Automatic Floating to Fixed Point Translation and its Application to Post–Rendering 3D Warping

M.P. Leong, M.Y. Yeung, C.K. Yeung, C.W. Fu, P.A. Heng and P.H.W. Leong
{mpleong,myyeung2,ckyeung,cwfu,pheng,phwl}@cse.cuhk.edu.hk
Department of Computer Science and Engineering
Chinese University of Hong Kong
Shatin, N.T. Hong Kong

Abstract

The automatic conversion of floating point software implementations of algorithms to a equivalent fixed point implementation which can be efficiently implemented in an FCCM remains an obstacle in the rapid systems prototyping design flow. Floating point to fixed point conversion is tedious, error prone and requires a good knowledge of fixed point computer arithmetic. This paper describes a software system called fp *designed to automate the process. It consists of a fixed point C++ class; a profiler which is used to determine the number of bits of precision required for each signal in the hardware implementation; an optimiser which finds the minimal number of bits required for a specified degree of accuracy in the implementation and finally and a compiler which takes the information collected by the system and outputs synthesisable VHDL code. A post-rendering 3D image warping application designed using this system is used as an example.*

1 Introduction

High level synthesis techniques have significantly improved productivity for designers of field-programmable custom computing machines (FCCM) in recent times, allowing them to concentrate on higher level design issues. However, for even larger productivity gains, the problem of converting algorithmic descriptions, which are typically implemented using a general purpose programming language such as C, to a hardware efficient fixed point description needs to be addressed. This is typically done by the designer manually translating a floating point description to a fixed point one, observing the ranges of variables and specifying enough bits in the fixed point implementation to ensure that the correct results are obtained. This process is tedious, time consuming and requires a good knowledge of fixed point arithmetic.

It is likely that a computer optimised fixed point implementation of a floating point algorithm would require less hardware than a human optimised version. This is because a designer is unlikely to reduce the wordlength of each variable to the absolute minimal value, and also, the computer can make low level optimisations which a human designer may find too tedious to apply. Using automatic translation tools, users that are not intimately familiar with fixed point arithmetic to design an FCCM, in the same way that programmers, not intimately familiar with how to implement, say the *log* function, can apply it through a math library.

The tools that we have developed, called *fp*, center around a profiler which collects information on the range that a C variable will take during execution of a program with fixed inputs and outputs. In the general case, it is not possible to know the range of an arbitrary variable inside a program since it is dependent on the inputs. However, for a given a set of inputs, the profiler can record the range of all the variables. The profiler collects information about each of the variables and then makes an initial, conservative estimate for the range of each variable.

Quantisation effects will, in general, degrade the performance of an algorithm. Increasing the wordlength of the fixed point arithmetic reduces quantisation effects at the cost of increased hardware. In many applications, it is possible to describe this tradeoff in terms of a cost function. For example, a compression algorithm might require that the outputs be identical for both fixed and floating point versions and a suitable cost function could be a weighted sum of the number of bytes which are different and the total wordlengths of all variables. The optimiser tool in *fp* minimises a user supplied cost function by changing

0-7695-0375-6/99 $10.00 © 1999 IEEE

the wordlengths of the variables in the program.

Finally, the program is translated from C into synthesisable VHDL code with the wordlengths discovered by the optimiser. It is also possible for the user to specify wordlengths for any variable, overriding the computer generated values.

Commercial design environments supporting the modeling of fixed point system have been mainly based upon block diagrams descriptions (e.g. [1, 2, 3]), blocks being implemented as parameterised libraries which can operate at different word lengths. Although this dataflow approach has the advantages of being very simple and intuitive, it is not possible to change the functionality of the blocks and it is difficult to express control intensive applications in these systems.

More general purpose design environments based on programming languages such as Mentor Graphics' Data Flow Language (DFL) [2] and C/C++ [4, 5] have concentrated on targeting fixed point DSP chips which have a fixed wordlength of between 16 and 32 bits. In this work, optimisation of wordlengths for hardware synthesis is addressed. All signals can be of different wordlengths and the objective is to minimise the hardware while at the same time meeting some quality criteria.

