Correction of Geometric Image Distortion Using FPGAs

David Eadie, Fergal Shevlin, Andy Nisbet

Department of Computer Science,
Trinity College, Dublin 2,
Ireland.

1. INTRODUCTION

Many image processing systems have real-time performance constraints. Systems implemented on general purpose processors maximise performance by keeping busy the small fixed number of available functional units such as adders and multipliers. In this paper we investigate the use of programmable logic devices to accelerate the execution of an application. Field Programmable Gate Arrays (FPGAs) can be programmed to generate application specific logic that alters the balance and type(s) of functional units to match application characteristics. In this paper we introduce a *correction of geometric image distortion* application. Real number support is a requirement in most image processing applications. We examine the suitability of fixed point, floating-point and logarithmic number systems for an FPGA implementation of this image processing application. Performance results are presented in terms of: i) execution time, and ii) FPGA logic resource requirements.

2. GEOMETRIC DISTORTION

Geometric distortion exists when the spatial relationships between pixels in the image do not equate to the spatial relationships between corresponding points in the scene. Such distortion is inevitable due to the perspective projection effected by normal lenses. Telecentric lenses are often used to effect near-parallel projection in machine vision metrology applications in order to minimise distortion. However some distortion will still exist due to imperfections in lens design and configuration.

Figure 1(a) illustrates simulated geometric distortion of a calibration target consisting of a regular grid of white spots on a black background, illuminated with monochromatic red light. Figure 1(b) shows a particular problem which arises when forming a colour image from several others acquired sequentially under different monochromatic illuminations. In this case the different wavelength-specific geometric distortions are commonly refered to as chromatic aberration.

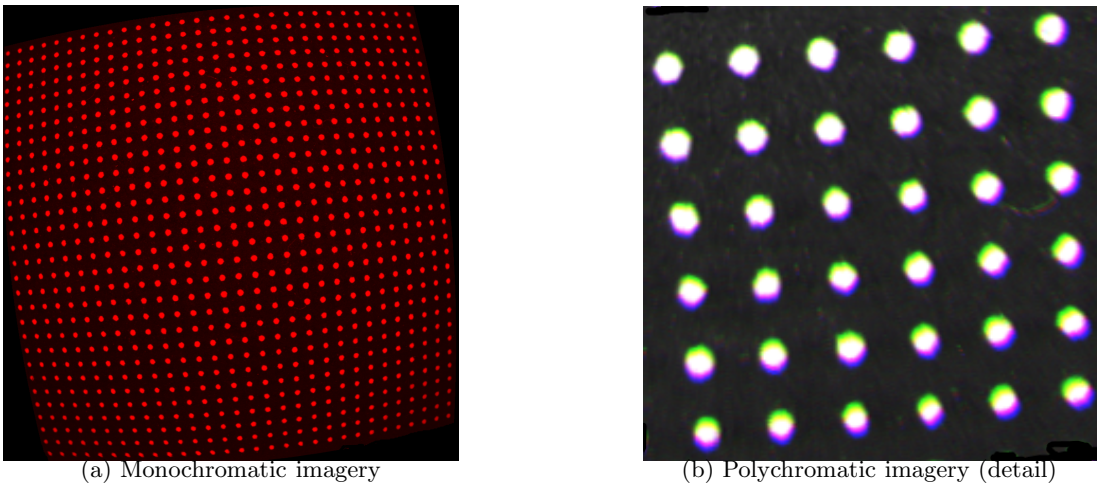
The distortion due to a reasonable-quality commercially-available telecentric lens has been found to be about 1% towards the periphery of the image—about 10 pixels in a 1024×1024 pixel image. The effect of this in actual polychromatic imagery is shown in figure 2(a).

3. GEOMETRIC RECTIFICATION

Rectification or correction of geometric distortion requires the synthesis of a new image through the transformation or warping of the acquired image, in accordance with some known inverse model of the distortion. For practical purposes the best way to implement this is through an *inverse mapping* strategy[1]—each pair of integer pixel coordinates in the corrected image is mapped to a pair of not-necessarily-integer coordinates in the image to be rectified. A pixel value for the corrected image is usually found through bi-linear interpolation of the pixel values in the neighbourhood of the mapped coordinates. A rectified version of the image shown in figure 1(a) is shown in figure 2(b).

