HIGH PERFORMANCE SCIENTIFIC COMPUTING USING FPGAS WITH IEEE FLOATING POINT AND LOGARITHMIC ARITHMETIC FOR LATTICE QCD

*Owen Callanan, David Gregg**
Dept. of Computer Science,
Trinity College Dublin,
Ireland.
email: {Owen.Callanan,
David.Gregg} @cs.tcd.ie

Andy Nisbet
Dept. of Computing and
Mathematics,
Manchester Metropolitan
University, UK.
email: A.Nisbet@mmu.ac.uk

Mike Peardon
Dept. of Mathematics,
Trinity College Dublin
email: mjp@maths.tcd.ie

ABSTRACT

The recent development of large FPGAs along with the availability of a variety of floating point cores have made it possible to implement high-performance matrix and vector kernel operations on FPGAs. In this paper we seek to evaluate the performance of FPGAs for real scientific computations by implementing Lattice QCD, one of the classic scientific computing problems. Lattice QCD is the focus of considerable research work worldwide, including two custom ASIC-based solutions. Our results give significant insights into the usefulness of FPGAs for scientific computing. We also seek to evaluate two different number systems available for running scientific computations on FPGAs. To do this we implement FPGA based lattice QCD processors using both double precision IEEE floating point and single precision equivalent Logarithmic Number System (LNS) cores and compare their performance with that of two lattice QCD targeted ASIC based solutions and with PC cluster based solutions.

1. INTRODUCTION

Recent multi-million gate equivalent FPGAs make it possible to implement complex, high-performance designs incorporating non-integer arithmetic. A significant branch of reconfigurable computing research now addresses problems of efficiently implementing the array and matrix operations that form the kernels of many scientific computations [1]. In this paper we investigate the performance of FPGAs as a platform for implementing *full* floating point based scientific applications.

In order to do this we implement a significant sub-atomic physics simulation called lattice QCD. Improving the performance of lattice QCD is the focus of considerable research work worldwide [2], including two competing ASIC based solutions targeted solely at lattice QCD simulations, [3] [4]. Several groups are also investigating optimizing PC clusters for lattice QCD, [5] [6] [7]. This considerable body of research allows us to compare the performance of our FPGA based solution with the state of the art for scientific computing.

The core of Lattice QCD is the *Dirac operator*, a large complex floating-point intensive matrix computation. Although the performance of the Dirac operator is central, Lattice QCD involves other operations which can have a significant impact on performance. We investigate this by implementing an example application, a lattice QCD conjugate gradient solver. The conjugate gradient solver uses the Dirac operator, a vector add-scale operator and a dot product operator, which are both normally memory bandwidth bound.

We also aim to compare the suitability of different non-integer arithmetic systems for performing high performance computing applications on FPGAs. We present a comparison of LNS arithmetic and IEEE double precision floating point and identify the strengths and weaknesses of both systems when used for real high performance scientific computing applications.

We present performance results for both LNS and IEEE double precision versions of the full conjugate gradient solver along with results for its component parts, including the performance critical Dirac operation. We also present quantitative comparisons for two categories of alternative lattice QCD systems: ASIC-based processors designed exclusively for lattice QCD and highly optimised PC cluster systems.

2. BACKGROUND

2.1. Quantum Chromodynamics

Quantum Chromodynamics (QCD) theory describes how quarks and gluons interact. QCD is believed to explain confinement, which is the observation that these constituents are inextricably bound inside hadrons like the proton and nucleon. This property of QCD explains how the proton and neutron are heavy, in spite of the fact the quarks are very light particles and the gluon is massless. However no free quarks or gluons have ever been observed experimentally. As a result numerical simulations of QCD performed on high-performance computers have been used for over thirty years in an attempt to make *ab initio*

* Sponsored by the Irish Research Council for Science Engineering & Technology under grant SC/02/288

1-4244-0 312-X/06/$20.00 ©2006 IEEE.

predictions about experimental results using QCD theory. This is done because the mathematical complexity of QCD makes it impossible to solve using traditional methods. These simulations provide vital inputs into the experimental searches for new physics at ever-increasing energy scales in the world's largest particle collider experiments. The simulations also attempt to explain the mechanism for confinement in QCD and so bring us to a better understanding of the fundamental nature of matter.