The main goal of the *fp* system is to allow for the automatic translation of arbitrary floating point algorithms to the equivalent minimal wordlength fixed point implementation. It has the following features

- no knowledge of fixed point arithmetic is required to use the system
- wordlengths can be different for each signal in the final implementation
- a user-supplied cost function is optimised which takes into account tradeoffs between wordlength and implementation quality
- a parameterised library of fixed point mathematical functions such as addition, multiplication, exponential, square root and reciprocal are provided
- output is in synthesisable VHDL, hardware optimisations being automatically performed in the process.

In Section 2 we will describe a post-rendering 3D image warping algorithm which is used as an example in this paper. A description of the *fp* floating point tools which were developed to support both automatic translation of floating point to fixed point code and manual analysis of floating point systems will be presented in Section 3. Section 4 describes the results of the *fp* system when applied to the 3D image warping problem and finally, conclusions are drawn in Section 5.

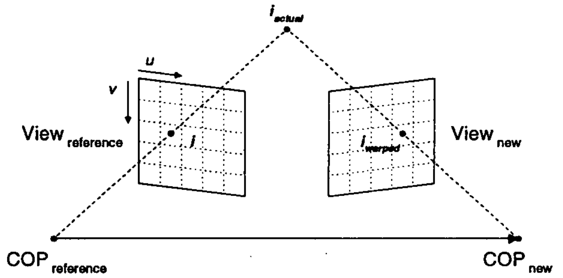

Figure 1: Post-rendering 3D warping : Warping of pixel i to i_{warped} (COP is center of projection)

2 Post-rendering 3D Image Warping

Traditional rendering systems use linearly interpolated triangles to render a 2D graphical image from a complex 3D internal representation so as to obtain realistic rendering result. The computational requirement is strongly dependent on the scene complexity. To address this problem, an image-based rendering system was introduced by McMillan and Bishop [6] and after that, post-rendering 3D warping [7] was proposed. Post-rendering 3D warping provides an order of magnitude improvement in apparent frame-rates over conventional rendering, with computational requirements not dependent on scene complexity but rather on the image size.

The post-rendering 3D warping algorithm is illustrated in Figure 1. The inputs to the algorithm are a reference image and its corresponding depth map. Each pixel is projected to its actual 3D space and then reprojected to the new view plane. The technique is typically used in conjunction with traditional rendering and the post-rendering 3D warping used for the small view angle changes between frames [7]. This rendering technique is very effective and has been an active field of computer graphics research [8, 9, 10, 11].

Figure 2 shows two different views of the same scene obtained using simple post-rendering 3D warping. The resultant image have dark lines which are caused by undersampling. This problem can be overcome by splat reconstruction or mesh reconstruction [7]. One other difficulty which must be addressed by the 3D warping problem is that there is no information about objects occluded in the reference image but which might appear in the new view. This problem, called the occlusion problem, can be handled by having multiple reference images [11] or layered depth images [9].

The post-rendering 3D warping algorithm is simple so its hardware and memory requirements are modest and high bandwidth memory is not necessary. These two features make it suitable for implementation on an FCCM. The undersampling and occulsion problems were not addressed in our implementation.

2.1 Algorithm

In our implementation, the 3D representation of the image is computed in software and the FCCM is responsible for the 3D warping process which is computationally intensive. We use a perspective projection model with image plane (reference view).

In the method described below, we denote the new viewpoint (new center of projection) as v and the new view plane's top left, top right, bottom left and bottom right points A_{00}, A_{10}, A_{01} and A_{11} respectively as shown in Figure 3. The vectors from v to A_{00}, A_{10}, A_{01} and A_{11} are named as $\vec{v}_{00}$, $\vec{v}_{10}$, $\vec{v}_{01}$ and $\vec{v}_{11}$ respectively.

For every pixel,

1. After projecting it from the reference image to its actual 3D position, we can have its actual position, $\vec{p}$ w.r.t. the coordinate system of the new viewpoint by a simple vector calculation.

2. Next, to find the projected position of $\vec{p}$ on the new view plane, we first parameterise the new view plane by

$$\begin{array}{rcccccc} \vec{I}(h,k) & = & (1-k) & ((1-h)\vec{v}_{00} & + & h\vec{v}_{10}) \\ & + & k & ((1-h)\vec{v}_{01} & + & h\vec{v}_{11}) \end{array} \quad (1)$$

3. By equating

$$\lambda\vec{p} = \vec{I}(h,k) \quad (2)$$

we will have three real unknown numbers, λ, h and k.

4. Since the above equation are in three dimensional space, we can obtain three independent equations along the x, y and z dimensions which can be solved to obtain h and k.

