FPGA BASED ACCELERATION OF THE LINPACK BENCHMARK: A HIGH LEVEL CODE TRANSFORMATION APPROACH

Kieron Turkington, Konstantinos Masselos, George A. Constantinides

Department of Electrical and Electronic Engineering, Imperial College London, UK
e-mail: {kjt01, k.masselos, g.constantinides}@imperial.ac.uk

Philip Leong

Department of Computing, Imperial College London, UK
e-mail: philip.leong@imperial.ac.uk

ABSTRACT

Due to their increasing resource densities, field programmable gate arrays (FPGAs) have become capable of efficiently implementing large scale scientific applications involving floating point computations. In this paper FPGAs are compared to a high end microprocessor with respect to sustained performance for a popular floating point CPU performance benchmark, namely LINPACK 1000. A set of translation and optimization steps have been applied to transform a sequential C description of the LINPACK benchmark, based on a monolithic memory model, into a parallel Handel-C description that utilizes the plurality of memory resources available on a realistic reconfigurable computing platform. The experimental results show that the latest generation of FPGAs, programmed using Handel-C, can achieve a sustained floating point performance up to 6 times greater than the microprocessor while operating at a clock frequency that is 60 times lower. The transformations are applied in a way that could be generalized, allowing efficient compilation approaches for the mapping of high level descriptions onto FPGAs.

1. INTRODUCTION

The vast majority of engineering and scientific applications running on conventional computers employ floating point arithmetic for accurate modeling and to reduce experimentation requirements [1]. Recently, technology scaling has made possible the fabrication of FPGA devices with resource densities that allow efficient implementations of floating point algorithms. Significant research effort has been spent on studying floating point arithmetic for FPGAs. The use of custom floating point formats in FPGAs is discussed in [2, 3]. Other works focus on the development of optimized floating point operators for FPGAs [4] and features that can improve floating point performance are described in [5]. The work presented in [6] studies single precision floating point matrix multiplication on FPGAs, compares it to a microprocessor and proves that FPGAs can achieve competitive sustained floating point performance. Performance comparisons between FPGAs and microprocessors for double precision floating point linear algebra functions are discussed in [7].

Although FPGAs and microprocessors are quite competitive in terms of performance, a large gap still exists in terms of design flows. FPGA programming is still performed to a large extent using conventional HDL based design flows. This approach is quite inefficient for exploiting the large resource densities of modern FPGAs within strict development time constraints. Higher level hardware compilation approaches are required to create an efficient, software-like design flow and allow the efficient implementation of complex applications on modern FPGAs. A number of commercial and academic approaches have been proposed in this direction [8, 9].

The main contribution of this work involves a design methodology of applying a sequence of optimizations before hardware compilation for coarse grained data level parallelization (not limited to the innermost loops of loop nests) and data transfer and storage optimization. These optimizations involve identifying opportunities for: data reuse, loop pipelining, loop level parallelism, parallel memory assignment, fine grained (instruction level) parallelism and pipelining. Such optimizations are not automatically applied in existing hardware compilation environments, especially in combination. Most existing hardware compilation approaches focus on performance optimization by applying loop unrolling and pipelining optimizations in the inner loops of loop nests while data transfers and storage issues are not aggressively optimized.

Applying this methodology to the popular LINPACK benchmark [10] we were able to achieve very high performance on a complex benchmark, outperforming a standard commodity processor. To the best of our knowledge, this is the first reported FPGA-based implementation of the LINPACK benchmark, and presents an indication of the performance achievable using current devices on real-world scientific benchmarks. The same methodology could be applied to other applications, though how successful the optimizations are will depend on the regularity of the application’s data access patterns.

```
begin
    -- generate linear system Ax = b --
    matgen (&A[ ][ ],&b[ ]);
    -- start timer --
    t1 = second();
    -- Solve system using LU decompositon --
    dgefa (&A[ ][ ], &ipvt[ ], 1000);
    dgesl (&A[ ][ ], &b[ ], ipvt[ ], 1000);
    -- store time taken --
    t1 = second( );
    -- calc average FLOPs --
    ops = 2/3*(1000)^3 + (1000)^2;
    flops = ops / t1;
    x[ ] = b[ ];
    -- regenerate original A[ ] & b[ ] --
    matgen(&A[ ], &b[ ]);
    -- calculate b = Ax - b --
    b[ ] = -b[ ];
    dmxpy(A[ ][ ], &b[ ], x[ ], 1000);
    -- find residual and normalise --
    resid = max (b[ ]);
    residn = norm(resid);
    print(t1, flops, resid, residn);
end;
```

