to track the location of the decimal point in the binary representation of a given value.

However, hardware implementations of floating-point operations are generally much bigger and more complex than implementations of fixed-point operations. The main difference is that floating-point numbers require additional complex logic to handle the computation of the exponent. In addition, the floating-point standard have special representations to indicate values such as infinities.

Although FPGAs have improved in implementing floating-point computations [de Dinechin et al. 2008; Langhammer 2011], even outperform GPUs in some

applications [Strickland and Langhammer 2008], their greatest strength is the ability to customize a datapath exactly to the needs of an application. For example, there are numerous applications where the dynamic range of floating point is unneeded, and can be more efficient when fixed point representations are used [Langhammer 2011]. Although fixed point is well understood in hardware design, it is incredibly complex to restructure an algorithm to use fixed-point types while producing a correct and efficient implementation that minimizes quantization error.

1.1. Proposed Solution

Given the two factors described before, we propose a fully automated floating-point to fixed-point conversion framework for applications written in C for a microprocessor coupled with an FPGA accelerator. The hybrid environment allows us to use techniques that are difficult or computationally inefficient when using only an FPGA. At a high level, our flow comprises the following steps.

- (1) *Parse* the C program in Clang [Lattner 2007] and apply IR-level optimizations in LLVM [Lattner and Adve 2004].
- (2) Automatically *insert profiling code* into the application to track the largest range of values experienced by each floating-point variable in the application. This of course assumes the existence of a representative test case. We use this collected data in combination with worst-case analysis to determine the range of representable values.
- (3) *Run* application test cases to obtain profiling information. In this step, we assume all profiling information is obtained by running the test case on the processor alone. This avoids the need to synthesize, or place and route any of the profiling code onto the FPGA itself.
- (4) *Convert Floating-Point data types to Fixed-Point* using LLVM IR transformations. The basis for the conversion algorithm is described by Cilio and Corporal [1999].
- (5) *Evaluate Error* in the result by running the fixed point application on the CPU. Although many previous works show theoretical models for error in simple cases, our approach *transforms* the application so the actual error can be experimentally determined. This is especially important in cases like loop accumulation where running sums may accumulate quantization error.
- (6) *Convert the fixed-point datapaths to HDL*. Our CAD tool automatically builds pipelined accelerators for regions of logic converted to fixed point. Our focus in this work is to examine the logic savings of these accelerator blocks in comparison to full floating-point implementations. We do not cover the interface between processors and accelerator in this article.

In contrast with previous works [Han et al. 2006; Olson et al. 1999; Gaffar et al. 2004; Brown et al. 2008; Aamodt and Chow 1999], we believe this article makes the following unique contributions.

- This is the first article which speaks about compiler-level floating-point transformations to the Intermediate Representation (IR). The LLVM-based framework allows us to profile and transform code, and retarget this IR to different, but functionally equivalent backends of a microprocessor and an FPGA.
- We are targeting the area of algorithm acceleration where the FPGA is coupled with a processor. This implies that there are often format conversions to and from floating point which need to be optimized. We have not seen this trade-off described previously.
- Because LLVM IR is target independent, we can easily instrument and optimize the characteristics of the application on a processor before we target the FPGA.

Fig. 2. Altera stratix v FPGA architecture [Verma and McElheny 2012].

- Our flow does not attempt to transform the entire application to fixed point. Since some floating-point operations may be more efficient than others on an FPGA, we use a high-level resource estimation and error model to decide which portions of the application should be converted.
- We present comparisons to modern, highly optimized floating-point cores and show detailed trade-offs and application characteristics that benefit from this flow.
- We perform a case study to evaluate our algorithm within a High-Level Synthesis (HLS) tool targeting FPGAs.

The rest of this article is organized as follows. First, we review relevant background and contrast with previous work. Next, we describe our LLVM-based compilation framework for automatic fixed-point conversion of floating-point applications. Using this framework, we evaluate the resource usage, speed, and Signal-to-Noise Ratio (SNR) performance of several benchmarks implemented on a Stratix IV-530 FPGA. Finally, our case study of automatic fixed-point conversion within HLS tools is presented.

2. BACKGROUND

In this section, we will briefly describe the complexity of the FPGA design flow, and the importance of fixed-point conversion in FPGA design. A detailed background of the fixed-point conversion process will be provided.

2.1. Microprocessors vs. FPGAs

Traditional microprocessor-based systems are highly efficient at performing floating-point operations. They contain a fixed datapath, with highly optimized functional units. These floating-point operations are often executed serially on each processor core, and can incur high runtimes. However, microprocessors are easy to program and are very easy for initial prototyping of algorithms.

In contrast, the FPGA is a programmable computational fabric with a regular grid-like architecture, as seen in Figure 2. The primary building block on the FPGA is the Logic Element (LE) surrounded by a configurable routing network. Each LE can implement a Boolean logic function of up to k inputs. The FPGA also contains customizable RAM and DSP blocks, typically arranged in columns on the device. Designers can customize the datapath exactly as needed, and replicate the datapath as the device would

Fig. 3. Bit format for (a) floating-point numbers; and (b) fixed-point numbers.

allow. Thus, the FPGA can parallelize execution of floating-point operations to achieve speedup versus traditional microprocessors.

However, the resources necessary to implement floating-point operations on an FPGA may be substantial. Although DSP blocks can perform floating-point operations efficiently, they are limited in quantity and constrained in location, and can limit the number of floating-point operations that can be implemented on one device.

The FPGA also has a different programming model than microprocessors. Designs targeted at FPGAs are described using Hardware Description Languages (HDL) such as VHDL or Verilog. These languages are meant to be used to describe digital circuits, rather than algorithms. These designs are then compiled through FPGA CAD tools to efficiently map the design onto the fabric. As FPGA devices grow in size and functionality, the compilation process can be quite lengthy, and detrimental for prototyping.

2.2. Floating Point to Fixed Point

One way of reducing resource usage on FPGAs for floating-point operations is to convert floating-point numbers to fixed-point representations. Traditional integer operations can then be performed on these fixed-point representations with a much lower resource requirement. An excellent overview of the conversion process can be found in Cilio and Corporaal [1999]. A fixed-point number consists of three parts: a *sign bit*, the *integer portion*, and the *fractional portion* as shown in Figure 3(b).

