Accelerating seismic computations using customized number representations on FPGAs

Haohuan Fu, William Osbourne, Robert G. Clapp, and Oskar Mencer

ABSTRACT

Field Programmable Gate Arrays (FPGA) offer significant potential speedups over conventional microprocessors for some applications. For downward continued migration, complex math and Fast Fourier Transforms (FFT) are the dominant cost. Convolution is the dominant cost for reverse time migration. We implement these core algorithms on a FPGA and show speedups ranging from 5 to 15, including the transfer time to and from the processors. We consider methods to further speed up these migration algorithms.

INTRODUCTION

Seismic imaging is the most computationally demanding technology of the oil and gas sector. Downward continued based migration (Gazdag and Sguazzero, 1985) is the most prevalent high-end imaging technique today, and reverse time migration appears to be one of the dominant imaging techniques for the future.

Downward continued migration comes in various flavors including Common Azimuth Migration (Biondi and Palacharla, 1996), shot profile migration, source-receiver migration, planewave or delayed shot migration, and narrow azimuth migrations. The different techniques have varying cost profiles but all share two meaningful computational bottlenecks: transforming to-and-from the wavenumber domain (FFT) and applying the single square root (SSR) or double square root (DSR) condition (complex exponentials).

The cost of explicit space-domain 3-D reverse time migration is dominated by the cost of continuing the source and receiver wavefield a given time step. To progress the wavefield a given time step requires applying a 3-D stencil that can range in size from 7 to 31 points depending on the finite-difference approximation that is chosen.

In this paper, we describe the implementation of DSR equation and the kernel of 3-D acoustic modeling on a FPGA. We begin by giving a basic background of FPGAs. We describe the FPGA programming environment and our methodology for determining the correct trade off between precision and speed. We then describe the implementation procedure for both algorithms. We conclude by discussing potential additional speedup opportunities of both reverse time and wavefield continuation based migration.

PROJECT OVERVIEW

As a first step it is useful to begin by defining some FPGA terminology.

Block RAM Small memory elements (store up to 512 single-precision floats) located within the FPGA. The Xilinx Virtex 4 FX100 FPGA contains 376. All BRAMs can be read and written in parallel and combined into larger memories, leading to very high internal bandwidth.

FIFO First-In-First-Out Memory queue built from Block RAMs. FIFOs exploit temporal locality in data streams.

Slice A unit of area on Xilinx FPGAs. Each slice contains 2 (on most current FPGAs) lookup tables (LUTs), the basic compute unit on an FPGA, each LUT implements any 4-input 1-output logical function. We connect LUTs to implement arithmetic and control logic.

PCI Express x8 State-of-the-art bus for FPGA acceleration. 4000MB/sec peak bandwidth.

FPGAs are Complementary metaloxidesemiconductor (CMOS) technology-based chips containing logic which can be configured to any sequential circuit and a limited number of memory elements including RAMs and registers. The price of reconfigurability is a 10x slower dynamic clock frequency compared to modern processors. We exploit the parallelism and ability to use custom number representations to overcome the lower clock frequency and obtain a higher performance.

The long term goal of this project is to speed up key seismic imaging application by at least a factor of 10x over conventional multi-core hardware.

Downward continued based migration

For downward continued based migration there are four potential computational bottlenecks that vary depending on the flavor of the downward continuation algorithm. In many cases the dominant cost is the FFT step. The dimensionality of the FFT varies from 1-D (tilted plane-wave migration (Shan and Biondi, 2007)) to 4-D (narrow azimuth migration (**?**). The FFT cost is often dominant due its $nlog(n)$ cost ratio, n being the number of points in the transform, and the non-cache friendly nature of multi-dimensional FFTs. The FK step, which involves evaluating a square root function and performing complex exponential is a second potential bottleneck. The high operational count per sample can eat up significant cycles. The FX step, which involves a complex exponential, or sine/cosine multiplication, has a similar, but computationally less demanding, profile. Creating subsurface offset gathers for shot profile or plane-wave migration, particularly 3D subsurface offset gathers, can be an overwhelming cost. The large op-count per sample and the non-cache friendly

usage can be problematic. Finally, for finite difference based schemes a significant convolution cost is involved.

Last summer the focus was on speeding up 1 and 2-D FFTs. Speedup ranged from 8x-16x depending on the required data precision. (Pell and Clapp, 2007) demonstrated that the subsurface offset calculation can be sped up by a factor of 20x-40x. This summer, the focus was speeding up the FK step by implementing both a table lookup and complex exponential on the FPGA.

Reverse time migration

The primary bottleneck of reverse time migration is applying the finite-difference stencil. In addition to the large operation count (5 to 31 samples per cell) the access pattern has poor cache behavior for real size problems. Beyond applying the 3-D stencil the next major cost is implementing damping boundary conditions. Methods such as Perfectly Matched Layers (PML) can be costly (Berenger, 1994). Finally, if you want to use reverse time migration for velocity analysis, subsurface offset gathers need to be generated. The same cost profile that exists in downward continued based migration exists for reverse time migration.

Last summer the focus was on implementing the 2-D elastic modeling convolutional kernel. We achieved a speed up of 8-16x, again depending on data precision. This summer we concentrated on 3-D acoustic modeling kernel.

BACKGROUND

Number Representation

Precision and range are key resources to be traded off against the performance of a computation. We looked at three different types of number representation: fixed-point, floating-point and logarithmic. Consider the case when U_i is represented as a fixed-point number, with an integer part I which is a bits in length, and a fraction part F which is b bits in length.

i_{a-1} ... i_2 i_1 i_0	f_0 f_1 f_2 ... f_{b-1}

The integer bit-width, which represents the dynamic range of the number, is calculated according to equation (1):

$$k \geq \lceil log_2(|max(U_i) - min(U_i)|) \rceil \tag{1}$$

For the floating-point number system, let U_i represent a floating-point number $(-1)^S \cdot M \cdot 2^E$, where S is the sign bit, M is the mantissa with a bit-width of m bits, and E is the exponent with a bit-width of e bits.