2.2. Lattice QCD

Lattice QCD is the term used to describe the application of computer simulations to QCD theory. Lattice QCD uses a set of matrices, usually known as the lattice, to simulate space and time at a sub-atomic level. Lattice QCD belongs to a general class of high performance computing algorithms called sparse matrix solvers. Other examples of sparse matrix solvers include Computational Fluid Dynamics and Finite Element Solvers. However lattice QCD is crucially different to these applications because in lattice QCD the matrix is constant whereas it must be re-formulated for every calculation in other applications. This allows the matrix representation to be built into the algorithm itself and so it is not explicitly represented. This means lattice QCD calculations sustain much higher performance compared to other sparse matrix solvers.
The lattice for a lattice QCD calculation takes the form of a set of large matrices. These matrices are hyper-cubes of four dimensions. The number of elements in these matrices is determined by the following formula.

$$NS = NX \times NY \times NZ \times NT$$

The four values dictate the number of points to be simulated in each of the three dimensions of space and also the dimension of time. NS tells us the size of the lattice. Current values of NS for large simulations are in the region of 2 million. Consequently obtaining a single scientific result for such a lattice needs approximately 6.6 Peta floating-point operations [6]. As such the computational requirements for lattice QCD are massive.

3. RELATED WORK

Lattice QCD is an important scientific application and is the focus considerable research work worldwide. Much of this work aims to improve the algorithm itself and make it more useful to the scientists and mathematicians who use it. Considerable effort is also expended on improving the performance of lattice QCD machines. This effort is in one of two broad fields. The first is the construction of massively parallel machines dedicated to lattice QCD calculations consisting of thousands of ASIC processors connected with a high bandwidth low latency interconnect. The other field uses commodity PC clusters for lattice QCD calculations focussing on using the vector processing extensions of PC processors along with interconnects such as Infiniband to build large clusters.

QCDOC [3] and apeNEXT [2] are both ASIC based machines specifically designed for lattice QCD. Both aim to deliver machines that can use tens of thousands of nodes on a single lattice QCD simulation. Both machines provide double precision IEEE floating point arithmetic. QCDOC prototypes can sustain over 5 Teraflops on a single problem by using 12,288 processing nodes on a single problem. ApeNEXT machines will have similar performance to the QCDOC machine. QCDOC nodes deliver about 396 MFLOPS each and apeNEXT nodes will deliver about 896 MFLOPS.

Clusters of commodity PCs are the focus of much research activity [4] [5] [7]. A cluster of dual Intel Xeon based PCs using Infiniband for interconnect and single precision arithmetic is described in [7]. This system returns around 1.1 GFLOPS per CPU for a realistic problem size on a sixteen CPU cluster.

Minimising communications overhead is vital for PC cluster performance. PC clusters have very good per node performance but have high communications overheads compared to the ASIC based machines. This restricts the number of nodes that can be applied to a single problem; QCDOC for example can use over 100 times more nodes on a given problem size compared to the cluster in [7]. The challenge for PC clusters is improving interconnect performance and not per node floating point performance.

In earlier work we presented an LNS implementation of the Dirac operator [8]. This operator has been substantially improved for the current paper and performance has been increased by 23% with a further 15% improvement by using a faster speed grade device.

4. NUMBER REPRESENTATIONS

IEEE floating point is the standard approach for performing non-integer calculations and is implemented on most commodity processors. The Logarithmic Number System (LNS) is an alternative approach to these calculations that uses fixed point logarithms to represent non-integer numbers. LNS requires vastly fewer resources for multiplication and division compared to IEEE floating point however addition and subtraction become significantly more complicated.

