FPGA Implementation of a Lattice Quantum Chromodynamics Algorithm Using Logarithmic Arithmetic

Owen Callanan[†], Andy Nisbet[‡], Emre Özer[†], James Sexton[Δ] and David Gregg[†].

[†]Dept.of Computer Science, Trinity College, Dublin, Ireland. {Owen.Callanan, Emre.Ozer David.Gregg} @cs.tcd.ie

[‡]Dept. of Computing and Mathematics, Manchester Metropolitan University, UK. A.Nisbet@mmu.ac.uk

[Δ]Dept. of Mathematics, Trinity College, Dublin sexton@maths.tcd.ie

Abstract

In this paper, we discuss the implementation of a lattice Quantum Chromodynamics (QCD) application to a Xilinx VirtexII FPGA device on an Alpha Data ADM-XRC-II board using Handel-C and logarithmic arithmetic. The specific algorithm implemented is the Wilson Dirac Fermion Vector times Matrix Product operation. QCD is the scientific theory that describes the interactions of various types of sub-atomic particles. Lattice QCD is the use of computer simulations to prove aspects of this theory. The research described in this paper aims to investigate whether FPGAs and logarithmic arithmetic are a viable compute-platform for high performance computing by implementing lattice QCD for this platform. We have achieved competitive performance of at least 936 MFlops per node, executing 14.2 floating point equivalent operations per cycle, which is far higher than the previous solutions proposed for lattice QCD simulations.

1. Motivation

We aim here to evaluate the viability of using FPGAs and logarithmic arithmetic [1] to accelerate high performance computing applications by implementing a specific application, lattice QCD, on this platform. FPGAs have considerable potential to accelerate computationally intensive applications and particularly those with exploitable instruction level parallelism such as lattice QCD. Traditionally floating point arithmetic units have been too large to fit in an FPGA; logarithmic arithmetic units provide a resource efficient alternative to these units. Logarithmic arithmetic has several advantages over conventional floating point systems and the most important is that multiplication, division and square root operations are very fast and require few resources as part of an FPGA design. Thus logarithmic arithmetic and FPGAs have the potential to provide a viable platform for many computationally intensive applications.

Lattice QCD [4] is a simulation of sub-atomic particle interactions and is used to verify theories about the Strong Force of Nature, which is described by QCD theory. Computer simulations must be used to verify QCD theories since it is impossible to observe these interactions experimentally. Thus, since QCD describes the very nature of matter, lattice QCD is a grand challenge high performance computing application. First, lattice QCD is converted into logarithmic floating-point domain [2], and then ported to a high-level hardware description language, Handel-C [3]. Handel-C describes behaviour at an algorithmic level rather than at a logical level, as is the case with conventional Hardware Description Languages (HDLs) such as VHDL and Verilog. This high-level approach eases the modelling of complex systems and reduces the time required to specify and implement an FPGA system

In this paper, we discuss the implementation of a custom processor for the lattice QCD application discussed in Section 2. Section 3 describes the logarithmic arithmetic cores used in this work and compares them with conventional IEEE floating point arithmetic. Section 4 describes how the application was implemented in hardware using FPGAs. Section 5 shows the experimental results. Section 6 discusses related work. Section 7 gives the conclusions and details of possible future work.

2. QCD and Lattice QCD

2.1. QCD

We know that all matter is made of atoms of one type or another. We also know that these atoms are constructed out of many different combinations of protons, neutrons and electrons. Protons and neutrons are examples of a class of particle called hadrons which we now know to be constructed from quarks and gluons. However we do not know whether quarks and gluons are indivisible or whether they can be sub-divided further, answering this question will require experimental analysis. Quarks and gluons are bound together by the Strong force of nature and QCD is the scientific theory that describes this force [11]. QCD predicts that quarks can never be individually seen and that they can only ever be seen bound with gluons and other quarks to form a hadron. Thus in order to obtain information about quarks QCD theory must be solved. Currently the best way to do this is through the use of computer simulations referred to as lattice QCD.

2.2. Lattice QCD

There are many aspects of lattice QCD that can be simulated using computer simulations. In this study, the aspect of lattice QCD implemented is a *Wilson Fermion Vector times Matrix Product.* Lattice QCD simulates a finite portion of space and time at a sub-atomic level. This portion of space and time is represented by a four dimensional grid of points, or sites, called a lattice. The data for each site in the lattice is held in a large matrix composed of groups of four small matrices of complex numbers. These small matrices are called *gl3* matrices. Data is also stored for the links between sites in the lattice. This data is stored in another large matrix composed of another type of small complex number matrices which are called *wfv* matrices. One wfv matrix is used per point in the lattice.