S	$i_0\ i_1\ i_2\ ...\ i_{m-1}$	$f_{e-1}\ ...\ f_2\ f_1\ f_0$

The value of the mantissa M is expressed as:

$$M = \sum_{i=0}^{m-1} a_i 2^{-i} \tag{2}$$

where $a_i \in \{0, 1\}$.

It is possible to relate the bit-width m of the mantissa of the node to the error when representing the mantissa by a finite bit-width $Errflt$, as follows:

$$Errflt(m) = \begin{cases} 2^{-m} \times 2^E & \text{if round-to-nearest} \\ 2^{-(m-1)} \times 2^E & \text{if truncation} \end{cases}$$

where E is the value of the exponent at the node.

Since there is no standard to encode logarithmic numbers, in this report we use a fixed-point format to store the logarithmic value.

A stream Compiler (ASC)

We use our object-oriented ASC FPGA programming tool to develop a range of different solutions. ASC, A Stream Compiler, was developed following research at Stanford University and Bell Labs, and is now commercialized by Maxeler Technologies. ASC enables the use of FPGAs as highly parallel stream processors. ASC is a C-like programming environment for FPGAs. ASC code makes use of C++ syntax and ASC semantics which allow the user to program on the architecture-level, the arithmetic-level and the gate-level. In contrast to other methodologies, ASC provides the productivity of high-level hardware design tools and the performance of low-level optimized hardware design. On the arithmetic level, PAM-Blox II provides an interface for custom arithmetic optimization. On the higher level, ASC provides types and operators to enable research on custom data representation and arithmetic. ASC hardware types are HWint, HWfix and HWfloat. Utilizing the data-types we build libraries such as a function evaluation library or develop special circuits to solve particular computational problems such as graph algorithms. A simple example of an ASC description for a stream architecture that doubles the input and adds '55' looks as follows:

```
%\lstset{language=ASC}
%\begin{figure}[ht]
%\begin{lstlisting}
```

```
// ASC code starts here
STREAM_START;

// Hardware Variable Declarations
HWint in(IN);
HWint out(OUT);
HWint tmp(TMP);

STREAM_LOOP(16);
tmp = (in << 1) + 55;
out = tmp;

// ASC code ends here
STREAM_END;
%\end{lstlisting}
%\caption{A simple ASC example} \label{fig:asc_example}
%\end{figure}
```

The ASC code segment shows HWint variables and the familiar C syntax for equations and assignments. Compiling this program with 'gcc' and running it creates a net-list which can be transformed into a configuration bitstream for an FPGA.

CUSTOMIZED NUMBER REPRESENTATIONS

FPGA-based implementations have the advantage over current software-based implementations of being able to use customizable number representations in their circuit designs. On a software platform, users are usually constrained to a few fixed number representations, such as 32/64-bit integers and single/double-precision floating-point; while the reconfigurable logic and connections on an FPGA enables the users to explore various number formats with arbitrary bit-widths. Furthemore, users are also able to design the arithmetic operations for these customized number representations, can thereby providing a highly customized solution for a given problem.

In general, to provide a customized number representation for an application, we have three questions to solve:

- Which number representation should we use?

  There are existing FPGA applications using fixed-point, floating-point and logarithmic numbers. Fixed-point has simple arithmetic implementations, while floating-point and logarithmic number systems (LNS) provide a wide representation range.

- How do you determine the bit-width of the variables in the design?