The number of bits representing each part, denoted by *IWL* (Integer Word Length) and *FWL* (Fractional Word Length) can vary depending on precision requirements. The total number of bits needed to represent the entire word is given by

$$WL = 1 + IWL + FWL. \quad (2)$$

The real value, v , represented in fixed point, is given by

$$v = \left(-sign * 2^{(WL-1)} + \sum_{i=0}^{WL-2} bit_i * 2^i \right) * 2^{FWL}. \quad (3)$$

The process of converting a floating-point operation to fixed-point is as follows:

- (1) Scale the floating-point number to a representation where the lower *FWL* bits represent fractional values by multiplying the number by 2^{FWL} .
- (2) Type conversion from floating point to integer, which shifts the floating-point significand bits such that the exponent is 0.
- (3) Perform the actual operation using integer units instead of floating-point units.
- (4) Convert the fixed-point integer type back to floating-point by shifting the bits such that the bit representation of the number is normalized, and create the floating-point exponent.
- (5) Perform inverse scaling of the floating-point number by dividing by 2^{FWL} .

An example of how fixed-point conversion works is shown in Figure 4. To add the two floating-point numbers 2.5 and 3.125 together, each must first be converted to

Fig. 4. Conversion example: (a) original floating-point implementation; (b) fixed-point implementation.

fixed-point format. Ignoring the bias in the exponent, and showing the implicit 1, these two numbers are shown in binary as 1.01 and 1.1001 with exponents of 1, respectively. These numbers are multiplied by 2^{FWL} , where FWL is 5 for this example, and then converted to integer format. An integer addition is performed, and the result is converted to floating-point format by normalization. We divide by 2^{FWL} to obtain the final floating-point result.

Automatic fixed-point conversion is not trivial and can be complicated. First, the alignment of the decimal point during computation is crucial for correct results. For example, when we add two integer numbers, both operands must have the same IWL and FWL for correctness. Also, since we only have IWL bits to represent the integer portion of the number, it is easy to overflow as the result gets larger. Therefore it is necessary to explicitly keep track of the IWL bits required at every stage of the conversion. To accommodate these issues, the following rules are used for fixed-point addition and multiplication.

—*Addition/Subtraction*: Given the addition of two fixed-point numbers of IWL_a and IWL_b , the IWL_c of the resulting sum is

$$IWL_c = \max(IWL_a, IWL_b) + 1. \quad (4)$$

Note that the $+1$ term is different than the formulation given in Cilio and Corporaal [1999] as we have found this is necessary to avoid overflow.

—*Multiplication*: The IWL_c of the product of two signed fixed-point numbers is

$$IWL_c = IWL_a + IWL_b + 1. \quad (5)$$

The additional logic required on the FPGA include a Floating-Point (FP) multiplier and divide for scaling operations, and barrel shifters for type conversions. This increase in logic may overwhelm any FPGA logic area savings from substituting the FP unit with an integer unit. However, we can amortize this area overhead if multiple FP operations are performed in sequence. Consider Figure 5(a), where we have 2 FP adders followed by an FP multiplier. There is no need to convert the output of the adders before the data is consumed by the multiplier. Therefore, we only need to perform conversion on the inputs to adders, and outputs of the multiply. The huge cones of FP operations in many computationally intensive designs benefit from this, leading to substantial area

Fig. 5. (a) Original floating-point design; and (b) Fixed-Point Implementation of an FP cluster.

savings. In addition, the multipliers and divides needed for scaling are very efficient to implement on the FPGA. Because scaling is by powers of 2, only the exponent of the floating-point representation needs to be changed.

2.3. Previous Work

Several previous works attempt to optimize floating-point designs by using fixed point. In Gaffar et al. [2004], the authors propose a C++ library with specially defined types. The programmer is required to change here source-code to use the library types rather than standard *float* and *double* data types. Operator overloading is then used to construct an internal representation of the expression to be evaluated. The internal graph is then used to create a Xilinx System Generator project. The authors do not describe the fixed-point conversion process as it seems to be left to the Xilinx backend to handle. In addition, the tool is driven by error sensitivity metrics which require differentiable computations and operating points. This adds complexity to the design flow. In contrast, our flow requires no source-code changes. The use of automatic profiling in our flow also reduces design flow complexity.

In Olson et al. [1999] and Han et al. [2006], the authors describe a Matlab-based environment for converting floating-point to fixed realizations. This is more suitable for DSP signal processing than general algorithm acceleration. The authors specifically study the optimization of the word length [Han et al. 2006] using an evolutionary algorithm, but do not detail the trade-offs in comparison to an optimized floating-point FPGA implementation.

The work in Brown et al. [2008] uses the Valgrind profiler to gather information about the range of floating-point values on x86 binaries. Valgrind is a x86-to-x86 JIT translation tool which allows instrumentation applications to be built. This information is used to reduce the size of shifters needed to implement floating-point operations on FPGAs. The main concern with this approach is that the captured values are from a binary executable which is highly optimized for an x86 target. This executable may include many code transformations. It is difficult to match the final assembly code back to the initial unoptimized compiler-intermediate representation. There is no automation shown to facilitate this process. Although this work shows good results for select benchmarks, the comparison made is to a parameterizable suite of floating-point cores [Belanovic and Leeser 2002]. This library is designed to allow research on varying floating-point bit widths; however, comparisons are not made against floating-point cores which are optimized for standard mantissa and exponent sizes defined by IEEE-754.

Finally, a highly relevant work is the Fused Datapath Compiler [Langhammer 2011]. Although the output of a design must be a normalized floating-point number by IEEE

Fig. 6. General FPGA usage scenario.

standards, the intermediate results do not need to be normalized. The Fused Datapath Compiler strips out the logic required by the normalization step from all FP operations, and only implements the normalization logic after all FP operations are completed. Since normalization is only performed once per contiguous block of FP operations, a reduction of 46.5% in logic (ALUTs) and 27.8% in register count were achieved. Our work looks at a further extreme where full fixed-point operations are used.

3. AUTOMATIC FIXED-POINT CONVERSION

FPGAs are often used in data streaming systems. In such systems, the FPGA receives data from external devices and performs computations on-chip. These results are then sent out over external I/O interfaces as shown in Figure 6. Many computational problems often have large datasets that must be processed and FPGA processing can be applied in a similar fashion by streaming the data from external memory banks into the FPGA fabric. The FPGA processes the data on-chip, and streams the result back to external memory for storage. The data streamed into the FPGA is often represented in a fixed point format, such as video streams containing values representing light intensity. In many cases, the application developer converts this input data into floating-point format to simplify development effort. When the floating-point algorithm is finalized, manual conversion to fixed point can be performed to create a smaller and faster implementation using the FPGA logic fabric. It is this very scenario that we address in this work. We provide a methodology for automatically converting to a fixed-point representation from high-level code written in C.

