UNIFYING FINITE DIFFERENCE OPTION-PRICING FOR HARDWARE ACCELERATION

Qiwei Jin, Wayne Luk *

Department of Computing
Imperial College London
qj04,wl@doc.ic.ac.uk

David B. Thomas

Department of Electrical and Electronic Engineering
Imperial College London
d.thomas1@imperial.ac.uk

ABSTRACT

Explicit finite difference method is widely used in finance for pricing many kinds of options. Its regular computational pattern makes it an ideal candidate for acceleration using reconfigurable hardware. However, because the corresponding hardware designs must be optimised both for the specific option and the target platform, it is challenging and time consuming to develop all the designs efficiently. This paper presents a framework for describing and automatically implementing financial explicit finite difference procedures in reconfigurable hardware, allowing parallelised fully pipelined implementations to be created from high-level mathematical expressions. The proposed methodology is demonstrated using three option pricing problems. Our results show that the implementation generated by our framework on a Virtex-6 device at 310MHz is more than 24 times faster than a software implementation fully optimised by the Intel compiler on a four-core Xeron CPU at 2.66GHz. In addition, the latency of the FPGA solvers is up to 90 times lower than the corresponding software solvers.

1 Introduction

Finite difference method [1] is a basic numerical procedure widely adopted in the finance industry. It is easy to specify mathematically and it is relatively easy to translate from a mathematical expression into sequential software code. Reconfigurable hardware can provide good performance and energy efficiency which are desirable in modern finance industry, but it is time consuming to manually exploit parallelism in explicit finite difference procedures and translate them into pipelined hardware.

This paper describes a unifying framework for describing and implementing financial explicit finite difference procedures in reconfigurable hardware, allowing parallelised fully pipelined implementations to be automatically created from simple mathematical expressions. The main contributions are:

- a scheme for classifying option pricing problems solvable by explicit finite difference method in terms of the option specification's dependence on space and time;
- a unifying framework that allows the automatic translation of any high-level one-dimensional option specification into pipelined hardware;
- an evaluation of our approach based on the Virtex-6 FPGA technology, showing a speed up of 24 times against a highly optimised software implementation.

*The research leading to these results has received funding from J.P. Morgan, EPSRC, Alpha Data, Xilinx and the European Union Seventh Framework Programme under grant agreement number 248976 and 257906. The author would also like to thank the suggestions and comments provided by Gary C.T. Chow.

2 Background

Existing work addresses physical problems [2] which do not require high arithmetical precision, or focus on accelerating individual options rather than the entire set of possible options [3].

In this paper we consider the explicit method, which has a regular computational stencil which fits well in hardware. Explicit finite difference method approximates the solution of the partial differential equation (PDE) of an option by discretising in both time and space to construct a grid as shown in Figure 1. The grid is then solved by updating each of its columns from right to left iteratively, with an update function update each element in the column.

A call option is a contract that gives party A the right to buy some asset S to party B at a fixed price K (called the strike price). The payoff of the Asian call option can be written as $(\bar{S}_T - K)^+$ where $\bar{S}_T$ is the arithmetic average of S between time 0 and T under continuous sampling. The continuously sampled Asian call option has the following PDE, which is derived from a none-Black-Scholes model from [4], in the assumption that price of the underlying asset follows a geometric Brownian motion:

$$\frac{\partial u}{\partial t} + \frac{1}{2}(z - e^{-\gamma t} q_t)^2 \sigma^2 \frac{\partial^2 u}{\partial z^2} = 0 \quad (1)$$

$$u(T, z) = u(t, z_N) = (z - K_1)^+ \quad (2)$$

We call Equation 2 the initialisation function, which is different for different option pricing problems. Applying explicit discretisation to Equation 1, we can get the update function, as are listed in the third row in Table 1. The details of the derivation will be included in a future journal paper. Figure 1 shows the grid used to discretise the PDE. Such a grid is typical for many option pricing problems, such as European options and American options. The rightmost column is initialised at the beginning; the stencil in the middle is comprised of three inputs to the update function; the grid is iteratively updated from right to left column by column.