A lattice QCD calculation consists of a very large number of *sweeps* of the lattice. A sweep is performed by updating the *wfv* matrices for each point of the lattice once, with each update taking the result of the previous update as the input *wfv* value for the current update. The update of each site is independent of the update of any other site, allowing a single sweep to be performed by multiple computing nodes. The data held in the *gl3* matrices is not changed.

A site update is performed by the following steps:

1. The *wfv* matrix for each of the neighbouring sites is put through a *Gamma* operation. This operation consists of twelve additions or subtractions and so is very simple. There are 8 matrices upon which this calculation is performed.
2. The result of step 1 is then multiplied by one of the *gl3* matrices for the current site resulting in a *wfv* matrix. Each site has eight neighbours (one in each dimension) so eight versions of this calculation (referred to here as *Mul* calculations) are performed.
3. The next step is to add together all 8 *wfv* matrices produced by step 1. The resulting *wfv* matrix is then multiplied by a scaling factor called *kappa.*
4. The result of step 2 is then subtracted from the existing value for the current point.

The *Mul* operations are the most computationally demanding part of the application since they require two complex number matrices to be multiplied. In comparison the calculations of stages 1, 3 and 4 are all variations on matrix additions and so are less computationally demanding. Each of the 8 *Gamma* operations in stage 1 are paired with a *Mul* operation from stage 2 and these pairs may be considered as one operation referred to here as *GammaMul* operations. All 8 versions of the *GammaMul* operations are independent and may be performed in parallel given sufficient functional units. The additions of the results of stage 2 may also be parallelised, but to a lesser extent, by performing the additions in 3 stages with 4 matrix additions in the first stage, 2 in the second and 1 in the third, with each stage not beginning until the results of the previous stage are available. It is also possible to exploit parallelism internally within the components of each stage by using multiple functional units to perform a single matrix operation. Since most of the operations are performed on small complex number matrices there is considerable exploitable parallelism within these steps, given sufficient availability of functional units.

A very important issue with a lattice QCD simulation is how many points the lattice contains. The total number of points in the lattice, *N,* is the product of each of the four dimensions in the lattice as follows;

$$\mathbf{N = NX \times NY \times NZ \times NT}.$$

Where *NX*, *NY*, *NZ* and *NT* are the number of points the lattice has in each of its four dimensions. *N* is important because calculations performed at higher values of *N* are more scientifically interesting than those performed with lower values for N; however since there are more points in the lattice the calculation takes longer to complete. Scientifically valuable results are now being obtained from very large problem sizes with upwards of 24^4 lattice points. Consequently it is unacceptably slow to complete a lattice QCD calculation on a single compute node; many nodes are needed for such calculations. To update a single lattice point requires approximately 2420 floating point operations thus a sweep of a 24^4 lattice requires over 800 million floating point operations and given that many millions of sweeps are required for a solution it can be seen that the computational requirements for a full lattice QCD simulation are vast.

3. Logarithmic Arithmetic

We aim here to explore the usefulness of logarithmic arithmetic and FPGAs as a platform for performing floating point based applications. We do this by implementing a specific floating point based application, lattice QCD, for FPGAs using logarithmic arithmetic. Conventional floating point does not translate well to FPGAs since some of the structures required for a floating point calculation, such as the multiplier tree used in the multiplier units, require considerable resources. Logarithmic arithmetic offers an alternative approach to such systems and it is the viability of this alternative that we aim to evaluate here.

Logarithmic arithmetic is an alternative approach to the IEEE standard for performing floating point arithmetic calculations. The main advantage of the Logarithmic Number System (LNS) lies in the simplicity and speed of the multiplication, division and square root operations. These operations are largely reduced to the complexity of an integer addition and consequently the hardware required to perform them is very small and fast. They are implemented in the package used here as 2 cycle latency pipelines which require only about 80 slices on an FPGA for each unit.

LNS addition, and consequently subtraction, uses the block RAMs available on Virtex II FPGAs to store look-up tables that are used to perform addition and subtraction operations. The Xilinx Virtex II includes a significant amount of on-chip RAM, called block RAM, which can be used for data storage in a design. This block RAM has two ports allowing two accesses to be made to the RAM in a single cycle. The Virtex II also includes 18 by 18 bit multipliers that can multiply two numbers each 18 bits wide without using any LUTs to perform the operation.