Fig. 1. Pseudo code for LINPACK 1000

The rest of the paper is organized as follows. Section 2 gives a brief description of the LINPACK 1000 benchmark. Section 3 describes the general platform this work seeks to target and Section 4 describes how the critical sections of the LINPACK benchmark are implemented on an FPGA. In Section 5 the results for this implementation are presented and Section 6 contains the conclusions drawn from this work.

2. LINPACK 1000

The LINPACK family of benchmarks solves a system of linear equations using LU decomposition and is typical of many matrix-based scientific computations. It is widely used to gauge the floating point performance of computer systems. This work will focus on the LINPACK 1000 benchmark which is most commonly used to evaluate general purpose microprocessors. LINPACK 1000 solves a random linear system of order 1000, measures the average floating point performance achieved and reports the residual error produced.

The pseudo-code for the operation of the LINPACK 1000 benchmark is given in Fig. 1. It first generates a random 1000x1000 element matrix, ***A***, and 1000 element vector, ***b***. The elements in ***A*** and ***b*** are all floating point numbers. The *dgefa* and *dgesl* subroutines are then used to find a 1000 element vector, ***x***, such that ***Ax*** = ***b***.

The *dgefa* subroutine performs LU decomposition by Gaussian elimination with partial pivoting on ***A***. A brief description of this method can be found in [11]. *dgefa* modifies ***A*** and returns a vector, ***ipvt,*** containing the pivot indices. The *dgesl* subroutine then solves the simplified LU version of the original system. The time taken to complete the *dgefa* and *dgesl* subroutines is measured, and the average FLOPs calculated. The remainder of the algorithm serves to estimate the normalized residual error in the result vector **x.**

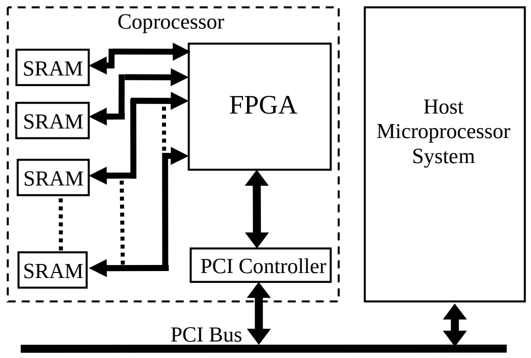

Fig. 2. Target Platform

The true LINPACK 1000 benchmark uses double precision floating point numbers. However, Celoxica's floating point library offers only single precision format. To enable a fair comparison between the microprocessor and Handel-C implementations, the benchmark must be modified to use single precision.

The single precision benchmark was profiled for an Intel Pentium 4 processor (3GHz, 1Mbyte L2 cache) running Windows XP using GCC 4.0.2 (running through Cygwin). The entire single precision benchmark takes 1.70 seconds to complete, with an average floating point performance of 427 MFLOPs across the *dgefa* and *dgesl* subroutines. 93% of the execution time is spent in the *dgefa* subroutine. *dgefa* calls a smaller function, named *daxpy*, 499500 times with the result that the latter alone accounts for 90% of the total execution time for the benchmark.

3. TARGET PLATFORM

This work targets the general platform shown in Fig. 2. It consists of a host microprocessor system linked to an FPGA based coprocessor via a PCI interface. The coprocessor features a high density FPGA, a PCI interface/controller and on-board memory resources. In this work the on-board memory is assumed to be SRAM with between 4 and 6 banks giving a total of up to 32Mbytes of storage. The data width of the memories is assumed to be 32 bits. Although this does not match any existing development boards the features specified are realistic and similar to those found on a number of boards including the Celoxica RC300 [12] and RC2000 [13].

