A nested Multilevel Monte Carlo framework for efficient simulations on FPGAs

Irina-Beatrice Nimerenco

St Hugh's College

University of Oxford

A thesis submitted for the degree of

MSc in Mathematical Modelling and Scientific Computing

Trinity 2024

Acknowledgements

First, I would like to thank our Course Director, Dr. Kathryn Gillow, for her dedication, her kindness and her availability throughout the year and for making our entire cohort feel supported at all times.

I would like to thank my master thesis supervisor, Prof. Mike Giles, for his precious feedback and directions, for the time he allocated to answering my questions and all the advice and insight he gave me.

I also really want to thank Prof. Yuji Nakatsukasa and Prof. Raphael Hauser for their encouragements and their guidance this year.

I am very grateful to all academics and internship supervisors that inspired and motivated me during my studies.

I would like to thank many people from my cohort in Oxford for being friendly and outgoing, and I hope that everyone will be successful and happy in the next steps of their studies and career. In particular, I would like to mention Gabriel, Luo, Botond, Makoto and Ben B. - thank you all for the fun and hard time we had together.

Then of course I would like to thank my parents and my close friends for the love and support they showed me. Finally, I would like to thank specifically my mother for proofreading my thesis.

Abstract

Multilevel Monte Carlo (MLMC) is a widely used method allowing to reduce the computational cost of stochastic simulations, notably for pricing financial options. The main idea is to approximate the expectation of a random variable using multiple resolutions in the numerical schemes that solve the Stochastic Differential Equation decribing the dynamics of the underlying asset of the option. This method is highly parallelisable which makes is suitable for efficient hardware implementations. An important avenue for decreasing the power consumption and speeding MLMC simulations further is the use of lower precision calculations which are efficiently performed on configurable hardware devices such as Field-Programmable Gate Arrays (FPGAs). A framework that mixed MLMC with customizable precision was proposed in [6], however in this thesis we propose a new framework that improves over the previous literature using two important directions : first, using a rounding error model we propose a procedure to adapt the precision of several variables at each level ; second, we assume that the random numbers used in path generation on FPGAs is performed with almost negligible cost, and we provide arguments and examples of random number generators that would allow this. This way, we argue that our proposed framework offers higher computational savings.

Contents

List of Figures

Chapter 1

Introduction

Monte Carlo simulations are used in various applications to approximate the expectation of random variables that can be simulated but do not have a known analytical expression. In particular the targeted application of our work and most of the literature cited in this thesis is in financial applications where the quantity of interest is the expected payoff of a financial option. Monte Carlo methods consist of generating a large number of samples and using their mean to estimate the expected payoff of an option that is based on a stochastic process. With the standard Monte Carlo estimator obtaining a desired variance of ϵ^2 requires $\mathcal{O}(\epsilon^{-2})$ samples which can be very expensive. Therefore a number of variance reduction techniques such as multilevel Monte Carlo, quasi Monte Carlo, control variates, antithetic variables and more have been developed to achieve higher accuracy for the same computational cost [18].

The Multilevel Monte Carlo (MLMC) method for SDEs was first proposed in 2008 in the paper [17] and was since adapted to numerous applications with PDEs and SPDEs and combined with other variance reduction methods (eg. with quasi-Monte Carlo in [15]). A good review of MLMC applications and extensions is [18]. The main idea of the method is to approximate the payoff on multiple grids of different resolutions and use an optimal number of samples on each level. The computational savings come from the fact that most samples are computed on the coarser levels and hence are less expensive to calculate while only a few samples from the finest levels are required.

Among the directions in which the computational complexity of MLMC methods could further be reduced an important avenue is the use of lower precision in calculations. Low precision is already successfully exploited in other domains of scientific computing : a few examples are in optimisation [9], machine learning [22], and iterative refinement [26, 4] in numerical linear algebra. An overview of the research on mixed precision for the standard Monte Carlo framework is provided in [8] but only a few references study the potential of low precision computation in the multilevel framework [39]. To the best of our knowledge,the only MLMC framework with optimised precision in the literature is [6], but their framework uses a uniform precision for each Monte Carlo level instead of allowing each intermediary operation to have an optimised precision.

An important motivation for an MLMC framework with variable precision would be to perform the low precision computations on reconfigurable hardware devices such as Field Programmable Gate Arrays (FPGAs). FPGAs contain customizable logic blocks and connectors that allow to easily adapt the digital circuit architecture for a specific application, leading to a highly parallelised and optimised implementation. Therefore they are successfully exploited in applications that require high speed and have high computational workload such as signal processing and real time applications like high frequency trading [30, 5].

The main goal of this thesis is to build a nested MLMC framework specifically to exploit FPGAs, which would perform the low precision computations in fixed-point arithmetic very efficiently. This would allow to accelerate tasks like pricing financial derivatives or simulations in biochemistry [18] and reduce significantly the energy consumption of the devices that are used.