The hardware implementation of LNS addition and subtraction involves the use of look-up tables to calculate the result and these look-up tables require significant amounts of on-chip storage. The process of performing the addition involves several look-up operations and consequently the adder pipelines have a 9-cycle latency. The implementation of the LNS adder for Virtex II FPGAs used here uses the dual ported on-chip block RAM incorporated into the Virtex II design to store these look up tables. This dual porting allows two adder pipelines to share the same set of look up tables and for this reason the LNS adder pipelines are always instantiated in pairs.

With IEEE floating point, the situation is somewhat reversed with multiplication being the more complex operation and requiring the greater amount of resources to implement when compared to IEEE addition. Complexity in IEEE addition primarily arises in the addition of the mantissas and in the normalisation of the additions result. In comparison, the IEEE multipliers principle source of logic complexity is the multiplication of the mantissas with the rest of the operation being relatively simple.

Table 1 shows resource requirements for fully pipelined implementations of the LNS multiplier along with the requirements for two different IEEE format multiplier units. The Clemson units [12] are the result of work published in 1998 and do not make use of the hardware multipliers available in Virtex II FPGAs. The USC multiplier [13] is the result of work published in 2004 makes use of the hardware multipliers. It can be seen that the LNS format multiplier is over an order of magnitude smaller than the Clemson multiplier. In addition the LNS multiplier requires significantly fewer flip flops and fewer LUTs than the USC multiplier and it also requires no hardware multipliers. The small size of the LNS multipliers along with their low pipeline latency of two cycles makes the LNS multipliers a viable alternative to IEEE format floating point multipliers including those that use the Virtex II hardware multipliers.

Table 1. Resource requirements for LNS multiplier and comparable units

Resource Type	**LNS**	**Clemson**	**USC**
Flip flops	67	1017	249
LUTs	159	759	196
Block RAM	0	0	0
On-Chip Multipliers	0	0	4

Table 2. Resource Requirements for LNS adder and comparable units

Resource Type	**LNS**	**Clemson**	**USC**
Flip flops	1778	1342	1040
LUTs	2539	1248	1096
Block RAM	28	0	0
On-Chip Multipliers	8	0	0

Table 2 shows the resource requirements of the LNS adder used in this work, the Clemson adder [12] and the USC adder [13]. LNS adder pipes can only be instantiated in pairs; this ensures that the pipes make efficient use of block RAM. It can be seen that whilst the resource requirements for the LNS adder are greater than those for either IEEE format adder, they are nonetheless reasonably comparable. This combined with the low pipeline latency and small size of the LNS multipliers makes the LNS arithmetic units a competitive solution for performing floating point calculations on FPGAs.

The lattice QCD application described here involves nearly equal numbers of floating point additions and multiplications, making it well suited to take advantage of the small size and high speed of the LNS multipliers.

4. Implementation

4.1. Conversion of Floating Point Numbers to LNS Format

The LNS functional units will only operate on operands that have already been converted to the LNS system; the operands are *not* converted on the fly. So when a design is implemented in LNS all floating point data are transferred to the chip and stored on chip in the LNS format. The host performs all necessary conversion prior to sending the data to the FPGA. This means that there is no conversion overhead to using the LNS system.

4.2. Porting Applications to Logarithmic Arithmetic

Included with the logarithmic ALUs are C-level logarithmic emulation libraries, which allow a C program to be converted to perform all its calculations in logarithmic arithmetic. Also included are Handel-C simulation libraries which allow Handel-C code that uses logarithmic arithmetic to be run simulated correctly. Conversion functions to/from the logarithmic/real domain are also provided. The approach taken here in converting a C application to Handel-C using logarithmic arithmetic is: 1) incrementally port functions to logarithmic arithmetic using ANSI-C emulation libraries, 2) verify that no error has been introduced by calling the converted function from the application inside a wrapper function which converts the operands into log format and the results back to IEEE format. The results will not be identical since the FPGA is running single precision and the original version is performed at double precision. 3) Convert the log ANSI-C application to simulated Handel-C incrementally, and verify the results produced are correct. Handel-C allows the calling of ANSI-C functions and this facilitates incremental porting, 4) add hardware-specific functionality to the Handel-C design using the ADM XRC-II board support package, 5) generate an EDIF netlist with the Handel-C compiler and finally 6) place and route the netlist using Xilinx tools and run it using the ADM-XRC II board.