3.1. LLVM Framework

LLVM is an open-source project that provides modular and reusable components for creating compilation tools. The LLVM infrastructure is based upon a strongly typed virtual instruction set that is both target and language independent. The intermediate representation (IR) of a program can be restructured in numerous ways. For example, consider the source-code for a simple dot-product function in Algorithm 1.

ALGORITHM 1: Dot-Product Computation

Input: float *a, float *b

Output: sum

sum = 0.0f

#pragma unroll

for int i=0; i < 2; i++ **do**

 sum += a[i] * b[i]

end

printf("dot prod = %f", sum);

First, this program is parsed by the Clang frontend which understands the syntactic and semantic meaning of the C program. After initial conversion to the LLVM virtual instruction set, a number of passes progressively optimize the IR. These include standard compiler optimizations such as dead code elimination, constant propagation, and removal of redundant instructions. After these passes, the original program is transformed into an LLVM IR shown in Algorithm 2.

ALGORITHM 2: Dot-Product IR.

```
define void @dotp(float* %a, float* %b) {
entry:
  %tmp8 = load float* %b, align 4
  %tmp3 = load float* %a, align 4
  %mul = fmul float %tmp3, %tmp8
  %add = fadd float %mul, 0.000000e+00
  %arrayidx.1 = getelementptr float* %a, i64 1
  %arrayidx7.1 = getelementptr float* %b, i64 1
  %tmp3.1 = load float* %arrayidx.1, align 4
  %tmp8.1 = load float* %arrayidx7.1, align 4
  %mul.1 = fmul float %tmp3.1, %tmp8.1
  %add.1 = fadd float %add, %mul.1
  %phitmp = fpext float %add.1 to double
  %1 = call @printf(..., double %phitmp)
  ret void
}
```

In this example, the loop has been completely unrolled. The resulting program has 4 *load* instructions, 2 *floating-point multiplies*, and 2 *floating-point additions*. The transformations we present work directly on this representation of the program.

The overall tool flow of automatic fixed-point conversion, shown in Figure 7, is implemented as a pass in LLVM. For each basic block in the LLVM IR,¹ we identify the *conversion set*, which is a set of Floating-Point (FP) operations to convert to fixed point. The process of selecting the set of operations to convert is described in Section 3.2. The floating-point inputs of the conversion set are converted to signed-integer formats denoted by the *fp_{to}si* step to prepare for fixed-point computation. Next, we create a fixed-point integer operator for every FP operation of the conversion set. This is discussed in Section 3.3.

After all fixed-point implementations have been created, the output must be converted back to floating point in the *si_{to}fp* step. We can then replace the original outputs with the newly created outputs from fixed-point implementations. A sweep operation will then remove all the unused original instructions from the IR. The modified IR can now be used to generate code to run on the CPU, or to generate RTL code for execution on the FPGA (described in Section 3.6).

3.2. Selection of Conversion Set

Before any FP operations can be converted to fixed point, we iterate through all instructions of the basic block to identify FP clusters. First, we denote I_{ma} as an FP add or multiply operation. We then define a *FP cluster C* as a collection of floating-point multiply, add, compare, and select² instructions (I) that are connected components in

¹A basic block is a straight line portion of code where branches are only permitted as the last instruction.

²A multiplexer instruction used to implement conditional operations.

Fig. 7. Automatic fixed-point conversion tool flow.

the DAG represented by the LLVM IR for the basic block (BB) under consideration.

$$I_{ma} = \{I \mid I \in BB \wedge (isFMul(I) \vee isFAdd(I))\} \quad (6)$$

$$C = I_{ma} \cup \{I \mid I \in BB \wedge (isSel(I) \vee isFCmp(I))\} \quad (7)$$

Each FP cluster maintains a list (Inp) of cluster inputs and a list (Out) of cluster outputs. Cluster inputs are instructions that produce values used by operations within the cluster. Cluster outputs are instructions within the FP cluster that produce values that are used by at least 1 instruction outside the cluster. Thus, these outputs must be visible outside of the FP cluster.

$$Inp = \{I \mid I \in Program \wedge I \notin C \wedge I \in Fanin(C)\} \quad (8)$$

$$Out = \{I \mid I \in C \wedge \neg(Fanout(I) \subseteq C)\} \quad (9)$$

The pseudocode for the cluster selection is shown in Algorithm 3.

As we look through the IR for FP operations, we also keep track of inputs and outputs to the FP operations. If an instruction is an FP operation, but its operands are not, then the operand instructions are added to the set of FP cluster inputs. If an instruction is not an FP operation or a select or compare instruction, but its operands are, then this instruction is added to the set of FP cluster outputs. This is so that we can identify the points where we need to convert between floating point and fixed point.

The initial FP cluster set is the maximum set of instructions we can convert to fixed point. However, it may not be to our advantage to convert all possibilities to fixed point. Fixed-point conversion incurs resource overhead as shown in Figure 5. If this overhead is greater than the logic savings of using an integer operator, then there is no reason to convert these operations. To determine the best logic savings, we create a resource

ALGORITHM 3: Selection of FP Clusters

```

for each instruction  $I$  in the Basic Block  $BB$  do
  if  $IsFloatingPointOperator(I)$  then
     $FPcluster.insert(I)$ 
    for each operand  $op$  of  $I$  do
      if not  $IsFloatingPointOperator(op)$  then
        add  $op$  to FP cluster Input set
      end
    end
  end
else
  if any operand  $op$  of  $I$   $IsFloatingPointOperator(op)$  then
    if  $I$  is a FP Select or Compare Block then
       $FPcluster.insert(I)$ 
    end
  else
    //  $I$  is not FP
    add  $I$  to FP Cluster Output set
  end
end
end

```

estimation model.

$$C_{mul} = \{I \mid I \in C \wedge isFMul(I)\} \quad (10)$$

$$C_{add} = \{I \mid I \in C \wedge isAdd(I)\} \quad (11)$$

$$C_{sel} = \{I \mid I \in C \wedge isSel(I)\} \quad (12)$$

$$C_{cmp} = \{I \mid I \in C \wedge isFCmp(I)\} \quad (13)$$

C_{mul} , C_{add} , C_{sel} , and C_{cmp} denote the floating-point multiply, add, select, and compare instructions in the cluster. The resource usage (A) after conversion to fixed point can be estimated by the following.