In the following sections of this chapter, we introduce preliminaries on FPGAs, fixed-point arithmetic and MLMC simulations to help the reader delve into our project. We make a short literature review and summarise our contributions in the last two sections. Then the thesis is organised as follows : In Chapters 2 to 4 we make the necessary arguments to build the nested framework and investigate the computational savings and limitations. We also explain how the optimal precisions evolve over levels and connect them to the evolution of the rounding errors with level (see [39]). Finally in Chapter 5 we discuss approximate random number generation methods that would be relevant for the FPGAs (including aspects of their implementation into the hardware) and would reduce the overall computational cost further.

Our numerical experiments were implemented in Matlab using the Fixed-Point Arithmetic toolbox.

1.1 Background on FPGAs

In this section we quickly introduce the main characteristics of FPGAs in order to highlight their potential for scientific computing applications. We will notably detail the main differences between FPGAs, CPUs and GPUs, as we assume the reader is more familiar with the latter two devices. We also present the main components of FPGAs and their applications.

FPGAs are devices made of a ”fine-grain grid” ([40]) of logic blocks and interconnects that can be reconfigured almost infinitely many times after manufacturing to adapt to the desired applications and perform new functions. An FPGA is able to reoptimise its architecture and change its configuration without requiring any change in the actual physical circuit. This is illustrated in Figure 1.1, where two independent functions (function A and function B) are implemented on the same FPGA. These functions can therefore be executed in parallel and so are the blocks AF5a and AF5b which are part of function A. Notice that in this example four blocks remain unconfigured and only the connections represented by the dark arrows are activated so most of the interconnect area is also unused. This configuration (blocks and connections) could be changed whenever a new one is needed.

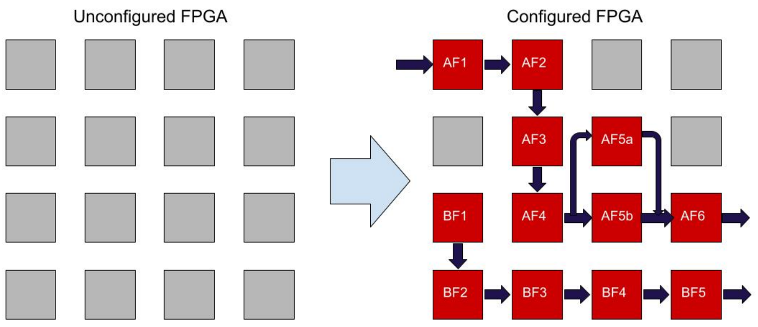

Figure 1.1: A simplified representation of an unconfigured FPGA (left) and a configured FPGA (right). Reproduced from [41].

This flexibility allows to reduce the time and cost of designing application specific platforms and makes FPGAs popular for hardware design testing and optimisation. However its configuration requires expertise as it is performed in a Hardware Description Language (HDL) such as VHDL or Verilog. ”HDL code is more like a schematic that uses text to introduce components and create interconnections” [29]. Other devices (especially CPUs) are easier to set up, which may explain the limited attention that FPGAs have received from the scientific computing community so far.

Thanks to their flexibility and efficiency, FPGAs are exploited in a number of applications namely telecommunications, automotive, aerospace, consumer electronics, medical imaging and robotics [29, 21]. They are also used in high-frequency trading to dramatically reduce latency [30] and some authors [5] accelerate the integration of trading algorithms from high level languages to FPGA circuits by using high-level synthesis (HLS) [1].Broadly speaking, FPGAs are a key component in a number of embedded systems because they ease hardware design updates and accelerate the development process (compared to developing an Application Specific Integrated Circuit (ASIC)), and they are important for streaming applications where high speed, parallelism and reduced power consumption are sought [41].

An FPGA is mainly made of logic blocks that can be linked to Look-Up-Tables, memory blocks or more specific units like DSPs (see Table A.1). The logic blocks on FPGAs perform bit-wise operations on a small number of bits. On the other hand, traditional devices like CPUs and GPUs act on larger blocks of data and although they have their own advantages their architecture is fixed. CPUs are efficient for implementing complex software as they can execute complicated functions but they suffer from latencies because the result of each operation has to be put into memory after being processed by a logic block. The data then goes back and forth between memory blocks and logic blocks which takes time. For this reason, CPUs are not adapted to streaming applications in which several tasks are concatenated and only a part of the outputs need access to memory. GPUs partially solve this problem as they are specialized for treating large amounts of data by performing vectorised operations. They are particularly efficient for parallelisable tasks. However once configured for a specific application FPGAs can be faster than GPUs and handle important workloads and more diverse sizes of data [40], especially because their implementation has more flexibility for optimisation which maximises their efficiency [41]. In early product development stages FPGAs can also be used instead of investing time and money in the development of ASICs and another important application of large modern FPGAs is in hardware design testing [41].

To conclude, we believe that the success of FPGAs in other applications and their advantages compared to other devices could incite more research on this topic from the scientific computing community. MLMC is a suitable application for FPGAs because it can make good use of very cheap, low-precision calculations.

