Design of Power Efficient FPGA based Hardware Accelerators for Financial Applications

Hegner, Jonas Stenbæk; Sindholt, Joakim; Nannarelli, Alberto

Published in:
2012 NORCHIP

Link to article, DOI:
10.1109/NORCHP.2012.6403096

Publication date:
2012

Design of Power Efficient FPGA based Hardware Accelerators for Financial Applications

Jonas Stenbæk Hegner, Joakim Sindholt and Alberto Nannarelli
Dept. Informatics and Mathematical Modelling
Technical University of Denmark
Kongens Lyngby, Denmark

***Abstract*—Using Field Programmable Gate Arrays (FPGAs) to accelerate financial derivative calculations is becoming very common. In this work, we implement an FPGA-based specific processor for European option pricing using Monte Carlo simulations, and we compare its performance and power dissipation to the execution on a CPU. The experimental results show that impressive results, in terms of speed-up and energy savings, can be obtained by using FPGA-based accelerators at expenses of a longer development time.**

I. Introduction

In recent years financial applications have gained a significant portion of the computer market as the number of financial computations and transactions over the network is increasing everyday. This large number of financial data is often to be processed fast in data centers which consume a large amount of electrical power. Both aspects, low latency and low power dissipation, can take advantage of specially designed processing systems based on hardware accelerators; where the term accelerator is not only used in its physical denotation (faster), but also to refer to more power efficient hardware.
Field Programmable Gate-Arrays (FPGAs) are good candidates for hardware accelerators as they can exploit data parallelism and can be fine tuned to match exactly the algorithm.
In this paper, we show how FPGA based Application Specific Processors, or ASPs, can accelerate option pricing algorithms [1] with high energy efficiency. The case study is a Monte Carlo approach for pricing of European options. The ASP development time is shortened by resorting to *FloPoCo* (*Flo*ating-*Po*int *Co*res) a tool for generating arithmetic cores optimized for FPGAs [2].
We compare the execution of the Monte Carlo algorithm on a soft-core processor implemented on FPGA with that of the ASP by measuring execution time and power dissipation. Furthermore, we compare our design with the previous work of [3] where the FPGA based accelerator is compared to a dual-core processor. The experimental results confirm that not only the ASP execution is much faster, but also that the energy consumption is order of magnitudes lower.

II. Background

An *option* is a financial instrument that gives the holder the right, but not the obligation, to buy (*call option*) or to sell (*put option*) an asset by a certain date for a certain price [1]. For example, a European put option gives the owner the right to sell an asset at a *strike price* K at a specific time T. If at time T the value of the underlying asset is lower than the strike price K, the owner can make a profit. Otherwise, the option is not exercised (the owner does not sell). *European options* can be exercised only on the expiration date, while *American options* can be exercised at any time before the expiration date. For this reason, *European options* are generally easier to analyze.
The asset, or *security*, price changes can be modeled in a risk-neutral model by the Brownian motion model

$$dS = \mu S dt + \sigma S dz \tag{1}$$

where S is the security price, μ is the *drift rate* (the expected return), σ is the volatility (a measure of the uncertainty about the return provided by the stock) and $z(t)$ is a *standard Brownian motion* process. At $t = 0$ the value of the security is S_0. By dividing the life of the option into intervals of lenght Δt (1) becomes

$$S(t + \Delta t) - S(t) = \mu S(t)\Delta t + \sigma S(t)\epsilon\sqrt{\Delta t} \tag{2}$$

where ϵ is a random sample from a normal distribution with mean zero and standard deviation 1.0 [1].
By evaluating the asset value by the Black-Scholes-Merton formula (see [1] for the detail of the derivations) the value of the security at time T is

$$S(T) = S(0) \cdot e^{\left[\left(\mu - \frac{\sigma^2}{2}\right)T + \sigma\epsilon\sqrt{T}\right]} \tag{3}$$

The value of an European option can be computed by using a Monte Carlo simulation by evaluating (3) for several samples and then by computing the mean value. This approach is shown in Algorithm 1 for a *risk-neutral world* [1].
In the algorithm, the inputs are:

- initial security price S_0
- strike price K
- risk-free interest rate r
- security volatility σ
- time to expiration T (in years)
- number of simulations n.