5. If the values of h and k are within $[0,1]$ then the pixel will be splated onto the new image plane at coordinate $(resx \times h, resy \times k)$ where $resx$ and $resy$ is the new view's resolution.

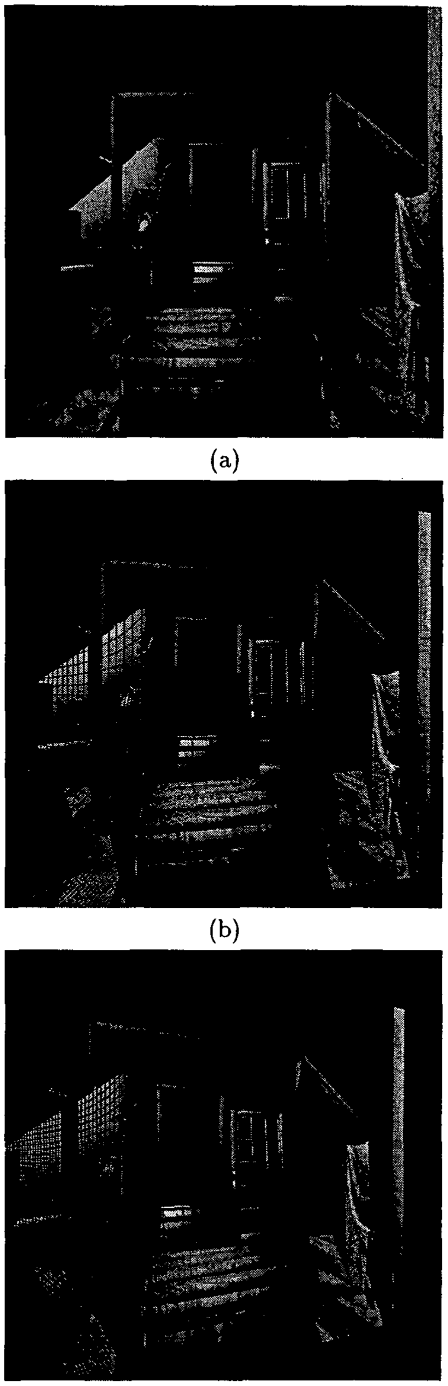
(a)
(b)
(c)

Figure 2: Original (a) and warped images (b) and (c).

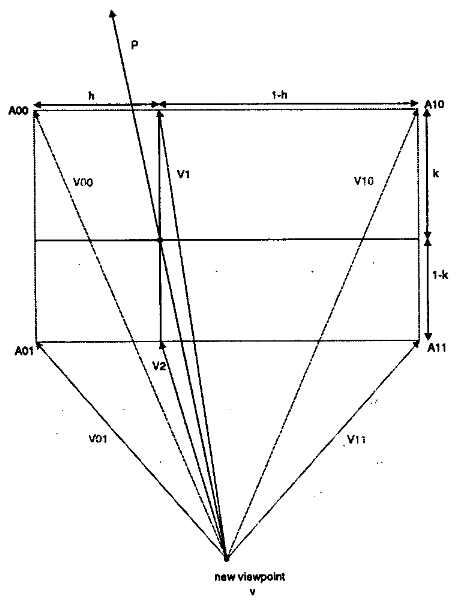

Figure 3: Framework of reprojecting w.r.t. a new viewpoint.

In a software implementation, all of the operations are performed using floating point arithmetic. Although the algorithm is very simple, it is not straightforward to implement it using fixed point arithmetic since

- it is difficult to know how many bits of precision are required for λ, h and k
- it is tedious and difficult to implement fixed point division, fractional multiplication and square root functions
- tradeoffs in time, area and quality need to be considered both for the algorithm and the fixed point operators.