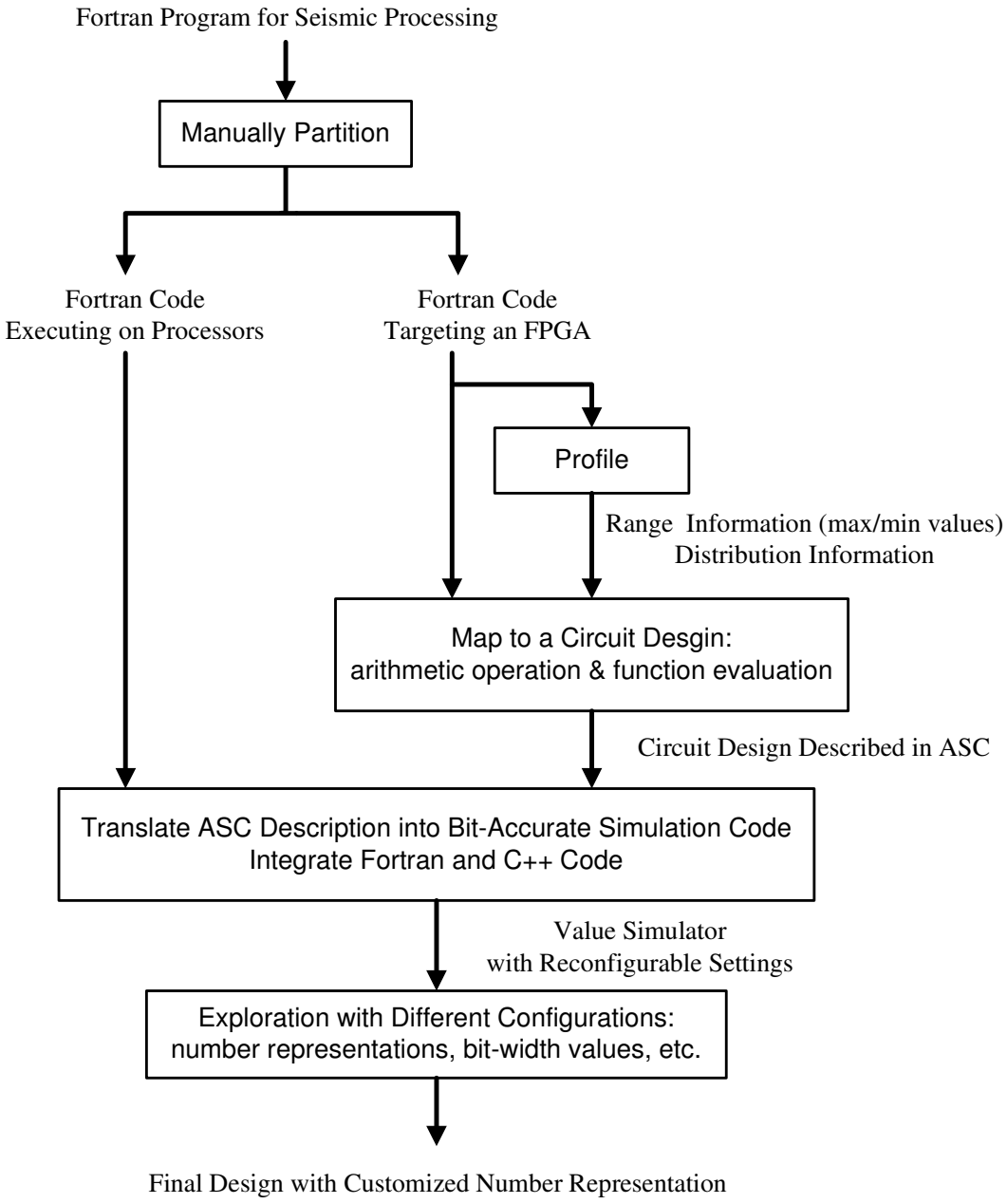


Figure 1: Basic steps to achieve a hardware design with customized number representations.

This problem is generally referred to as bit-width or word-length optimization (Lee et al., 2006; Constantinides et al., 2001). We can further divide this into two different parts: **range analysis** considers the problem of ensuring that a given variable inside a design has a sufficient number of bits to represent the range of the numbers; while in **precision analysis**, the objective is to find the minimum number of precision bits for the variables in the design such that the output precision requirements of the design are met.

- How do you implement the arithmetic operations for the customized number representations?

  The arithmetic operations of each number system are quite different. For instance, in LNS, multiplication, division and exponential operations become as simple as addition or shift operations, while addition and subtraction become non-linear functions to approximate. The arithmetic operations of regular data formats, such as fixed-point and floating-point, also have different algorithms with different design characteristics. Evaluation of elementary functions also plays a large part in seismic applications (trigonometric and exponential functions). Different evaluation methods and configurations can be used to produce evaluation units with different accuracies and performance.

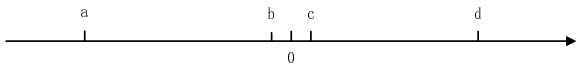
This section discusses our approaches to finding a solution to these three problems. The approaches are partly based on our previous work on bit-width optimization (Lee et al., 2006) and comparison between different number representations (Fu et al., 2006, 2007). As shown in Fig. 1, we manually partition the Fortran program into two parts: one part will run in software and the other in hardware (target code). The first step is to profile the target code to acquire information about the distribution of values that each variable can take. In the second step, based on the range information, we map the Fortran code into a hardware design described in ASC format, which includes implementation of arithmetic operations and function evaluation. In the third step, the ASC description is translated into bit-accurate simulation code, and merged into the original Fortran program to provide a value simulator for the original application. Using this value simulator, explorations can be performed with configurable settings such as different number representations, different bit-widths and different arithmetic algorithms. Based on the exploration results, we can determine the optimal number format for this application with regards to certain characteristics such as circuit area and performance.

Profiling

In the profiling stage, the major objective is to collect range and distribution information for the variables. The idea of our approach is to instrument every target variable in the code, adding function calls to initialize data structures for recording range information and to modify the recorded information when the variable value changes.

For the range information of the target variables (variables to map into the circuit design), we keep a record of four specific points on the axis, shown in figure 2.

Figure 2:



The points a and d presents the values far away from zero, i.e., the maximum absolute values that need to be represented. Based their values, the integer bit-width of fixed-point numbers can be determined. Points b and c represent the values close the zero, i.e., the minimum absolute values that need to be represented. Using the minimum and maximum values, the exponent bit-width of floating-point numbers and integer bit-width of logarithmic numbers can be determined.

For the distribution information of each target variable, we keep a number of buckets to store the frequency of values at different intervals. Fig. 3 shows the distribution information recorded for the real part of variable `wfld` (a complex variable). In each interval, the frequency of positive and negative values are recorded separately. The results show that, for the real part of variable `wfld`, in each interval, the frequencies of positive and negative values are quite similar, and the major distribution of the values falls into the range 10^{-1} to 10^4.

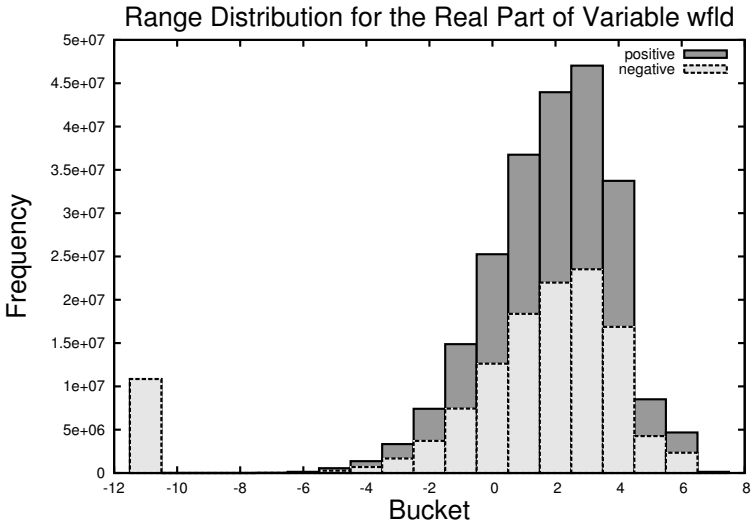


Figure 3: Range distribution of the real part of variable 'wfld'. The leftmost bucket with index= -11 is reserved for zero values. The other buckets with index= x store the values in the range $(10^{x-1} - 10^x$.