Lattice QCD calculations require at least single precision arithmetic and the end users *prefer* double precision as it gives more accurate results. However accuracy can also be increased with a larger problem size, which increases the demand for FLOPs. Single precision is used on PC clusters since they usually return double the performance this way. Double precision is used on other systems where the gap is not so large.

Table 1. Resource Requirements for LNS and Comparable IEEE Units

	LNS Multiplier [1]	Underwood Multiplier [9]	LNS Divider [1]	Underwood Divider [9]	LNS Adder (2 pipes) [1]	Underwood Adder [9]
Slices	83	598	82	1929	1648	496
Multipliers	0	4	0	0	8	0
Block RAM	0	0	0	0	28	0
Latency	1	16	1	37	8	13
MHz	250	124	250	100	90	165
Pipes per FPGA	407	36	412	17	10	68

4.1. Single precision log and floating point

We use commercial LNS cores from the High-Speed Logarithmic Arithmetic system (HSLA) by Matousek et al [9]. These are 32-bit logarithmic cores and support the full range of exceptions from the IEEE floating point standard. The cores are highly-optimized for both performance and space. A comparison of logarithmic arithmetic and IEEE floating point was presented in [10] The LNS cores we use are significantly different from those presented in [10] so we make our own comparison here.

Table 1 shows the resource requirements of the LNS arithmetic units compared to the requirements for comparable IEEE arithmetic units published in [11]. The Underwood cores' latencies are variable; shorter latencies are possible but with lower clock rates.

LNS has a clear advantage for multiplication and division. The LNS multiplier is substantially smaller than the Underwood multiplier and has a lower latency. Also the fully pipelined LNS divider returns similar performance to the Underwood divider but uses less than 5% of the resources and has a much lower latency.

The penalty for the small and low latency LNS multiplier and divider is that the LNS adder requires 66% more slices than the IEEE adder, and a significant quantity of block RAM and hardware multipliers. This block RAM requirement limits the number of LNS adder units to 10 per FPGA but requires only 25% of slices on our Xilinx Virtex-II-6000 FPGA, leaving plenty of space for control logic and other arithmetic units.

The resource requirements for the LNS cores favour applications that have a high proportion of multiplications compared to additions. Also the very small size of the divider is a particular advantage for applications which use division only rarely. A large IEEE divider is a waste of resources in such designs.

5. IMPLEMENTATION

We implement our designs using Handel-C, Celoxica DK 4.0 for synthesis, and Xilinx ISE8.1i for place and route. Our designs are tested in hardware using prototype boards from by Alpha Data (Model: ADM-XRCII) which include a Virtex-II XC2V6000 (speed grade 6) along with 6 banks of 32- bit wide SRAM.

5.1. Algorithm Analysis

Understanding an algorithm is essential to getting good performance for a hardware implementation. Table 2 shows the most relevant information about the lattice QCD codes. The first column shows the ratio of floating point operations to memory operations for each part of the algorithm. The second column shows the proportion of floating point operations that are adds or subtracts (almost all remaining operations are multiplies). The final column shows the proportion of each application part as a percentage of the whole application.

The Dirac operator is the largest consumer of non-integer calculations in the conjugate gradient application and is where most time is spent. It has low memory bandwidth requirements compared to calculation and has a balanced requirement for adds and multiplies. By comparison the dot product and vector add scale operations have much higher memory bandwidth requirements, so their performance is restricted by memory bandwidth, not non-integer calculation performance.

The Dirac operator is the most compute intensive and is where we concentrate our efforts. It is constructed from 4 operations, which operate on the *gl3* (3×3) and *wfv* (4×3) matrices. They are:

1. *Gamma* functions
2. Matrix-multiply; $wfv \times gl3 = wfv$
3. Matrix addition/subtraction; $wfv + wfv = wfv$
4. Matrix scale; $wfv \times value = wfv$

The *gamma* functions multiply a *wfv* matrix by an identity matrix to produce a *wfv* matrix. The Dirac operator uses 8 slightly different versions of this function. The identity matrix is constant so this is done using a small number of additions; this is the standard practice for the Dirac operator. The $wfv \times gl3$ complex number matrix multiply is the most compute intensive part of the calculation needing 264 floating point calculations per operation. The matrix addition is a straightforward matrix addition. The matrix scale scales every element in a *wfv* matrix by a particular value.