3 Fixed Point Tools

Fp consists of a C++ class called "Fixed" for the representation of fixed point values; a profiler that determines the number of bits of precision of every value in a fixed point function; an optimiser that interacts with the profiler to minimise the total number of bits of precision while retaining a certain degree of accuracy; a simulator that allows users to verify the results when using the optimised precisions; and a compiler

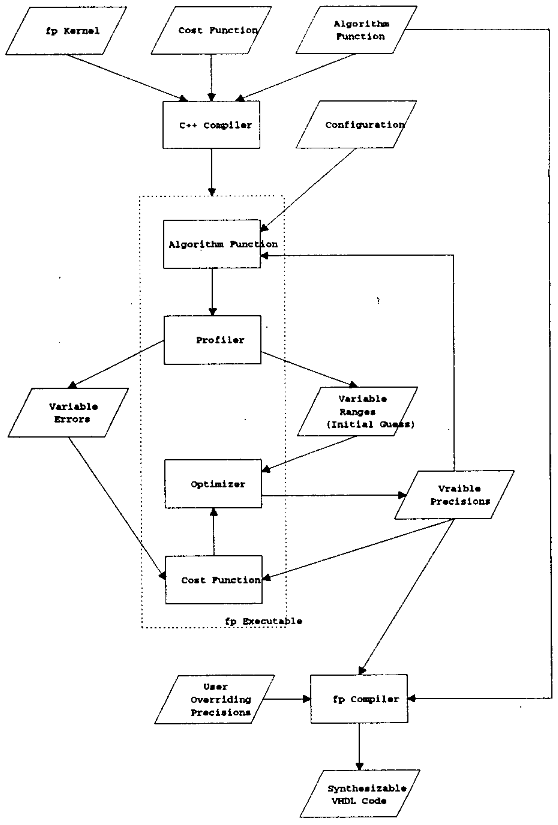

Figure 4: *Fp* tools design flow.

that translates the C function and the required number of bits of precision from the profiler into synthesisable VHDL code.

Figure 4 illustrates the *fp* design flow. A design begins by writing a C function to represent the algorithm to be implemented in hardware. We will call this function the "algorithm function". Variables (both integer and fixed point) in the algorithm function should be declared as Fixed, which is a C++ class that is capable of representing a fixed point value under different precisions. The function takes a number of Fixed variables as input and produces a number of Fixed variables as the output. Users are also required to supply a configuration file, which specifies the precision formats of the inputs variables. This information is useful for worst case analysis and is used as the initial guess for the optimiser.

The optimiser routine is used to minimise a user–defined cost function. The cost function usually reflects the total number of bits in the design and the error between the floating point and fixed point computations. During optimisation, the optimiser modifies the precision formats of the algorithm function's Fixed variables to minimise the cost function. During execution, the profiler routine can be called by the cost function to extract the precision information of any of the Fixed variables.

The algorithm function, cost function, profiler routine, optimisation routine and Fixed class are combined to produce a single C++ program. When this program is compiled and executed it performs optimization and profiling on itself (see Figure 4), outputting optimised precision information for each variable used in the algorithm function.

Finally, the compiler uses the algorithm function together with the precision information generated by the profiler to generate synthesisable VHDL code.

3.1 Fixed Class

The built-in C data types such as *int* and *double* support only pre-defined wordlengths. The pre-defined lengths for each data type are determined by the compiler and computing platforms. These built-in data types are not suitable for describing fixed point values where wordlengths vary from one variable to another. To describe hardware that utilises fixed point computation, a data type that supports variable bit lengths must be introduced.

In the *fp* system, a "Fixed" C++ class is used to implement this data type. The reasons for using C++ are [12]

- C++ supports class operators so that if x and y are variables of type Fixed, it is possible to write x+y instead of add(x,y)
- C++ supports overloading so that implicit type conversions can be made (e.g. we can write expressions such as x + 1.5 and the constant will be implicitly converted to Fixed)
- C++ supports exception handling so that either standard or user overridden exception handlers can be called in the case of arithmetic exceptions such as division by zero and taking the square root of a negative number

A Fixed variable consists of the following major components

1. the precision of the value in (ilen.flen) format, where ilen is integer wordlength and flen is fractional wordlength
2. the range of the variable (minrange, maxrange) – this is a worst case information which is collected by the profiler but not used by the compiler
3. a value fval that represents the value under the current precision
4. a value dval that represents the floating point value of the variable
5. the actual minimum and maximum values that have been assigned to the variable (minval, maxval)

For worst case analysis, the two class methods, SetLen and SetRange, allow users to specify the precision and range of a Fixed value respectively. For example, f.SetLen(8, 4) sets the integer and fractional wordlengths of variable f to 8 and 4 respectively; g.SetRange(-2.5, 3.5) sets $-2.5 \leq g \leq 3.5$. When SetLen is applied to a variable, the corresponding range is automatically computed but users can override this using SetRange. Similarly, when SetRange is applied to a variable, the corresponding integer wordlength is determined but the fractional wordlength remains unassigned. Users can apply both SetLen and SetRange on the same variable provided they do not conflict.