The distribution information provides a rough metric for the users to make an initial guess about which number representations to use. If the values of the variables cover a wide range, floating-point and logarithmic number formats are usually more suitable. Otherwise, fixed-point numbers shall be enough to handle the range.

Circuit Design: Basic Arithmetic & Elementary Function Evaluation

After profiling range information for the variables in the target code, the second step is to map the code into a circuit design described in ASC.

As a high-level FPGA programming language, ASC provides hardware data-types, such as HWint, HWfix and HWfloat. Users can specify the bit-width values for hardware variables, and ASC automatically generates corresponding arithmetic units for the specified bit-widths. It also provides configurable options to specify different optimization modes, such as AREA, LATENCY and THROUGHPUT. In the THROUGHPUT optimization mode, ASC automatically generates a fully-pipelined circuit. These features make ASC an ideal hardware compilation tool to re-target a piece of software code onto the FPGA hardware platform.

ASC does not have inherent support for LNS numbers. This part is covered by our previous work on the LNS arithmetic library generator (Fu et al., 2007), which produces optimized LNS arithmetic units with customizable bit-width values, also in ASC format.

Thus, with support for fixed-point, floating-point and LNS arithmetic operations, the target Fortran code can be transformed into ASC C++ code in a straightforward manner. We also have interfaces provided by ASC and the LNS library generator to modify the internal settings of these arithmetic units.

In seismic applications, evaluation of elementary functions takes a large part in the application. For instance, in the first piece of target code we try to accelerate, the 'complex exponential function'. A large part of the computation is to evaluate the square root and sine/cosine functions. To map these functions into efficient units on the FPGA board, we use a table-based uniform polynomial approximation approach, based on Dong-U Lee's work on optimizing hardware function evaluation (Lee et al., 2005). The evaluation of the two functions can be divided into three different phases (Muller, 1997):

- Range Reduction: reduce the range of the input variable x into a small interval that is convenient for the evaluation procedure. The reduction can be multiplicative (e.g. $x' = x/2^{2n}$ for square root function) or additive (e.g. $x' = x - 2\pi\dot{n}$).
- Function Evaluation: approximate the value of the function using a polynomial within the small interval.
- Range Reconstructions: map the value of the function in the small interval back into the full range of the input variable x.

To keep the whole unit small and efficient, we use degree-one polynomial so that only one multiplication and one addition are needed to produce the evaluation result.

Meanwhile, to preserve the approximation error at a small scale, the reduced evaluation range is divided into uniform segments. Each segment is approximated with a degree-one polynomial, using the minimax algorithm. In the case of the 'complex exponential' code segment, the square root function is approximated with 384 segments in the range of [0.25, 1] with a maximum approximation error of 4.74×10^{-7}, while the sine and cosine functions are approximated with 512 segments in the range of [0, 2] with a maximum approximation error of 9.54×10^{-7}.

Bit-accurate Value Simulator

As discussed earlier, based on the range information, we are able to determine the integer bit-width of fixed-point and LNS numbers and the exponent bit-width of floating-point numbers. The remaining bit-widths, such as the fractional bit-width of fixed-point and LNS numbers, and the mantissa bit-width of floating-point numbers, are predominantly related to the precision of the calculation in order to find out the minimum acceptable values for these precision bit-widths, we need a mechanism to determine whether a given set of bit-width values produce satisfactory results for the application.

In our previous work on function evaluation or other arithmetic designs, we set a requirement of the absolute error of the whole calculation, and use a conservative error model to determine whether the current bit-width values meet the requirement (Lee et al., 2006). However, a specified requirement for absolute error does not work for seismic processing. To find out whether the current configuration of precision bit-width is accurate enough, we need to run the whole program to produce the resulting image, to find out whether the image contains the correct pattern information. Thus, to enable exploration of different bit-width values, a value simulator for different number representations is needed to provide bit-accurate simulation results for the hardware designs.

With the requirement to produce bit-accurate results as the corresponding hardware design, the simulator also needs to be efficiently implemented, as we need to run the whole application (which takes days using the whole input dataset) to produce the image.

In our approach, the simulator works with ASC format C++ code. It re-implements the hardware data-types, such as HWfix, HWfloat and HWlns, and overloads their arithmetic operators with the corresponding simulation code.

For HWfix variables, the value is stored in a 64-bit signed integer, while another integer is used to record the fractional point. The basic arithmetic operations are mapped into shifts and arithmetic operations of the 64-bit integers.

For HWfloat variables, the value is stored in a 64-bit double-precision floating-point number, with two other integers used to record the exponent and mantissa bit-width. To keep the simulation simple and fast, the arithmetic operations are

processed using double-precision floating-point values. However, to keep the result bit-accurate, during each assignment, using functions `frexp` and `ldexp`. The double-precision value is decomposed into mantissa and exponent, truncated according to the exponent and mantissa bit-width, and combined back into the double value.

The arithmetic operations of HWlns are implemented using HWfix numbers. Thus, we call the HWfix simulation code to perform the calculations of HWlns.

Number Representation Exploration

Based on all the above modules, we can now perform exploration of different number representations for the FPGA implementation of a specific piece of Fortran code.

The current tools support two different number representations: fixed-point and floating-point numbers (the value simulator for LNS is still in progress). For all the different number formats, the users can also specify arbitrary bit-widths for each different variable.

There are usually a large number of different variables involved in one circuit design. In our previous work, we usually apply heuristic algorithms, such as ASA (Ingber, 2004), to find out a close-to-optimal set of multiple values for the bit-widths of different variables. The heuristic algorithms may require millions of test runs to check whether a specific set of values meet the constraints or not. This is acceptable when the test run is only a simple error function and can be processed in nanoseconds. In our seismic processing application, depending on the problem size, it takes half an hour to several days to run one test set. Thus, heuristic algorithms become impractical.