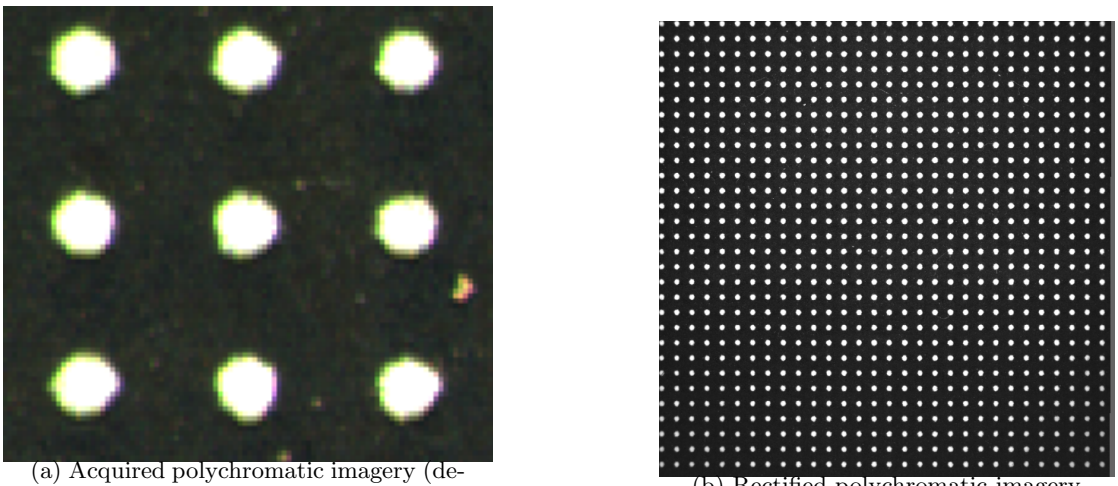
The model of distortion used needs to be accurate. Lens distortion is often radial about the optical axis and can be modelled with low-order polynomials. The coefficients of these polynomials can be found through

Further author information—Email: David.Eadie@cs.tcd.ie; Telephone: +353-1-608-1435; Fax: +353-1-677-2204.



(a) Monochromatic imagery (b) Polychromatic imagery (detail)

Figure 1. Simulated geometric distortion (exaggerated)



(a) Acquired polychromatic imagery (detail) (b) Rectified polychromatic imagery

Figure 2. Acquired and rectified imagery

a process of calibration where the distorted image coordinates of points on a target are associated with their precisely-known coordinates.[2] The relationship between the distorted image coordinates (f, g) and their rectified homologues (u, v) is expressed as follows, where n is the order of the polynomial,

$$f = \sum_{i=0}^{n} \sum_{j=0}^{n-i} a_{ij}\, u^i v^i$$

$$g = \sum_{i=0}^{n} \sum_{j=0}^{n-i} b_{ij}\, u^i v^i.$$

Third-order polynomials were found to be sufficient to correct the distortion of the lens used* with a root-mean-squared (RMS) error of order 10^{-2} pixels. Therefore $2 \times \frac{(n+1)(n+2)}{2} = 20$ coefficients were found (for each of the three wavelengths used) through the aforementioned calibration process.

These polynomials must be evaluated at run-time for every pixel (u, v) in the rectified output image (whose dimensions may or may not be the same as those of the distorted input image). The computational cost is 25 real number multiplies and 18 additions. However further computation is typically required because the (f, g) distorted image coordinates found through the evaluation of the polynomial are typically non-integer. Therefore an image intensity value needs to be synthesised for the output. Bi-linear interpolation adds to the computational cost 4 real number multiplies and 2 additions.

Thus the total computational cost of correcting geometric distortion using third-order polynomials is 29 real number multiplies and 20 additions or subtractions for each rectified image pixel. A non-optimised program† running on a 1.7 GHz Pentium 4-based system with 512MB 133MHz SDRAM running Windows 2000 took 780 ms to synthesise a 600×600 pixel corrected image. Such correction is required for each channel of a polychromatic image, meaning that $3 \times 780 = 2340$ms would be required to synthesise a polychromatic image from three images acquired under typical red, green, blue illumination. This is clearly unacceptable performance for "real-time" machine vision applications, therefore it was decided to investigate an alternative implementation using FPGAs.

4. INTRODUCTION TO FPGAS

Programmable logic devices[3] are VLSI circuits that can be configured after manufacture. Such devices consist of configurable logic blocks (CLBs), block RAM, input/output blocks for connecting to external pins and programmable interconnects. Field-programmable gate arrays (FPGAs) are one family of programmable logic devices that can be reconfigured dynamically as required.