1.2 Fixed-point arithmetic

FPGAs are very efficient for fixed-point calculations and in fact, although recent models have integrated new components that implement floating-point arithmetic, FPGAs originally could compute in floating-point only if they were configured with

[1]HLS : According to Wikipedia HLS is ”an automated design process” that takes high level code and turns it into HDL code which then be ”synthesised to the gate level”.

$$\begin{tikzcd} {} \arrow[r, "x_0 = \theta"] & \boxed{f_1} \arrow[r, "x_1"] & \boxed{f_2} \arrow[r, "x_2"] & \boxed{f_3} \arrow[r, "x_3 = P"] & {} \end{tikzcd}$$

Figure 1.2: Forward computation of the output P from the input θ.

user-defined functions for that, as the extensive literature on floating-point implementation on FPGAs shows. In this work we exploit fixed-point arithmetic as it allows us to formulate a mathematical problem to optimise the precision used in every operation that is performed to compute a sample of the payoff. The key difference between fixed-point and floating-point arithmetic is that the latter allows the exponent of a variable (which is represented by the position of a *point*) to adapt in order to represent the variable with as high accuracy as possible, whereas in fixed-point arithmetic this scaling is fixed when the variable is created. We define here the notation and the bounds on the rounding errors on the fixed-point variables.

Say a parameter or input used to compute P (in full precision) is denoted as θ and every elementary operation performed to compute P defines an intermediary variable denoted $\theta = x_0, x_1, ..., x_m$ as illustrated in 1.2.

Now each variable is represented in fixed-point arithmetic as

$$x_i = (-1)^s 2^{e_i}(2^{-d_i} n), \quad \text{with} \quad n \in [0, 2^d - 1]. \tag{1.1}$$

where e_i is the exponent and defines the range of the variable such that $|x_i| < 2^{e_i}$, s_i is the sign bit, n is the represented integer and d_i is the word length or *bit-width*, ie. the number of bits assigned to represent n. Due to its finite precision the variable x_i is computed with a rounding error δx_i. This error comes only from the last operation performed to obtain x_i and must not be confused with the overall error between the accurate value of x_i and the one obtained after all the fixed-point operations with possibly propagating errors. Additionally we use the convergent round-to-nearest rounding mode at every fixed-point operation, which is presented in [36]. Therefore the error δx_i can be bounded by :

$$|\delta x_i| \leq 2^{e_i - d_i - 1}. \tag{1.2}$$

Cost of elementary operations :
To set the precision of each variable in our MLMC framework we will minimise the computational cost under a constraint on the variance. Therefore we use a model introduced by [32] to approximate the cost of the elementary operations in fixed-point arithmetic. The cost of addition and multiplication of variables x_i, x_j is

$$\begin{aligned} x_i + x_j &\quad \to d_i + d_j \\ x_i \times x_j &\quad \to d_i \times d_j \end{aligned} \tag{1.3}$$

The reasoning behind these approximates is fairly natural as illustrated in Figures 1.3 and 1.4. A multiplication is performed bitwise : each bit of variable a is multiplied by the variable b then the results are summed over the bits of a. For addition in fixed point arithmetic the numbers are aligned so that their fixed points are superposed. Then the fraction bits are summed and the integer bits are summed separately, and the result of $a + b$ is obtained. The position of the fixed point can then be adapted if the new variable has a different specified fraction length.

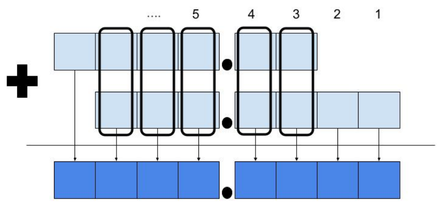

Figure 1.3: Illustration of additions in fixed-point arithmetic.

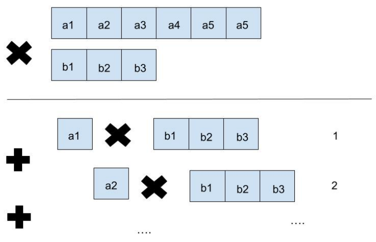

Figure 1.4: Illustration of multiplication of two variables in fixed-point arithmetic.

Matlab fi objects :

In Matlab we use the Fixed-Point Arithmetic toolbox to represent fixed point variables. The parameters are the "signedness" (1 if the variable is signed, 0 otherwise) the word length w (which should include an extra bit if the variable is signed) and the fraction length f, which is $f = d - e$, so that $x = 2^{-f}N$, where for a signed variable $N \in [-2^w, 2^w - 1]$ is an the integer represented on $w = d + 1$ bits using the two's complement representation, and for an unsigned variable $N \in [0, 2^w - 1]$ with $d = w$. Note that the fraction length can be larger than the word length, which corresponds to $|x| < 1$. Another important point is that Matlab offers several rounding modes. We used the "Convergent" mode [36] which is a round-to-nearest mode that rounds ties to the nearest even number. This treatment of ties is the most statistically accurate (it does not introduce any bias).

1.3 Pricing an European vanilla option with the Euler-Maruyama scheme

In this section we introduce the work example used throughout the thesis and the values of the parameters (as they will be the same throughout our experiments). This also allows to introduce the notation used in error modelling and analysis sections.

