An FPGA-based Parallel Processor for Black-Scholes Option Pricing Using Finite Differences Schemes

Georgios Chatziparaskevas, Andreas Brokalakis, Ioannis Papaefstathiou
Department of Electronic and Computer Engineering
Technical University of Crete, Greece
{gehatzip, abrokalakis, ygp}@mhl.tuc.gr

Abstract— **Financial engineering is a very active research field as a result of the growth of the derivative markets and the complexity of the mathematical models utilized in pricing the numerous financial products. In this paper, we present an FPGA-based parallel processor optimized for solving the Black-Scholes partial derivative equation utilized in option pricing which employs the two most widely used finite difference schemes: Crank-Nicholson and explicit differences. As our measurements demonstrate, the presented architecture is expandable and the speedup triggered is increased almost linearly with the available silicon resources. Although the processor is optimized for this specific application, it is highly programmable and thus it can significantly accelerate all applications that use finite differences computations. Performance measurements show that our FPGA prototype triggers a 5x speedup when compared with a 2GHz dual-core Intel CPU (Core2Duo). Moreover, for the explicit scheme, our FPGA processor provides an 8x speedup over the same Intel processor.**

Keywords- Option pricing, finite differences, FPGA

I. Introduction

Financial (and in particular derivative) markets have seen an outstanding growth during recent years. The need for higher computational capacity has accordingly escalated as financial models become more complicated and client portfolio sizes increase, in an operating field, where real time market calculations are required. Moreover, financial problems often don't have a closed form solution, thus requiring computationally intensive approximate techniques to solve them. In general, increased calculations' speed is becoming a strategic advantage for all financial institutions.

As it has been demonstrated in numerous applications, FPGAs represent one of the most convenient solutions for high performance computing as they balance the performance of hardware with the flexibility of software [1] - [4]. The clock frequencies of these reconfigurable devices have been considerably increased in the last couple of years, as well as their capacity, effectively widening the application areas where they can be successfully deployed. Finally they provide better performance per watt ratios when compared to other types of accelerators such as GPUs.

In this paper, an FPGA-based parallel processor optimized for the Black-Scholes equation, utilized in option pricing, which employs two distinct finite differences schemes, is presented. In particular both the explicit and the Crank-Nicholson finite difference methods have been implemented on the processor; however since it is programmable more schemes can be easily adopted. The system architecture is flexible; it incorporates a number of cores connected on a ring-type bus and it can support different configurations that make it suitable for implementation in virtually any FPGA device. Our real-world results demonstrate that performance scales almost linearly with the number of cores. For a maximum of 64 cores that could be fitted on a high-end Virtex-5 FPGA, the speedup achieved over optimized software implementations on a 2GHz Core 2 Duo dual-core CPU is 8 times for the explicit method and almost 5 times for the Crank-Nicholson one.

II. Related Work

The increased capacity and speed of recent FPGAs has allowed the implementation of a series of algorithms belonging to diverse scientific fields ranging from bioinformatics to image processing and cryptography. Financial Engineering has also become a prospective field of application for these accelerators [5], [6] and option pricing in particular has been considered by researchers. For instance, a reconfigurable architecture that computes a Monte-Carlo technique has been implemented for real-time evaluation of derivatives and risk management [7]. Binomial pricing [8] has also been implemented on an FPGA as a pipelined architecture allowing the computation of multiple binomial trees in parallel [9]. Furthermore, in [10] an explicit finite-difference scheme implementation as a parallel reconfigurable co-processor exploiting the partitioning of the solution domain has been presented, while there has been a similar approach focused on option pricing models in [11]. However, this is the first, to the best of our knowledge, reconfigurable system for option pricing based on both explicit and semi-implicit finite differences applied to the Black-Scholes model with which certain financial products can be priced relatively accurately. Finally, there are also available GPU implementations of option pricing methods including Binomial [12], Monte-Carlo pricing [13] and explicit finite difference [11], along with an analytical solution of the Black-Scholes formula [14].

III. Motivation

Monte-Carlo methods for option pricing can be efficiently implemented on reconfigurable hardware since they can be easily parallelized. Although they are suitable for financial models with complex parameters, they cannot calculate all kinds of options [17]. Finite-difference (FD) methods on the other hand, are iterative techniques that approximate over a grid of discrete points the differential equations which model the prices of options. Their advantage is that they calculate many values of the unknown function at a single step and they

978-3-9810801-8-6/DATE12/©2012 EDAA

can be applied to other fields apart from financial engineering (e.g. Laplace heat [15] and Maxwell wave equations [10]).