Internally each of these blocks has exploitable parallelism. The real and imaginary components of the numbers can be calculated in parallel. Also many of the blocks are independent and thus can be parallelized.

Table 2. Lattice QCD algorithm information.

	Calc : Mem	Add %	CG %
Dirac	6.81	55%	95.6%
Dot Product	1	50%	1.8%
Add-Scale	0.66	50%	2.6%
Total	5.07	54.8%	100%

The *gamma* and matrix multiply blocks are paired and we call them *gamma-mul* pairs. The eight pairs are independent and can be performed in parallel given sufficient resources. The *wfv* add blocks accumulate the eight results of the gamma-mul blocks into one *wfv* matrix. We can parallelise by performing an accumulate in three stages; add the 8 gamma-mul results into 4 matrices, these 4 into 2 and finally these 2 into one. Finally the scale block is dependant on the results from the set of add blocks.

5.2. Improved LNS Dirac operator

Our initial implementation of the Dirac kernel for LNS is outlined in detail in [8]. Performance for the LNS Dirac operator is restricted by the availability of a maximum of 10 on-chip adders (details in section 0). Due to this constraint we designed our architecture to make maximum use of each adder pipe. Significant improvements to the design presented in [8] raised performance for the LNS Dirac operator by 23%. Improvements to the architecture reduced the number of cycles required to calculate the result for each site from 190 to 168. Improvements to the application logic reduced logic delays enabling a higher clock rate for the design. Also moving to a faster speed grade device has boosted performance by a further 15%

5.3. LNS Conjugate Gradient Solver

The conjugate gradient application uses the Dirac operator from section 5.2. It also uses a vector dot product operator and scale-add operators which operate on vectors of 4×3 *wfv* matrices. Both operations are memory bandwidth bound for our implementation, as they are on most platforms. Memory bandwidth is dictated by the memory system used and by the data layout in that system. For our implementation the vector inputs to both of these functions are stored in paired banks of 32-bit on-board SRAM memory; the real components in one bank and the imaginary components in the other.

$$y_i = (k \times x_i) + y_i \qquad (1)$$

The scale-add operation is shown (1). *Y* is both read from and written to which restricts us to one multiply-accumulate operation per cycle. Vector *y* is stored linearly in a pair of memory banks to which we can either read or write once per cycle but not both. Data is streamed from memory to arrive at the adder or multiplier on precisely the cycle it is required. This ensures maximum throughput for the arithmetic units.

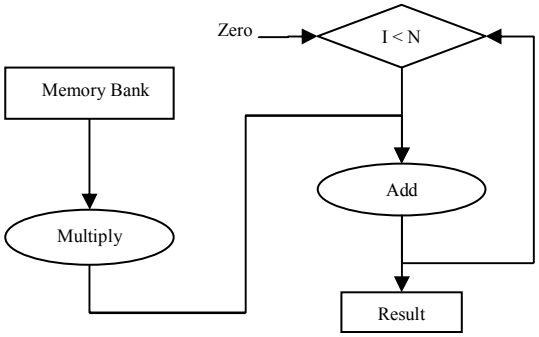

Fig 1. Architecture of the dot-product operator.

The dot-product is a global sum of the square of all the real and imaginary values in a vector. This is used to find how close the solver is to a solution by finding a global sum on every iteration and comparing it to the sum from the last iteration. If the difference is less than a certain threshold then the run is complete.