$$\begin{aligned}
 A = & |C_{mul}| * ResourceUsed(FixedPtMul) \\
 & + |C_{add}| * ResourceUsed(FixedPtAdd) \\
 & + |C_{sel}| * ResourceUsed(FixedPtSel) \\
 & + |C_{cmp}| * ResourceUsed(FixedPtCmp) \\
 & + |Inp| * ResourceUsed(fptosi) \\
 & + |Out| * ResourceUsed(sitofp)
 \end{aligned} \quad (14)$$

This equation assumes the existence of a function *ResourceUsed* which returns the logic resource usage of each operation. This resource usage includes both LEs used as well as the number of DSP blocks needed for each operation, since both metrics can be the limiting factor. This function is easily obtainable by running a few small sample circuits through the FPGA CAD tool to derive the characteristics of different floating-point and fixed-point cores. We can then run the resource estimation model with various scenarios, and pick the configuration that will yield the greatest resource savings. For example, the resource savings from converting FP multiplies to fixed point may not be substantial enough to offset the logic required to implement *fptosi* and

Fig. 8. Original IR graph.

sitofp conversions. However, we may still yield resource savings if the arithmetic tree is deep enough.

3.3. Fixed-Point Conversion

After the conversion set is selected, the inputs to this set are converted to fixed-point formats as shown in Figure 5. This is performed by first multiplying the input by 2^{FWL} , and the LLVM *fptosi* instruction for type conversion.

Unlike floating-point operations, we must explicitly keep track of the *IWL* of every instruction to ensure functional correctness of the output. To do so, we compute the output *IWL* for every instruction within the set using Eqs. (4) and (5). When we create fixed-point implementations for each FP operation, we need to make sure the output *IWL*s are consistent with the original *IWL*s. If the *IWL*s are not consistent, we must insert right shift (LLVM *ashr*) operators in the IR. For integer multiplication, the output data width is twice the input data width. We need to first right shift the result by *WL*, and then truncate (LLVM *trunc*) the result back to *WL*. After all fixed-point units are created, the conversion back to floating point is done by inserting a LLVM *sitofp* instruction to convert the signed integer back to floating point, and a *fdiv* instruction to divide by 2^{FWL} . This procedure is illustrated by the sample C program code of matrix multiplication shown in Algorithm 4.

ALGORITHM 4: Matrix Multiplication

Input: float *a, float *b

Output: float *c

```

for int row=0; row < 3; row++ do
  for int col=0; col < 3; col++ do
    float sum = 0.0f;
    #pragma unroll
    for int i=0; i < 3; i++ do
      sum += a[row*3+i] * b[i*3+col];
    end
    c[row*3+col] = sum;
  end
end

```

The original FP cluster identified from the C program is shown in Figure 8. The blue rectangles indicate the inputs and outputs of the conversion set.

Figure 10 shows the resulting transformation after running our automatic conversion tool. This figure is simply a graphical representation of the LLVM IR. First, each

Fig. 9. Balanced IR graph.

Fig. 10. IR graph after fixed-point conversion.

input is multiplied with a constant value (2^{FWL}) through the *fcmul*³ block. It is then casted to a signed integer by the *(int)(..)* block. For an integer multiplier denoted by *imul*, we must first truncate its inputs to *WL* bits, before sign-extending it to $2WL$ bits. This is to allow proper computation. The $2WL$ output is then right shifted by *WL* bits. For an integer adder (*iadd*), its operands must have the same *IWL*. Its *IWL*

³LLVM does not have special multiply-by-constant instructions, but we display *fcmul* to indicate that it is efficiently implemented in HW.

Fig. 11. Aligning decimal point in fixed-point computation.

also needs to increase by 1 to avoid overflow. Thus, shift operations are performed on the operands before computation. After all the integer operations, the result is converted back to float via the casting instruction (*float(...)*) which is followed by a divide operation by 2^{FWL} with the *fcdiv* block.

Note that in Figure 10, there are a number of operations that shift data to the right (denoted by $\gg k$). The reason why we need these shift operations is explained in greater detail in Figure 11. Suppose two numbers with different IWL and FWL are to be added together. If added as is, the result will be incorrect. These right shift operations are necessary to align the decimal point of the number formats. Note that the IWL of the sum has increased by 1 according to Eq. (4).

As FP units cascade, the output IWL can increase very quickly. If the required IWL exceeds the total bit width (*WL*), we may not be able to perform fixed-point conversion on the entire floating-point cluster due to the loss of precision. To counteract this, our tool allows for arbitrarily large integer widths (up to 64 bits) in the fixed-point implementation.

3.4. Arithmetic Tree Balancing

The maximum number of operators in any given path in Figure 8 is 4. This is an example of a vine-like code structure that can unnecessarily increase the maximum IWL required. If the code is restructured such that Figure 9 is achieved, we can reduce max path in the circuit to 3. This both reduces the latency in the system, and gives better SNR since our worst-case IWL is reduced.

To do this, we created another LLVM compiler pass which attempts to balance arithmetic trees by taking advantage of *commutative* property of multiplies and adds. The goal of the balancing is to minimize the worst-case IWL. Note that this type of optimization is not performed by LLVM automatically since IEEE-754-compliant FP operators are not commutative due to possible differences in the least significant digits. However, our fixed-point conversion pass already introduces some quantization noise so any changes due to commutativity are quite negligible.

3.5. Profile-Guided Optimizations

So far in the IWL computation, we have assumed a *default* IWL value is selected for FP cluster inputs. However, static selection can be dangerous if data values exceed the representable range of the IWL. Consider a floating-point number, represented by *sign*, *mantissa*, and *exp* fields, which is to be converted into fixed-point representation

with IWL integer bits. First we create an intermediate bit vector⁴ of length WL .

$$B = \{0, 1, mantissa[22 : 0], (WL - 25)\{0\}\} \quad (15)$$

This bit vector can then be shifted to provide IWL integer bits using the following.

$$Fixed = B \gg (IWL - 1 - (exp - bias)) \quad (16)$$

Finally the *sign* bit must be applied where the two's complement operator is applied to *Fixed* if the *sign* bit is 1. Note, that the floating-point value will overflow IWL bits if the value of $(IWL - 1 - (exp - bias))$ is less than 0. Our *fptosi* core checks this condition and produces a signal which indicates that the floating-point value is outside the expected range. Our general approach is to signal an interrupt to the microprocessor whenever such an exception should occur. The processor can instead run this portion of code in software to ensure functional correctness. However, we would like to provide automation so that this situation is unlikely to ever occur. Thus we provide a methodology, based on profiling, for guiding the selection of the fixed-point parameters for the cluster inputs.