The Fixed class provides support of basic arithmetic operations, such as addition and subtraction, multiplication and division, square, square root, and absolute value. More complex functions such as *log* and *exp* can be also included in libraries. When an Fixed class operator is applied, the following four operations are carried out

1. the worst case range of the result is determined
2. the worst case precision of the result is determined
3. the exact fixed point result value under the degree of precision calculated in (2) is computed
4. the floating point result value is computed

As an example, consider the addition of two Fixed variables, a and b with precisions (8.6) and (4.8) respectively, the result being assigned to the Fixed variable c. The worst case result would require a precision of (9.8) to guarantee no loss of precision. However, in practice, the worst case estimate is too pessimistic. The maximum and minimum values that a variable takes during execution of the algorithm function is a best case estimate. Profiling may discover that during the course of program execution the variable takes on the range (-12.0, 10.0). Then a precision of (5.8) would be sufficient to store the result. The precision is overridden when the user applies SetLen or SetRange to c.

The fractional wordlength of a variable affects its precision only. However, the integer wordlength affects the range that the value can represent. When a manually set integer wordlength is not able to represent a value, overflow occurs and a flag is set to indicate this situation. There is also a not-a-number (NaN) flag which is set after, for example, taking the square root of a negative number.

The constructor function for a Fixed variable causes its creation to be recorded in a global data structure called *Fixedvars*. It is thus possible for other parts of the *fp* system to deduce how many variables exist in the algorithm function, their ranges and precisions etc.

3.2 Profiling

The profiler serves to extract the range and error of each Fixed variable in the algorithm function after its execution. The range is defined by the minimum and maximum value that a Fixed variable was assigned, while the error is defined as the absolute difference between the value using fixed point and floating point representations.

The range information collected by the profiler is useful for estimating the initial guess, while the error information is used by the cost function.

3.3 Optimisation

The optimiser uses the method of Nelder–Mead [13] to minimise a nonlinear cost function without requiring the computation of derivatives. It obtains its initial guess from the profiler and adjusts the precisions of every Fixed variable inside the function. The algorithm function is called twice for every pass of the optimisation routine. The first call is used for range extraction and the second for error extraction.

Due to quantisation effects, different sets of fractional wordlengths result in different sets of ranges for Fixed variables. Before calling the algorithm function, the optimiser sets all Fixed variables to a large integer wordlength thus preventing overflow. After the execution of the algorithm function, the optimizer determines the minimal integer wordlengths according to the runtime minimum and maximum values and computes error statistics on the Fixed variables. This information is then passed to the cost function.

Different applications require varying degrees of accuracy. The allowable degradation of an algorithm is specified in the cost function. Commonly, cost functions are a weighted sum of the the total number of bits in the fixed point algorithm and the error at the output. Different cost functions allow tradeoff between wordlengths and output errors.

3.4 Simulator

The simulator sets the wordlengths of variables in the algorithm function to the optimised wordlengths prior to executing it. It allows users to verify the results from the optimiser.

3.5 Compiler

The compiler was constructed by modifying the code generator of an ANSI C compiler, *lcc* [14]. The code generator currently can handle most C declarations, expressions and conditionals. Loops are not currently supported.

The compiler begins by constructing the expression tree from the algorithm function. A tree node corresponds to either a register or an operator. The bit-widths of registers are obtained from the output of the optimiser and the bit-widths of operators hence determined. The operators all have different latencies and a time-alignment technique [15] which adds stage latches into the expression tree is used to address this problem. The stage latches are constructed from linear feedback shift registers [16].

A library of parameterised synthesisable VHDL modules are associated with the compiler. In each case, minimal area implementations were chosen in order to reduce hardware resources. The addition and subtraction operators are implemented using the appropriate VHDL operators. N bit multiplication is implemented using a shift/add algorithm in order to minimise hardware requirements. The fixed point division is implemented using restoring division [17] and the square root function is implemented by "completing the square" [17].

4 Results

4.1 Fixed Point Optimisation

A post-rendering 3D warping routine was written in C++ and all integer and floating point variables changed to be of the Fixed class. The routine used for the hardware implementation uses a sequential warping order rather instead of that of Figure 1 in order to allow for burst mode memory accesses in the hardware implementation. The design flow of Figure 4 was followed, most of the effort and time being spent deciding on a suitable cost function.