Fig 1 shows our implementation of the dot-product operation. We stream data in from memory and issue it directly to the multiplier to square it. Then we issue the multiply result directly to the adder along with the current add result eliminating any need for intermediate storage of the accumulate result. Zeroes are issued for the first *n* additions (*n* is the adder's latency); then the current output of the adder is returned to the adder as an operand. Thus the results of the multiplications are added to *n* running totals. Once the vector has been processed the *n* running totals summed. The real and imaginary components of the vector are stored in separate on-board RAM banks so we run two instances of the architecture shown in Fig 1 in parallel adding the result from each to give the final result.

5.4. Double precision Dirac operator kernel

Our implementation of the Dirac operator for IEEE double precision uses the Moloney cores [12]. The Dirac operator consumes 9 additions for every 8 multiplications (see Table 1) so ideally we want a balance of arithmetic units in about this ratio. Using 8 multipliers and 10 adders for the design gives a balance of units broadly in line with this, using 51% of the available slices. This leaves sufficient chip resources for application control logic and so is the balance we chose for our design.

Fig 2 shows the structure of the double precision Dirac operator and inset is the structure of the *gamma-mul* block implementation. Each *gamma-mul* block uses 1 adder and 1 multiplier to perform its operation which multiplies a 3x3 *gl3* matrix by a 4x3 *wfv* matrix to produce a *wfv* result matrix. Our implementation uses the multiplier to produce 6 partial products for each component (real and imaginary) of each point in the result *wfv* matrix. The partial products must then be summed, giving the result for each point. The first two multiply results are partial products of site [0,0],

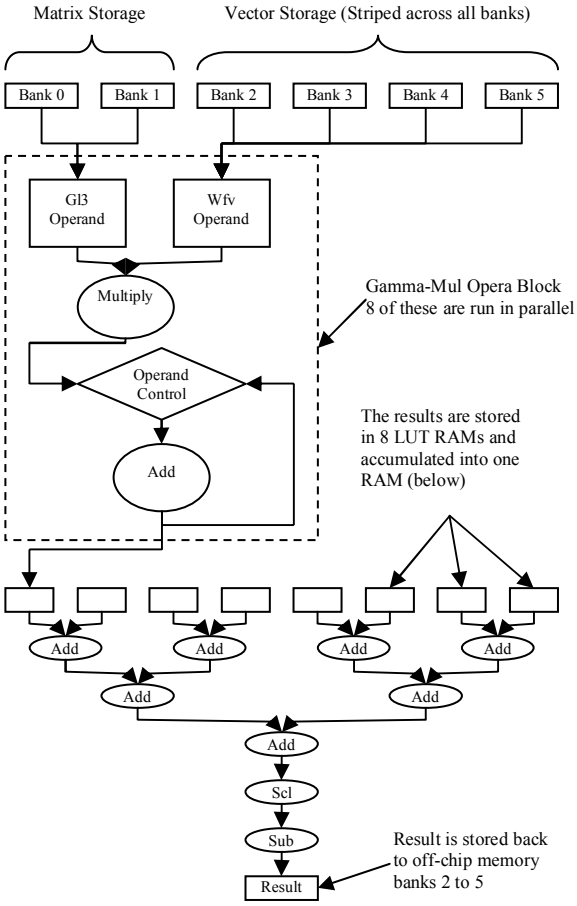

Fig 2. Structure of IEEE Dirac operator.

the next two are of site [0,1] and so on to site [3,2]. This sequence is repeated three times. The first multiply result is held in a register and issued in the next cycle, with the current multiply result, to the adder pipe. This is the first stage of summing the partial products.

These adds can only be issued on every second cycle, leaving spare slots in the adder pipeline. These are filled by accumulating the results of the first stage of partial product summation into 24 running totals (one for the *r* and *i* components of each point in the result *wfv* matrix). These totals are stored in a pair of LUT RAMs. Once all the partial products are summed these RAMs hold the result. Once all the first stage sums are complete second stage sums are issued on every cycle making best use of the adder.