The payoff function for the European vanilla option is

$$P = max(S - K, 0) \tag{1.4}$$

Where K is the strike. The underlying asset follows the Geometric Brownian Motion : $dS_t = rS_t dt + \sigma S_t dW_t$, where r is the interest rate and σ is the volatility of the asset. In our numerical experiments $r = 0.05$ and $\sigma = 0.2$. This SDE is solved numerically using the Euler-Maruyama scheme

$$S_{i+1} = S_i + rS_i h + \sigma S_i \sqrt{h} dW_i. \tag{1.5}$$

In practice we will scale the asset price S by dividing it by K so the payoff becomes $P = max(S-1, 0)$. Then the forward code to compute a path where every elementary operation is written separately to show the intermediary variables is presented in Algorithm 1.

Algorithm 1 Geometric Brownian Motion path calculation (in fixed-point arithmetic)

```
Inputs: interest rate $r$, volatility $\sigma$, maturity $T$, initial asset price $S_0$, number of time steps $N$
$h \leftarrow T/N$
$rh \leftarrow r * h$
$sh2 \leftarrow \sqrt{h}\sigma$
generate random normals $dW_i$ for $i = 1, \dots, N$
for $i = 1, \dots, N$ do
    $mult1_i \leftarrow sh2 * dW_i$
    $sum1_i \leftarrow rh + mult1_i$
    $mult2_i \leftarrow S_i * sum1_i$
    $S_{i+1} \leftarrow S_i + mult2_i$
end for
```

1.4 Monte Carlo methods for SDEs applications

In financial applications, Monte Carlo approaches are used to estimate the expectation of some payoff functional P that is computed on an underlying asset price S_t that follows a stochastic differential equation (SDE). The standard Monte Carlo approach consists of computing the asset price with a finite difference scheme to approximate the solution of the SDE. In this thesis we use the Euler-Maruyama scheme [25]. Noting

$\hat{P}^{(i)}$ a sample of P and using $\hat{N}$ samples the standard Monte Carlo estimate for the expectation is

$$\frac{1}{\hat{N}} \sum_{i=1}^{\hat{N}} \hat{P}^{(i)}. \tag{1.6}$$

To achieve a Mean Square Error of ε^2 with the standard estimator requires a time step $h = \mathcal{O}(\varepsilon)$ and a number of samples $\mathcal{O}(\varepsilon^{-2})$ leading to an overall computational cost of $\mathcal{O}(\varepsilon^{-3})$, while the multilevel Monte Carlo allows to decrease the cost to $\mathcal{O}(\varepsilon^{-2})$ up to a logarithmic factor [17].

The idea of the multilevel Monte Carlo is to use different levels of time discretisations in order to split the computational work such that the total cost is minimised and the desired overall accuracy is achieved. Say we use two different resolutions where the coarser is denoted by the index 0 and the finest by the index 1. The two level Monte Carlo estimator is

$$\mathbb{E}[P_1] \approx \frac{1}{N_0} \sum_{i=1}^{N_0} P_0^{(i)} + \frac{1}{N_1} \sum_{i=1}^{N_1} (P_1 - P_0)^{(i)}. \tag{1.7}$$

In each sample of $P_1 - P_0$ both quantities are based on the same random normal increments. To make this clearer, note $S_{i,0}, S_{i,1}$ the Euler-Maruyama estimates of the asset price that are used to compute P_0 and P_1. Let's suppose that the asset price is a Geometric Brownian Motion (GBM) with drift rS_t and volatility $\sigma S_t dW_t$. Note h is the time step of the fine mesh and T the final time. To compute the finer path $S_{i,1}$, $N = T/h$ random normal increments ΔW_i are generated and used to update the price at each time step as

$$S_{i+1,1} = S_{i,1}(1 + rh + \sigma\sqrt{h}\Delta W_i) \qquad i = 0, 1, \ldots, N. \tag{1.8}$$

Then the price $S_{i,0}$ is updated using the same increments ΔW_i as in [20] by

$$S_{i+1,0} = S_{i,0} + S_{\underline{i},0}(rh + \sigma\sqrt{h}\Delta W_i) \qquad i = 0, 1, \ldots, N, \tag{1.9}$$

where $\underline{i} = 2\lfloor i/2 \rfloor$. This is how the asset price at two consecutive resolutions is computed to obtain a sample of $P_1 - P_0$.

Then noting C_0 the cost of a sample of P_0, C_1 the cost of a sample of $P_1 - P_0$ the total computational cost of the estimation is $N_0C_0 + N_1C_1$, and noting $V_0 = \mathbb{V}[P_0]$ and $V_1 = \mathbb{V}[P_1 - P_0]$ the total variance is $N_0^{-1}V_0 + N_1^{-1}V_1$. Therefore we can minimise the total computational cost for a fixed value of the total variance using a Lagrange multiplier approach and considering N_0, N_1 as real variables in the optimisation.

