High-Level Design Tools for Floating Point FPGAs

Deshanand P. Singh
Altera Corporation
150 Bloor Street West, Suite 400
Toronto, Ontario, Canada
dsingh@altera.com

Bogdan Pasca
Altera Corporation
Westwood, High Wycombe,
Buckinghamshire HP12 4PU,
United Kingdom
bpasca@altera.com

Tomasz S. Czajkowski
Altera Corporation
150 Bloor Street West, Suite 400
Toronto, Ontario, Canada
tczajkow@altera.com

ABSTRACT
This tutorial describes tools for efficiently implementing floating point applications on FPGAs. We present both the SDK for OpenCL and DSP Builder Advanced Blockset and show that they can be effectively used to implement many floating point applications. The methods for optimizing application performance are also described.

In this tutorial we focus on a few applications, including Fast Fourier transform, matrix multiplication, finite impulse response filter and a Cholesky decomposition. In all cases we show what the tools are capable of achieving, and more importantly how a user can take advantage of the various floating-point centric features that are made available. We also discuss how these tools can automatically use FPGA architectural features such as hardened floating-point DSP available on Altera Arria 10 family.

Categories and Subject Descriptors
C.1.3 [**Computer System Organization**]: Other Architecture Styles, Data-flow architectures.

General Terms
Design

Keywords
Floating Point; Optimization; FPGAs

1. INTRODUCTION
Many applications in a variety of different domains are first simulated or modeled using floating-point data processing. This is done using either programming languages such as C/C++ or tools such as Matlab. The final implementation on platforms such as FPGAs has usually been performed using fixed-point arithmetic because of area considerations. To do this successfully, the algorithms are carefully mapped into a limited dynamic range, and scaled through each function in the datapath.

Over the last 10 years FPGAs have grown sufficiently large to facilitate native floating point based applications. However, there has been a lack of support for floating point functions, which meant that designers were left on their own to ensure that the floating point implementation of a given function satisfies their application's criteria. This meant most designers chose not to use floating point operations simply because there was no convenient way to do so.

To truly enable floating point application development, it is imperative to provide both FPGAs and tools to program them. To that end, in this tutorial we introduce two tools that can enable users to take advantage of floating point capabilities on FPGA devices. These tools are: SDK for OpenCL and DSP Builder Advanced Blockset. We will also discuss how these tools can take advantage of architectural features of modern FPGAs, and specifically focus on Arria 10 device family as an example of how hardened FP DSP blocks can benefit designs in many application domains.

SDK for OpenCL [1] enables users to describe an application using a C-like description, as described by the OpenCL Standard [4]. One of the key advantages this standard brings to FPGAs is a front-end support for floating-point operations, enabling end users to seamlessly use floating point data types that they are used to when programing a wide variety of applications, while not having to worry too much about the low-level implementation details of floating point functions. DSP Builder Advanced Blockset [2] is a tool that uses Matlab's Simulink as a front end to describe an application, both fixed and floating point, to enable the user to abstract away low-level details of hardware implementation. In this tutorial, we will demonstrate how these two tools can be used to implement efficient floating-point benchmarks.

The remainder of this paper is organized as follows: Section 2 discusses the floating point formats and the key challenges floating point application designers face. In Sections 3 and 4, we discuss how many of the challenges of such design are alleviated by tools such as Altera's SDK for OpenCL [1] and DSP Builder Advanced Blockset [2]. We discuss these tools using a case study of several applications to illustrate the novelty and productivity the tools bring to end users. Finally, we summarize the paper in section 5 with concluding remarks and future work.