This target platform has a key characteristic that affects the implementation of the LINPACK 1000 benchmark, namely that the benchmark must be partitioned between the FPGA and the microprocessor to achieve good performance. However, communication between the microprocessor and the FPGA will be slow compared to accessing the local SRAM. Hence a partition must be chosen that limits the total communication required.

For simplicity the benchmark is partitioned so that the entire *dgefa* subroutine is implemented on the FPGA co-processor. This partition limits the communication across the PCI bus to two large block transfers of data (totaling 2001000 32-bit transfers), one at the start of the subroutine and the other at the end. It also provides potential opportunities for data reuse and parallelism on the FPGA since *dgefa* contains a nested loop with three levels. Finally, since *dgefa* accounts for 93% of the execution time of the benchmark, there is good potential for overall acceleration of the algorithm.

4. FPGA ACCELERATION OF THE *DGEFA* ROUTINE

The design of LINPACK on FPGAs was done in Handel-C and compiled in hardware using the DK Design Suite [14]. Compared with other HDLs, Handel-C allows optimizations to be made at a high level, improving programmer productivity.

With the benchmark partitioned, the next step is to translate the *dgefa* subroutine to Handel-C and optimize it with respect to data storage and parallelization. This work uses an ANSI C version of the LINPACK benchmark as its starting point. Before any optimizations are considered some simple translation steps are performed such as removing side effects and replacing 'for' loops with 'while' loops (which are implemented more efficiently in Handel-C). Pointer-based array access is converted into direct array addressing, and any floating point arithmetic is replaced with Handel-C floating point macro procedures.

To complete the translation to Handel-C, the problem of assigning widths to all of the variables in the code must be considered. Fortunately, since the main arithmetic in the LINPACK benchmark is floating point, only the widths of the control signals and array indices need to be considered. These can be inferred relatively easily from loop bounds and array sizes.

The code can now be compiled as Handel-C and synthesized to hardware. However, the hardware generated will not be very efficient as no parallelism has been exploited and no embedded memories specified. These issues are tackled in the five optimization stages that follow.

Before any optimizations are considered, the arrays in the algorithm are provisionally assigned to a layer of the available memory hierarchy (registers, on chip memory or off chip memory) depending on their sizes. This provisional assignment establishes the maximum bandwidth available for accessing each array. Data reuse can then be focused on the most frequently accessed arrays with the lowest available bandwidth.

The *dgefa* subroutine uses two arrays. The first is used to store the **A** matrix and the second stores the **ipvt** vector. The **A** matrix has 1000 x 1000 floating point elements and therefore requires 32Mbits of storage. This is four times greater than the total storage available on even the largest FPGAs, so **A** must be assigned to off-chip memory. The **ipvt** vector has only 1000 floating point elements and so can be stored on the FPGA.

```
dgefa(*A[ ][ ], *ipvt[ ])
begin
    for (k = 0 : 998)                      -- loop 1 ---
        -- find the pivot index of column k --
        -- loop 4 in idamax --
        piv = idamax(A[ ][k], k) + k;
        ipvt[k] = piv;
        if (A[piv][k] != 0)
            -- swap matrix elements if k != piv--
            swap(&A[piv][k], &A[k][k]);
            -- floating point divide --
            t = -1/(A[k][k]);
            -- scale column k by t --
            --loop 5 in dscal --
            dscal(&A[ ][k], t, k );
            for (j = (k+1) : 999)      -- loop 2 --
                -- swap matrix elements if k != piv --
                swap(&A[piv][j], &A[k][j]);
                t = A[piv][j];
                -- loop 3 in daxpy –
                    --A[ ][j] = t*A[ ][k] + A[ ][j] --
                daxpy(&A[ ][j], A[ ][k], t, k );
            end for;                   -- end loop 2 --
        end if;
    end for;                           -- end loop 1 --
    ipvt[999] =999;
end;
```

Fig. 3. Pseudo code for *dgefa* subroutine

4.1. Data Reuse Exploitation

In the first optimization stage the potential for data reuse is investigated. The goal is to search for any reuse pattern that could be exploited to reduce external memory accesses or reduce array accesses within loop nests. This could also allow greater parallelism to be exploited in later stages of the implementation.