This idea is naturally generalised to more levels. Say we use $L + 1$ levels that we denote by the index $l \in \{0, 1, \dots, L\}$. For each level l we define an approximate P_l computed in full-precision with a time step of size $h_l = h_0 2^{-l}$. Then for L sufficiently large we have $\mathbb{E}[P] \approx \mathbb{E}[P_L]$. We then decompose $\mathbb{E}[P_L]$ as follows

$$\mathbb{E}[P_L] = \sum_{l=0}^{L} \mathbb{E}[P_l - P_{l-1}] \tag{1.10}$$

with the convention $P_{-1} = 0$. In each sample of $P_l - P_{l-1}$ both terms are computed using the same random (normal) increments as explained previously in the two level case. We also define the notation $\Delta P_l = P_l - P_{l-1}$ for all levels l with the convention $\Delta P_0 = P_0$. For each level the expectation $\mathbb{E}[\Delta P_l]$ is approximated using the standard Monte Carlo estimator (1.6) with N_l samples. Then noting by C_l the cost of computing a sample of ΔP_l and $V_l = \mathbb{V}[\Delta P_l]$ the overall cost of the estimation is

$$\sum_{l=0}^{L} N_l C_l \tag{1.11}$$

and the overall variance is

$$\sum_{l=0}^{L} N_l^{-1} V_l. \tag{1.12}$$

Using a Lagrange multiplier $\lambda^2 \in \mathbb{R}$ and minimising the cost under the constraint $Variance = \varepsilon^2$ gives that the number of optimal samples is $N_l = \lambda\sqrt{V_l/C_l}$. Then from the variance constraint we get $\lambda = \varepsilon^{-2} \sum_{l=0}^{L} \sqrt{V_l C_l}$, therefore the overall cost of the multilevel Monte Carlo estimator is

$$Cost_{MLMC} = \varepsilon^{-2} \left(\sum_{l=0}^{L} \sqrt{V_l C_l} \right)^2 . \tag{1.13}$$

For comparison, noting $V = \mathbb{V}[\hat{P}]$ and C the variance and cost in the standard Monte Carlo estimator the overall cost would be $\varepsilon^{-2} VC$. As [18] shows, if the factor $V_l C_l$ decreases (resp. increases) with level then the total cost of MLMC is approximately $\varepsilon^{-2} V_0 C_0$ (resp. $\varepsilon^{-2} V_L C_L$) so it is smaller than the standard Monte Carlo cost by a factor C_0/C_L (resp. V_L/V_0). The cost of a sample increases with level and for Lipschitz payoffs for the Euler-Maruyama scheme the variance V_l decreases exponentially with level, which leads to the MLMC estimation being cheaper than the standard Monte Carlo estimation.

1.4.1 Accuracy achieved by Multilevel Monte Carlo

To measure the accuracy of Monte Carlo estimation we will use the Mean squared error (MSE) or the Root Mean Square deviation (RMS) which are defined below :

Definition 1. *The Root Mean Square deviation of an estimator $\underline{P}$ of the expected payoff $\mathbb{E}[P]$ is*

$$RMS = \sqrt{MSE} = \sqrt{\mathbb{V}[\underline{P}] + Bias(\underline{P})^2} = \sqrt{\mathbb{V}[\underline{P}] + \mathbb{E}[\underline{P} - P]^2}. \tag{1.14}$$

The following theorem from [17, 10] makes precise the convergence of the MLMC algorithm.

Theorem 2. *Let P denote a random variable, and let P_l denote the corresponding level l approximation. If there exist independent estimators Y_l based on N_l Monte Carlo samples and positive constants $\alpha, \beta, \gamma, c_1, c_2, c_3$ such that $\alpha \geq \frac{1}{2} min(\beta, \gamma)$ and*

i) $$|\mathbb{E}[P_l - P]| \leq c_1 2^{-\alpha l} \tag{1.15}$$

ii) $$\mathbb{E}[Y_l] = \begin{cases} \mathbb{E}[P_0] & l = 0 \\ \mathbb{E}[P_l - P_{l-1}] & l > 0 \end{cases} \tag{1.16}$$

iii) $$\mathbb{V}[Y_l] \leq c_2 N_l^{-1} 2^{-\beta l} \tag{1.17}$$

iv) $$\mathbb{E}[C_l] \leq c_3 N_l 2^{\gamma l}, \text{ where } C_l \text{ is the computational complexity of } Y_l \tag{1.18}$$

then there exists a positive constant c_4 such that for any $\varepsilon < e^{-1}$ there are values L and N_l for which the multilevel estimator

$$Y = \sum_{l=0}^{L} Y_l \tag{1.19}$$

has a MSE with bound

$$MSE = \mathbb{E}[(Y - \mathbb{E}[P])^2] < \varepsilon^2 \tag{1.20}$$

with a computational complexity C with bound

$$\mathbb{E}[C] \leq \begin{cases} c_4 \varepsilon^{-2}, & \beta > \gamma \\ c_4 \varepsilon^{-2} (\log \varepsilon)^2, & \beta = \gamma \\ c_4 \varepsilon^{-2-(\gamma-\beta)/\alpha}, & \beta < \gamma \end{cases} \tag{1.21}$$

The assumptions made in the theorem are satisfied for a large class of payoff functions, in particular for the European vanilla option that we use for most numerical experiments the parameter values are $\alpha = 1, \beta = 1, \gamma = 1$.