It is extremely useful if some sample data can be obtained so that the minimum and maximum data ranges can be determined. We can then choose the IWL such that the entire range can be correctly represented. This does not guarantee functional correctness since it is possible that data can fall outside this range. However, as discussed earlier, we can detect these situations. By using representative sample data, we minimize the likelihood of data outside this representable range. To accomplish this, we have created another LLVM pass which inserts value profiling instructions directly into the IR. This is best illustrated using our initial *dot-product* example. After our value profiling pass is run, the IR, shown in Algorithm 5, contains the following instructions.

Our pass first inserts a global array named *OptFloatRanges* which has two entries per floating-point variable in the original program, to track the minimum value and maximum values seen. The IR has code inserted to track the minimum and maximum values of each variable using *fcmp* and *select* instructions. The IR in Algorithm 5 shows the instructions required to track the value of the *%tmp8* variable. Before the program exits, we have inserted code (not shown) to dump the value of the *OptFloatRanges* array to a file. These values can be used by our fixed-point conversion pass to determine appropriate IWL s for the FP cluster inputs. A value of the IWL which leads to representable fixed-point values that completely enclose the $[min, max]$ range is selected.

The method of profiling value ranges is similar to traditional compiler optimizations involving feedback-directed compilation [Smith 2000]. These techniques use program transformations to attempt to optimize the runtime of the most common use cases for a particular program. Representative data inputs must be used to allow the compiler to select appropriate transformations. If the dataset is not representative of the true workload, then the program will still work correctly but may function slower than predicted.

In a similar manner, the use of value profiling allows for the creation of a highly efficient fixed-point accelerator on the FPGA. If the values of the true datasets fall outside the range that is able to be represented by the accelerator circuit, then this situation is detected and the computation is performed in software. In this case, the application will not be able to take advantage of the capabilities of the FPGA for acceleration.

The use of representative benchmark examples and experimental evaluation of error characteristics is common in the fields of audio, video, and image processing [Winkler

⁴The notation $(WL - 25)\{0\}$ indicates $(WL - 25)$ consecutive zero bits.

ALGORITHM 5: Value Profiling

```

define void @dotp(float* %a, float* %b) {
entry:
    %tmp8 = load float* %b, align 4
    %tmp3 = load float* %a, align 4
    %mul = fmul float %tmp3, %tmp8
    %add = fadd float %mul, 0.000000e+00
    %arrayidx.1 = getelementptr float* %a, i64 1
    %arrayidx7.1 = getelementptr float* %b, i64 1
    %tmp3.1 = load float* %arrayidx.1, align 4
    %tmp8.1 = load float* %arrayidx7.1, align 4
    %mul.1 = fmul float %tmp3.1, %tmp8.1
    %add.1 = fadd float %add, %mul.1
    %phitmp = fpext float %add.1 to double
    %1 = call @printf(..., double %phitmp)
; Track the MIN for %tmp8
    %minval = load float* @OptFloatRanges, ...
    %compare.min = fcmp olt float %tmp8, %minval
    %sel.min = select i1 %compare.min, %tmp8, %minval
    store float %sel.min, float* @OptFloatRanges, ...
; Track the MAX for %tmp8
    %maxval = load float* @OptFloatRanges, ...
    %compare.max = fcmp ogt float %tmp8, %maxval
    %sel.max = select i1 %compare.max, %tmp8, %maxval
    store float %sel.max, float* @OptFloatRanges, ...
; ...
    ret void
}

```

2012; Lin et al. 2011; Winkler and Mohandas 2008]. In these areas, a certain amount of error in the outputs is acceptable as these small differences can often be unnoticed by a human observer. In many cases, these algorithms are highly nonlinear; experimental evaluation of error bounds is the only option available. We believe that use of profiling and automated floating- to fixed-point conversion is practical for these scenarios. In other cases, where representative data is not possible to obtain and quantitative evaluation of error cannot be performed, then this style of automation may not apply.

3.6. LLVM FPGA Backend

Once the application has been transformed to a fixed-point representation in the LLVM IR, we need to map the clusters of operations to a datapath circuit implemented on the FPGA. In this work, we use a very simple approach where we create pipelined circuits which aim to produce 1 result per clock cycle. There are two reasons for this. First, we focus on computations with large datasets where the latency of the pipeline (on the order of hundreds of cycles) is small in comparison to the quantity of data that must be processed, which can be on the order of millions of data points. Secondly, optimizing for latency is critical in applications that involve processing data from FPGA I/O, such as ethernet packet processing. Our techniques also help here because they reduce the number of register stages required to maintain a high operating frequency.

To achieve this, we translate the LLVM IR directly into a structural Verilog representation. Each operation such as *fadd* instantiates a structural block which implements the FP add function on the FPGA. These blocks communicate with each other using a handshaking mechanism of Synchronous Interlocked Pipelines [Jacobson et al. 2002]. An example of this is shown in Figure 12. To generate the actual hardware

Fig. 12. Synchronous interlocked pipelining scheme using valid and stall signals.

Fig. 13. Inserting register delays to balance the pipeline.

implementation of the LLVM netlist, we implement each edge in the graph with 3 components: data, valid, and stall. The *data* edge is the operand value. A *valid* bit signals the data consumer when the data is valid. A *stall* signal is sent back from the consumer to the data producer if it cannot process the incoming data. When the *stall* signal is asserted, the stall propagates up the chain of blocks, thus halting the pipeline.

To ensure the resulting datapath can run free of stalls in the steady state, we adopt a simple ASAP (As Soon As Possible) clock cycle scheduling algorithm as shown in Figure 13. Since every operation has an associated latency, we can compute the minimum allowable latency required at every stage. For example, in our framework, an *fmul* has a latency of 5 cycles, and an *fadd* has a latency of 12 cycles. The *fmul* block is implemented using hard Digital Signal Processing (DSP) blocks, which are very area efficient. In contrast, the *fadd* block is implemented entirely in soft logic. To ensure a high operating frequency, circuits consisting of soft logic (FPGA logic elements) must be pipelined to a greater degree than those dominated by hard blocks. This leads to a significant difference in the latency of the floating-point adder and multiplier. Given these latencies, we can compute the minimum number of cycles before data is available at the output of a given block. This is shown on the right of the graph on Figure 13. For this example, it takes 41 cycles to generate the result at the output. However, because paths through the graph do not have equal latencies, data may get to a block before it can be consumed. Thus, registers are inserted to add delay to some paths to ensure that the pipeline latencies are completely balanced. In this example, a 12-stage and a 24-stage register path are inserted into the graph to fully balance this pipeline. If the number of register stages is large, we automatically translate these into shift register implementations using block RAMs.