The pseudo code for the *dgefa* subroutine is shown in Fig. 3. It is possible to reuse (up to) a column of **A** matrix data between iterations of both *loop 1* and *loop 2*. The *daxpy* function takes two matrix columns as inputs, denoted as **A[ ][j]** and **A[ ][k]**. The same **A[ ][k]** data is used for each iteration of *loop 2* meaning that, if the correct **A[ ][k]** data is buffered on the FPGA at the start of *loop 1*, it can be reused for every iteration of *loop 2*. More importantly, the **A[ ][k]** data for iteration *k+1* of *loop 1* is calculated as a result column in iteration *k*. If this row of data is stored on the FPGA as it is generated then only the first **A[ ][k]** column will need to be read from the external memory. As a result, inside the loop nest in Fig. 3, only the **A[ ][j]** data must read from the external memory instead of the **A[ ][k]** and **A[ ][j]** data. This effectively halves the number of reads from the external memory and halves the bandwidth required.

This data reuse can be implemented using two new arrays (provisionally assigned to on-chip RAM), each capable of storing one column (32000 bits) of matrix data. One array, called *k_col*, stores the **A[ ][k]** data for the current iteration of *loop 1*. The second array, called *k_next*, stores the **A[ ][k]** data needed for the next iteration as it is generated. During the *idamax* and *dscal* functions the new **A[ ][k]** data stored in *k_next* is modified and transferred across to *k_col*.

4.2. Loop Pipelining

Stage 2 of the optimization process explores the possibility of single dimension software pipelining, using the method presented in [15]. The potential for pipelining at each level in the loop nest is considered, starting at the outermost loop and working inwards. At each level data the dependence constraints, the possible initiation rate of the pipeline and the possibilities for data reuse are used to determine the suitability of that level for pipelining.

It is not possible to pipeline iterations of the outer loop (*loop 1* in Fig. 3) since there are loop carried dependencies on all of the input data. For the *idamax* and *dscal* functions this leaves only their internal loops as candidates for pipelining. There are no loop carried dependencies within *idamax* and *dscal* so their internal loops can be pipelined into the single floating point resource present in each function. *idamax* contains a floating point comparator with a pipeline depth of 2 and *dscal* contains a floating point multiplier with a pipeline depth of 7.

The *daxpy* function contains a floating point multiplier whose output feeds a floating point adder. This gives a total pipeline depth of 17. Iterations of either *loop 2* or *loop 3* (Fig. 3) can be pipelined into these resources, but *loop 2* is not perfectly nested. Since both have the same bounds it makes sense to pipeline *loop 3* since it is perfectly nested and therefore more easily implemented.

4.3. Coarse Grained Parallelization

The goal of this optimization step is to exploit coarse grained parallelism by scheduling blocks of code to execute together, specifically entire loops and/or loop iterations. In the LINPACK 1000 case the main focus is on loops involving the *daxpy* function since it accounts for 99% of the time spent in the *dgefa* subroutine.

The first step is to analyze the input data patterns and data dependencies at each loop level in the *dgefa* subroutine. There is a loop carried dependency in *loop 1* (Fig. 3) such that all of the input data used by iteration *n+1* is generated in iteration *n*. This means that, if multiple *loop 1* iterations are run in parallel with their start times offset correctly, then data could effectively flow from the output of one iteration to the input of the next without going through the external memory. This allows arbitrary *daxpy* parallelism with only two ports to the external memory.