1.4.2 Multilevel Monte Carlo algorithm

We can now introduce the initial MLMC algorithm described in [18] which is presented in Algorithm 4. The algorithm computes an initial user-specified number of paths and uses the estimated level variances V_l and costs C_l to determine the number of additional samples that need to be computed. If the assumptions of Theorem 2 hold the algorithm converges and achieves the desired variance $\varepsilon^2/2$. To achieve an $MSE \leq \varepsilon^2$ the algorithm ensures that $Bias(P_L)^2 \leq \varepsilon^2/2$ and $\mathbb{V}[P_L] \leq \varepsilon^2/2$ [2]. As mentioned in [18], to enforce the weak convergence $\mathbb{E}[P - P_L] \leq \varepsilon/\sqrt{2}$ we consider that $\mathbb{E}[\Delta P_l] \propto 2^{-\alpha l}$, which leads to

$$\mathbb{E}[P - P_L] = \sum_{l=L+1}^{\infty} \mathbb{E}[P_l - P_{l-1}] \approx \mathbb{E}[P_L - P_{L-1}]/(2^\alpha - 1). \tag{1.22}$$

Then we obtain the following test for the weak convergence, which is used in Algorithm 4 :

$$\mathbb{E}[\Delta P_L]/(2^\alpha - 1) > \varepsilon/\sqrt{2}. \tag{1.23}$$

1.4.3 Nested multilevel Monte Carlo

In this thesis we attempt to reduce the computational work by computing some paths in lower precision, so here we formulate the output estimator that includes low precision computations. We decompose each level expectation $\mathbb{E}[\Delta P_l]$ into the sum of an estimate computed in low precision and a correction term. This leads to the identity

$$\mathbb{E}[P] \approx \sum_{l=0}^{L} \left(\mathbb{E}[\widetilde{\Delta P_l}] + \mathbb{E}[\Delta P_l - \widetilde{\Delta P_l}] \right). \tag{1.24}$$

Throughout the thesis we denote by $\tilde{P_l}, \widetilde{\Delta P_l}$ the samples that are computed in low precision (on the FPGA). Again each expectation is obtained with a standard Monte Carlo estimator using $\tilde{N_l}$ samples for $\mathbb{E}[\widetilde{\Delta P_l}]$ and N_l samples for $\mathbb{E}[\Delta P_l - \widetilde{\Delta P_l}]$. We note $\tilde{C_l}$ the cost of computing a sample of $\widetilde{\Delta P_l}$ and C_l the cost of a sample of $\Delta P_l - \widetilde{\Delta P_l}$ (which is necessarily larger than $\tilde{C_l}$), and similarly $\tilde{V_l} = \mathbb{V}[\widetilde{\Delta P_l}]$ and $V_l = \mathbb{V}[\Delta P_l - \widetilde{\Delta P_l}]$. Then the total computational cost and the total variance of the

[2]The case of a different split of the MSE is discussed in [23, 11].

nested estimator of the output is

$$Cost = \sum_{l=0}^{L} \tilde{N}_l \tilde{C}_l + N_l C_l \tag{1.25}$$

$$Variance = \sum_{l=0}^{L} \tilde{N}_l^{-1} \tilde{V}_l + N_l^{-1} V_l. \tag{1.26}$$

Suppose we would like the overall variance to be smaller than $\varepsilon^2/2$. Similarly to the standard MLMC case, using a Lagrange multiplier $\lambda_M^2 \in \mathbb{R}$ gives $N_l = \lambda_M \sqrt{V_l/C_l}$ and $\tilde{N}_l = \lambda_M \sqrt{\tilde{V}_l/\tilde{C}_l}$ for all levels l. Plugging the expressions of the number of samples back in $Variance = \varepsilon^2/2$ gives

$$\lambda_M = 2\varepsilon^{-2} \left(\sum_{l=0}^{L} \sqrt{\tilde{V}_l \tilde{C}_l} + \sqrt{V_l C_l} \right) \tag{1.27}$$

and in turn the total cost is

$$Cost_{nestedMLMC} = 2\varepsilon^{-2} \left(\sum_{l=0}^{L} \sqrt{\tilde{V}_l \tilde{C}_l} + \sqrt{V_l C_l} \right)^2. \tag{1.28}$$

This expression is very similar to the total cost (1.13) achieved by the classical MLMC but due to how we defined the notation in this section the cost C_l is not the same as in the previous sections. Similarly, here the factor 2 in the total cost only comes from the different definition of the desired variance compared to (1.13). Then [20] shows that if $V_l/\tilde{V}_l \ll \tilde{C}_l/C_l \ll 1$ then the nested estimation leads to a computational saving of a factor approximately $\max_l \tilde{C}_l/C_l$ compared to the standard MLMC framework.

1.5 Literature overview

In this section we aim to summarise the previous research on the use of FPGAs and low precision calculations in MLMC simulations. As mentioned above FPGAs have been used as accelerators in various applications including financial forecasting [30, 5]. It is therefore only natural that a number of previous works implemented MLMC frameworks using FPGAs as accelerators in their hardware architecture. This work has so far been performed mainly by the computer science community : In [13] experiments on a Xilinx Virtex 6 FPGA show that FPGA architectures can be significantly faster and power-efficient for MLMC simulations than full CPU architectures. In order to simplify the configuration of FPGAs and integration with other hardware devices for MLMC applications, [34] proposed a Domain Specific Language. Both

[34, 13] discuss the speed and power efficiency of FPGAs but they focus on the hardware design and testing aspects without optimising the precision in the computations, keeping all variables to single precision. The first to use lower precision were [6]. The authors introduce a heuristic to fix the precision used on each level and apply it to price an Asian option in the Heston market model. The paper [6] also includes a description of their hardware design and tests on real a Xilinx Virtex 6 FPGA. Contrarily to our model, [6] do not allow the variables from the same Monte Carlo level to have different word lengths and the random variables they use are still in single precision. Finally, for classical Monte Carlo simulations reduced precision was already investigated as detailed in the overview [8].