A simple and straightforward method to solve the problem is to use uniform bit-width over all the different variables to either iterate over a set of possible values or use a binary search algorithm to jump to an appropriate bit-width value.

Based on the range information and the internal behavior of the program, we can also try to divide the variables in the target Fortran code into several different groups, and assign a different uniform bit-width for each different group. For instance, in the 'complex exponential' function, there is a clear boundary that the first half performs square, square root and division operations to calculate an integer result, and the second half uses the integer result as a table index, and performs sine, cosine and complex multiplications to get the final result. Thus, in the hardware circuit design, we divide the variables into two groups based on which half they belong to. Furthermore, in the second half of the function, some of the variables are trigonometric values in the range of $[-1, 1]$, while the other variables represent the seismic image data and scale up to 10^6. Thus they can be further divided into two groups and assigned bit-widths separately.

```
! generation of table step%ctable
```

```
do i=1,size(step%ctable)
    k=ko*step%dstep*dsr%phase(i)
    step%ctable(i)=dsr%amp(i)*cmplx(cos(k),sin(k))
end do

! the core part of function wei_wem

do i4=1,size(wfld,4)
    do i3=1,size(wfld,3)
        do i2=1,size(wfld,2)
            do i1=1,size(wfld,1)

                k = sqrt(step%kx(i1,i3)**2 + step%ky(i2,i4)**2)
                itable =max(1, min(int(1 + k/ko / dsr%d) , dsr%n))
                wfld(i1,i2,i3,i4,i5)=wfld(i1,i2,i3,i4,i5)*step%ctable(itable)

            end do
        end do
    end do
end do
```

CASE STUDY I: COMPLEX EXPONENTIAL

Brief Introduction

The code above is the computationally intensive portion of the FK step in a downward continuation based migration. The governing equation for the FK step is the Double Square Root Equation (DSR) (**?**). The DSR equation describes how to downward continue a wavefield U one depth Δz step. The equation is valid for a constant velocity medium v and is based on the wave number of the source k_s and receiver k_g. The DSR equation can be written as,

$$U(\omega, k_s, k_g, z + \Delta z) = \exp\left[-i\omega v\left(\sqrt{1 - \frac{vk_g}{\omega}} + \sqrt{1 - \frac{k_s v}{\omega}}\right)\right] U(\omega, k_s, k_g, z), \quad (3)$$

where ω is frequency. The code takes the approach of building *a priori* a relatively small table of the possible values of $\frac{vk}{\omega}$. The code then performs a table lookup that converts a given $\frac{vk}{\omega}$ value to an approximate value of the square root.

In practical applications `wfld` contains millions of elements. The computation pattern of this function makes it an ideal target to map to a streaming hardware circuit on an FPGA.

Circuit Design

The mapping from the software code to a hardware circuit design is straightforward for most parts. Fig. 4 shows the general structure of the circuit design. Compared with the software Fortran code shown above, one big difference is the handling of the sine and cosine functions. In the software code, the trigonometric functions are calculated outside of the five-level loop, and stored as a look-up table. In the hardware design, to take advantage of the parallel calculation capability provided by the numerous logic units on the FPGA, the calculation of the sine/cosine functions are merged into the processing core of the inner loop. Three function evaluation units are included in this design, to produce values for the square root, cosine and sine functions separately. As mentioned in earlier, all three functions are evaluated using degree-one polynomial approximation with 386 to 512 uniform segments.

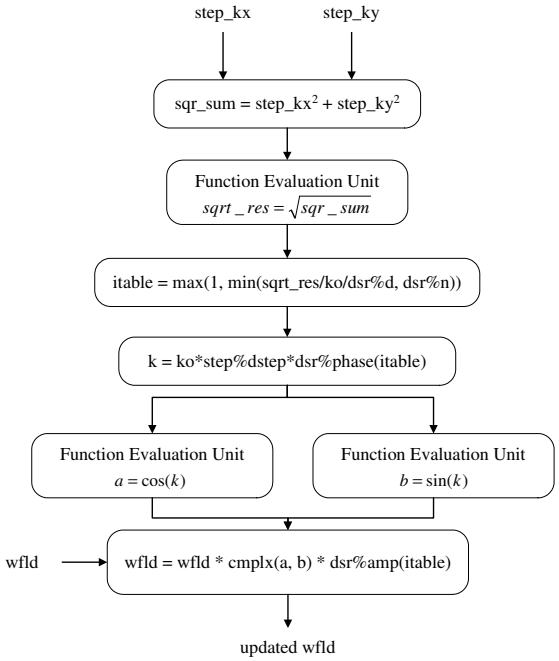


Figure 4: General structure of the circuit design for the 'wei_wem' function.

The other task in the hardware circuit design is to map the calculation into arithmetic operations of certain number representations. The previous table shows the value range of some typical variables in the 'wei_wem' function. Some of the variables (in the part of square root and sine/cosine function evaluations) have a small range within [0, 1], while other values (especially 'wfld' data) have a wider range from 10^{-14} to 10^{6}. If we use floating-point or LNS number representations, their wide representation ranges are enough to handle these variables. However, if we use fixed-point number representations in the design, special handling is needed to achieve acceptable accuracy over wide ranges.