The parameters r and σ are constant, and, therefore variables `VsqrtT`, `drift` and `expRT` are constant as well and can be precomputed.

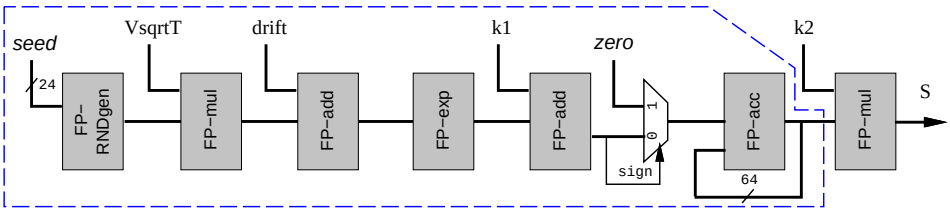

Fig. 1. ASP: single path implementation.

Algorithm 1 Monte Carlo European option pricing.

```
VsqrtT = σ√T
drift = (r − σ²/2)T
expRT = e^{rT}
sum = 0
for i = 1 to n do
    St = S0 · e^(drift + VsqrtT · Vrnd)
    if (St − K > 0) then
        sum = sum + (St − K) · expRT
    end if
end for
S = sum/n
```

III. Architecture of Accelerator

Algorithm 1 can be mapped on the Application Specific Processor (ASP) sketched in Fig. 1. Instead of performing the multiplication $S_0 \cdot e^{(\cdots)}$ in each cycle of the loop, we can divide K by S_0 off-line (or precompute it) and compare $e^{(\cdots)}$ directly to k1=K/S_0. Similarly, we can remove the multiplication by expRT out af the loop. The correct value of S is restored in the last stage by performing the multiplication by k2= S_0·expRT·$\frac{1}{n}$.

The units shown in Fig. 1 are *binary32*[1] floating-point (FP) units pipelined to work at a frequency of 100 MHz. Most of them were generated by *FloPoCo* by applying some modifications. The internal *FloPoCo* format has been changed to support $\pm inf$ (infinity) and NaN (not-a-number). Moreover, in all units subnormal numbers are flushed to zero.

Because of the special FP-accumulator, described in Sec. III-D, the ASP can sustain a throughput of 1 result every 10 ns (100 Mops/s).

The latency, expressed as number of clock cycles, of the *binary32* units composing the ASP is reported in Table I.

In the following, we first explain in detail the FP-units generated, especially the *non-standard* ones, and then, we show how the throughput of the ASP can be improved.

A. FP-Add and FP-Multiply

Both the floating-point adder and multiplier are standard *binary32* units generated by *FloPoCo* with some modifications to handle special values and subnormals.

[1]In the revision on the IEEE standard 754 *binary32* replaces the wording *single-precision*.

Unit	Function	Latency
FP-mul	$x \times y$	2
FP-add	$x + y$	3
FP-add3	$x + y + z$	3
FP-acc	$s = s + x$	3
FP-exp	e^x	4
FP-RNDgen	random	6

TABLE I
Overview of created floating point units

It is worth noting that adders, and to some extent multipliers, are mapped into the DSP blocks in the FPGA. This allows for significantly better area utilization and speed.

The number of stages (latency) are two for the FP-mul and three for the FP-add.

B. Random Number Generator

The detail of the unit to generate the random numbers, FP-RNDgen, is shown in Fig. 2 (left). To create a small area floating-point pseudo random number generator, four Linear Feedback Shift Registers (LFSRs) [4] are used. The period of the random number generator is 2^n-1 where n is the word-length of the LFSR output ($n = 24$ in our case). These four LFSRs produce uniformly distributed numbers, which are added together in an adder tree to get a Gaussian distribution. The average is found by shifting the result accordingly (divide by 4). This pseudo random bit pattern is then converted into a floating-point binary32 number in the range $(-1.0, 1.0)$[2] by mapping the random bit pattern in a positive fractional number and by subtracting an offset (using a FP-adder).