The FD methods can be divided in two main categories: implicit and explicit. The explicit scheme is more widely adopted as it is the most computationally efficient. The implicit one, however, is typically more stable and converges faster, at the cost of increased computational complexity and much more difficult parallelization. A third alternative is the Crank-Nicholson scheme which lies in-between the implicit and explicit ones outperforming both in stability and convergence [16]. The presented system implements both the popular explicit scheme as well as the Crank-Nicholson one.

IV. The Black-Scholes Option Pricing Model

An option is a contract that gives its owner the right to buy (call option) or sell (put option) an underlying asset (stock) at a specific price before (American options) or when (European options) a time interval elapses [17]. The question raised is how a fair market value for an option is determined. A number of factors influence the price of an option, some of which are the current price of the underlying asset (stock price), the time remaining until the expiration date, the volatility of the value of the asset and the strike price (price at which the asset is going to be sold or bought). The way these drivers affect the price of an option is simulated by various models; the most widely used is the Black-Scholes option pricing model [18]. If we consider that the option price and the stock price depend on the same underlying source of uncertainty we can form a portfolio consisting of the stock and the option which eliminates this source of uncertainty and is instantaneously riskless ("arbitrage free") [19]. This assumption of no arbitrage opportunities leads to the Black-Scholes formula:

$$rS = \frac{\partial S}{\partial t} + rx\frac{\partial S}{\partial x} + \frac{1}{2}\sigma^2 x^2 \frac{\partial^2 S}{\partial x^2}, \quad S = f(t,x) \qquad (1)$$

where S is the option price, r the risk-free rate of interest, σ the volatility of the stock, x the price of underlying stock and t the time to expiration date ($0 \le t \le T$, T: expiry date). At expiration date the price of the stock is known:

$$f(T,x) = \max\{s - K, 0\}, \quad K : strikeprice \qquad (2)$$

Equation 1 is a parabolic partial derivative equation (Black-Scholes PDE), which can be solved analytically only for some types of derivative securities such as European Call options. In general its solution can be approximated with numerical methods. The algorithms implemented in the presented system belong to the finite difference family of methods which approximate the value of the unknown function over discrete points. The partial derivatives in Eq. 1 are replaced by approximations based on Taylor series expansion over the points of interest (Eq. 3). The exact FD methods implemented are the explicit and the Crank-Nicholson schemes. The explicit scheme is first order accurate in the time dimension and second order accurate in the asset price dimension. It is not however always stable and the necessary condition for stability declares that the time steps used should be equal to the square of the asset price points. On the other hand, the Crank-Nicholson scheme is unconditionally stable, second order accurate and converges quadratically with respect to time ($O(\delta t^2)$) while fully-implicit and explicit schemes converge linearly. The combination of accuracy and unconditional stability allows the use of larger time steps in the Crank-Nicholson method.

$$\frac{\partial S}{\partial t} \approx \frac{S(x, t+\delta t) - S(x,t)}{\delta t} \qquad (3)$$

V. The Explicit Scheme

The explicit scheme uses a forward difference quotient to approximate the time derivative and a central difference for both the first and second order derivatives in the asset price direction. The scheme calculates the option values $S_{i,j+1}$ of the $j+1$ time step over all the internal (non boundary) values of the asset price ($0 < i < N_S$), using the option values of the current time step $S_{i,j}$. The option values for the first time step (j = 0) are known from the initial condition $S_{i,0} = \max\{s_i - K, 0\}$.

Discretizing the first and second order derivatives and applying them to Eq. 1, Equation 4 is reached.

$$rS_{i,j} = -\frac{S_{i,j+1} - S_{i,j}}{dt} + rx_i \frac{S_{i+1,j} - S_{i-1,j}}{2dx} + \frac{1}{2}\sigma^2 x_i^2 \frac{S_{i+1,j} + S_{i-1,j} - 2S_{i,j}}{sx^2} \qquad (4)$$

After executing the computations on Eq. 4, the formula that calculates $V_{i,j+1}$ using the known values $V_{i-1,j}$, $V_{i,j}$, and $V_{i+1,j}$ is the following:

$$S_{i,j+1} = a_i S_{i-1,j} + b_i S_{i,j} + c_i S_{i+1,j} \qquad (5)$$

The parameters a_i, b_i, c_i for $0 < i < N_S$ remain constant in the course of time and their values are:

$$\begin{aligned} a_i &= -\frac{1}{2}ri + \frac{1}{2}\sigma^2 i^2 \\ b_i &= 1 - ri - \sigma^2 i^2 \\ c_i &= \frac{1}{2}ri + \frac{1}{2}\sigma^2 i^2 \end{aligned} \qquad (6)$$