4.3. Handel-C

Handel-C is a C level hardware description language. It is similar to C but it has several extensions to support the instruction level parallelism available in FPGA architectures. The principle difference between Handel-C and conventional HDLs, such as VHDL or Verilog, is that conventional HDLs are good for describing the behaviour at a logical level, whereas Handel-C describes behaviour at an algorithmic level. As a result of this Handel-C hides much of the complexity that is inherent in the use of conventional HDLs. Thus for small designs Handel-C is often not an ideal choice since it can prevent a designer from constructing precisely the right circuit, however for large designs, such as the one described in this paper, it is much more suitable since by hiding much of the detail it makes large designs much more tractable.

Handel-C was chosen for this project since the original application was written in C and by using a C based hardware design tool it was possible to significantly simplify the process of converting the application to an FPGA-based implementation. Handel-C includes functionality to allow a design consisting of part ANSI C and part Handel-C to be simulated. The facility allows the process of porting a C application to Handel-C to be performed incrementally, one function at a time. The feature is a considerable advantage for the work described here. Also given that the design is significant in terms of size and complexity the level of abstraction offered by Handel-C was very attractive.

4.4. Optimisation and Parallelisation Strategy

The key to obtaining good performance with this application is to maximise the number of floating point operations that are performed per cycle. The first priority is to include as many floating point units as possible in the design, this maximises the potential peak floating point performance. Analysis of the resource requirements of the two types of floating point units required, adders (which can also perform subtractions) and multipliers, reveals that a maximum of 5 pairs of adder pipes will fit on the available chip at one time, giving a total of ten adder pipes. The reason for this limit is that each pair of adders requires 28 block RAMs and there are a total of 144 block RAMs available on the device available. The multipliers meanwhile require few resources and consequently a large number can be instantiated in a design. Thus it is important to maximise the use of the adders in the design in order to obtain good performance.

Analysis of the application reveals that there is good match between the number of available adders and the number of functions in the application. The application contains 8 pairs of functions (steps 1 and 2 described in **Section 2.2**) which will be referred to as the *GammaMul* functions and each pair has identical requirements for floating point functional units. The 8 functions are all independent of each other and thus can be performed in parallel. Each function involves 144 multiplications and 156 additions. The remainder of the application consists mainly of matrix additions and a matrix scale operation where each element in the matrix is multiplied by a constant. This section of the application requires 192 additions and 24 multiplications to be performed. We determined that if a single adder pipe is devoted to each of the *GammaMul* functions and a pair is devoted to the set of matrix addition functions then the *GammaMul* functions require a similar number of cycles to the set of addition functions to complete.

This approach allows the 8 *GammaMul* functions to be performed in parallel. Each of the *GammaMul* functions is then internally optimised using pipelining techniques and by performing other calculations, such as loop counter increments, in parallel with the floating point calculations so as to make best use of the adder pipes assigned to it. Sufficient multiplier units are instantiated for each function to ensure that the adder pipes never have to stall whilst waiting for multiplications to complete. The collection of addition and subtraction functions is integrated into one function so as to make best use of the allocated adders by minimising pipeline flushes.