The first issue to consider in fixed-point designs is the division after the evaluation of the square root, $\frac{\sqrt{step\%x^2+step\%y^2}}{ko}$. Suppose the error in the square root result $sqrt\_res$ is E_{sqrt}, and the error in variable ko is E_{ko}, assuming the division unit itself

Variable	step%x	ko	wfld_real	wfld_img
Max	0.377	0.147	3.918e6	3.752e6
Min	0	7.658e-3	4.168e-14	5.885e-14

Table 1: Profiling results for the ranges of typical variables in function 'wei_wem'. 'wfld_real' and 'wfld_img' refer to the real and imaginary parts of the 'wfld' data. 'Max' and 'Min' refer to the maximum and minimum absolute values of variables.

does not bring extra error, the error in the division result is given by $E_{sqrt} \cdot \frac{sqrt\_res}{ko} + E_{ko} \cdot \frac{sqrt\_res}{ko^2}$. As ko holds a dynamic range from 0.007658 to 0.147, and $sqrt\_res$ has a maximum value of 0.533 (variables step%x and step%y have similar ranges), in the worst case, the error from $sqrt\_res$ can be magnified by 70 times, and the error from ko magnified by approximately 9000 times. The values of $step\%x$, $step\%y$ and ko come from the software program as input values to the hardware circuit.

To solve this problem, we perform shifts at the input side to keep the three values $step\%x$, $step\%y$ and ko in a similar range. For ko and the larger value between $step\%x$ and $step\%y$, we perform the shifts so that the leading one of them is just right to the fractional point (in the form of $0.1\cdots$); for the smaller value between $step\%x$ and $step\%y$, we assure it is shifted by the same distance as the larger value. The shifting distance difference between the ko and $step\%x$ is recorded, so that after the division, the result can be shifted back into the correct scale. In this way, the $sqrt\_res$ has a range of $[0.5, 1.414]$ and ko has a range of $[0.5, 1]$. Thus the division only magnifies the errors by an order of 3 to 6. Meanwhile, as the three variables $step\%x$, $step\%y$ and ko are originally in single precision floating-point representation in software, when we pass their values after shifts, a large part of the information stored in the mantissa part can be preserved. Thus, a better accuracy is achieved through the shifting mechanism for fixed-point designs.

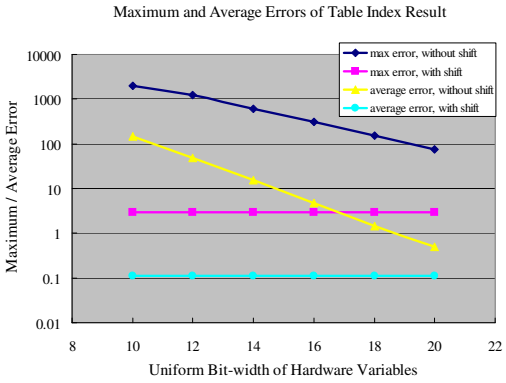


Figure 5: Maximum and average errors for the calculation of the table index when using and not using the shifting mechanism in fixed-point designs, with different uniform bit-width values from 10 to 20.

Fig. 5 shows experimental results about the accuracy of the table index calculation when using shifting or not using shifting, with different uniform bitwidths. The possible range of the table index result is from 1 to 2001. As it is the index for tables with smooth sequential values, an error within five indices is generally acceptable.

We assume that the table index results calculated with double precision floating-point representation are accurate enough and use them as the true values for error processing. When the uniform bit-width of the design changes from 10 to 20, designs using the shifting mechanism show a stable maximum error of 3 and an average error around 0.11. On the other hand, the maximum error of designs without shifting vary from 2000 to 75, and the average errors vary from approximately 148 to 0.5. These results show that the shifting mechanism provides much better accuracy for the part of the table index calculation in fixed-point designs.

The other issue to consider is the representation of 'wfld' data variables. As shown in the table above, both the real and imaginary parts of 'wfld' data have a wide range from 10^{-14} to 10^{6}. Generally, fixed-point numbers are not suitable for representing such wide ranges. However, in this seismic application, the 'wfld' data is used to store the processed image information. It is more important to preserve the pattern information shown in the data values rather the data values themselves. Thus, by omitting the small values, and using the limited bit-width to store the information contained in large values, fixed-point representations still have a better chance to achieve accurate image in the final step. In our design, for convenience of bit-width exploration, we scale down all the 'wfld' data values by a ratio of 2^{-22} so that they fall into the range of [0, 1).

Bit-width Exploration Results

In the first step, we apply uniform bit-width over all the variables in the design. The approach for accuracy evaluation, introduced earlier, is used to provide a value that indicates the quality of the resulted seismic image.

The original software Fortran code of the `S_G_wem` application performs the whole computation using single-precision floating-point. We firstly replace the original Fortran code of function `wei_wem` with a piece of C++ code using double-precision floating-point to generate a full-precision result for comparsion. After that, to investigate the effect of the variables' bit-widths in function `wei_wem` on the accuracy of the whole application, we replace the code of function `wei_wem` with our simulation code that can be configured with different bit-widths, and generate results for different bit-width settings.

As mentioned earlier, according to their characteristics in range and operational behavior, we can also divide the variables in the design into different groups and apply a uniform bit-width in each group. In the design of function 'wei_wem', the variables are divided into three groups: SQRT, the part from the beginning to the table index calculation, which includes an evaluation of the square root; SINE, the part from the end of SQRT to the evaluation of the sine and cosine functions; and WFLD, the part that multiplies the complex values of 'wfld' data with a complex value consisting of the sine and cosine values (for phase modification), and a real value (for amplitude modification). To perform the accuracy investigation, we keep two of the bit-width

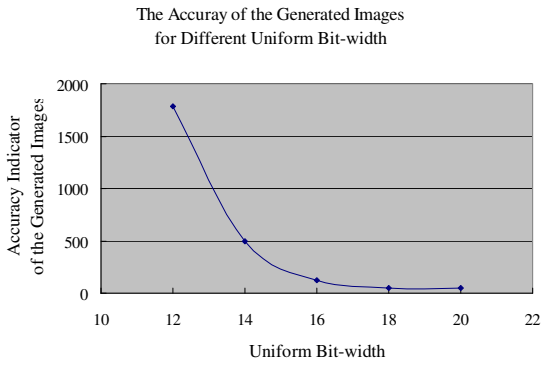


Figure 6: Variation of accuracy for a varying bit-width.

values constant, and change the other one gradually to see its effect on the accuracy of the entire application.