In computer science, fixing appropriate bit-widths is an important factor to reduce latency, area and power consumption in hardware designs. Therefore procedures to tweak the range and precision of fixed-point variables are an extensively studied topic. According to the classification in [32] precision analysis can be divided into two main directions : static analysis and dynamic analysis. Dynamic analysis requires a potentially large amount of stimuli input signals to analyse the design with a sufficient confidence, which makes it prohibitive for MLMC applications. On the other hand, static analysis makes use of analytical error models to guarantee a desired accuracy [32]. In this thesis we use a number of simulated signals to determine the range of the variables and an analytic model for the precision that ensures accuracy assuming no overflow. Our error model is based on Algorithmic Differentiation, which was used to bridge bit-width optimisation methods for fixed-point and floating-point hardware designs in [14]. Algorithmic differentiation is widely used in financial applications as it allows to compute the sensitivity of the output to every input at every intermediary step, making it a good tool for tweaking some empirical parameters in a model as well.

Modelling rounding errors is also of interest in the scientific computing research : rounding errors occur whenever a problem is solved numerically with finite precision computations and sometimes limit the accuracy that can be achieved within a realistic computational time. Therefore various methods that attempt to compensate rounding errors or extrapolate inaccurate outputs to obtain an accurate estimate of the solution exist, in particular for numerical solutions of differential equations. A good example is the Kahan summation [28]. In our application we are interested in the joint effect of time stepping and finite precision in numerical schemes for the solution of SDEs. Theoretical analysis and results on this issue are surprisingly scarce, although instead many recent works try to explain the efficiency of stochastic rounding [12], which is a related topic and also appeals to rounding error modelling. For the Euler-Maruyama scheme, two important contributions in the analysis of net rounding errors are [3, 39]. [3] show how the rounding errors at each step accumulate and provide an assymptotic bound on the error that depends on the time step and the unit roundoff. Based on this result, [39] provide further justification of the rounding error model from [3] and generalise the framework to the case where the exact random numbers in the Euler-Maruyama scheme are replaced by approximate random numbers.

1.6 Objective and summary of the contribution of this work

The aim of this project is to show the advantage of using FPGAs in conjunction with CPUs to reduce the computational cost of MLMC simulations. We suggest using a nested MLMC framework in which the low precision computations are performed on an FPGA and the full-precision ones are done on a CPU. The FPGA offers high flexibility allowing us to define a separate bit-width for every variable in the FPGA path simulation, which has not been done in the previous literature. A method to globally optimise the bit-widths and the number of samples that need to be calculated is introduced and tested in Matlab. This optimisation is performed before the main path generation loop is executed and is based on a linear approximation of the error made on the payoff when computing in low precision. We also show that starting from a certain level it is more efficient to use only the CPU (so going back to standard MC levels in our MLMC framework) and our global optimisation method allows to determine the number of samples needed on both the nested and standard MC levels.

Another strength of our framework compared to [6] is that the precision of the random variables that is used is also optimised, which allows to significantly reduce the cost since random number generation in full precision is expensive. We suggest to use approximate random normal variables instead to reduce the cost for the paths generated on the FPGA. Three relevant approximate random number generators for Gaussian variables are compared in the last chapter and the solution suggested in the end costs only a few bitwise operations per approximate Gaussian number. Our framework allows to use them on the FPGA without violating the assumption on the telescoping summation (1.24), which is important to ensure that the estimated expectation is accurate.

Chapter 2

Rounding error model via linear approximation

In order to choose the right precision in the FPGA calculations we need to approximate the overall error and variance caused by the accumulation of rounding errors in the low precision path simulation. In this chapter, focusing on the single level Monte Carlo estimator to simplify the exposition, we model the overall rounding error $P-\tilde{P}$ that is committed when approximating the full precision payoff P by the fixed-point payoff $\tilde{P}$. We then derive a bound on the corresponding variance $\mathbb{V}[P-\tilde{P}]$, which is a proxy for the accuracy of the estimation $\mathbb{E}[\Delta P_l]$.

2.1 Linearised model of the total error

In order to approximate the overall error we will consider that the rounding errors are small and use a first order Taylor expansion as follows :

$$P-\tilde{P} \approx \frac{dP}{dx} \approx \sum_{i}^{m} \frac{\partial P}{\partial x_i}\delta x_i \tag{2.1}$$

where the variables x_i are defined as the intermediary variables that are computed to calculate a sample of the payoff P and δx_i is the rounding error made on x_i in the fixed-point calculation, as in the Section 1.2 of the introduction.