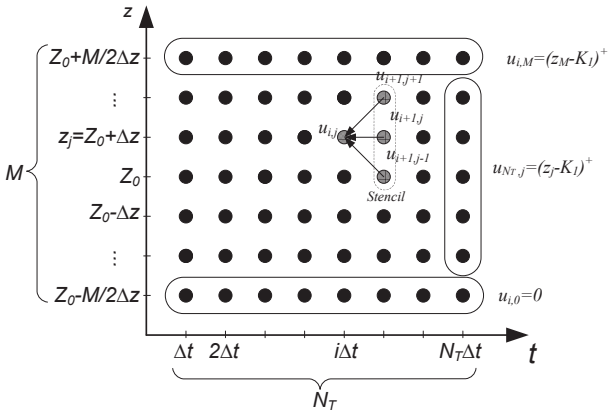

Fig. 1: Explicit Finite Difference Grid for Asian Option Pricing. Note that z is the random process under consideration, Z_0 is the value of z at the present time, u is the option value function, $u_{i,j}$ is the value of the option when $z = z_j$ and $t = i\Delta t$, t is the timeline and K_1 is the strike price of the option.

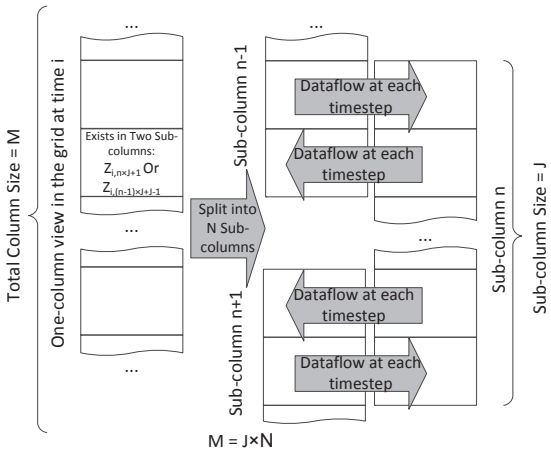

Fig. 2: One column view of the grid shown in Figure 1 at time step i, and the original column is split into N equally sized sub-columns each with size J. This single column is updated along the time line, syncronisation occurs between adjacent sub-columns just before stepping from time i to $i-1$.

3 Framework for Explicit Finite Difference Procedures

We propose that the explicit mechanism be classified into four categories in terms of change of coefficients in the stencil over space and time: **a) Constant** problems (e.g. European option pricing with change of variable) are the problems where the coefficients does not change with the position of the stencil in the grid. They are the most hardware resource efficient, as its coefficients can be pre-calculated by CPU and transferred to the FPGA at startup. **b) Time variant** problems (e.g. European option pricing) are those where the coefficients changes only when the stencil is in different time steps. They can be optimised to allow pre-calculation of the coefficients for the next time step which update the current time step, so are more logic resource intensive than constant problems. **c) Space variant** problems (e.g. American option pricing) are those where the coefficients changes only with value of the random process (e.g. z) but not with time. They can be optimised to construct a lookup table of coefficients beforehand, which can be reused at each time step, and so it is more memory intensive but requires less calculation than time variant problems. **d) Time-Space variant** problems (e.g. Asian option pricing) are the problems whose coefficients changes with the position of the stencil in the grid. They cannot be optimised like the other three, and so all coefficient calculations need to be done in hardware, so it is the most logic resource intensive yet most general amongst the four.

We now describe the parallelised general explicit finite difference procedure mathematically. Each option can be represented by a tuple $Q \equiv (K, S_0, T, r, \sigma)$, where K is the strike price, S_0 is the current underlying asset price, T is the time to maturity r is the interest rate and σ is the volatility of the underlying asset. We discretise in both the time dimension and price (space) dimension to get a grid Z,as shown in Figure 1. At a particular time i each element in column i is updated based on the values from column $i+1$. Figure 2 shows how intra-column spatial parallelism is achieved. We split each column in the grid into N equally sized sub-columns on the price dimension, with each sub-column containing $J = \frac{M}{N}$ elements, where M is the number of rows in the original grid.