2. BACKGROUND
Traditionally FPGAs have been used for non-floating-point applications due to the fact that floating point operations can take considerable area when implemented using Lookup Tables (LUTs). The reason for this is the representation of floating point numbers as specified in IEEE754 standard [3]. Each floating point number consists of a single bit sign, an exponent and a mantissa. The exponent specifies the order of magnitude for a given number, whereas the mantissa specifies the value with more precision. Table 1 shows a variety of mantissa and exponent sizes commonly used by many applications.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
FPGA'14, February 22–24, 2015, Monterey, California, USA.
Copyright © 2015 ACM 978-1-4503-3315-3/15/02…$15.00.
http://dx.doi.org/10.1145/2684746.2689079

Table 1. Commonly used floating point formats

Precision	Exponent Bits	Mantissa Bits
Half	5	10
Single	8	23
Double	11	52

In addition to supporting computation in normal range, thre are two special values: infinity and not-a-number (NaN). Infinity is used by floating point operations to signify that the result of an operation is outside of the range representable by a given floating point format. A NaN on the other hand signifies an operation that does not provide a valid result. For example, adding +Infinity to –Infinity does not produce a valid result.

One of the many opportunities afforded by using FPGAs is the optimization of floating point operations for a given application. Both Altera SDK for OpenCL [1] and Altera DSP Builder Advanced Blockset [2] enable users to take advantage of floating-point compiler mode [7]. The floating point compiler mode enables the user to reduce the area of floating point operations by removing support for infinity and NaN special values if the user can guarantee that their application will never make use of them.

In the following sections we will discuss various methods that can be used by FPGA system designers to take advantage of the floating point support offered by Altera OpenCL SDK and Altera DSP Builder Advanced Blockset flows as well as device families they can target.

3. SDK for OpenCL

Altera OpenCL SDK is a complete suite to facilitate the use of OpenCL Standard [4] for designing applications on FPGAs. An OpenCL application comprises a kernel and a host program. In SDK for OpenCL, the kernel is implemented using an automatically generated datapath on an FPGA, while the host program is executed on a processor that communicates with the kernel on an FPGA board via mechanisms such as PCIe. We demonstrate the utility of this tool for the implementation of an FFT and a general matrix-matrix multiplication.

3.1 FFT

The Fast Fourier transform is a classic application used in digital signal processing applications. Its regular structure yields itself nicely to an efficient hardware implementation. While many FFT architectures have been described in literature, as an example we implemented a particular one described in [5] for 4K-point FFT on an Altera Stratix V FPGA.

Floating point optimizations are especially important for algorithms such as FFT, where floating point operations comprise almost all of the required resources. Aside from minimal control and data movement logic, the complete data pipeline is effectively a sequence of floating point operators comprising addition, subtraction and multiplication.

The optimizations stem partly from proprietary optimizations known as the Floating Point Compiler (FPC) [7]. The FFT data path benefits from three such optimizations: removal of NaN and Inf support, changing rounding mode to round-to-zero, and fusing addition and subtraction into a single operation.

In algorithms such as FFT, support for non-finite results is typically superfluous when the input data ranges are known and bounded. The area required for adders and multipliers can therefore be reduced by not supporting these exceptions. Round-to-zero operations are simply truncations; consequently they require no hardware resources to implement. Further optimization is possible by fusing addition and subtraction operators when both inputs are the same (a+b, a-b). It is a well-known technique [6] that does not require reordering of operations and is thus safe to perform at any time. This transform is beneficial because most of the logic in a single adder module can be reused within the subtractor, avoiding logic duplication.

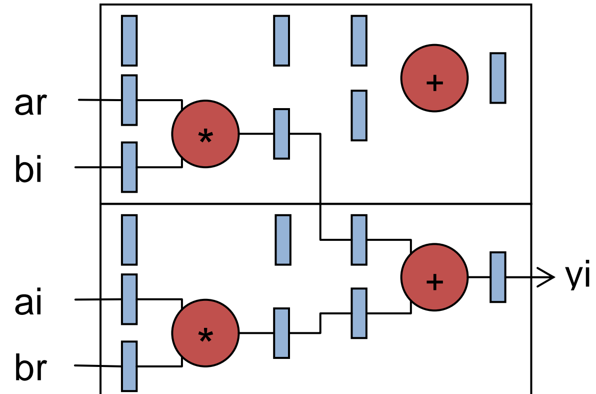

Figure 1. Imaginary part of complex FP multiplication implemented using FP DSP Blocks.