The main approaches to programming FPGAs are through high-level C-like languages and low-level hardware description languages. Hardware description languages (HDLs) such as VHDL and Verilog are powerful, but require significant hardware specification details from the programmer. Such HDLs focus on specifying the structure of a system capable of performing an algorithm whereas high-level languages such as Handel-C[4] and SystemC[5] allow the programming effort to focus on the specification of an algorithm capable of solving a problem. Both HandelC and SystemC have constructs to specify the width in bits of C data types, parallel execution of code, and mechanisms to support interfaces to hardware devices such as busses, I/O pins and memory storage. Synthesis is the process of converting the HandelC/SystemC program into an HDL and ultimately into an EDIF netlist that is *placed and routed* onto a target FPGA device. Place and route generates a configuration or *bit file*. Downloading the bit file to the FPGA configures its CLBs, I/O pins, programmable interconnect and block RAM resources in a manner satisfying the semantics of the original HandelC/SystemC program.

*Computar telecentric video lens, 55mm focal length.

†This program was written in the C language and compiled by the Microsoft C/C++ optimising compiler version 12.00.8168 (which does not perform SSE and SSE2) with performance optimisation (O2) specified. However application-specific memory-access optimisations, etc. were not implemented.

Conventional microprocessors have a static architecture with a small fixed number of on-chip functional units. The rules concerning the number and types of assembly language instructions that may be executed in parallel by a particular microprocessor are fixed for all applications. Performance degradation occurs when an application fails to keep all functional units in a microprocessor busy. This can occur when the application cannot be restructured to offer the correct balance and mix of parallel instructions and when overheads such as cache-misses, translation look-a-side buffer misses, synchronisation and communication occur.

A significant attraction of using FPGAs is the ability to generate application specific logic where the balance and mix of functional units can be altered dynamically. This has the potential to generate orders of magnitude increases in performance for computationally intensive algorithms. There have already been many demonstrations of such speedups in signal and image processing applications using integer and fixed-point arithmetic.[6] The number and type of functional units that can be instantiated on an FPGA are only limited by the silicon area available on the FPGA device in question. Current FPGA devices are sufficiently large to instantiate many entire microprocessor cores, albeit at a lower clock rate than their custom silicon counterparts. Thus, applications targeted for FPGA execution can generate performance benefit at low clock rates because they exploit more parallelism than conventional microprocessors.

The speed of current FPGAs is an order of magnitude slower then their general processor counterparts. To achieve better performance from an FPGA solution, the design must produce results in much fewer clock cycles than the general purpose processor can achieve. Well known hardware techniques such as pipelining and multiple clock domains can significantly speed up FPGA designs. Pipelining breaks complex operations into a series of simpler operations with shorter logic propagation delays that can be clocked at a faster speed. The latency of generating a single result is increased, but the time taken to generate many consecutive results can be dramatically reduced. Pipelining an image processing application allows the computations performed on multiple pixels to be overlapped. Thus, if a design can be fully pipelined, on each clock cycle the next pixel values to be processed enter the pipeline and a processed pixel leaves the pipeline (initially the pipeline must be filled with pixel values which introduces a latency equal to the number of stages in the pipeline times the clock rate). Multiple clock domains allow different parts of the chip to be run at different clock rates. If a design consists of some logic that must run at a slow speed and other logic that can run at a high speed using one global clock will force the fast logic to run at a slow clock rate. Multiple clock domains solve this problem increasing the throughput of the overall design.

The large number of I/O pins on an FPGA allow more data (e.g. 256 bits) to be accessed on each clock cycle compared with a general purpose processor. This allows greater data throughput, a distinct advantage for image processing applications. Even greater throughput can be achieved with suitable data organisation. It is important to organise data so that the FPGA accesses the maximum number of bits each clock cycle. The Celoxica RC1000 FPGA development board used in the research described in this paper, has 8M of SRAM organised as 4 banks of independently addressable memory where a single access to each bank can be made in a single clock cycle.