Fig. 14. Effect of total bit width on SNR.

4. RESULTS

To evaluate our automatic fixed-point conversion tool, we used a set of 6 benchmarks written in C. This set represents standard floating-point applications commonly used to evaluate algorithm accelerator devices. These designs include the following.

- 8-point Discrete Cosine Transform (dct)
- 16×16 Matrix Multiply (mm16)
- 32-tap Finite Impulse Response (FIR) filter (fir)
- 16-point Fast Fourier Transform (FFT) (fft)
- Sum of Absolute Differences (sad)
- Matrix Inversion Algorithm (cholesky)

The data inputs for these benchmarks are randomly generated single-precision floating-point values. After running through the default LLVM flow targeting a processor, the output can be saved as a golden reference.

Next, we run these benchmarks using our automatic fixed-point conversion tool, which instruments the code to analyze the range of the input data. While still targeting the processor backend, we quickly obtain results for different fixed-point bit widths, ranging from 16 bits to 40 bits. We can then compare the SNR ratio of our tool at different fixed-point widths as shown in Figure 14. As expected, the lower integer widths show more error in the output results, ranging from 0dB (sad) to 34dB (mm16). As integer widths increase, the SNR improves until plateauing at approximately 70dB. For most benchmarks, a total bit width of 32 is sufficient enough to achieve SNR above 60dB. However, for some designs that have more levels in the circuit, total bit widths of 36 is required. The trade-off of choosing the best number of total bits to satisfy an error criteria is guided by user trials running their application on the processor.

Once the appropriate bit width is chosen, the FPGA backend can generate Verilog accelerator blocks to implement customized datapaths for the clusters. To demonstrate the trade-offs, we generated HDL representations of all benchmarks for all bit widths tested. Verilog accelerators were created both before and after fixed-point conversion. The generated designs are compiled through Altera's Quartus II v11.0 [Altera 2010] to synthesize and place and route each benchmark on a Stratix IV-530 device. From Quartus II, we can extract the total logic and registers used to implement each benchmark and the maximum operating frequency (F_{max}) of the final placed and routed circuit.

Table I. Baseline Floating-Point Results

Design	ALUTs	Reg	RAM	DSP	F_{max} (MHz)
dct	19,623	30,027	16,384	112	214
fir	21,671	29,013	4,096	128	223
fft	42,515	65,959	18,432	40	201
mm16	10,861	14,492	2,048	64	242
sad	22,996	35,472	2,048	64	218
cholesky	6,880	11,226	0	40	254

Fig. 15. Logic utilization of automatically converted fixed-point benchmark circuits.

The results of the original floating-point implementation are shown in Table I. These results are created using Altera's highly optimized floating-point IP cores and serve as our point of reference.

The logic utilization of each benchmark is shown in Figure 15, normalized to the number of combinational ALUTs used by the original floating-point core. From the graph, we can see that the fixed-point implementation of each benchmark substantially reduces the amount of logic required. On average, the fixed-point version is 38% (at 16 bit) to 63% the size of the original circuit (1.6x–2.6x times smaller).

Figure 16 shows the total number of registers required by each benchmark, also normalized by the original floating-point core. We see great reductions in the number of registers used, ranging from 33% to 50% of the original register count (2x–3x smaller). In both logic and register usage, the 16-bit core is the smallest, and the resource usage increases linearly with respect to the total bit width.

Logic savings are heavily dependent on the topology of the benchmark. Designs with huge cones of FP units (fft) achieve the best resource savings. We see the least reduction for designs with many FP cluster inputs, but without a deep cone of FP units (cholesky).

Our conversion tool does not change the number of DSP blocks used because DSPs in 36×36 multiply mode are used for *imuls*. On the other hand, the RAM bits used in the original core were completely unnecessary in the fixed-point version. This is due to the dramatically reduced latency of the fixed-point cores (1 cycle) in comparison to the floating-point cores (5–12 cycles). In many cases, the pipeline balancing registers are so long in the floating-point case that it is necessary to implement them using embedded RAMs.

Fig. 16. Register usage in automatically converted fixed-point benchmark circuits.

Fig. 17. F_{max} of automatically converted fixed-point benchmark circuits.

Since fixed-point circuits use fewer logic resources, they present as simpler placement and routing problems. Thus, a higher operating frequency should be achievable. To illustrate, we compare the F_{max} of the fixed-point benchmark with the original design in Figure 17. To minimize the variation in placement due to the CAD tool, a sweep of 10 seeds was run, and the F_{max} of the 10 compiles are geometrically averaged. The F_{max} of each fixed-point circuit is then normalized by the F_{max} of the original floating-point circuit. In most cases, we see that the F_{max} of the fixed-point circuit is higher than the original circuit. On average, the F_{max} of the fixed-point circuit is 13%–22% higher than the original core. There is no clear relationship between F_{max} and the fixed-point bit width; there seems to be a general downward trend in F_{max} as total bit width increases. The slight dip in F_{max} at bit widths of 32 may be the result of a CAD anomaly.

Although we show good results from the presented techniques, the limiting factor in achieving greater gains are the conversion functions. For each input and output of the FP cluster, 220 ALUTs and 186 ALUTs are required for the *fptosi* and the *sitofp* instructions, respectively. For a number of applications, the programmer may

Fig. 18. OpenCL programming model.

use casting operations in C to convert from integer types to floating-point types. In the future, we plan to extend our clustering algorithm to go traverse backwards to these points so that the integer-to-float and the subsequent float-to-integer conversions cancel.

5. CASE STUDY: AUTOMATIC FIXED-POINT CONVERSION WITH HLS

Thus far in our discussion of automatic fixed-point conversion, we have focused the analysis on standard benchmark circuits. We automatically convert these floating-point benchmarks to fixed point, and compare its performance against the original implementation. To further illustrate the significance of our technique, we present a case study where automatic fixed-point conversion is used in conjunction with a High-Level Synthesis (HLS) tool targeting FPGAs. First, the HLS tool, Altera's OpenCL Compiler, will be presented. We then discuss the results of using automatic fixed-point conversion within this tool.

5.1. OpenCL and Altera's OpenCL Compiler (ACL)

OpenCL is a platform-independent standard where data parallelism is explicitly specified. It is based on C and contains extensions to specify parallelism and memory hierarchy. The application, shown in Figure 18, is composed of two sections: the host program and the kernel. The host program is the serial portion of the application and is responsible for managing data and controlling the overall flow of the algorithm. The kernel program is the highly parallel part of the application to be accelerated on a device such as a GPU or an FPGA.