Boundary conditions on three edges of the grid ($t= T$, $x= 0$, $x = X$) are necessary; in the current implementation the following conditions are used (Von Neumann conditions):

$$f(T,x) = \max\{S - K, 0\}, f(T,0) = 0 \text{ and } \left.\frac{\partial^2 S}{\partial x^2}\right|_{x=X} = 0 \qquad (7)$$

The first boundary condition is the value of the option at expiration date, the second one denotes that the value of an option over a zero amount of stock is also zero, and the third is based on the assumption that the value of the option grows linearly for large stock amounts (x).

VI. The Crank-Nicholson Scheme

The Crank-Nicholson scheme is a one step iterative semi-implicit finite difference method [20]. It utilizes a central difference in time and a second order central difference for the stock price derivative. The domain over which the values are calculated is a grid of points (t_i, x_i) with $t_i = T - i\,\delta t$, $0 \le t_i \le T$ and $x_i = j\,\delta x$, $0 \le x_j \le X$.

The Crank-Nicholson scheme utilizes also a backward difference to approximate the time derivative and an average central difference over $t_{i+1/2}$ to approximate the first and second order derivatives in the direction of the stock price. After the discretization of the first and second order derivatives, the initial PDE is transformed to a linear equation with unknowns the values of *S* at *i+1* time step ($S_{i+1,j-1}$, $S_{i+1,j}$, $S_{i+1,j+1}$). The equation produced by this process has the following form:

$$a_j S_{i+1,j-1} + b_j S_{i+1,j} + c_j S_{i+1,j+1} = e_{i,j} \qquad (8)$$

with:

$$\begin{aligned} e_{i,j} &= d_{0,j} S_{i,j-1} + d_{1,j} S_{i,j} + d_{2,j} S_{i,j+1} \\ a_j &= \frac{1}{4} rj - \frac{1}{4}\sigma^2 j^2 \\ b_j &= \frac{1}{\delta t} + \frac{1}{2}\sigma^2 j^2 + \frac{1}{2} r \\ c_j &= -\frac{1}{4} rj - \frac{1}{4}\sigma^2 j^2 \\ d_{0,j} &= -a_j,\ \ d_{1,j} = -b_j,\ \ d_{2,j} = -c_j \end{aligned} \qquad (9)$$

A system of linear equations has to be solved in each time step to find the value of S over all the internal points of the grid. The main diagonals of the system remain the same whilst the right hand side is updated with the values of the stock price calculated in the previous time step (Eq. 9).

The system to be solved in each time step is tridiagonal (Eq. 10). It can be solved with LU-decomposition in linear time with respect to its size. LU decomposition, though simple and fast, cannot be parallelized by nature, thus it cannot take advantage of a state-of-the-art FPGA device. A parallelizable algorithm for solving tridiagonal systems is the cyclic odd-even reduction one. A variant of this method is adopted in the proposed system and it is described in detail in the next section.

$$\begin{bmatrix} b_0 & 0 & \cdots & & 0 \\ a_1 & b_1 & c_1 & \ddots & \vdots \\ 0 & \ddots & \ddots & \ddots & 0 \\ \vdots & \cdots & a_{N-2} & b_{N-2} & c_{N-2} \\ 0 & \cdots & 0 & a_{N-1} & b_{N-1} \end{bmatrix} \begin{bmatrix} S_0 \\ S_1 \\ \vdots \\ S_{N-2} \\ S_{N-1} \end{bmatrix} = \begin{bmatrix} e_0 \\ e_1 \\ \vdots \\ e_{N-2} \\ e_{N-1} \end{bmatrix} \qquad (10)$$

VII. Cyclic Odd-Even Reduction

Cyclic Odd-even reduction is a recursive technique for solving tridiagonal systems [21]. Its principle is the recursive elimination of the odd-indexed unknowns till a unit-size system is produced. The remaining unknowns are calculated by moving backwards after the unit-size system is solved.

A. Forward Phase (Reduction)

The odd-indexed unknowns are eliminated and a new system with half the size of the previous one is produced in each recursion step (Fig. 1). The process (Alg. 1) includes successive multiplications that produce rapidly increasing products resulting to overflow errors after a certain number of steps. Our proposed scheme overcomes this weakness by normalizing the main diagonal of the initial system and keeping its elements equal to one throughout the algorithm. This is done by dividing each equation of the system with the corresponding element of the main diagonal during the initialization phase. Considering $a_{-1} = c_{-1} = e_{-1} = 0.0$ and $a_N = c_N = e_N = 0.0$, the process is summarized in Alg. 3.