The *add/subtract wfv* blocks are implemented by streaming data from the result RAMs of the *gamma-mul* blocks into two adder pipes, one pipe handling the real and the other the imaginary components.

The Dirac operator is then split into 4 pipeline stages, parallelizing operand retrieval, *gamma-mul* calculations, *add/subtract wfv* calculations and result write. In order to prevent Read After Write data dependencies between the *gamma-mul* stage and the *add/subtract wfv* stage it is necessary to copy some of the results from the *gamma-mul* stage into temporary storage.

Table 3. Performance of FPGA based solutions.

	Clock Rate (MHZ)	MFLOPS	FP Ops per cycle
IEEE Dirac	85	1200	14.1
IEEE CG	85	918	11.1
LNS Dirac	85	1320	15.5
LNS CG	85	1050	12.35

5.5. Double precision conjugate gradient solver

The double precision conjugate gradient solver requires double precision versions of the *dot-product* and *vector scale-add* operations. Data layout in memory determines how the operations are implemented. Double precision variables are 64 bits wide, so the ratio of memory bandwidth to calculation is doubled compared to single precision, making data layout very important. The *wfv* vectors are stored across 4 memory banks, using all six would increase bandwidth but would make the memory access hardware unfeasibly complex.

Our *vector add-scale* implementation streams data for one vector from memory into a multiplier where it is scaled by a fixed value. The result is then added to the appropriate value for the other vector. Both vectors are stored in the same memory banks so retrievals must be alternated between the two input vectors. The results are buffered in block RAMs before being written out to memory when the block RAM is full.

For the dot product implementation we use a similar architecture to the one employed successfully for the LNS version. Data is streamed directly from external RAMs into the multiplier to scale it. The results are then accumulated using the architecture shown in Fig 1. The adder's shorter latency means there are fewer partial products to be accumulated so performance is improved slightly compared to the LNS implementation.

6. RESULTS

In Table 3 we present performance results for our IEEE double precision implementations of the Dirac operator and the conjugate gradient solver. We also present results for our LNS implementation of the conjugate solver along with results for a substantially improved version of the LNS Dirac operator presented in [8]. All designs have been placed and routed for a Xilinx Virtex II XC2V6000 speed grade 6 device.

We obtain excellent performance for our double precision implementation of the Dirac operator with over 1200 MFLOPS sustained and for the double precision conjugate gradient solver with over 940 MFLOPs

Improvements to the LNS Dirac operator have boosted performance to 1320 MFLOPS. We also use this improved

Table 4. Performance of double precision implementations and comparable systems

	Dirac	CG
FPGA	1200	940
apeNEXT	894	-
QCDOC	396	351
PC	550	-

operator to implement an LNS conjugate gradient solver with performance of over 1050 MFLOPS.

We also present data for the average number of floating point operations performed per cycle for each design. The double precision Dirac operator achieves over 14 operations per cycle, whilst the LNS Dirac operator performs over 15 per cycle. This clearly demonstrates the level of parallelism that is exploited in our designs.

7. CONCLUSIONS

In recent years large FPGAs and the availability of arithmetic cores have made high-performance scientific computing increasingly practical on FPGAs. Lattice QCD is an important scientific application and is the focus considerable research work worldwide, with a variety of PC based and custom ASIC implementations. Thus it is ideal for evaluating a computing platforms' suitability for scientific computing. We have presented the design and implementation for FPGAs of the Dirac operator and a full Lattice QCD application using LNS and IEEE double precision floating point.

As discussed in Section 4, either single or double precision can be used for lattice QCD, however double precision is preferred since it is more accurate for a given problem size. PC clusters have significantly higher performance for single precision compared to double, so for these machines single precision is normally used with a larger problem size to compensate for the lower precision.