At a high level, Altera's OpenCL Compiler [Singh 2011] translates an OpenCL kernel to hardware by creating a circuit which implements each operation. These circuits are wired together to mimic the flow of data in the kernel. For example, consider a simple vector addition kernel shown in Figure 19. This kernel describes a simple C function where many copies of this function conceptually run in parallel. Each parallel thread is associated with an ID (`get_global_id(0)`) that indicates the subset of data that each thread operates on. For brevity, we do not include a detailed discussion of the OpenCL programming model. The interested should refer to Khronos OpenCL Working Group [2008].

The translation to hardware will result in the high-level circuit structure shown on the right side of Figure 19. The loads from arrays A and B are converted into *load units* which are small circuits responsible for issuing addresses to external memory and processing the returned data. The two returned values are fed directly into an adder unit responsible for calculating the floating-point addition of these two values. Finally, the result of the adder is wired directly to a *store unit* that writes the sum back to external memory.

Fig. 19. Vector addition FPGA implementation.

Fig. 20. Example of pipeline parallelism.

Fig. 21. Altera OpenCL hardware system.

The most important concept behind the OpenCL-to-FPGA compiler is the notion of pipeline parallelism. For simplicity, assume the compiler has created 3 pipeline stages for the kernel as shown in Figure 20. On the first clock cycle, thread 0 is clocked into the two load units. This indicates that they should begin fetching the first elements of data from arrays A and B. On the second clock cycle thread 1 is clocked in at the same time thread 0 has completed its read from memory and stored the results in the registers following the load units. On cycle 3, thread 2 is clocked in, thread 1 captures its returned data, and thread 0 stores the sum of the two values that it loaded. It is evident that in the steady state, all parts of the pipeline are active, with each stage processing a different thread.

Figure 21 shows a high-level representation of a complete OpenCL system containing multiple kernel pipelines and circuitry connecting these pipelines to off-chip data interfaces. In addition to the kernel pipeline, ACL creates interfaces to external and internal memory. The load and store units for each pipeline are connected to external memory via a *global* interconnect structure that arbitrates multiple requests to a group of DDR

Table II. 3D Finite Differences Results

Design Flow	ALUTs	Reg	F_{max} (MHz)	Runtime	Ratio (ALUTs)	Ratio (Reg)
Altera OpenCL Default	88250	135519	190	0.08s	1.0	1.0
Altera OpenCL Default + FPC	78583	122175	196	0.08s	0.89	0.90
Altera OpenCL Default + Float-to-Fixed	73590	103388	195	0.08s	0.83	0.76

Fig. 22. Example of a 3D stencil computation.

DIMMs. Similarly, OpenCL *local* memory accesses are connected through a specialized interconnect structure to on-chip M9K RAMs.

5.2. Finite Differences in Fixed Point

To demonstrate our techniques on a complete example, we consider the 3D Finite Difference computation described in Micikevicius [2009]. The key computation in this application is a 3D convolution shown in Figure 22.

This stencil template is moved across each element in a 3D volume. Each box in the stencil is given a predetermined weight. The application of the stencil to an element in the volume is simply the weighted sum of neighboring elements.

The source-code for this application is freely downloadable from NVIDIA [2010] and written in OpenCL [Khronos OpenCL Working Group 2008]. We have incorporated the algorithm described in this article into Altera's OpenCL-to-FPGA compiler [Singh 2011; Czajkowski et al. 2012]. Altera's OpenCL compiler is also based on the LLVM compiler framework. This allows us to easily plug in our profiling and floating- to fixed-point conversion passes and evaluate the benefits.

The results of three different variants are shown in Table II. First, Altera's default OpenCL flow is used where standard IEEE-compliant floating-point cores are used in pipelined fashion just as described in Section 5.1. Second, we use the OpenCL compiler with the fused datapath techniques published in Langhammer [2011]. Finally, we show the results of our fixed-point conversion algorithms.

The use of these techniques results in a 17% savings in ALUTs and 24% savings in registers. We feel that these ratios are representative of complete applications as circuits require more than just the core logic to perform computation. There is also a significant chunk of circuitry required to marshal data in an efficient manner from external sources such as SDRAM to the accelerators. The runtime results are reported for a volume of $384 \times 384 \times 384$. Note that the highly pipelined implementation on an FPGA performs the stencil computation across the entire volume in 0.08s in all cases. In contrast, the floating-point algorithm runs on a single-core CPU (Intel Core2 Q9950

at 2.83 GHz) completed in 21s. The fixed-point implementation on the CPU did not yield much runtime savings, since this algorithm still took 20.6s. Our fixed-point FPGA implementation represents a 257x–262x speedup over CPU runtime. We also compared the results from the fixed-point implementation with the original floating-point cores, and determined that our fixed-point implementation yields a signal-to-noise ratio (SNR) of 125 dB. This is well within the required tolerance for this application.

6. CONCLUSION

We have implemented an automatic, profile-guided, fixed-point conversion tool in LLVM which converts floating-point operations to fixed-point to reduce the resources required to implement these computations in customized hardware. This tool offers programmers the opportunity to prototype and perform design space exploration on the microprocessor while creating an efficient FPGA design for algorithm acceleration. This alleviates the long turnaround times often experienced by large FPGA designs. Through our benchmarks, we demonstrate that fixed-point conversion can yield great resource savings, up to 2x–3x reductions in logic and registers used. We also minimize embedded RAM usage, and achieve an average of 13%–22% higher F_{max} than the original floating-point implementation. Our flow uses a novel technique of adding instrumentation code directly into the compiler IR and using that to guide fixed-point code transformations. The transformed IR can be executed directly on a standard microprocessor to evaluate various trade-offs. We have demonstrated that high SNR can be achieved by performing automatic data range analysis and varying total fixed-point bit widths. All steps are completely automatic; we present a set of SNR, area, and operating frequency trade-offs allowing the designer to control the operating point that best meet her requirements.

To illustrate the effectiveness of our conversion tool, we incorporated it into a High-Level Synthesis (HLS) tool that targets FPGAs. In our case study using a 3D finite differences algorithm, we show that our methods reduced logic and register usage by 17% and 24%, respectively. This incurred no performance penalty, and the SNR remained very high at 125 dB.

An interesting future research area would be to combine the techniques shown in this article with Altera’s Fused Datapath Compiler [Langhammer 2011]. The use of value profiling along with reduced dynamic range floating-point (variable sized fixed-point mantissa along with a small dynamic exponent) may lead to even more substantial savings.