The operations inside the loop (Alg. 1, 3) are independent of previous loop executions and they can all be executed in parallel. Operations belonging to the same recursion step are independent of one another but they depend on data produced on the previous recursion step. Thus only a single step can be broken into parallel calculations.

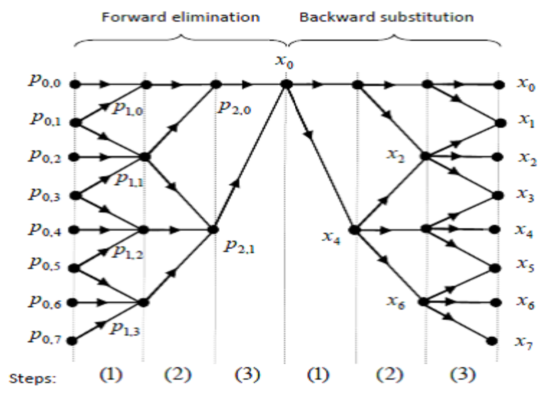

Figure 1. Cyclic Odd-Even reduction forward and backward steps.

Algorithm 1 Forward Phase step of traditional Cyclic Odd-Even Reduction

```
for $i = 1$ to $N_k - 1$ do
    $a_{k+1,i/2} = -a_{k,i-1} a_{k,i} b_{k,i+1}$
    $b_{k+1,i/2} = b_{k,i-1} b_{k,i} b_{k,i+1} - c_{k,i-1} a_{k,i} b_{k,i+1} - b_{k,i-1} c_{k,i} a_{k,i+1}$
    $c_{k+1,i/2} = -b_{k,i-1} c_{k,i} c_{k,i+1}$
    $e_{k+1,i/2} = b_{k,i-1} e_{k,i} b_{k,i+1} - e_{k,i-1} a_{k,i} b_{k,i+1} - b_{k,i-1} c_{k,i} e_{k,i+1}$
    $i \leftarrow i + 2$
end for
```

Algorithm 2 Backward Phase step of traditional Cyclic Odd-Even Reduction

```
for $i = 1$ to $N_k - 1$ do
    $x_{i2^k} = (e_{k,i} - a_{k,i} x_{(i-1)2^k} - c_{k,i} x_{(i+1)2^k}) / b_{k,i}$
    $i \leftarrow i + 2$
end for
```

Algorithm 3 Forward Phase step of the implemented Cyclic Odd-Even Reduction Variant

```
for $i = 1$ to $N_k - 1$ do
    $temp = 1.0/(1.0 - c_{k,i-1} a_{k,i} - c_{k,i} a_{k,i+1})$
    $a_{k+1,i/2} = -temp \cdot (a_{k,i-1} a_{k,i})$
    $c_{k+1,i/2} = -temp \cdot (c_{k,i} c_{k,i+1})$
    $e_{k+1,i/2} = temp \cdot (e_{k,i} - e_{k,i-1} a_{k,i} - c_{k,i} e_{k,i+1})$
    $i \leftarrow i + 2$
end for
```

Algorithm 4 Backward Phase of the implemented Cyclic Odd-Even Reduction variant

```
for $i = 1$ to $N_k - 1$ do
    $x_{i2^k} = e_{k,i} - a_{k,i} x_{(i-1)2^k} - c_{k,i} x_{(i+1)2^k}$
    $i \leftarrow i + 2$
end for
```

B. Backward Phase (Back Substitution)

When the size of the tridiagonal system becomes one, the value of x_0 can be trivially computed. Backtracking from x_0 and substituting the calculated $x_{i2}{}^{k}$ (*i = 0, ... ,N_k*), $x_{i2}{}^{k-1}$ is computed for all *i = 0, ... , N_{k-1}* (k is the current algorithm step and $N_k = N / 2^k$ is the current system size). Only calculations belonging to the same backward step are parallelizable similarly to the forward phase.

C. *Performance Evaluation*

The number of elementary operations of both the traditional and the implemented Odd-even Reduction schemes depends on the number of forward and backward loop executions, which are the same and equal to N-1. Concerning LU decomposition, the equivalents of the forward and backward phase loops are executed N-1 times as well (see Alg. 5). LU-decomposition presents the lowest overall number of operations (Table 2), followed by the implemented variation of the odd-even scheme. When p processors are deployed, the time complexity of Odd-Even Reduction becomes O(N/p+p) while the order of LU decomposition remains O(N) as it cannot be efficiently parallelized due to its serial nature. Consequently Odd-Even reduction becomes faster for sufficiently large systems.