The implementation of an FFT becomes even more optimized on an Arria 10 device, where hardened floating point DSP blocks are available [8]. In particular, for FFT applications we can take advantage of the DSP blocks to perform complex multiplication compactly, as shown in Figure 1. In the figure, we show an abstract representation of two adjacent DSP blocks configured in a floating-point mode. Each DSP block comprises two operations, addition (or subtraction) and multiplication. The operators are connected by programmable paths that may take advantage of pipelining registers to connect to one another or to an adjacent DSP block. In this case we show the computation of the imaginary part of complex multiplication. To do this we use two DSP blocks, taking advantage of two multipliers and an adder, while one adder is left unused. If it is the case that complex multiplication is followed by complex addition the unused adder may be selected to perform the addition of the imaginary parts of the complex multiplication result and another complex number. A similar implementation is used for the real part of the computation. Table 2 summarizes the area for each optimization.

Table 2. FFT Optimization Results

Optimization	ALMs	DSPs
IEEE754 Conformant	62126	60
FPC	39662	60
FPC+Fused add/sub	34102	60
Arria 10 (Hardened FP)	6208	98

3.2 Matrix-Matrix Multiplication

In the matrix-matrix multiplication algorithm, shown in Figure 2, the multiplication is performed using blocks of data, where on

each iteration of a loop a block of size BLOCKxBLOCK of each of the input matrices is read in, a dot product is computed and added to the sum until the entire column is processed for each element. To speed up the computation, an attribute *num_simd_work_items* is used to vectorize the application, thus increasing the throughput by a factor of V.

The key to an efficient design from a floating-point perspective is in how lines 19-20 are implemented in hardware. Due to the *#pragma* statement, the loop of multiplication and addition is unrolled into a chain of multiply and add operations. Usually, a balanced tree of adders works better than a chain as it reduces the area of the circuit. In SDK for OpenCL the users are not required to rewrite the application to do this; it is sufficient to supply a flag *--fp-relaxed* to the compiler. This flag signifies that the user is aware that reordering floating point operations may change the output, but it is acceptable for this application. The compiler will then examine the sequence of floating point operations and rearrange them to produce a more efficient implementation.

```
__attribute((reqd_work_group_size(BLOCK,BLOCK,1)))
__attribute((num_simd_work_items(V)))
__kernel void matmult(global float *A,
             global float *B, global float *C) {
  int w = get_global_size(0);
  int x = get_local_id(0); int y = get_local_id(1);
  int bx = get_group_id(0); int by = get_group_id(1);
  local float lA[BLOCK*BLOCK], lb[BLOCK*BLOCK];
  float sum = 0.0f;
  for(int i=0; i< w; i+=BLOCK) {
    lA[x+BLOCK*y] = A[i+x+(y+by)*w];
    lB[x+BLOCK*y] = B[bx+x+(i+y)*w];
    barrier();
    #pragma unroll
    for(int k=0; k < BLOCK; k++)
      sum += lA[k+y*BLOCK]*lB[k*BLOCK+x];
    barrier();
  }
  C[get_global_id(0)+get_global_id(1)*w] =  sum;
}
```

Figure 2. Matrix-Matrix Multiplication pseudo code.

Similarly to the example of the FFT, we can take advantage of the FPC flow using the *--fpc* flag as an argument to the compiler. In this case, the compiler will optimize the tree of adders to minimize their area. Doing this allows us to more than double the throughput of the application.

Finally, we can take this design to the next level by implementing it on an Altera Arria 10 device and take advantage of hardened floating point adder and multiplier blocks to reduce the area of the design. This particular optimization occurs automatically, when a user choses to target an Arria 10 device.

The results of synthesizing, placing and routing this design are shown in Table 3. This shows the use of hardened FP (HFP) on Arria 10 to achieve extreme area savings.