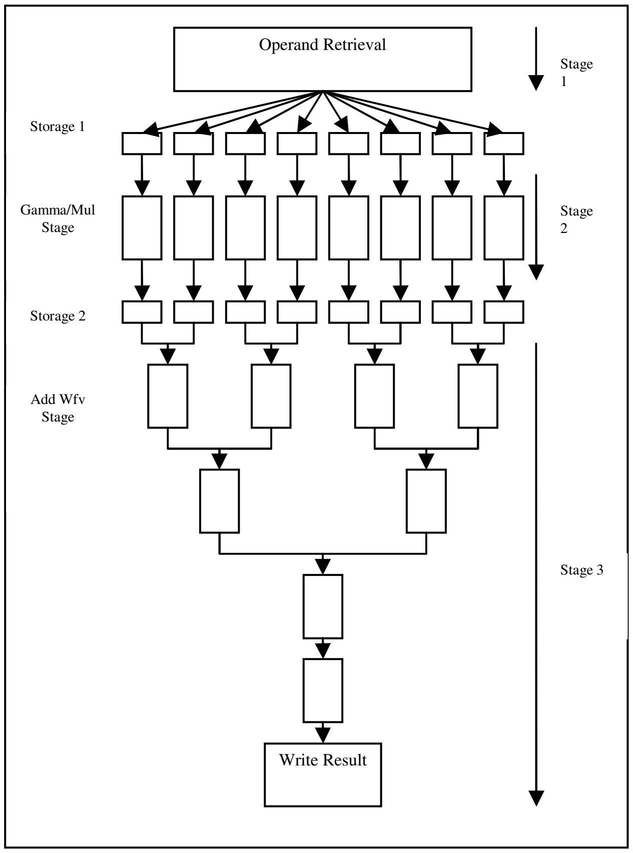

Figure 1 - Structure of pipelined optimized application

Also since the addition functions cannot be performed until the *GammaMul* functions are complete it was decided that the best approach would be to convert the application itself into a pipelined design. Thus operand retrieval, the *GammaMul* functions and the matrix addition functions are parallelised by performing each set of operations on its own pipeline stage. **Figure 1** shows the structure of the fully optimised and pipelined application. This approach allows very high overall utilisation levels for the adder units. The allocation of the functional units is important since the scheme chosen means that each of the pipeline stages requires a similar number of cycles to complete creating a well balanced pipeline. It is important to note that Handel-C is a synchronous language in which each statement takes exactly one cycle. Thus no explicit synchronisation is needed.

An important aspect of the application structure shown in **Figure 1** is that since each function only uses of a small number of functional units nearly all data storage are implemented by using distributed RAM. Distributed RAM is constructed using the Look Up Table (LUT) elements of the FPGA. LUTs are normally used for synthesising logic within the FPGA however they may also be used to construct RAMs for data storage. Distributed RAMs require fewer resources than arrays of registers, which are constructed using the flip flop elements of the FPGA. However distributed RAMs, unlike register arrays, only allow a maximum of two parallel accesses to be made to a single RAM in a single cycle. The data access pattern created by the chosen distribution of the functional units does not require more than two parallel accesses to any bar one set of arrays; consequently distributed RAM is ideally suited for storing most of the application data. Using distributed RAM significantly reduces both the resource requirements and the routing complexity of the design.

Once the structural optimisations outlined above were completed optimisation aimed at maximising clock rate began. The original clock speed of the design was about 30 MHz which was much slower than the maximum clock rate of the arithmetic cores. Consequently we felt that there was much scope for improving this figure. This was done by placing and routing the design and then running the result through the Xilinx Timing Analyser tool. Timing Analyser can be used to find the paths in the design with the longest delays, which are the paths that are limiting clock rate. The names of all the nets in the placed and routed design contain the name and line number of the Handel-C source file that generated them. Tracing back to the Handel-C source allows the source delay to be found so that logic with a shorter delay can be used instead. This process was then repeated several times until no more clock rate improvements could be made. Examples of long delays removed are complex condition checks in if statements and while loops, complex arithmetic operators such as modulus and nested if then else statements. The clock rate of the design was improved from about 30 MHz to 66 MHz by this process.

5. Experimental Results

The design described here is implemented on a six million gate equivalent Xilinx Virtex II FPGA. The specific FPGA used is a Xilinx VirtexII xc2v6000-ff1152-4. This FPGA is part of an Alpha Data ADM XRC-II development board. This board has 6 banks of external RAM with 2MByte of storage per bank. Both the host computer and the FPGA can access these RAMs using an Alpha Data board support package. FPGA access to RAMs is fully pipelined. In this paper, the host PC is used to initialize data and convert it to the logarithmic domain, the data is then transferred to the RAM and the FPGA performs the lattice QCD calculation; writing the result back to the RAMs. The host computer then retrieves the results from the RAMs and checks them for correctness.

5.1. Performance and Area Results

Table 3 presents performance results for an FPGA implementation of lattice QCD using single precision logarithmic arithmetic running at 66 MHz. All measurements are lattices of the dimensions N^4. The clock rate of 66 MHz was the best clock rate obtained after aggressive timing analysis aimed at reducing long logic and routing delays in the design along with maximum effort place and routing. Results are presented for a fully pipelined version of the design described in **Section 4.4**. We have achieved very good results with no degradation for increasing values of *N*. Hence our solution is scalable with increasing problem size and suffers no degradation.