Algorithm 5 LU-Decomposition
```
{Forward phase}
$b'_0 = b_0$
$e'_0 = e_0$
for $i = 1$ to $N-1$ do
    $a'_i = a_i / b'_{i-1}$
    $b'_i = b_i - a'_i c_{i-1}$
    $e'_i = e_i - a'_i / e'_{i-1}$
end for
{Backward phase}
$x_{N-1} = e'_{N-1} / b'_{N-1}$
for $i = N-2$ to 0 do
    $x_i = (e'_i - c_i x_{i+1}) / b'i$
end for
```

Space complexity of all three methods is linear with respect to the system size: LU decomposition requires 5N memory words for the three diagonals (a, b, c), the right hand side (e) and the solution of the system (x). Traditional Odd-even Reduction requires a surplus of 4(N – 1) memory words while the implemented variant requires 4N memory words initially for a, c, e, x and 3(N – 1) words for the produced a, c, e.

TABLE I. NUMBER OF OPERATIONS OF A SINGLE FORWARD/BACKWARD LOOP PASS FOR THE INSPECTED ALGORITHMS

	Forward Loop Pass			**Backward Loop Pass**		
Operation Type	***#mul***	***#add***	***#div***	***#mul***	***#add***	***#div***
Traditional Odd-Even Reduction	16	4	0	2	2	1
Implemented Odd-Even Reduction	9	4	1	2	2	0
LU Decomposition	1	2	2	1	1	0

TABLE II. TOTAL NUMBER OF OPERATIONS

Operation Type	***#mul***	***#add***	***#div***
Traditional Odd-Even Reduction	18(N-1)	6(N-1)	N-1
Implemented Odd-Even Reduction	11(N-1)	6(N-1)	N-1
LU Decomposition	2(N-1)	3(N-1)	2N-1

Considering the accuracy of Cyclic Odd-even Reduction, it is equivalent to Gaussian elimination without pivoting on a system with permutated rows and columns [22]. As LU decomposition is a simplified Gaussian elimination it can be inferred that both methods are equally accurate.

VIII. ODD-EVEN REDUCTION PARALLELIZATION

The odd-even reduction algorithm can be parallelized at both instruction and task level. In the computations inside the main loop of the forward phase (Alg. 3), the individual products of the diagonal elements belonging to the same formula are independent. The same applies to operations with two terms belonging to separate formulas; they can be executed simultaneously allowing for instruction level parallelism.

When considering different loop executions of the same forward step, their independence can be translated to more coarse grain parallelism (the same applies to the backward phase). Finer-grain parallelism can be achieved by using multiple adders and multipliers. An alternative approach includes a pipelined FPU consisting of a single adder, divider and multiplier that executes the independent operations in a row in order to achieve maximum overlap. This FPU can then be replicated to take advantage of the available coarser-grain parallelism based on the different loop executions. This procedure infers serial execution of a number of loops of the forward and backward phase inside each FPU and parallel execution of these batches of loops in separate FPUs (Fig. 2).

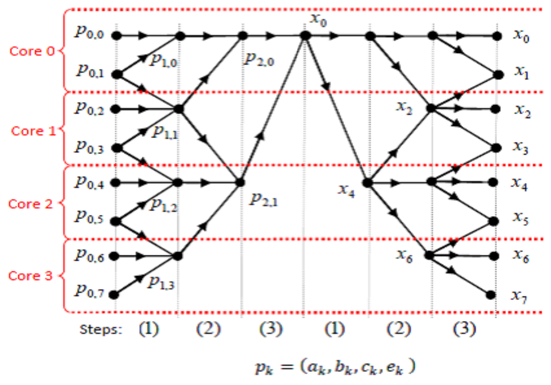

Figure 2. Odd-Even Reduction steps for a system of size 8, allocated to 4 processors.

IX. ARCHITECTURE OF THE PROPOSED SYSTEM

In the proposed system the FPGA receives data from a host PC and performs all the computationally intensive processing. Internally, it consists of a number of programmable cores and peripherals for the required I/O activities. Each core has a limited RISC-like custom ISA. It includes a local memory subsystem, decoding and control logic and a floating point unit.

Inter-core communication is message driven. There is a point-to-point connection between neighboring cores and the whole system forms a ring topology. Examining thedata exchanges between the cores, we discovered that the vast majority of the data transfers on both the explicit and the odd-even reduction schemes, is only between neighboring cores. Therefore, this topology offers great scaling, resource savings and allows for high clock frequencies.