5. REAL NUMBER SYSTEMS

Many image processing algorithms use real numbers for part or all their numerical computation. There are many ways of representing non-integer numbers, however floating-point numbers are most widely used in general purpose processors. Floating-point allows a wide range of values to be represented and its numerical stability has been well studied in the research literature. However floating-point arithmetic units consume significantly greater hardware resources than their integer counterparts. The relatively recent availability of large multi-million gate FPGA devices such as the the VirtexII platform FPGAs from Xilinx has made it feasible to implement sufficient floating-point execution units, to give comparable performance to general purpose processors.[7] Another problem is that the topology of an FPGA makes operations such as large barrel shifters, required by floating-point units, difficult to implement efficiently.[7]

An advantage of an FPGA implementation is that the designer is not restricted to using the real number support provided by the chip manufacturer, and may choose one or more non integer representations depending on their suitability for the algorithm being implemented. Two alternatives are fixed-point and logarithmic

representations. Fixed-point arithmetic is essentially integer arithmetic coupled with scaling, this can be implemented with much simpler hardware than floating-point. This translates to less area used on the chip and faster speeds. The drawback is a smaller range of values represented for a given number of bits when compared to floating-point and increased complexity in coding the application.

The logarithmic number system takes advantage of the fact that multiplications and divisions simplify to additions and subtractions in the logarithmic domain. This leads to smaller circuits and therefore more non integer execution units and greater throughput. The cost of using logarithmic representation is that adds, subtracts and conversions between the integer and logarithmic domain can be complex.[8]

6. FPGAS FOR GEOMETRIC CORRECTION

Two approaches will be compared:

- Real Time Computation—Calculate the image coordinates through evaluating the polynomials and perform bilinear interpolation to determine intensity values for each pixel.
- Look Up Table—Calculate the image coordinates through evaluating the polynomials *in advance*, storing them in look-up tables which are referenced at run-time. Bilinear interpolation for each pixel is the only computation required at run-time.

The significant difference between these two approaches is that the former requires more real-time computation while the latter requires more memory accesses.

A non-pipelined and a pipelined hardware solution were implemented on an FPGA. The Celoxica RC1000 FPGA development platform was used. The RC1000 consists of a 2 million gate Xilinx Virtex E FPGA device and 8 MB of SRAM mounted on a PCI card and hosted in a PC. The RAM consists of four banks of 512 K $\times$ 32 bit SRAM. This allows the FPGA to access 16 bytes of SRAM in parallel on each clock cycle. Note, in a real image processing system performance could be improved by using i) faster SRAM devices, and ii) by connecting FPGA I/O pins directly to a fast bus.

These implementations were then modified to use different real number representations: fixed-point, floating-point and logarithmic number systems.

It was relatively straightforward to port conventional C program implementations of these programs to Handel-C. Handel-C allows operations to be executed in parallel by enclosing them in a `par` block. It also requires that the width in bits of each variable is specified. This gives the developer the flexibility to use the least amount of hardware to represent each piece of data and to perform arithmetic operations on them. This flexibility combined with parallelism allows designs to execute faster on FPGAs then on general processors which have fixed data widths and degree of parallelism. Data reorganisation and parallelism were added to the Handel-C code and this resulted in a non-pipelined version of the two approaches. The performance of the two FPGA implementations was then improved by fully pipelining them. Performance tuning of the HandelC program was easily accomplished as each statement executes in one clock cycle. Thus, the clock cycle time is dependent on the complexity of the statements in the program.

The results achieved are presented in Tables 1 and 2. Before discussing them it is important to appreciate the issues discussed in the following sections.

Real Number Representation

Polynomial evaluation and bilinear interpolation require the use of real number arithmetic. Three different number systems were implemented:

- Fixed Point— Two fixed-point implementations were evaluated: a simple fixed-point system developed by the authors and Celoxica's fixed-point library. This device independent library allows the width of the the fractional and integer part of the number to be defined and provides macros to execute arithmetic operations. This has the advantage that the API is easy to use, however the compile times can be

long. To give one decimal place precision on the final pixel intensity value, the resampled coordinates are represented with 24 bits (14 bits for fractional part) and the polynomial coefficients are represented with 69 bits (40 bits for fractional part). A simple fixed-point scheme was implemented by scaling all the real numbers by the number of bits that are used to represent the fractional and then using normal integer arithmetic. The 69 bit arithmetic used in the Real Time Computation approach requires a large amount of logic and consequently a large amount of chip area. The simple fixed-point system implemented uses integer arithmetic and scaling. This compiles quicker but without the convenience of the task being performed in a library.