Table 3. FPGA lattice QCD implementation performance results in MFLOPS

Lattice Dimension, N	Performance (MFlops)	FP-IPC
4	936.6	14.2
5	936.9	14.2
6	937.1	14.2
7	937.3	14.2
8	937.5	14.2
9	937.6	14.2
10	937.6	14.2

Also we can achieve 14.2 floating point instructions per cycle (FP-IPC) for all values of *N*. Floating point instructions per cycle is an excellent indicator of the level of parallelism that a design achieves and a high figure indicates that the functional units in a parallel processor are heavily utilised. FP-IPC is calculated by dividing the number of floating point instructions performed per second by the clock rate of the processor.

Table 4 presents resource requirement data for the design implemented on a Xilinx VirtexII XC2V6000. As can be seen from the data in the table the design uses nearly all (97%) of the available block RAM along with a substantial number of LUTs (55%). Comparatively few flip flops are used (24%) since registers are not used extensively for data storage. The reason for the high utilisation of block RAM in the device is due to the substantial number of block RAMs required by each pair of adders. It can be seen that the design fit easily into the available FPGA barring the high requirement for block RAM.

Table 4. Resource requirement data for complete design

Resource Type	Total Used	Percentage Used	Total Available
LUTs	37,752	55%	67,584
Flip Flops	16,276	24%	67,584
Block RAM	141	97%	144
Multipliers	40	28%	144

6. Related Work

There are two existing approaches to performing lattice QCD calculations that are relevant for comparison to the work described here. These are PC based clusters and specialised lattice QCD machines based around customised ASIC processors. **Table 5** shows per node performance data in MFLOPS for the two types of relevant lattice QCD platforms. The PC cluster solution described here is the result of work described in [7] [8]. This solution uses a cluster of 128 PCs connected using Gigabit Ethernet and makes use of the SSE2 capabilities of the Pentium 4 processor to improve the floating point performance of the design. These systems use single precision floating point. It also makes use of the pre-fetch instruction included in the Pentium 4 instruction set to reduce cache misses in the design. PC based solutions can show extremely high performance, where a single node is used for a very small problem size. Under these conditions nearly 2 GFlops is possible for a 1.7 GHz Pentium 4 in [7] [8]. However this performance drops away dramatically to about 1.5 GFlops as the problem size increases. This is because the application dataset no longer fits in the processor's cache and the cache misses that this causes degrade performance. Also once communication is turned on so that more than one node can be used the performance degrades even further to 1 GFlops for a 4 node system down to 750 MFlops for a 64 node system. This is because the processor must stop calculating in order to perform the inter-node communication. Results presented in [8] show a per node problem size of 8^4 is the minimum size that each node can calculate without inter-node communication dominating the calculation and thus degrading performance. Obviously this is not ideal since the PC nodes return their best performance at very small problem sizes and this minimum problem size contributes to the limited scalability of PC based clusters. These figures for a PC based system are all for single precision arithmetic, the figures for double precision are roughly half those for single.

The ApeNEXT [9] and QCDOC [10] systems both have ASIC processors designed specifically for performing lattice QCD calculations. Both of these systems use double precision floating point arithmetic. Being custom designs they have been tailored to the demands of lattice QCD calculations so that their inter-node communication has a very low latency and is performed in parallel with the calculation. This means that whilst these systems return similar per node performance to PC cluster nodes they allow clusters of many more nodes can be constructed, up to 4000 nodes compared to 128 for the PC clusters. Also the minimum per node problem size is much smaller due to the low latency of the communication and that it is performed in parallel with calculation. The sustained figures presented for these systems here are the measured performance of the chips performing real lattice QCD calculations at double precision; these systems are ASIC based and consequently only double precision floating point hardware is included on chip. **Table 5** shows the sustained per node performance in MFLOPS, clock-rate in MHz and IPC number of these studies.

Table 5. Performance of existing lattice QCD machines

	Single Node	Multiple Node	Clock Rate	IPC
PC Cluster	1500	750	1700	0.88/ 0.44
QCDOC	-	535	500	1.07
ApeNEXT	-	800	200	4
FPGA	936	-	66	14.2

Comparison between the performance data in **Table 3** and **Table 5** shows that the single node sustained performance of our FPGA implementation compares very well with each of the relevant systems discussed here. Whilst our single node performance is not as good as that of the single node PC it is still compares well. Unfortunately no single node performance data is available for either the QCDOC or ApeNEXT systems. However both systems use very low latency interconnects and both scale to very large numbers of nodes. This gives us reason to believe that the single node performance of either system would be quite close to the multiple node performance. Once again our performance compares well with the performance figures for either of the two solutions.