Each core consists of three main parts (Fig. 3); a local memory, a memory controller and an FPU. The local memory stores all the required control and information data, while the memory controller (Fig. 4) resolves the different addresses for reading and writing data, synchronizes FPU operations and handles the communication with the other cores.

The FPU has four single-precision floating-point arithmetic units: an adder, a multiplier and two dividers (Fig. 5). It should be noted that the explicit scheme does not include any divisions and therefore no FP dividers were instantiated for this design. Since loop iterations are data independent, and in order to increase parallelism, each FP unit is fully pipelined – except for the dividers. During design exploration it was found that, in the targeted FPGA device, implementing two dividers with a throughput equal to half of their latency yielded a smaller circuit than having a single fully pipelined divider while the performance of the two approaches was identical.

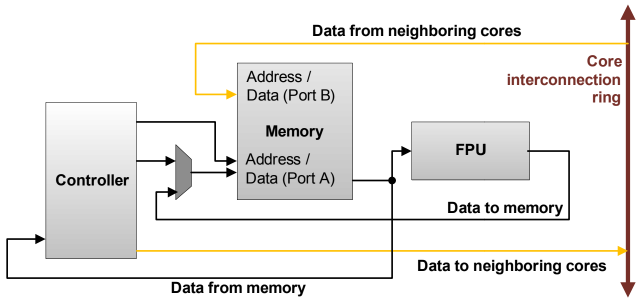

Figure 3. Overall core organisation.

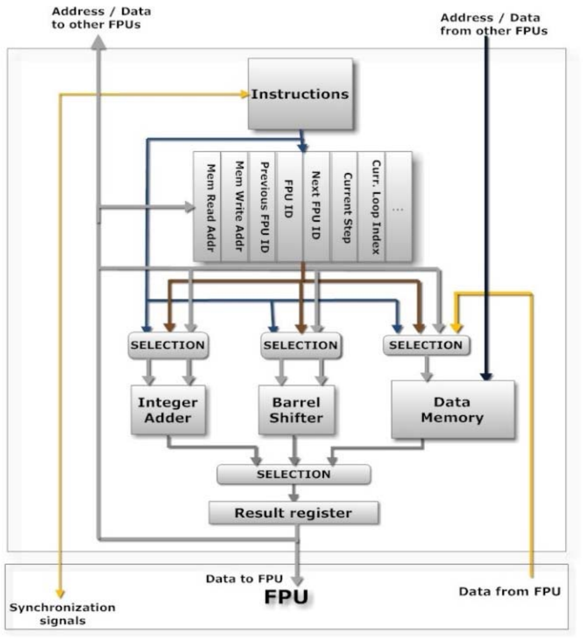

Figure 4. Internal architecture of memory controller.

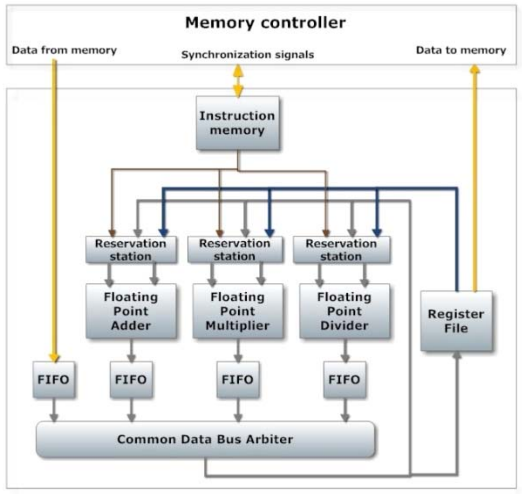

Figure 5. Internal architecture of Floating Point Unit.

The FP operations are stored as entries in reservation station structures attached to each FPU. The operations stored in these structures are triggered with a predefined priority just when their operands arrive from memory.

X. IMPLEMENTATION

The proposed architecture has been implemented on a Xilinx Virtex5 FPGA device. FP multipliers, adders and dividers have been generated by Xilinx's COREGen tool and the selected latencies are 4, 8 and 14 clock cycles respectively. Table 3 provides the resource usage information.