Fig.4. shows how the *loop 1* iterations can be scheduled across multiple parallel loop processing units. The number of processors can be varied using a '# define' in the Handel-C source code. In Fig. 4 *idamax[k]* and *dscal[k]* represent the execution of the *idamax* and *dscal* functions respectively in iteration *k* of *loop 1*. *daxpy[k][j]* represents the execution of the *daxpy* function in iteration *j* of *loop 2* and iteration *k* of *loop 1*. All of the rest of code in the *dgefa* subroutine has been moved into one of these three functions to simplify the system. *idamax*, *dscal* and *daxpy* all contain loops iterating over the same bounds so they take the same number of clock cycles to run (+/- 2%). This allows instances of the three functions to be scheduled as if they were individual instructions.

Processor 0	Processor 1	Processor 2	Shared
wait	wait	wait	idamax[0]
wait	wait	wait	dscal[0]
daxpy[0][1]	wait	wait	idamax[1]
daxpy[0][2]	wait	wait	dscal[1]
daxpy[0][3]	daxpy[1][2]	wait	idamax[2]
daxpy[0][4]	daxpy[1][3]	wait	dscal[2]
daxpy[0][5]	daxpy[1][4]	daxpy[2][3]	idamax[3]
daxpy[0][6]	daxpy[1][5]	daxpy[2][4]	wait
■	■	■	■
■	■	■	■
■	■	■	■
■	■	■	■
daxpy[0][999]	daxpy[1][998]	daxpy[2][997]	dscal[3]
daxpy[3][4]	daxpy[1][999]	daxpy[2][998]	idamax[4]
daxpy[3][5]	wait	daxpy[2][999]	dscal[4]
daxpy[3][6]	daxpy[4][5]	wait	idamax[5]
daxpy[3][7]	daxpy[4][6]	wait	dscal[5]
daxpy[3][8]	daxpy[4][7]	daxpy[5][6]	idamax[5]
daxpy[3][9]	daxpy[4][8]	daxpy[5][7]	wait
■	■	■	■
■	■	■	■
■	■	■	■

Fig. 4. Schedule for 3 parallel processing units

idamax[k+1] operates on the data generated in *daxpy[k][k+1]*. As the output data from *daxpy[k][k+1]* is generated it is sent to *idamax[k+1]* through global variables, allowing the two functions to run in parallel. *daxpy[k+1][k+2]* runs in parallel with *daxpy[k][k+3]* but uses the data generated by *daxpy[k][k+2]* during the previous iteration of *loop 2*. Hence, each processor must include a circular buffer (capable of storing a column of matrix data) to store the data generated by the previous processor in the chain until it can be used in the next iteration of *loop 2*.

As can be seen in Fig. 4, *idamax* and *dscal* are never required by more than one processor at once, allowing them to be shared across all of the processors. This means that the *k_next* buffer (generated in Section 4.1) can also be shared since only *idamax* and *dscal* access it. Each processor must have its own *k_col* buffer however, since each processor uses a different **A[ ][k]** column as its input.

Only the last loop processor in the chain must write its **A[ ][j]** output data to the external memory. However, the **A[ ][k]** data stored in the *k_col* buffer in each processor must be written to the external memory at some point since

Table 1. Performance results for Altera Stratix II devices

No. loop processors	Device	ALUTs	Memory (kbits)	DSPs	Fmax (MHz)	Cycles for *dgefa*	Time for *dgefa* (s)	Speedup Total (*dgefa* only)
2	EP2S15	11704	160	24	67.7	172922066	2.58	0.63 (0.61)
3	EP2S30	15584	224	32	63.7	115453199	1.84	0.87 (0.86)
4	EP2S30	19195	288	40	63.4	86718956	1.40	1.12 (1.13)
8	EP2S60	33635	352	72	61.8	43618207	0.74	1.99 (2.15)
16	EP2S90	62095	1056	136	64.1	22069153	0.37	3.48 (4.27)
36	EP2S180	135487	1920	296	44.4	10104098	0.26	4.51 (6.14)