Fig. 7 shows the accuracy of the generated images when we change the bit-width of the SQRT part from 6 to 20. The bit-widths of the SINE and WFLD parts are set to 20 and 30 respectively. Large bit-widths are used for these two parts so that they do not contribute much to the errors and the effect of variables' bit-width in SQRT can be extracted out. The case of SQRT bit-widths shows a clear cut at the bit-width value of 10. For bit-width values smaller than 10, the generated images show a large accuracy indicator value at the level of 10^5, which means the pattern in the generated images are highly different from the correct ones. For bit-width values equal to or larger than 10, the accuracy indicator value drops to the level of 10^2, which indicates a similar accuracy to single-precision floating-point results.

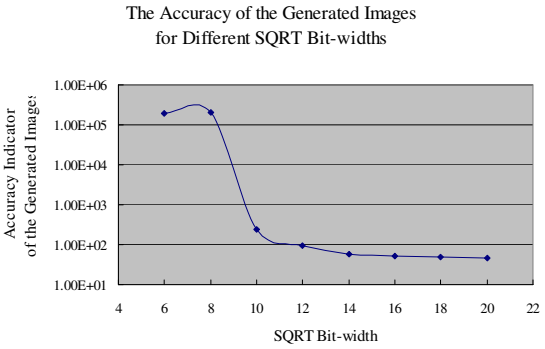


Figure 7: Accuracy of the generated images for different SQRT bit-widths. The accuracy indicator value shows the difference between the pattern in the generated images and the pattern in the full-precision image.

Similarly, Fig. 8 shows the case when we change the bit-width of the SINE part. The SINE bit-width changes from 6 to 20, while the bit-widths of the SQRT and WFLD parts are set to 20 and 30 respectively. There is also a fast decrease at the bit-width value of 8 (not quite evident in a logarithmic scale). The indicator value drops to the level of 10^2 when the bit-width increases to 12.

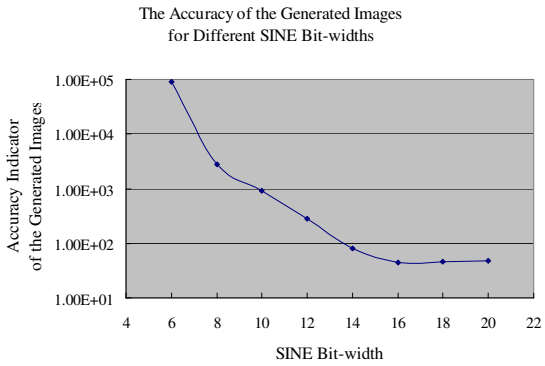


Figure 8: Accuracy of the generated images for different SINE bit-widths. The accuracy indicator value shows the difference between the pattern in the generated images and the pattern in the full-precision image.

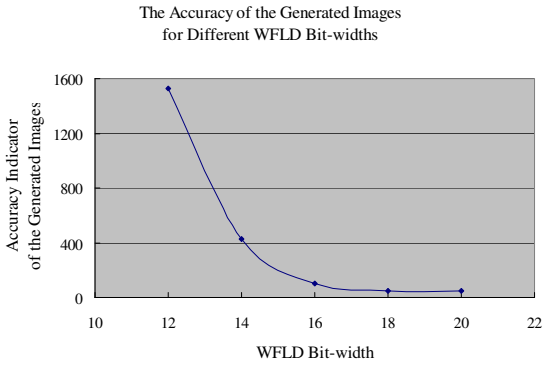


Figure 9: Accuracy of the generated images for different WFLD bit-widths (floating-point). The accuracy indicator value shows the difference between the pattern in the generated images and the pattern in the full-precision image.

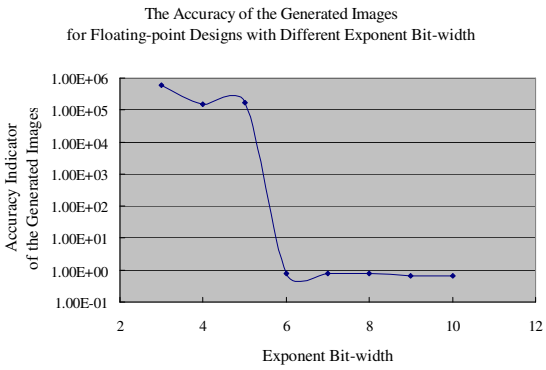


Figure 10: Accuracy of the generated images for different exponent bit-widths. The accuracy indicator value shows the difference between the pattern in the generated images and the pattern in the full-precision image.

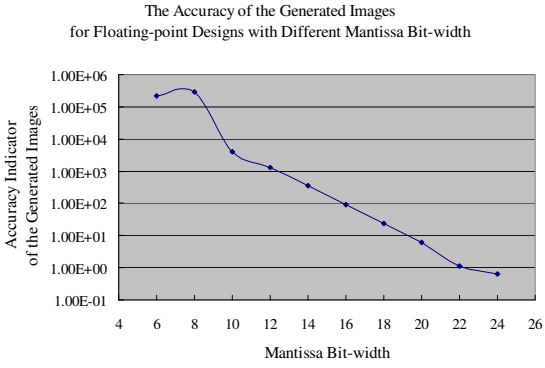


Figure 11: Accuracy of the generated images for different mantissa bit-widths (floating-point). The accuracy indicator value shows the difference between the pattern in the generated images and the pattern in the full-precision image.

Hardware Acceleration Results

The hardware acceleration tool used in this project is the FPGA computing platform MAX-1, provided by Maxeler Technologies. It consists of a high performance Xilinx Virtex IV FX100 FPGA, and provides a high bandwidth interface of PCI Express X8 to the software side residing in CPUs. We found a speedup of 8x compared to the CPU implementation.