Table 3. Matrix-Matrix multiplication area results

Configuration	ALMs	DSPs
BLOCK=128, V=8, FPC	315061	1034
BLOCK=128, V=8, HFP	61293	1034

4. DSP Builder Advanced Blockset

The DSP Builder Advanced Blockset (DSPBA) is a high-level design tool with a model-based design entry which integrates with Matlab's Simulink Frontend. With DSPBA, users functionally verify and debug their designs at the Simulink level using scopes and variables. This allows for considerably faster algorithm iterations as opposed to traditional FPGA development using RTL languages and simulators. Once the desired functionality is achieved, DSPBA efficiently maps the implementation to a user-defined FPGA target and automatically pipelines the design to achieve a target clock frequency.

DSPBA offers users full flexibility when implementing datapaths allowing for a mixture of fixed or floating-point types. Moreover, both fixed and floating-point types are parametrizable: total width, fraction width and sign are used for fixed-point types and exponent and fraction width are used for floating-point types. For floating-point datapaths users may choose from implementing parts using the floating-point compiler technology [7] or IEEE-754 conformant implementation to trade-off resources for numerical conformance. The provided floating-point library of components uses state-of-the-art techniques [9] to efficiently map to modern DSP-enabled FPGAs.

In the following we exemplify the use case for DSPBA on two designs: a floating-point FIR filter and linear system solver based on the Cholesky decomposition.

4.1 FIR Filter

The Finite Impulse Response (FIR) filter is expressed using the following equation for a filter of order *N*:

$$y[n] = \sum_{i=0}^{N} c_i x[n-i]$$

DSPBA provides users a full library of efficient FIR filter implementations including decimating, interpolating, single-rate and fractional-rate. However, if users desire to manually create a simple FIR example they may do so using the scalar product block which receives on one input the twiddle coefficients and on the second input the input x and its delayed versions using the z^{-1} block. Figure 2 shows how such a filter would be used for N=4.

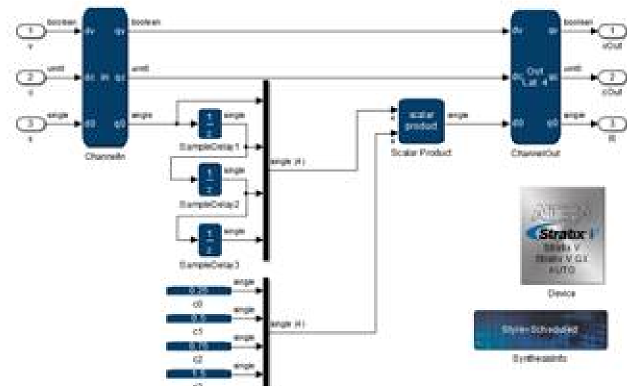

Figure 2. FIR Filter Design in DSPBA

One must note that DSPBA provides specialized blocks for the tapped delay line, which allow for a more compact description of the delay line. The scalar product block receives the two vectors of 4 elements and outputs the filter result to the channel out. The

data types used for this example is floating-point single precision. This can be easily updated to other floating or fixed-point data types by updating the type of the input and allowing the default data type propagation. Table 4 shows the implementation results for this benchmark on Stratix V using soft logic and Arria 10 using FP DSP Blocks.

Table 4. FIR Filter area results

Configuration	ALMs	DSPs
Stratix V, 128-tap	60881	128
Arria 10, 128-tap, HFP	1676	131

4.2 Cholesky Decomposition

DSPBA allows users to generate larger circuits which perform more complex tasks, such as solving linear systems of equations, $Ax=b$, using the Cholesky decomposition. The Cholesky decomposition relies on decomposing the matrix A into a lower triangular matrix L such that $A = L L^*$ where L^* is the transpose conjugate (if A is a complex matrix). The system $Ax=b$ now becomes $L(L^* x) = b$ and using the variable change $L^* x=y$ we obtain $L y = b$. Solving this system for y can be achieved using forward substitution. Having obtained y the next system $L^*x=y$ can be solved for x using backward substitution. Hence, solving the linear system is composed of 3 steps: decomposition, forward substitution followed by backward substitution.

There are multiple possible DSPBA implementations for this problem each trading latency and throughput for area. One possible implementation performs the decomposition and forward substitution in one module and the backward substitution in another. To maximize performance it is desirable to balance the latency of the two modules. Therefore, the first module would use extended vector-products whereas the second module would use fewer resources and perform the process iteratively. The two modules are depicted in Figure 3.