- Floating Point— The floating-point implementation used Celoxica's floating-point library. This is a device independent hardware library that allows floating point numbers of different widths to be specified allowing designers to use the minimum number of bits to represent data and consequently generate smaller hardware. The polynomial coefficients are represented by IEEE754 single precision (32 bit) floating-point values and the bilinear interpolation is performed with 23 bit floating-point numbers. Using floating-point numbers allows the same range of values to be represented with fewer bits, but logic required for computation is more. It should be noted that the accuracy of IEEE754 single precision is less than that achieved by the fixed-point implementations, the RMS error in the greylevel intensity values is order 10^{-3}.

- Logarithmic— The 32bit LNS core of Kadlec et al.[9] was used to implement the logarithmic version of the run-time calculation. This core has precision and range that is equivalent to IEEE754 single-precision. On the XCV2000E device it is possible to place a twin logarithmic 32bit ALU core that can be clocked at 60MHz. This enables up to 2 add/subtract operations to be issued every clock cycle with a latency of 8 cycles whereas multiply/divide/square root operations have a latency of 2 cycles. Add/subtract operations are complex operations in the logarithmic domain and the ALU core makes extensive use of on-FPGA block RAM resources to perform such operations. The number of ALU cores that can be instantiated is largely limited by the availability of block RAM resources. A significantly greater number of multiply, divide and square root functional units can be instantiated. For example, the logarithmic run-time calculation makes use of 72 multiplier functional units.

Pipelining

Pipelining was used to significantly improve the performance of the initial port to Handel-C. The non-pipelined implementation of the Look Up Table approach takes nine FPGA clock cycles per pixel coordinate. The fully pipelined FPGA implementation takes one clock cycle per pixel coordinate after an initial latency. It achieves this by interleaving the computation of twelve pixel coordinates. At any one time twelve pixel coordinates are in different stages of computation. This is a major performance enhancement and makes the implementation I/O bound. On the RC1000 board the SRAMS can be accessed in a single cycle with a clock speed of 51MHz. At this stage the solution can only be clocked as fast as the data can be got on and off the chip. Once the design can be clocked as fast as the SRAMs can be accessed there is little point in optimising the design further although multiple clock domains could be exploited to gain a small performance improvement. If the logic is too complicated to be clocked at the speed data can be accessed, it must be simplified by pipelining it further or using some other technique.

Many image processing applications are well suited to being fully pipelined as they generally follow a single path of execution. In applications where multiple paths of execution are possible, pipelining becomes difficult. It might only be possible to pipeline certain sections of the design and techniques like multiple clock domains become necessary to increase performance.

Data Organisation

It is important in an FPGA design to ensure that the maximum amount of data can be read by the FPGA on each clock cycle. The Real Time Calculation approach needs to read four 8-bit intensity values for the bilinear interpolation and write one 8-bit intensity result value for each pixel coordinate. The RC1000 board allows four 32 bit SRAM locations to be accessed on each clock cycle. In order to access 5 SRAM locations each cycle requires data organisation. The Look Up Table approach needs to read two 28 bit numbers and four

8-bit intensity values for the bilinear interpolation, and write one 8-bit intensity result for each pixel coordinate. Again to access 7 SRAM locations per clock cycle requires some form of efficient data organisation.

In the non-pipelined Real Time Computation approach each pixel coordinate takes 10 FPGA cycles. As a result the four 8-bit reads do not need to occur with the 8-bit intensity result write. A copy of the input image is placed in each of the four memory banks, which allows the four reads to occur in parallel. The write which occurs 7 clock cycles later can be to any memory bank. Each pixel coordinate in the non-pipelined Look Up Table approach takes 9 cycles to compute. The reading of the two resampled pixel coordinates can occur in parallel. The reading of the four 8-bit intensity values can also occur in parallel. Placing one pixel coordinate in bank0, the other in bank1, and a copy of each image in the four banks allows this parallelism.

The pipelined implementations require a more sophisticated data organisation. The pipelined Look Up Table approach has 12 stages. At any one time 12 pixel coordinates are in different stages of computation. A fully pipelined implementation requires that all 6 reads and 1 write must occur in parallel. Compressing each pixel coordinate into 16 bits each allows them to fit into one memory bank and be read as one 32 bit value. The location of the top left pixel in the interpolation grid allows you to work out the locations of the other three pixels needed for the bilinear interpolation. These four 8-bit intensity values of the uncorrected image can be formed into a 32-bit word, placed on one line of a SRAM bank and read in one cycle. The output image can then be placed in a separate memory bank. Pixel coordinates are stored in bank0, the uncorrected image is stored in bank1 and bank2, and the output image is stored in bank3. A similar technique is used for the Real Time Calculation approach except bank0 is not used.