To allow inter-column parallelism, we define a new indexing scheme for the original grid as shown in Figure 1:

$$U_{n,i,j} \equiv u_{i,n\times J+j}\ ,\ Z_{n,i,j} \equiv z_{i,n\times J+j}.$$

The initialisation procedure is defined by $init(\aleph, \Re, Q) \mapsto \Re$, where $\aleph$ is the set of nature numbers and $\Re$ is the set of real numbers, and $q \in Q$:

$$U'_{n,0,j} \leftarrow init(n \times J + j, Z_{n,0,j}, q),\ where\ 0 \leq j \leq J-1.$$

The rightmost column will be initialised by initialisation procedure, and then the stepping procedure can begin. We now define the stepping procedure based on the update function $f(\aleph, \aleph, \Re, \Re, \Re, Q) \mapsto \Re$:

$$\begin{aligned} U'_{n,i-1,j} &\leftarrow f(i, n\times J+j, U_{n,i,j-1}, U_{n,i,j}, U_{n,i,j+1}, q) \\ &\quad where\ 0 < j < J-1 \quad (3) \\ U'_{n,i-1,0} &\leftarrow U'_{n-1,i,J-2} \quad (4) \\ U'_{n,i-1,J-1} &\leftarrow U'_{n+1,i,1}. \quad (5) \end{aligned}$$

For different types of option pricing problems, different initialisation functions and stepping functions will be defined by the user. Table 1 lists such functions for various option pricing problems.

There are two kinds of data dependencies exhibited in this procedure. The first one is the intrinsic inter-column temporal data dependency from the model, as in Equation 3. The second one is the intra-column spacial dependency which is introduced by allowing parallelism in the model, as in Equation 4 and Equation 5. The idea is illustrated in Figure 2. In the nth sub-column, the value of the first element depends on the second to last value of $n-1$th sub-column, which are not available locally.

Intra-column data dependencies greatly reduce the efficiency of deeply-pipelined reconfigurable architectures. Pipeline draining will occur at each time step since the second to last element of any sub-column is usually the last element to enter the pipeline. However, with careful arrangement of execution order, full pipelining is achievable (this is further explained in Section 4).

4 Mapping the Framework to Hardware

We identify four main components in explicit finite difference solvers: a) the **data module** to store the grid, which is updated every time step; b) the **initialiser** to initialise the data module prior to the commencement of the stepping procedure; c) the **data path** to realise the stencil function d) the **data control** module controls the stepping procedure, and the output function to extract the final result from data module. It controls the data module to produce one set of inputs for the data path, and write one result back to the data module each clock cycle.

Our architecture exploits both intra-option parallelism and inter-option parallelism. The former is used to process multiple sub-columns of the grid at the same time, each sub-column is processed by one processing module. To allow data exchange between adjacent sub-columns at the end of each time step, adjacent pairs of data modules have dedicated communication channels between them, to handle the intra-column data dependency described in Section 3.

For naive implementations where the elements in the sub-column are processed one by one in the order $1 \ldots (J-2)$, the intra-column data dependency greatly reduces the efficiency of deeply-pipelined reconfigurable architectures. Since the $(J-2)$th element of any sub-column, which is needed in the swap at the end of a time step, is the last element to enter the pipeline, pipeline draining will occur at each time step. We propose the inwards interleaving accessing technique to avoid all pipeline draining while iterating – instead of iterating in the normal order, the elements in the sub-column are updated in the following sequence:

$$1\ ,\ (J-2)\ ,\ 2\ ,\ (J-3)\ \ldots\ \frac{J}{2}-1\ ,\ \frac{J}{2}.$$

In this way, the $(J-2)$th element is always updated at the beginning of each time step, so with enough elements in each sub-column to fill the pipeline, full pipelining is achievable.

Inter-option parallelism allows more than one option to be priced at the same time. It is straight forward but leads to a less efficient design: if there are more options to be priced than the number of pipelines, the options in the queue need to wait until the pipeline is drained and the last element in the grid is calculated for its predecessor. Option level pipelining can be used to overcome this problem, by duplicating the data module within the processing module. While one data module is providing data to the datapath, the other module is being initialised, in this way the processing module can switch to process other option and avoid pipeline draining. Figure 3 shows a detailed architecture for the data module, which allows inwards interleaving memory access.