TABLE III. RESOURCE UTILIZATION ON A VIRTEX5 XC5VSX240T

Explicit Scheme Implementation			
Resource	***1 Core***	***32 Cores***	***64 Cores***
Slice Registers	1,051 (0.7%)	39,092 (26.1%)	78637 (52.5%)
Slice LUTs	1,268 (0.8%)	22,584 (15.1%)	36222 (24.2%)
BRAMs	5 (1%)	176 (34.1%)	368 (71.3%)
DSPs	3 (0.3%)	96 (9.1%)	192 (18.1%)
Crank-Nicholson Scheme Implementation			
Resource	***1 Core***	***32 Cores***	***64 Cores***
Slice Registers	1,564 (1%)	55,444 (37%)	112,469 (75.1%)
Slice LUTs	2,239 (1.5%)	76,274 (50,9%)	146,532 (97.8%)
BRAMs	9 (1.7%)	192 (37.2%)	380 (73.6%)
DSPs	3 (0.3%)	96 (9.1%)	192 (18.1%)

The clock frequencies (for 64 instantiated cores) are 160 and 145 MHz for the explicit and Crank-Nicholson schemes implementations respectively. The difference in frequency is attributed to the higher resource utilization of the latter design.

XI. EVALUATION

To evaluate the efficiency of the proposed processor, its performance was compared with that of an Intel Core 2 Duo processor clocked at 2.0GHz. For the Crank-Nicholson case, the acceleration triggered by the proposed architecture has been measured against both Odd-Even Reduction and LU-Decomposition implementations, as for the single-thread case LU is faster by nature. The software implementations were based on widely used C++ reference implementations provided by [20]. These implementations were further modified to match the algorithmic variations and specific system specs. The software was threaded to use both cores on the processor and has been hand-optimized for this Intel CPU. G++ has been used with settings for maximum performance (-O3). The software performance was measured using Intel VTune performance analyzer. The benchmark grid utilized for the comparison has 8K points in the direction of Stock price (space dimension) and 8K points in the time direction. The exact same data have been used on both cases.

The speedup achieved for the explicit scheme by the proposed parallel processor (with 64 cores) is 8x over its software counterpart. Fig. 6 demonstrates how performance scales with regard to the number of cores instantiated in the parallel processor. There is an almost linear increase in performance as more cores are added. The less-than-ideal scaling is attributed to two primary reasons. First, when more cores are used, the efficiency of the ring topology decreases and secondly, and most important, there is a small clock frequency drop as the device resource utilization increases.

Fig. 7 compares the performance comparison for the Crank-Nicholson scheme. The proposed parallel processor is almost 5 times faster when Odd-Even Reduction is considered and ~3,7 times faster when compared with the software LU decomposition executed on the Intel CPU. It should be stressed that our FPGA-based prototype is clocked 14 times slower than the Intel processor whereas the complexity of our 64-core system is certainly smaller than that of the dual-core Intel CPU.

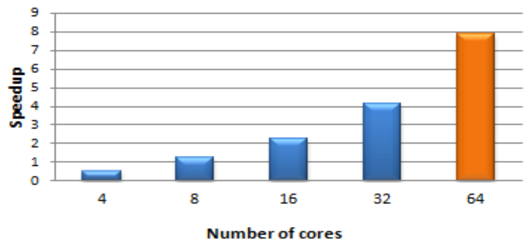

Figure 6. Speedup of proposed parallel processor over Core2Duo (2GHz) for the explicit scheme.

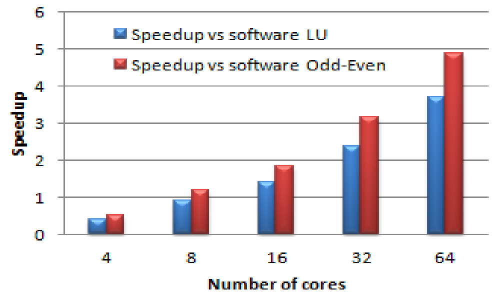

Figure 7. Speedup of proposed parallel processor over software running on a Core2Duo CPU for the Crank-Nicholson scheme Software implementations of LU decomposition and Odd-Even Reduction are considered.

XII. Conclusions and Future Prospects

This paper presents an FPGA-based parallel processor implementing both the explicit and the Crank-Nicholson finite difference schemes in order to calculate the prices of financial options, based on the Black-Scholes model. Great consideration has been given to the solution of the tridiagonal system produced in each time step of the Crank-Nicholson method, as this is the most computationally intensive part of the algorithm. We are solving this system by utilizing our novel variant of the Odd-Even Reduction scheme.

The architecture of the proposed system is highly flexible; functional units and cores can be easily added to adapt to the application requirements and available hardware resources. Evaluation results demonstrated that the performance of our system scales linearly with the number of instantiated cores. This is important as devices with logic resources almost five times as many as the one that has been used in this paper are already available from FPGA vendors and even larger ones are soon to be introduced (e.g. Virtex 7). As a result, the presented speedups are expected to be valid even when compared with a much more powerful processor.