7. RESULTS

Table 1. Characteristics of the two general purpose processor implementations and four FPGA implementations

Implementation	**Area (%)**	**SRAM (MB)**	**Clock** speed	**Time (ms)** 600×600 pixels
Pentium 4 real time calculation	n/a	n/a	1.7 GHz	780
Pentium 4 lookup table	n/a	n/a	1.7 GHz	760
non-pipelined FPGA real time calculation	63	5	10 MHz	360
non-pipelined FPGA lookup table	7	7	22 MHz	130
pipelined FPGA real time calculation	64	5	25 MHz	15
pipelined FPGA lookup table	6	6	34 MHz	11

All FPGA implementations except the logarithmic implementation are several times faster than the Pentium 4 implementations. It is also interesting to see that there is not much difference between the general processor Real Time Computation and Look-up Table approaches. This is due to the task being RAM-access limited. There are two main reasons why the FPGA implementations perform better than the Pentium 4 implementations. The FPGA is performing more multiplies and additions on every clock cycle than the Pentium 4 and the FPGA is able to access RAM much faster than the Pentium 4 can. It is worth noting again that the Pentium 4 implementations have not been optimised.

The pipelined FPGA implementations are an order of magnitude faster than their non-pipelined versions. This is because the clock cycles required per pixel have been reduced from about ten to one. The execution time for both approaches is also the same. Both fully pipelined designs generate a result each clock cycle (after a fixed latency) and therefore take the same time to process an image. The execution time for the non-pipelined Real Time Calculation implementation is comparatively very slow. This is due to the slow clock cycle (10 MHz) which could be increased by splitting the complex arithmetic logic into several smaller stages.

Area

The area of the two million gate Xilinx Virtex E device used by each FPGA implementation is also shown in table 1. It is clear that the Real Time Calculation approach consumes a much greater amount of logic resources than the Look Up Table approach. This is due to the 69 bit arithmetic operations. The calculation of image coordinates requires 25 multiplies and 18 additions. Separate logic was generated for each operation. This is necessary for the pipelined implementation as all these operations must happen in parallel. For the non-pipelined approach multipliers and adders could be reused several times which would reduce the logic requirement for this implementation.

Clock speed

Any design that accesses the 100 MHz SRAM banks on the RC1000 board is restricted to an input/output maximum clock speed of 51 MHz, despite the fact that the FPGA itself can be clocked at speeds greater than 100 MHz. The use of faster SRAM banks or another fast method of transferring data onto the FPGA would allow fully pipelined designs to be clocked faster than 51 MHz.

The initial non-pipelined implementations contained complicated logic that would only allow them to be clocked at speeds below 25 MHz. The non pipelined 69 bit arithmetic operations present in the Real Time Computation approach introduce significant logic delays that severely limit the speed that this implementation can run at. Further optimisation of the pipelined implementations could bring the clock rate up to 51 MHz at the expense of a longer latency. This would reduce the time/image to about 7ms.

Table 2. Characteristics of the three different real number representations

Implementation	**Number System**	**Area** %	**SRAM** MB	**Latency** cycles	**Clock** speed	**Time (ms)** 600×600 pixels
real time calculation	Celoxica Fixed Point Library	65	5	23	20 MHz	19
lookup table	Celoxica Fixed Point Library	8	7	24	32 MHz	12
lookup table	Celoxica Floating Point Library	35	6	24	15 MHz	25
non-pipelined real time calculation	Logarithmic	78	5	227	50 MHz	890

7.1. Fixed Point Implementations

The performance and area requirements of both fixed-point implementations is better than both the floating-point and logarithmic implementations. This is because the logic required is much less complicated (equivalent to integer arithmetic with some extra shift registers). Fixed-point arithmetic is appropriate for this application because the range of the values is small. A significantly greater number of bits would be needed to represent a much larger range of values. The associated large logic requirement would justify the use of a different number system such as floating-point or logarithmic. It is easier to use a fixed-point library than to implement the number system, however the compile times for Celoxica's fixed-point library are significantly longer (12 hours as opposed to 3 hours).