It is possible to connect multiple FPGAs using a low latency high bandwidth inter-connect in a similar fashion to either the QCDOC or ApeNEXT systems to perform lattice QCD calculations. The performance of FPGAs running lattice QCD interconnected in such a way would not cause a significant degradation of per-node performance relative to a single node system. Also the customisable nature of an FPGA design would allow the inter-node communication to be performed in parallel with calculation thus avoiding stalling the calculation as is the case for a PC based system.

7. Conclusions and Future Work

Our goal is to investigate the suitability of FPGAs combined with logarithmic arithmetic for performing high performance computing calculations. Lattice QCD was chosen as an example application to investigate this suitability. Lattice QCD is the focus of much research work worldwide and the results of some of this work are presented in this paper.

We have shown that the combination of logarithmic arithmetic and FPGAs can be an effective platform for implementing high performance computing calculations. Our performance results for a lattice QCD application implemented on FPGAs are competitive with the results of previous research work carried out in this field. This means that FPGAs and logarithmic arithmetic are a viable platform for performing high performance computing applications such as lattice QCD.

Two directions for further work on this subject are open at this point. One is to investigate the potential for using multiple FPGAs to accelerate a single high performance computing calculation, such as a lattice QCD simulation. This could be pursued by using the results of the work described here as the basis for a two node design. The results of such work would show what future potential there is in using clustered FPGAs for high performance computing. Also double precision IEEE compatible floating point cores have very recently become available which would allow us to implement lattice QCD at double precision using FPGAs. The cores are very resource efficient and would allow such an implementation to have floating point with performance at the level of the work described here.

References

[1] R. Matousek, M. Tichy, Z. Pohl, J. Kadlec and C. Softley, “Logarithmic Number Systems and Floating-point Arithmetics on FPGA”, *12th International Conference on Field Programmable Logic and Applications*, Montpellier (France), Sep. 2002.

[2] I. Kohen, “Computer Arithmetic Algorithms”, *A. K. Peters*, Natick, Massachusetts, 2002.

[3] Celoxica, *Handel-C Language Reference Manual*, Version 3.1, 2002, http://www.celoxica.com

[4] C. T. H. Davies and S. M. Playfer, "Heavy Flavour Physics: Theory and Experimental Results in Heavy Quark Physics", *Institute of Physics*, June 2002.

[5] Xilinx, *Xilinx Virtex-II Architecture Manual*, Sep. 2002.

[6] Alpha Data Parallel System Ltd., ADM-*XRC-II PCI Mezzanine Card User Guide Version 1.5*, 2002.

[7] Zoltán Fodor, Sándor D Katz, Gábor Papp; “Better than $1/Mflops sustained: a scalable parallel computer for lattice QCD”; Comput.Phys.Commun. 152 (2003) 121-134, hep-lat/0202030.

[8] Zoltán Fodor, Sándor D Katz, Gábor Papp; “A Scalable PC-based parallel computer for lattice QCD”; Nucl.Phys.Proc.Suppl. 106 (2002) 177-183, hep-lat/0209115.

[9] R Alfieri et al.; “The apeNEXT project”; Talk from Computing in High Energy Physics 2003, La Jolla, California, USA, March 2003.

[10] P A Boyle, D Chen et al., “Status and performance estimates for QCDOC”; Nucl.Phys.Proc.Suppl. 106 (2002) 177-183, hep-lat0210034

[11] Christine Davis, Sara Collins, “Theorists get to grips with the strong force”, http://www.physics.gla.ac.uk/lattice_EU_network/physics_world.pdf

[12] W Ligon, S McMillan, G Monn, F Stivers, K Underwood, “A Re-evaluation of the Practicality of Floating-Point Operations on FPGAs”; In Proc *The 6th Annual IEEE Symposium on Field-Programmable Custom Computing Machines FCCM '98* Napa, California, USA April 1998.

[13] G Govindu, L Zhuo, S Choi, V Prasanna, “Analysis of High-Performance Floating-point Arithmetic on FPGAs”; In *Proc of 11th Reconfigurable Architectures Workshop,* Santa Fe, New Mexico, USA, April 2004