The latency of the first stages of the Random Number Generator is 3 plus another 3 cycles for the FP-add for a total of 6 cycles.

C. Exponential Function

The arbitrary functions computation is based on a second degree polynomial interpolator [5], sketched in Fig. 2 (center), implementing: $f(x) = x(ax + b) + c$.

For the exponential function, e^x, the constants table, range reduction and reconstruction is generated directly by *FloPoCo*. The unit's latency is 4 cycles.

D. FP-Accumulate

The floating-point accumulator is the critical unit of the ASP of Fig. 1. Because the FP-add has latency 3, it is only possible to accumulate a new value every 3 clock cycles by degrading the ASP performance. For this reason, a special FP

[2]Subnormals are flushed to zero.

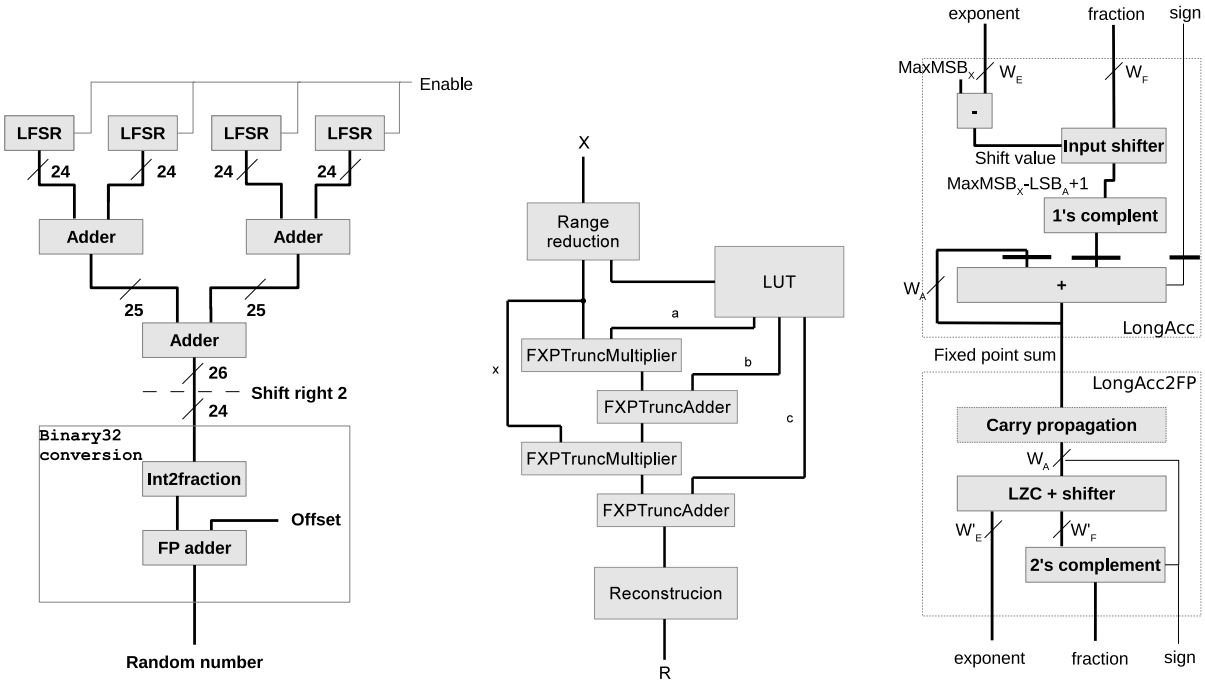

Fig. 2. Random Number Generator (left). Second degree polynomial evaluator (center). Floating-point Accumulator (right).

accumulator, derived from [6], is used to accumulate `sum`. The unit, depicted in Fig. 2 (right), converts the new value to be accumulated from *binary32* to 64-bit fixed-point by aligning the significand to the partial result stored as a 64-bit fixed-point. By taking advantage of the FPGA's fast carry-chains, this addition can be done in one clock cycle, and, consequently, the throughput of one result per clock cycle be maintained. In the last stage of the unit, the 64-bit fixed-point number is converted back to *binary32*.