The cost function consists of the weighted sum of the following components

- wordlength – the sum of wordlengths of all variables
- variable error – the sum of errors of all variables
- output error – the sum of errors of all outputs

The weighted sum is then multipled by the the percentage of outputs that exceed the error bound.

In order to save memory, we restricted the wordlengths of the 3D representation of the image to 16 bits in order to limit the external memory requirements of the algorithm. This constraint made it impossible to achieve a pixel exact correspondence between the floating and fixed point implementations.

A total of 35 Fixed variables were used in the 3D warping program. After optimisation, the total wordlength of the result was 723 bits, an average of approximately 21 bits per variable. The average errors at the outputs h and k (as described in Section 2.1) were 4.191e-4 and 5.248e-4 respectively. The maximum errors were 9.798e-4 and 1.016e-3 respectively.

The algorithm function was automatically translated to VHDL code and implemented using the Synopsys' FPGA Compiler with the Xilinx XC4085XL FPGA as the target device. A manually designed memory interface was added to interface the generated core design with an external memory.

The resulting images from the floating point and fixed point implementations are shown in Figure 5(a) and (b). In order to explore cost/performance tradeoffs, the weightings of the optimisation cost function were changed to get smaller area at the expense of image quality. This resulted in the image of Figure 5(c) which has a total wordlength of 501 bits (an average of 14 bits per variable). The number of CLBs required were 3099 and 2334 for the 723 and 501 bit implementations respectively and the Xilinx software's estimated maximum frequency for both versions was 20 MHz.

Due to our choice of operators which reduced area rather than time, throughput of the design was limited by the operator with the longest latency. The execution time of the 723 and 501 bit implementations were 37 and 33 cycles per pixel respectively. Both

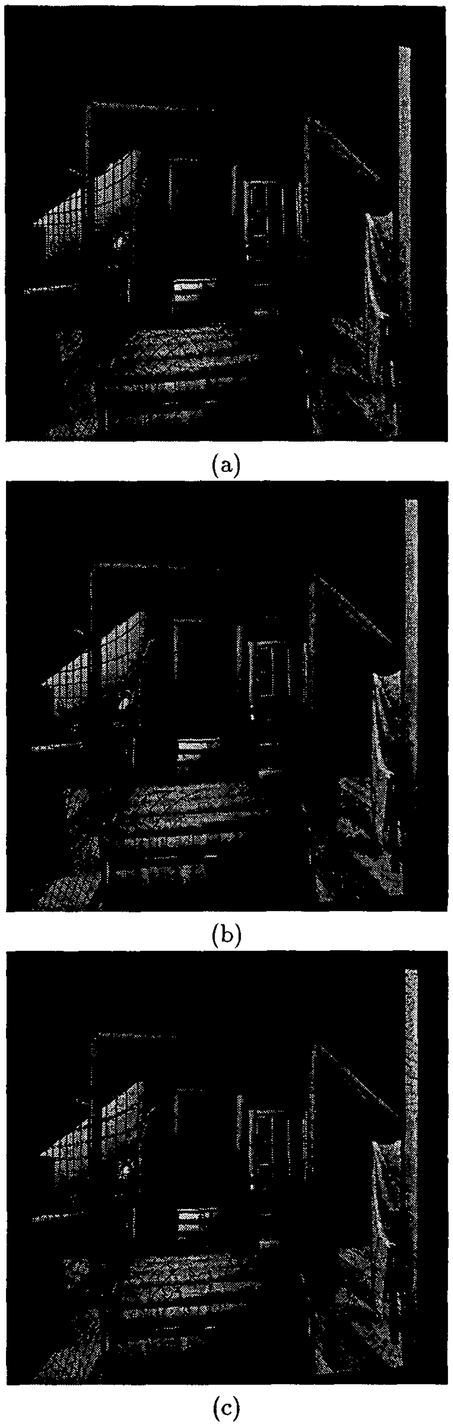

Figure 5: Post–rendering 3D warped images obtained from (a) Floating point (b) fixed point results using a total wordlength of 723 bits (c) fixed point results using a total wordlength of 501 bits.

implementations were successfully tested at 40MHz on an Annapolis Micro Systems Wildfire board, and achieved a frame rate of $40\text{MHz}/(37*512*512) = 4.12$ fps and $40\text{MHz}/(33*512*512) = 4.62$ fps for a image size of 512×512 pixels. This can be compared with an execution time of 3.70 fps on an Sun Ultra-5 270MHz machine with 512MB of RAM. The speed improvement was insignificant because the throughput was limited by our implementation of the multiply, divide and square root fixed point operators.