Table 2. Performance results for Xilinx Virtex 4 devices

No. loop processors	Device	Slices	Memory (kbits)	DSP Slices	Fmax (MHz)	Cycles for *dgefa*	Time for *dgefa* (s)	Speedup Total (*dgefa* only)
2	XV4LX25	9754	160	12	70.0	172922066	2.50	0.65 (0.63)
3	XV4LX40	13230	224	16	69.0	115453199	1.70	0.93 (0.93)
4	XV4LX40	16371	288	20	64.1	86718956	1.38	1.13 (1.15)
8	XV4LX100	28850	352	36	56.2	43618207	0.81	1.83 (1.96)
16	XV4LX200	52475	1056	68	53.8	22069153	0.44	3.04 (3.59)
28	XV4LX200	89086	1824	96	42.1	12837274	0.33	3.75 (4.72)

these columns form part of the final result matrix. Fortunately the final processor in the chain has sufficient 'wait' slots to output all the necessary data so only a single write port to the external memory is required. Since there are now multiple write sources an additional function must be included to arbitrate over the memory bus. The processors communicate with this function via global variables, allowing it to run in parallel with the normal operation of the system.

4.4 Memory Assignment

Memory assignment for the parallelized *dgefa* subroutine is quite simple as there are no opportunities to share memory resources between arrays since all of the arrays are active all of the time. All of the local buffers generated in the previous sections must be assigned to simple dual port memory (M4K blocks for Altera devices and Block RAM for Xilinx devices) as each is often accessed by two processing units in parallel.

The **ipvt** vector is assigned to a single port on chip memory. The **A** matrix is assigned to two blocks of off chip SRAM. The arbitration function which controls the external write port can be written so that, when an element from a row with an even number is being read, the element being written belongs to an odd numbered row. There is sufficient slack in the schedule to allow a write to be delayed by a cycle when necessary to avoid conflict. This scheme allows all even numbered rows to be assigned to one block of SRAM while the odd rows are assigned to the other. This allows the two blocks of SRAM to behave as a simple dual port RAM.

4.5 Fine Grained Parallelism & Pipelining

A number of steps are executed to increase the maximum clock frequency. These include breaking up complex calculations into several smaller calculations, inserting pipeline registers into the data paths and simplifying control statements. The read ports of all the embedded memory blocks are also pipelined to prevent Celoxica's Handel-C compiler from creating an inverted clock signal and halving the maximum operating frequency. Once these modifications have been made the final step is to go through the code and find any instructions that can be implemented in parallel and to enclose them in 'par' tags.

5. RESULTS

Table 1 shows the performance results achieved when between 2 and 36 parallel processing units are targeted to Altera Stratix II devices. 36 processors utilize 94% of the largest device available. Table 2 shows comparable results for Xilinx Virtex 4 LX devices. 28 processors occupy 99% of the slices on the largest Virtex 4 LX device, but this is without unrelated logic packing. Each test case is targeted to the smallest device (fastest speed grade) that will accommodate it, without filling the device to such an extent that timing suffers adversely.

The Handel-C source code was complied to EDIF through Celoxica DK Suite 4.0. The Altera syntheses were performed using Altera Quartus II 4.2 and the Xilinx syntheses used Xilinx ISE 7.1i. All of the test syntheses used single pass place and route and standard effort settings. The functionality of dgefa coprocessor was verified using the Handel-C simulator in DK suite. A four processor version of dgefa was downloaded to a Stratix EP1S40 device on a NIOS development board and verified for a 32x32 matrix.

The *dgefa* times listed in both tables are those taken to complete the processing for the *dgefa* subroutine plus an estimate of the time taken to transfer data between the FPGA and the host microprocessor across a 66MHz PCI bus. This has been estimated to take 0.03 seconds (2001000 32 bit transfers at 66MHz). The speedup time given inside the brackets is an estimate of the speedup over the 3GHz Pentium 4 for just the *dgefa* subroutine. The total speedup figure quoted is an estimate for the speedup factor for the whole benchmark, assuming that the remaining code is implemented on the Pentium 4 processor.

The optimized Handel-C implementation can execute the *dgefa* subroutine around 6 times faster than the Pentium 4 when targeted to the largest Altera device, with an average performance of 2570 MFLOPs. For the larger designs it should be possible to improve these figures by further pipelining the connections (global variables) used to send data between processing units. However, it should be noted that, up to 16 processors, the critical path for the Altera devices lay within the Celoxica floating point comparator. Hence these particular figures could not be improved by further pipelining the LINPACK code.