CASE STUDY II: CONVOLUTION

To test the speedup potential for reverse time migration we implemented a 6th order acoustic modeling kernel. The 3D convolution uses a kernel with 19 elements. Once each line of the kernel has been processed, it is scaled by a constant factor. We extend the approach to the 2D convolution used last year which works by indexing into the stream to obtain values already sent to the FPGA. These values are stored in BRAM FIFOs, automatically generated and assigned by ASC. The convolution was tested on a data size of $700 \times 700 \times 700$.

The main reason for a speedup is that the processor has limited computational resources. Furthermore, the processor uses floating-point units as opposed to fixed-point units. We exploit the parallelism of the FPGA to calculate one result per cycle. When ASC assigns the elements to BRAMs it does so in such a way as to maximize the number of elements that can be obtained from the BRAM every cycle. This means that consecutive elements of the kernel must not in general be placed in the same BRAM. Since we can use variable precision, we reduce the computation overhead, increasing the throughput. To compute the entire computation all at the same time (as is the case when a high-performance processor is used) requires a large local memory (in the case of the processor, a large cache). The FPGA has limited resources on-chip (376 BRAMs which can each hold 512 32-bit values). To solve this problem we break the large data-set into cubes. To utilize all of our input and ouput bandwidth, we assign 3 processing cores to the FPGA resulting in 3 inputs and 3 outputs per cycle at 125MHz (constrained by the throughput of the PCI-Express bus). This gives us a theoretical maximum throughput of 375M results a second.

The disadvantage to breaking the problem into smaller blocks is that the boundaries of each block are essentially wasted (although a minimal amount of reuse can occur) because they must be reused when the adjacent block is calculated. We do not consider this a problem since the blocks we use are at least $100 \times 100 \times 700$ which means only a small proportion of the data is resent. The amount of BRAM assigned to each block is calculated as follows:

$$\left\lfloor Total\ BRAM \times \left\lfloor \frac{Input\ bandwidth}{Input\ precision} \right\rfloor \right\rfloor \tag{4}$$

which assumes that the output precision is the same as the input precision. From

this we can work out the size of the block. In our case we get $\lfloor 376 * \lfloor (64/21) \rfloor \rfloor = 125$. Due to the number of multipliers and adders required, we cannot fit 3 cores onto the FPGA directly because the number of slices used would be too high. If all of the operations are assigned to the DSP blocks we wouldn't have enough. We therefore choose a hybrid approach in which we break each multiply into 2 parts. We use one 18-bit hard multiplier (1 DSP block) and put the rest of the calculation (3 smaller multipliers) directly into logic.

In software, the convolution we try to accelerate executes in 11.2 seconds on average. The experiment was carried out using a dual-processor machine (each quad-core Intel Xeon 1.86GHz) with 8GB of memory.

In hardware, using the MAX-1 platform we obtain a 5 times speedup. The design uses 48 DSP blocks (30%), 369 (98%) RAMB16 blocks and 30,571 (72%) of the slices on the Virtex -4 chip. This means that there is room on the chip to substantially increase the kernel size. For a larger sized kernel (31 points) the speedup should be virtually linear resulting in a 8x speedup compared to the CPU implementation.

FURTHER POTENTIAL SPEEDUPS

All of the speedups in this paper include the transfer time to and from the processor. If multiple portions of the algorithm are performed on the FPGA without returning to the CPU the additional speedup can be considerable. In the cases shown in this paper the limiting factor is the transfer time. For example if the FFT and FK step can reside simultaneously on the FPGA the cost of the FK step disappears. In the case of acoustic modeling multiple time steps could be applied simultaneously.

CONCLUSIONS

We describe a software methodology for implementing and evaluating algorithmic performance on a FPGA. We found a 8x speedup in implementing (including transfer time) the FK step of downward continued migration on FPGA. In addition we found a 5-8x speedup in implementing a acoustic 3-D convolution kernel.

ACKNOWLEDGMENTS

This project is a joint research effort between the Center for Computational Earth and Environmental Science, Stanford Exploration Project, Computer Architecture Research Group at Imperial College London, and Maxeler Technologies.

REFERENCES

Berenger, J., 1994, A perfectly matched layer for the absorption of electromagnetic waves: Journal of Computational Physics, **114**, 185–200.

Biondi, B. and G. Palacharla, 1996, 3-d prestack migration of common-azimuth data: Geophysics, 1822–1832, Soc. of Expl. Geophys.

Constantinides, G. A., P. Y. K. Cheung, and W. Luk, 2001, Heuristic Datapath Allocation for Multiple Wordlength Systems: Proc. Design Automation and Test in Europe Conference (DATE), 791–796.

Fu, H., O. Mencer, and W. Luk, 2006, Comparing Floating-point and Logarithmic Number Representations for Reconfigurable Acceleration: Presented at the Proc. IEEE International Conference on Field-Programmable Technology (FPT).

———, 2007, Optimizing Logarithmic Arithmetic on FPGAs: Presented at the Proc. IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM).

Gazdag, J. and P. Sguazzero, 1985, Migration of seismic data by phase shift plus interpolation, 323–330. Society Of Exploration Geophysicists.

Ingber, L., 2004, Adaptive simulated annealing (asa) 25.15. http://www.ingber.com.

Lee, D.-U., A. A. Gaffar, R. C. Cheung, O. Mencer, W. Luk, and G. A. Constantinides, 2006, Accuracy Guaranteed Bit-Width Optimisation: IEEE Trans. Computer-Aided Design, **25**, 1990–2000.

Lee, D.-U., A. A. Gaffar, O. Mencer, and W. Luk, 2005, Optimizing Hardware Function Evaluation: IEEE Trans. Comput., , no. 12.

Muller, J., 1997, Elementary functions: algorithms and implementation: Birkhauser Boston, Inc.

Pell, O. and R. G. Clapp, 2007, Accelerating subsurface offset gathers for: SEP-Report, **129**, 253–260.

Shan, G. and B. Biondi, 2007, Angle domain common image gathers for steep reflectors: SEP-Report, **131**.