REFERENCES

- AAMODT, T. AND CHOW, P. 1999. Numerical error minimizing floating-point to fixed-point ANSI C compilation. In *Proceedings of the 1st Workshop on Media Processors and Digital Signal Processing*.
- ALTERA. 2010. *Quartus II Handbook v11.0*. <http://www.altera.com/literature/archives/lit-archive-index.jsp?doctype=Handbooks&prodCat=Quartus%20II%20Software>
- BALOUGH, C. 2011. Strategic considerations for emerging SoC FPGA’s. Altera whitepaper.
- BELANOVIC, P. AND LESSER, M. 2002. A library of parameterized floating-point modules and their use. In *Proceedings of the 12th International Conference on Field-Programmable Logic and Applications*. Springer, 657–666.
- BROWN, A. W., KELLY, P. H., AND LUK, W. 2008. Profile-Directed speculative optimization of reconfigurable floating point data paths. In *Proceedings of the Workshop on Reconfigurable Computing at HiPEAC*.
- CILIO, A. G. M. AND CORPORAAL, H. 1999. Floating point to fixed point conversion of c code. In *Proceedings of the International Conference on Compiler Construction (CC’99)*. Lecture Notes in Computer Science, vol. 1575. Springer, 229–243.
- COPE, B., CHEEUNG, P. Y. K., LUK, W., AND WITT, S. 2005. Have GPU’s made FPGA’s redundant in the field of video processing? In *Proceedings of the IEEE International Conference on Field Programmable Technology (FPT’05)*. 111–118.

- CZAJKOWSKI, T., AYDONAT, U., DENISENKO, D., FREEMAN, J., KINSER, M., ET AL. 2012. From openCL to high-performance hardware on FPGA's. In *Proceedings of the 22nd International Conference on Field-Programmable Logic and Applications*.
- DE DINECHIN, F., DETREY, J., CRET, O., AND TUDORAN, R. 2008. When FPGA's are better at floating-point than microprocessors. In *Proceeding of the International ACM SIGDA International Symposium on Field Programmable Gate Arrays (FPGA'08)*. 260.
- FOWERS, J., BROWN, G., COOKE, P., AND STITT, G. 2012. A performance and energy comparison of FPGA's, GPU's and multicores for sliding-window applications. In *Proceedings of the International ACM SIGDA International Symposium on Field Programmable Gate Arrays (FPGA'12)*. 47–56.
- GAFFAR, A. A., MENCER, O., LUK, W., AND CHEUNG, P. Y. K. 2004. Unifying bit-width optimization for fixed-point and floating-point designs. In *Proceedings of the Annual International IEEE Symposium on Field Programmable Custom Computing Machines (FCCM'04)*. 79–88.
- HAN, K., OLSON, A. G., AND EVANS, B. L. 2006. Automatic floating-point to fixed-point transformations. In *Proceeding of the 40th ASILOMAR Conference on Signals, Systems and Computers*. 79–83.
- JACOBSON, H. K., KUDVA, P. N., BOSE, P., COOK, P. W. AND SCHUSTER, S. E. 2002. Synchronous interlocked pipelines. In *Proceedings of the International Symposium on Advanced Research in Asynchronous Circuits and Systems*. 3–12.
- JOHNSON, J., CHAGNON, T., VACHRANUKUNKIET, P., NAGVAJARA, P., AND NWANKPA, C. 2008. Sparse LU decomposition using FPGA's. In *Proceedings of the International Workshop on State-of-the-Art in Scientific and Parallel Computing (PARA)*.
- KHRONOS OPENCLWORKING GROUP. 2008. The openCL specification, version 1.0.29. <http://www.khronos.org/registry/cl/specs/opencl-1.0.29.pdf>
- LANGHAMMER, M. 2011. Teraflop FPGA design. In *Proceedings of the 20th IEEE Symposium on Computer Arithmetic (ARITH)*. 187–188.
- LATTNER, C. 2007. New LLVM C front-end: “clang”. In cfe-dev mailing list. <http://lists.cs.uiuc.edu/pipermail/llvmdev/2007-July/009817.html>
- LATTNER, C. AND ADVE, V. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO'04)*.
- LIN, W., TAO, D., KACPRZYK, J., LI, Z., IZQUIEDO, E., ET AL. 2011. *Multimedia Analysis, Processing and Communications*. Studies in Computational Intelligence. Springer.
- MICKEVICIUS, P. 2009. 3D finite difference computation on GPU's using CUDA. In *Proceedings of the 2nd Workshop on General Purpose Processing on Graphics Processing Units (GPGPU-2)*. ACM, New York, 79–84.
- NVIDIA. 2010. NVIDIA openCL SDK code samples. <https://developer.nvidia.com/opencl>
- OLSON, H., JANTSCH, A., AND TENHUNEN, H. 1999. Floating-To fixed-point refinement in matlab with an object-oriented library. In *Proceedings of the 17th IEEE Nordic Micro Electronics Event (NORDIC'99)*.
- SINGH, D. 2011. Implementing FPGA design with the OpenCL standard. Altera whitepaper.
- SMITH, M. D. 2000. Overcoming the challenges to feedback-directed optimization (keynote talk). In *Proceedings of the ACM SIGPLAN Workshop on Dynamic and Adaptive Compilation and Optimization (DYNAMO'00)*. ACM, New York, 1–11.
- STRICKLAND, M. AND LANGHAMMER, M. 2008. FPGA coprocessing evolution: Sustained performance approaches. Altera whitepaper.
- VERMA, S. AND McELHENY, P. 2012. Introducing innovations at 28 nm to move beyond moores law. Altera whitepaper.
- WINKLER, S. 2012. Analysis of public image and video databases for quality assessment. *J. Sel. Topics Signal Process.* 6, 6, 616–625.
- WINKLER, S. AND MOHANDAS, P. 2008. The evolution of video quality measurements: From PSNR to hybrid metrics. *IEEE Trans. Broadcast.* 54, 3, 660–668.
- XILINX. 2011. Zinq 7000 EPP product brief. <http://press.xilinx.com/phoenix.zhtml?c=212763&p=irol-newsArticle&ID=1678259&highlight=>
- ZHANG, Y., SHALABI, Y. H., JAIN, R., NAGAR, K. K. AND BAKOS, J. D. 2009. FPGA vs GPU for sparse matrix vector multiply. In *Proceedings of the International Conference on Field-Programmable Technology*.

Received June 2012; revised November 2012; accepted November 2012