Last but not least, since the architecture is programmable and not a fixed-function one, it is suitable to execute numerous applications that are based on finite differences schemes and are not restricted to the financial domain only. Although not presented in this paper, we have already used the same system for solving the heat diffusion equations described in [15], while our overall approach is suitable for other similar applications for which custom hardware is utilized, such as the calculation of flying object's optimal noise reduction paths [23].

References

[1] P. S. Graham and M. Gokhale, "Reconfigurable Computing Accelerating Computation with Field-Programmable Gate Arrays", 1st ed. Dordrecht, The Netherlands: Springer, 2005.

[2] R. Tessier and W. Burleson, "Reconfigurable Computing for Digital Signal Processing: A Survey", Journal of VLSI Signal Processing 28, 7-27, 2001.

[3] T. Ramdas, Gr. Egan,"A Survey of FPGAs for Acceleration of High Performance Computing and their Application to Computational Molecular Biology", TENCON 2005 2005 IEEE Region 10.

[4] S. Z. Ahmed, G. Sassatelli, L. Torres, L. Rougé, "Survey of new trends in Industry for Programmable hardware: FPGAs, MPPAs, MPSoCs, Structured ASICs, eFPGAs and new wave of innovation in FPGAs", FPL 2010.

[5] R. Baxter, R. Booth et al., "Maxwell a 64 fpga supercomputer." in Proc. 2nd NASA/ESA Conference on Adaptive Hardware and Systems (AHS 2007), UK, Aug. 2007, pp. 287–294.

[6] XtremeData, "Fpga acceleration in hpc: A case study in financial analytics version 1.0. whitepaper," Nov. 2006.

[7] M. Elm and J. K. Anlauf, "Pricing of derivatives by fast, hardware based monte carlo simulation." in Proc. SIAM Conference on Financial Mathematics and Engineering (FM 2006), USA, Jul. 2006.

[8] J. C. Cox, S. A. Ross, and M. Rubinstein, "Option pricing: A simplified approach."

[9] Q. Jin, D. B. Thomas, W. Luk, and B. Cope, "Exploring reconfigurable architectures for binomial-tree pricing models," in Proc. 4th international workshop on Reconfigurable Computing: Architectures, Tools and Applications (ARC 2008), UK, Mar. 2008, pp. 245–255.

[10] E. Motuk, S. Bilbao, and R. Woods, "Implementation of finite difference schemes for the wave equation on fpga," in Proc. IEEE ICASP 2005, USA, May 2005, pp. 237–240.

[11] Q. Jin, D. B. Thomas, and W. Luk, "Exploring reconfigurable architectures for explicit finite difference option pricing models," in *Proc. Int. Conf. on Field-Programmable Technology*. IEEE, 2009.

[12] NVIDIA, "Binomial option pricing model whitepaper," Apr. 2008.

[13] "Monte carlo option pricing whitepaper," Jun. 2008.

[14] "Black-scholes option pricing whitepaper," Jun. 2007.

[15] L. Elden,"Numerical solution of the sideways heat equation by difference approximation in time",*Inverse Problems*,11: 4, pp. 913-923, 1995.

[16] P. Wilmott, S. Howison, and J. Dewynne, "The Mathematics of Financial Derivatives: A Student Introduction." New York, USA: Cambridge University Press, 1995.

[17] J. C. Hull, "Options, Futures, and Other Derivatives", 6th ed. Prentice Hall, 2005.

[18] F. Black and M. Scholes, "The pricing of options and corporate liabilities," in The Journal of Political Economy, 1973.

[19] S. N. Neftci, "An Introduction to the Mathematics of Financial Derivatives", 2nd ed. Academic Press, 2000.

[20] J. Kerman, "Numerical methods for option pricing: Binomial and finite difference approximations," Courant Institute of Mathematical Sciences, New York University, Tech. Rep., Jan. 2002.

[21] H. S. Stone, "Parallel Tridiagonal Equation Solvers", ser. ACM Transactions on Mathematical Software (TOMS). New York, USA: ACM, 1975, vol. 1.

[22] Gander and G. H. Golub, "Cyclic reduction - history and applications," in Proc. of the Workshop on Scientific Computing, Hong Kong, 1997.

[23] Kontos, I. Papaefstathiou, D. Pnevmatikatos, "Design Space Exploration of Reconfigurable Systems for Calculating Flying Object's Optimal Noise Reduction Paths", in FPL 2009.