E. Higher Throughput ASP

Once the main problem of the floating-point accumulation has been solved, the loop of Algorithm 1 can be partially unrolled, depending of the size of the FPGA, to increase the throughput. A number of ASPs of Fig. 1, called ASP-1P in the following, can be placed in parallel as shown in Fig. 3. We refer to the latter implementation as ASP-nP. A "funnel" FP-adder-tree is necessary to compute the final result of the Monte Carlo simulation.

To reduce the latency of this final FP-adder-tree, we developed a 3-input FP-adder (FP-add3) which has the same latency of the 2-input FP-add (3 cycles). FP-add3 is derived by FP-add by modifying the significand alignment, exponent computation and update, and by performing the sum of the three aligned significands.

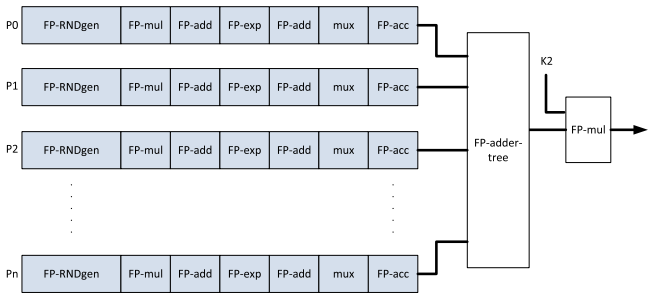

Fig. 3. ASP: 8-path implementation.

IV. Experiments

The platform used to run the experiments is a Xilinx ML550 board equipped with a Virtex-5 FPGA (XC5VLX50T) [7]. The board gives access to Kelvin resistors connected to the voltage regulators to monitor the power dissipation of the different parts of the FPGA chip.

We performed the experiments by running the Monte Carlo simulation for $n = 100,000$ on a desktop computer equipped with an Intel Core2 Duo E6600 processor running at 2.4 GHz, and on three different hardware configurations:

1) We mapped the MicroBlaze (*μBlaze*) Xilinx's soft processor core [8] on the FPGA (clock frequency of 100 MHz) and run the simulation on the processor. We added a 64-bit clock cycle counter, used as a stop-watch, to have a cycle-accurate measure of the execution time.
2) ASP-1P (Fig. 1).
3) ASP-4P (Fig. 3 with 4 P_i ASPs).

The implementation and measurement results are shown in Table II. The energy for the whole simulation is obtained by integrating the average power dissipation (measured from the board) over the execution time:

$$\text{energy} = P_{ave} \cdot (\text{n. cycles}) \cdot \frac{1}{f_{CLK}} \qquad [J]$$

For the execution on the desktop computer, we ran the simulation several times and averaged the execution time to have the cleanest data possible in a multicore and multitasking

	n. cycles	Timing exec. time $[ms]$	speed-up	Power P_{ave} $[mW]$	energy $[\mu J]$	ratio
Desktop computer	26,000,000	10.800	6.25*	N.A.	N.A.	-
μBlaze	163,065,258	1,603.653	-	387.5	631,878	-
ASP-1P	100,025	1.000	1,630	374.5	374	1,687
ASP-4P	25,031	0.250	6,515	684.8	171	3,686

* Ratio between cycle count of desktop computer and *μBlaze*.

TABLE II
RESULTS OF EXPERIMENTS FOR MONTE CARLO SIMULATION WITH $n = 100,000$.

	device	Timing exec. time $[ms]$	speed-up
μBlaze	LX50T	1,603.653	-
ASP-1P	LX50T	1.000	1,630
ASP-4P	LX50T	0.250	6,515
ASP-8P	LX330T	0.125	13,009
ASP-16P	LX330T	0.063	25,929

TABLE III
COMPARISONS OF *μBlaze* AND ASPS SIMULATIONS ($n = 100,000$).

	freq. clock $[MHz]$	n. cycles	exec. time $[ms]$	speed-up$^{(*)}$
CPU	3000	49,182,000	16.394	100
FPGA	80	10,000	0.125	13,045

$^{(*)}$ Speed-up (execution time) with respect to *μBlaze*.