We are working on the optimisation of the generated VHDL code and using pipelined computational modules instead of folded ones. The pipelined implementation, with an estimated size of 6000 CLBs, should fit in larger devices such as Xilinx XC40150XV FPGA. The throughput of a pipelined implementation is limited by the memory bandwidth. For each pixel, 3×16 bits are used to represent the 3D location of the object in space and another 16 bits are used to represent the RGB value of the pixel. A 16 bit RGB result is also written to memory. Thus in total 2.5 32 bit memory accesses are required for each pixel, corresponding to 7 clock cycles if non-burst mode accesses are assumed. At a 40MHz clock rate, a frame rate of $40\text{MHz}/(7*512*512) = 21.80$ fps can be achieved.

5 Conclusion

A flexible design environment which supports the automatic translation of floating point algorithms to a hardware implementable fixed point representation was presented. These tools enable FCCM designers to concentrate on higher level algorithmic issues thus increasing productivity and being able to explore more of the design space in a given time. The tools were successfully applied to a post-rendering 3D warping application in which synthesisable VHDL code was generated automatically from a floating point description written in C.

6 Acknowledgements

This work was supported by a Chinese University of Hong Kong direct grant.

References

[1] Alta Group Inc., *HDS User's Manual.*

[2] Mentor Graphics., *DSP Station User's Manual.* 1995.

[3] The Mathworks Inc., *Simulink Reference Manual.* 1998.

[4] S. Kim, K. Kum, and W. Sung, "Fixed point optimization utility for c and c++ based digital signal processing programs," in *Workshop on VLSI and Signal Processing*, (Osaka, Japan), pp. 197–206, 1995.

[5] M. Willems, V. Bursgens, H. Keding, T. Grotker, and H. Meyr, "System level fixed-point design based on an interpolative approach," in *Proceedings of the 34th Design Automation Conference*, (Anaheim, California), pp. 293–298, 1997.

[6] L. McMillan and G. Bishop, "Plenoptic modeling: An image-based rendering system," in *Computer Graphics Proceedings, Annual Conference Series (Proc. SIGGRAPH '95)*, pp. 39–46, August 1995.

[7] W. Mark, L. McMillan, and G. Bishop, "Post-rendering 3d warping," in *Symposium on Interactive 3D Graphics*, (RI), pp. 7–16, 1997.

[8] P. Rademacher and G. Bishop, "Multiple-center-of-projection images," in *Computer Graphics Proceedings, Annual Conference Series (Proc. SIGGRAPH '98)*, pp. 199–206, July 1998.

[9] J. Shade, S. Gortler, L. wei He, and R. Szeliski, "Layered depth images," in *Computer Graphics Proceedings, Annual Conference Series (Proc. SIGGRAPH '98)*, pp. 231–242, July 1998.

[10] W. R. Mark and G. Bishop, "Memory access patterns of occlusion-compatible 3d image warping," in *Proceedings of the 1997 Siggraph/Eurographics Workshop on Graphics Hardware*, pp. 35–44, August 1997.

[11] M. M. Rafferty, D. G. Aliaga, V. Popescu, and A. A. Lastra, "Images for accelerating architectural walkthroughs," in *IEEE Computer Graphics and Applications, Vol. 18, No. 6*, pp. 38–45, November/December 1998.

[12] B. Stroustrup, *The C++ Programming Language.* Addison–Wesley, 3rd ed., 1997.

[13] J. Nelder and R. Mead, "A simplex method for function minimization," *Computer Journal*, vol. 7, pp. 308–313, 1965.

[14] C. Fraser and D. Hanson, *A retargetable C Compiler: Design and Implementation.* Addison Wesley, 1995.

[15] R. Hartley, *Digit-serial computation.* Kluwer Academic Publishers, 1995.

[16] P. Alfke, "Efficient shift registers, LFSR counters, and long pseudo-random sequence generators," tech. rep., Xilinx Inc, 1996. Application Note XAPP 052.

[17] I. Koren, *Computer Arithmetic Algorithms.* Prentice-Hall, 1993.