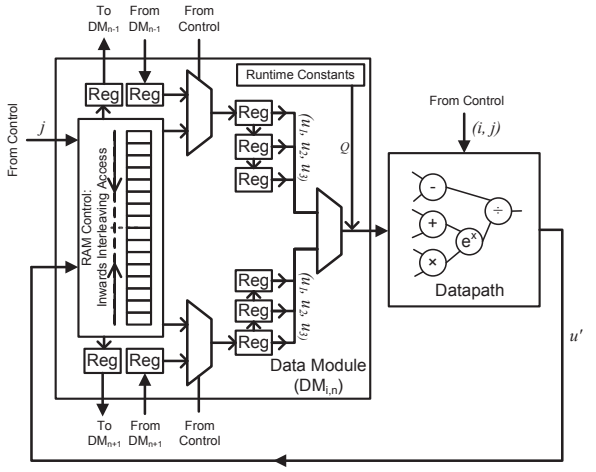

Fig. 3: Detailed architecture for the Data Module, which is connected to the Datapath logic automatically generated from a update function in Table 1.

5 Implementation and Evaluation

Our implementation of the framework is based on a Virtex-6 XC6VLX550T device. Flopoco 2.2 [5] floating point library is used to generate the floating point pipeline. The framework is mapped to hardware by manually following the automatic mapping procedure proposed in the previous section. A fully automated implementation will be developed in the future. The architecture is optimised to achieve full pipelining, results for implementations exploiting full FPGA computing capacity are reported. The reference C++ software implementation is compiled by both the Intel *icpc* compiler and the standard *gcc* compiler with maximum optimisation level turned on. We then exploit process level parallelism to extract the highest possible CPU performance by fully utilising the 4-core Intel Xeron E5640 platform at 2.66GHz. The FPGA (36nm) and the CPU (32nm) are based on similar technology, therefore comparable. Performance comparison with GPUs is outside the scope of this paper, and will be included in a future journal paper.

The hardware resource utilisation result of solvers with maximum number of duplicatable processing modules (PMs) is shown in Table 2. We find that single precision implementations are usually memory bounded, while double precision implementations are usually logic resource bounded. This is due to the fact that a double precision implementation is usually three times more Logic Element (LE) intensive than the corresponding single precision implementation; on the other hand, double precision implementations only use about twice as much Block RAMs (BRAMs) compared to single precision implementations.

The performance is evaluated in the number of nodes updated in the grid per second (MNodes/s). If a design executing at c MHz has x processing modules, then its performance is $x \times c$ MNodes/s. This is a standard peak sustained performance measure used in previous research [6], [7], [8]. In our case, the unaccounted overhead is the parameter setup and the retrieval of results, which will be of the order of 10K-50K cycles for every 10M-500M node update, and so is hidden when processing multiple options.

Name	$init: (\aleph, \Re, Q) \mapsto \Re$	$f: (\aleph, \aleph, \Re, \Re, \Re, Q) \mapsto \Re$
European	$(K - S_0 e^{(M/2-j)\Delta z})^+$	$u'_{EU} = \alpha u_1 + \beta u_2 + \gamma u_3$
American	$(K - S_0 e^{(M/2-j)\Delta z})^+$	$u'_{US} = \max(K - S_0 e^{(M/2-j)\Delta z}, u'_{EU})$
Asian	$(j\Delta z - K_1)^+$	$u'_{AS} = \frac{1}{2} \cdot A(i,j)(u_1 + u_3 - 2u_2) + u_2$

Table 1: Example Option Pricing problems. Note that for Time-Space variant Asian options the coefficient A is defined as $A(\aleph, \aleph) \mapsto \Re$ which is sensitive to time i and space j; for constant European option pricing problems the update function f is not sensitive to either i or j

.

	n	LE	DSP	BRAM
Asian32	50	602920 **(88%)**	700 (81%)	450 (71%)
Asian64	15	486158 (71%)	864 **(100%)**	270 (43%)
American32	70	477661 (69%)	770 (89%)	632 **(100%)**
American64	34	592962 (86%)	864 **(100%)**	612 **(97%)**
European32	70	268569 (39%)	420 (49%)	632 **(100%)**
European64	34	315573 (46%)	864 **(100%)**	612 **(97%)**

Table 2: Resource utilisation of the option solvers implemented on a Virtex-6 XC6VLX550T device.