TABLE IV
RESULTS OF [3] FOR $n = 100,000$.

environment[3]. The number of cycles for the Core2 simulation was computed (estimated) by multiplying the execution time by the clock frequency.

The data in Table II show that the *μBlaze* simulation is much less efficient than the Core2 simulation. However, even if the cycle count for the *μBlaze* were like the one obtained for the Core2, the execution on ASP-1P would have been about 260 times faster.

A significant part of the power dissipated in the FPGA is the FPGA intrinsic and static part: about 220 mW (50-55%) for *μBlaze* and ASP-1P. This constant amount of energy makes the simulations with shorter latency more favorable.

Next, we implemented the ASP of Fig. 3 on a larger FPGA (XC5VLX330T) to further exploit parallelism. Table III show the execution time and speed-up over the *μBlaze* simulation for ASPs with 8 and 16 parallel paths[4].

Finally, we compared our experiments with those of [3]. In [3], the same Monte Carlo simulation of Algorithm 1 was executed on a dual-CPU system equipped with Intel Core2 processors running at 3 GHz and on a Xilinx Virtex-5 FPGA platform part of the LabView development suite. The ASP implemented on the FPGA has a parallelism of 10 paths running at 80 MHz. The speed-up of the FPGA based simulation over the CPU one is 131 in [3].

By comparing the results of our experiments with those of [3], summarized in Table IV, we can conclude that

- The simulation on the Core2 in the two cases have similar latencies considering that many factors affect the execution time of desktop computers.
- The FPGA implementation of [3] shows the same performance of ASP-8P: 125 μs to simulate 100,000 elements.

[3]We shut-down all unnecessary services and set the clock to 2.4 GHz for the test.

[4]The board mounting the XC5VLX330T device is not provided with the power monitor.

V. CONCLUSIONS AND FUTURE WORK

The main purpose of this work was to study the energy efficiency of hardware accelerators. As the energy consumption is the product of average power dissipation and execution time, for speed-ups of 100 or more, unless the accelerator has a huge power dissipation (100 times or more that of the CPU), it will be more power efficient.

The measurements done on the case study confirmed the impressive speed-up achieved by FPGA based acceleration and demonstrated accelerators are order of magnitude more energy efficient than CPU execution.

One drawback of FPGA based accelerators is the development time. However, by using library of standard components and tools like *FloPoCo* these development times can be significantly shortened.

REFERENCES

[1] J. C. Hull, *Options, Futures and other Derivatives*, 8th ed. Prentice Hall, 2012.

[2] "FloPoCo Project". [Online]. Available: http://flopoco.gforge.inria.fr/

[3] T. Stratoudakis. "Hardware Acceleration of Monte Carlo Simulation for Option Pricing". 2010. [Online]. Available: http://www.wallstreetfpga.com/index.php?option=com_content&view=article&id=3&Itemid=2

[4] Xilinx Application Notes. "Efficient Shift Registers, LFSR Counters, and Long Pseudo-Random Sequence Generators". XAPP 052 July 7, 1996 (Version 1.1). [Online]. Available: http://www.xilinx.com/support/documentation/application_notes/xapp052.pdf

[5] M. Ercegovac and T. Lang, *Digital Arithmetic*. Morgan Kaufmann Publishers, 2004.

[6] F. de Dinechin, B. Pasca, O. Cret, and R. Tudoran, "An FPGA-specific Approach to Floating-Point Accumulation and Sum-of-Products," in *Proc. of International Conference on Field-Programmable Technology (FPT 2008)*, Dec. 2008, pp. 33–40.

[7] Xilinx User Guides. "ML550 Networking Interfaces Platform". UG202 (v1.4) April 18, 2008. [Online]. Available: http://www.xilinx.com/support/documentation/boards_and_kits/ug202.pdf

[8] Xilinx Inc. "MicroBlaze Soft Processor Core". [Online]. Available: http://www.xilinx.com/tools/microblaze.htm