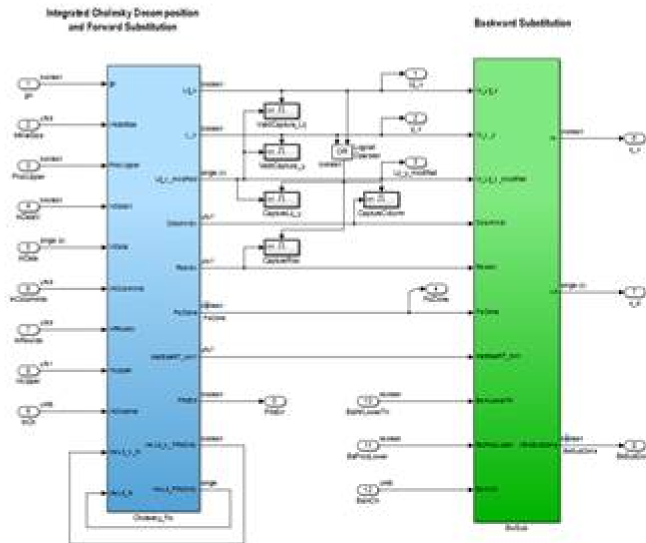

Figure 3. Cholesky Decomposition in DSPBA

The main computing kernel in the decomposition and forward substitution is the scalar-product. It is used in both stages using configurable multiplexers to feed the desired input data. The forward substitution stage overlaps with the decomposition stage.

Table 5 shows the implementation results for this benchmark on Stratix V using mostly soft logic and Arria 10 using hardened FP DSP Blocks.

Table 5. Cholesky Filter area results

Configuration	ALMs	DSPs
Stratix V	109914	260
Arria 10, HFP	12716	270

5. CONCLUSION

In this paper we have briefly presented a two high-level design tools and described how they can be used to optimize floating point benchmarks. The key advantage for the end user is the ability to quickly create a fully functioning circuit that can be programmed onto an FPGA. The tools themselves contain many floating-point specific optimizations that significantly reduce the amount of time the user is required to consider low-level implementation details.

Finally, we showed the impact of FPGA architectural features such as HFP to enable extreme area reductions. Both presented tools take advantage of HFP seamlessly, enabling the implementation of ever more advanced applications on FPGAs.

6. ACKNOWLEDGMENTS

The authors would like to thank Simon Finn, Michael Kinsner, Martin Langhammer in providing benchmarking data and advice throughout this work.

7. REFERENCES

[1] Altera Corporation, Altera SDK for OpenCL, http://www.altera.com/products/software/opencl

[2] Altera Corporation, Altera DSP Builder Advanced Blockset, http://www.altera.com/technology/dsp/advanced-blockset

[3] IEEE standard for binary floating-point arithmetic. ANSI/IEEE Std. 754-1985, pages 1-58, 2008.

[4] Khronos OpenCL Working Group. The OpenCL Specification, version 1.1.48, June 2009.

[5] M. Garrido, J. Grajal, M. Sanchez, and O. Gustafsson. "Pipelined radix-2k feedforward FFT architectures", Very Large Scale Integration (VLSI) Systems, IEEE Transactions on, 21(1):23-32, 2013.

[6] E. Swartzlander and H. Saleh, " FFT implementation with fused floating-point operations", IEEE Transactions on Computers, 61(2):284-288, 2012.

[7] M. Langhammer, "Floating Point Datapath Synthesis for FPGAs", International Conference on Field Programmable Logic and Applications, pp. 355-360, 2008.

[8] B. Pasca, and M. Langhammer, "Floating Point DSP Block Architecture for FPGAs", ACM/SIGDA International Symposium on FPGAs, Monterey California, Feb, 2015.

[9] de Dinechin, F.; Joldes, M.; Pasca, B., "Automatic generation of polynomial-based hardware architectures for function evaluation," Application-specific Systems Architectures and Processors (ASAP), 21st IEEE International Conference on , pp.216-222, 7-9 July 2010