6. CONCLUSIONS

An optimized Handel-C implementation of a coprocessor for a single precision version of the LINPACK 1000 benchmark was produced through a series of optimizations that identify opportunities to exploit data reuse, loop pipelining, coarse grained parallelism, parallel memory assignment, fine grained parallelism and pipelining. When targeted to the latest generation of FPGAs this coprocessor can execute parts of the benchmark up to 6 times faster than a 3GHz Pentium 4 processor and accelerate the execution of the complete benchmark by a factor of 4.5. This suggests that it is currently possible to accelerate scientific computing algorithms using FPGAs, even when using high level design methods and tools.

However, it has also been shown that significant effort is required to convert the standard ANSI C algorithms into efficient, parallel Handel-C. As a result the development times for hardware are still much longer than those for software, even when high level languages are used. Furthermore, an awareness of how specific high level code will be synthesized in hardware is still required to produce efficient designs. This prevents software designers from migrating seamlessly to hardware. It seems that further design tools, possibly automating optimizations similar to those presented here, are still required to make hardware design run as quickly and efficiently as software design.

7. REFERENCES

[1] K. Underwood, “FPGAs vs. CPUs: Trends in Peak Floating Point Performance”, *Proc. Int. Symp. Field Programmable Gate Arrays*, pp. 171–180, February 2004.

[2] P. Belanovic, M. Leeser, “A library of parameterized floating point modules and their use”, *Proc. Int. Conf. Field Programmable Logic and Applications*, pp 657-666, September 2002.

[3] J. Dido, et. al., “A flexible floating point format for optimizing data paths and operators in FPGA based DSPs”, *Proc. Int. Symp. Field Programmable Gate Arrays*, pp. 50-55, February 2002.

[4] X. Wang, B. E. Nelson, “Tradeoffs of designing floating point division and square root on Virtex FPGAs”, *Proc. IEEE Symp. Field-Programmable Custom Computing Machines*, pp. 195-203, April 2003.

[5] E. Roesler, B. Nelson, “Novel optimizations for hardware floating point units in a modern FPGA architecture”, *Proc. Int. Conf. Field Programmable Logic and Applications*, pp. 637 – 646, September 2002.

[6] W. Ligon, et. al., “A re-evaluation of the practicality of floating point on FPGAs”, *Proc. IEEE Symp. Field-Programmable Custom Computing Machines*, pp. 206–215, April 1998.

[7] K. Underwood, K. Hemmert, “Closing the gap: CPU and FPGA trends in sustainable floating point BLAS performance”, *Proc. IEEE Symp. Field Programmable Custom Computing Machines*, pp 219-228, April 2004.

[8] http://www.mentor.com/products/c-based_design/catapult_c_synthesis/index.cfm

[9] M. Weinhardt, W. Luk, “Pipeline vectorization”, *IEEE Trans. Comput.-Aided Design*, vol 20, no. 2, pp. 234–248, February 2001

[10] J. Dongarra, P. Luszczek, A. Petitet, “The LINPACK benchmark: Past, Present and Future”, http://www.netlib.org/utk/people/JackDongarra/PAPERS/hpl.pdf

[11] http://www.cs.toronto.edu/~bonner/courses/2006s/csc338/lectures/chap02.pdf, pp. 35-42.

[12] http://www.celoxica.com/products/rc300/default.asp

[13] http://www.celoxica.com/products/rc2000/default.asp

[14] http://www.celoxica.com/products/dk/default.asp

[15] H. Rong, Z. Tang, R. Govindarajan, A. Douillet, G Gao, “Single-Dimension Software Pipelining for Multi-Dimensional Loops”, *Proc. IEEE Int. Symp. Code Generation and Optimization*, pp. 163-174, March 2000

ACKNOWLEDGEMENTS

This work was supported by the EPSRC (EP/C549481/1)