Table 5 shows the performance of our single precision equivalent LNS designs compared to an Intel Xeon PC Cluster node from [7]. We achieve 1320 MFLOPS for the Dirac operator and 1050 MFLOPS for the full application. This compares well with the *single precision* performance of a PC cluster node of 1100 MFLOPS for the Dirac operator. Nonetheless, Lattice QCD, like most scientific applications that operate on matrices, has roughly the same numbers of adds and multiplies, and few divides. The large block RAM tables required by LNS adders were always the limiting factor in the LNS design.

Floating point units have no such limitations, so it was possible to build IEEE *double precision* floating point implementations that achieve **1200** MFLOPS for the Dirac operator and **940** MFLOPS for the full application using ten double precision adders and eight multipliers. Our double precision implementations are far more complex, however, because the pipelines are deeper and memory bandwidth, available block RAMs and slices are all critical constraints on the design.

Table 5. Performance of LNS FPGA implementation and comparable system

	Dirac	CG
FPGA	1320	1050
Intel Xeon	1100	-

Table 4 shows the performance of our implementation with comparable systems. This result compares extremely well with the QCDOC nodes which return 396 MFLOPS, with the apeNEXT nodes which return 894 MFLOPS per node and also with the PC cluster nodes which return about 550 MFLOPS at double precision [7].

Our results show that FPGAs can be competitive with general purpose processors and even custom ASIC processors for scientific computing applications such as Lattice QCD. To our knowledge this is the first FPGA implementation of Lattice QCD and one of the first full implementations of a large scientific application using IEEE double precision arithmetic.

References

[1] L Zhuo, V Prasanna, "Sparse Matrix-Vector Multiplication on FPGAs", *Proc. 2005 ACM/SIGDA 13th Int Symposium on Field-Programmable Gate Arrays*, pp.63.

[2] T Wettig; "Performance of machines for lattice QCD simulations", *Lattice 2005*, Proceedings of Science, ref. PoS(LAT2005)019.

[3] F Belletti et al, "Computing for LQCD: apeNEXT" Computing in Science & Engineering, vol.8, no.1 pp. 18- 29.

[4] P A Boyle et al., "Overview of the QCDSP and QCDOC Computers," *IBM J. Research and Development*, vol. 49, nos. 2–3, 2005, pp. 351–365.

[5] Dom Holmgren; "PC Clusters for Lattice QCD", Published in *Nuclear Physics Proceedings Suppl.*140:183-189,2005.

[6] A Gellrich, D Pop, P Wenger, H Wittig, M Hasenbusch, K Jansen, "Lattice QCD Calculations on Commodity Clusters at DESY", in *Proc. Computing in High Energy Physics 2003,* Published by eConf, ref C0303241.

[7] Don Holmgren; "Cluster Development at Fermilab", All Hands Meeting, Jefferson Lab, Virginia, USA, June 2005, available at *http://lqcd.fnal.gov/allhands_holmgren.pdf*

[8] O Callanan, A Nisbet, E Ozer, J Sexton, D Gregg, "FPGA Implementation of a Lattice Quantum Chromodynamics Algorithm Using Logarithmic Arithmetic," in *Proc Parallel and Distributed Processing Symposium 2005*, pp. 146b.

[9] R Matousek, M Tichý, Z Pohl, J Kadlec, C Softley, N Coleman, "Logarithmic Number System and Floating-Point Arithmetics on FPGA", in Proc. FPL 2002, LNCS, Volume 2438, Jan 2002, p.627

[10] M Haselman, M Beauchamp, A Wood, S Hauck, K Underwood, K S Hemmert, "A comparison of floating point and logarithmic number systems for FPGAs," in *Proc. IEEE FCCM 2005*, pp. 181- 190.

[11] K Underwood, "FPGAs vs. CPUs: trends in peak floating-point performance" in *Proc. 2004 ACM/SIGDA 12th Int. Symposium on Field Programmable Gate Arrays*, pp.171.

[12] D Moloney, D Geraghty, and F Connor, "The performance of IEEE floating-point operators on FPGAs", *IEE ISSC 2004* IEE Conf. Pub. 2004, Vol. CP506 p.601.