7.2. Floating Point Implementations

The floating-point look up table implementation uses significantly more logic resources and must be clocked at a lower speed than its fixed point counterpart. The pipelined floating point implementation would not compile as the logic resources required for the 25 multipliers and 18 adders cannot be satisfied by the 2 million gate

Xilinx Virtex E FPGA. The pipelined design could be placed on a larger FPGA or a non pipelined design that reuses a smaller number of multipliers and adders would fit on the 2 million gate device, but would take longer to process an image.

7.3. Logarithmic Number System (LNS) Implementation

Note: the performance results of the LNS presented in the paper are based on simulation. The performance of the LNS is poor because i), only 2 adds/subtracts can be issued in a single cycle having a latency of 8 cycles, ii) a single conversion from the integer to the logarithmic domain can be achieved in a single cycle, whilst the reverse takes 24 cycles, iii) the run-time calculation cannot easily be pipelined for execution using the LNS core because insufficient add/subtract operations can be issued in a single cycle. Further, the current LNS implementation is only an initial port rather than an optimised version. Significant performance gains can still be made even with the restriction of issuing 2 adds/subtracts per cycle. However a large portion of the program runs sequentially because insufficient add/subtract operations can be performed in parallel.

The initial port of the logarithmic run-time calculation attempts to calculate 8 pixel values concurrently. If more add/subtracts could be issued in a single cycle then it would be sensible to attempt to pipeline the logarithmic calculation. We have not attempted to optimise the current logarithmic implementation because we plan to use a larger XC2V6000 device on an Alpha Data ADM-XRC-II board which will ameliorate this difficulty. It will also be necessary to revisit the mathematics of logarithmic number system representations to determine if a more efficient mechanism can be found to truncate a real number in log base 2 to an integer in log base 2. We also plan to investigate the use of a 19 bit LNS core having smaller block RAM requirements making it possible to instantiate 24 19 bit cores on the XCV2000E device facilitating the issue of 48 add/subtract operations per cycle. It should then be possible to pipeline the logarithmic run-time calculation. However, the 19 bit core does not offer precision equal to IEEE 754 single precision. In order to generate results to sufficient accuracy it will be necessary to use the core in extended precision mode by scaling the data range to the interval [-1,1]. Details of extended precision mode for the core can be found in Kadlec et al.[9]

8. CONCLUSIONS

An obvious conclusion is that FPGAs can be used to achieve speeds not possible with general purpose CPUs for this class of image processing application. The potential speed up that can be achieved by fully pipelining a design make pipelining a serious performance goal for FPGA designs.

Real number systems such as logarithmic and floating-point representations have large resource requirements and consequently lower performance. If the range of real number values that must be represented is small or can be scaled so that it is small, fixed point arithmetic is one way of providing cheap fast non integer support.

Even using high level software tools and languages , there is significantly more development effort required, especially if prior FPGA experience is limited. FPGA development boards with large FPGA devices are expensive and large computing resources are required to compile FPGA designs. High level tools speedup the design cycle, but it is still significantly longer than the software design cycle.

REFERENCES

1. G. Wolberg, *Digital Image Warping*, IEEE Computer Society Press, 1990.
2. R. G. Willson, *Modelling and calibration of automated zoom lenses*. PhD thesis, Carnegie-Mellon university, 1994. CMU robotics institute technical report CMU-RI-TR-94-03.
3. A. Sharma, *Programmable Logic Handbook, PLDs, CPLDs and FPGAs*, McGraw-Hill, 1998.
4. Celoxica Limited, "http://www.celoxica.com."
5. SystemC, "http://www.systemc.org."
6. B. Draper, W. Najjar, W. Bohm, J. Hammes, R. Rinker, C. Ross, and J. Binns, "Compiling and optomizing image processing algorithms for FPGAs," in *International Workshop on Computer Architecture for Machine Perception*, IEEE, Italy 2000.

7. W. B. Ligon III, S. McMillan, G. Monn, K. Schoonover, F. Stivers, and K. D. Underwood., "A re-evaluation of the practicality of floating point operations on FPGAs," in *Symposium on FPGAs for Custom Computing Machines*, IEEE, 1998.
8. J. Coleman, E. Chester, C. Softley, and J. Kadlec, "Arithmetic on the European Logarithmic Microprocessor," *IEEE Trans on Computers* **49**, pp. 702–715, 2000.
9. J. Kadlec, A. Hermanek, C. Softley, R. Matousek, and M.Licho, "32-bit Logarithmic ALU for HandelC 2.1 and Celoxica DK1," in *User Conference*, Celoxica, England 2001.