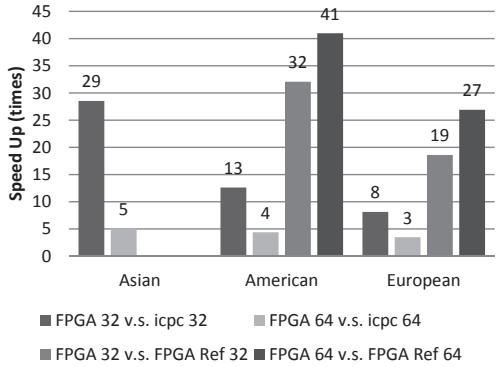

Fig. 4: Maximum speed up for FPGA implementation against fastest software implementation and reference FPGA implementation [3] .

We now discuss the performance of the FPGA implementations. If we replicate PMs on the FPGA, the single precision designs run at around 310 MHz and double precision designs run at around 240MHz. As shown in Figure 4 in single precision the maximum speed up against the fastest software implementation is 28 times. Solver latency is also an important factor which indicates how long it takes the solver to get the option price. The FPGA implementations have many PMs per solver executing in parallel, for complex Asian option pricing, the latency is more than 90 times lower compared to the fastest software implementation. Double precision implementations are typically 25% slower than their corresponding single precision implementations on FPGA. It can be seen that the maximum speed up achievable over the fastest software implementation is 6 times. On the other hand, the double precision FPGA implementations have upto 25 times lower latency than the software implementation.

The result also shows that our FPGA implementation is up to 41 times faster than the manually optimised reference FPGA implementation. This is because our implementation is based on a highly optimised framework with an efficient floating point library, while the reference is based on a high level synthesis language, Hyperstreams [3]. . The other reason is that our FPGA platform is based on newer technology and has more available resource than the reference platform.

Note that we do not show the comparison with the results reported in [9] as there is no well-defined benchmark to relate the performance and accuracy for Asian options, although other options have been compared in [10].

6 Conclusion and future work

This paper has presented a framework for describing and implementing financial explicit finite difference procedures in reconfigurable hardware. Our results show the parallelised fully pipelined implementation generated by our framework using simple mathematical expressions on a Virtex-6 device at 310MHz is more than 24 times faster than the software implementation fully optimised by the Intel compiler on a four-core Xeron CPU at 2.66GHz. In addition, the latency of the FPGA solvers is more than 90 times lower than the corresponding software solvers.

7 References

[1] J. Hull, *Options, Futures and Other Derivatives*, 6th ed. Prentice Hall, 2005.

[2] E. Motuk, R. Woods, and S. Bilbao, "Implementation of finite difference schemes for the wave equation on FPGA," in *Proc. IEEE Int. Conf. on ASSP*, 2005, pp. 237–240.

[3] Q. Jin, D. Thomas, and W. Luk, "Exploring reconfigurable architectures for explicit finite difference option pricing models," in *Int. Conf. on Field Programmable Logic and Applications*, 2009, pp. 73–78.

[4] J. Vecer, "Unified pricing of Asian options," *Risk*, vol. 15, no. 6, pp. 113–116, 2002.

[5] F. De Dinechin, J. Detrey, C. Octavian, and R. Tudoran, "When FPGAs are better at floating-point than microprocessors," Laboratoire de lInformatique du Paralllisme research report RR2007-40, 2007.

[6] A. H. Tse, D. B. Thomas, and W. Luk, "Accelerating quadrature methods for option valuation," in *Proc. IEEE Symp. on Field-Programmable Custom Computing Machines*, 2009.

[7] Q. Jin, D. B. Thomas, W. Luk, and B. Cope, "Exploring reconfigurable architectures for tree-based option pricing models," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 2, pp. 21:1–21:17, September 2009.

[8] D. Thomas, J. Bower, and W. Luk, "Automatic generation and optimisation of reconfigurable financial Monte-Carlo simulations," in *Proc. Int. Conf. on Application-Specific Systems, Architectures and Processors*, 2007, pp. 685–689.

[9] A. H. Tse, D. B. Thomas, K. Tsoi, and W. Luk, "Reconfigurable control variate Monte-Carlo designs for pricing exotic options," in *Proc. Int. Conf. on Field Programmable Logic and Applications*, 2010, pp. 364–367.

[10] Q. Jin, T. David, and W. Luk, "On comparing financial option price solvers on FPGA," in *Proc. IEEE Symp. on Field-Programmable Custom Computing Machines*, 2011.