The partial derivatives of the payoff represent how sensitive the output is to an error made on the variable x_i. We define the *sensitivity* of the payoff P to variable x_i as

$$\bar{x}_i = \frac{\partial P}{\partial x_i}. \tag{2.2}$$

This allows to express the overall error caused by rounding errors in the fixed-point calculation as

$$P - \tilde{P} = \sum_{i=1}^{m} \bar{x}_i \delta x_i. \tag{2.3}$$

and the variance of the overall error is then

$$\mathbb{V}[P - \tilde{P}] = \sum_{i=1}^{m} \mathbb{V}[\bar{x}_i \delta x_i] + 2 \sum_{i \neq j \in [1,m]} Cov\left(\bar{x}_i \delta x_i, \bar{x}_j \delta x_j\right). \tag{2.4}$$

For each available sample of P the sensitivities are computed by algorithmic differentiation as detailed in the next section.

2.2 Algorithmic differentiation for sensitivity analysis

In this section we detail how we used Algorithmic Differentiation (AD) to compute the sensitivities for the European vanilla option path calculation. We limit our example to the first level of the Monte Carlo estimator. A similar backward algorithm is used for higher levels with two additional sensitivities being computed at each time step : $\overline{mult2c}$ and $\overline{Sc}$ ("c" stands for "coarse path"). For each path computed with the forward Algorithm 1 the corresponding backward sensitivities are computed as detailed in Algorithm 2 below.

Algorithm 2 Geometric Brownian Motion backward sensitivities calculation

```
Inputs: variables $rh$, $sh2$, asset price $S_i$ and normal increments $dW_i$ (that were used to compute $S$) for all time steps $i = 0, \ldots, N$, number of time steps $N$
$\bar{S}_N = \mathbb{1}(S_N > 1)$
initialise all the other sensitivities to zero
for $i = N : -1 : 1$ do
    $\overline{mult2}_i \leftarrow \overline{S}_i$
    $\overline{S}_{i-1} \leftarrow \overline{S}_i * \overline{sum1}_i + \overline{mult2}_i \quad (= \overline{S}_i * (1 + rh + sh2 * dW_i))$
    $\overline{sum1}_i \leftarrow \overline{mult2}_i * S_i$
    $\overline{mult1}_i \leftarrow \overline{sum1}_i$
    $\overline{dW}_i \leftarrow sh2 * \overline{mult1}_i$
    $\overline{rh} \leftarrow \overline{rh} + \overline{sum1}_i$
    $\overline{sh2} \leftarrow \overline{sh2} + \overline{mult1}_i * dW_i$
end for
```

Considering how the sensitivities are computed, they will account for the link between the variables in the formulation of the error. It is also intuitive that the variables used early in the forward calculation usually have relatively higher sensitivities than the ones used only towards the end of the sample computation, but the use of normally distributed increments in our path calculation complicates the analysis. In this thesis, AD only serves as a tool to compute the sensitivities numerically.

2.3 Estimates of the overall variance

In this section we carry on the analysis of the overall error in order to find an estimate on the variance of this error that will depend explicitly on the bit-widths d_i of the variables x_i.

Note that in our application some inputs are random normally distributed variables, therefore the sensitivities and the rounding errors are also random. It is sensible to assume that the sensitivities $\bar{x}_i$ and the rounding errors δx_i are independent for all $i = 1, \ldots, m$, and we will make this assumption throughout the thesis. It always holds that $\mathbb{V}[\bar{x}_i \delta x_i] \leq \mathbb{E}[\bar{x}_i^2 \delta x_i^2]$ so using the upper bound on δx_i from Section 1.2 we obtain that

$$\mathbb{V}[\bar{x}_i \delta x_i] \leq \mathbb{E}[\bar{x}_i^2] 4^{e_i - d_i - 1}. \tag{2.5}$$

This gives an upper bound on the first sum in (2.4). Then if we assume that the individual errors $\bar{x}_i \delta x_i$ are independent we get an *optimistic* upper bound on the path error variance (2.4) :

$$\mathbb{V}[P - \tilde{P}] \leq \sum_{i=1}^{m} \mathbb{E}[\bar{x}_i^2] 4^{e_i - d_i - 1} \triangleq V_{indep}(d_1, \ldots, d_m) \tag{2.6}$$

On the other hand, if we assume perfect correlation between errors, ie. $Cov\left(\bar{x}_i \delta x_i, \bar{x}_j \delta x_j\right) = \sqrt{\mathbb{V}[\bar{x}_i \delta x_i] \mathbb{V}[\bar{x}_j \delta x_j]}$ then we obtain a *pessimistic* upper bound :

$$\mathbb{V}[P - \tilde{P}] \leq \left(\sum_{i=1}^{m} \sqrt{\mathbb{E}[\bar{x}_i^2]} \quad 2^{e_i - d_i - 1} \right)^2 \triangleq V_{corr}(d_1, \ldots, d_m). \tag{2.7}$$

These expressions do not contain the rounding errors and depend explicitly on the bit-widths d_i of the variables x_i. The exponents e_i will be fixed before the bit-width optimisation by computing 10^6 paths to determine the range of each variable.

2.4 Numerical validation of the variance model

In Section 2.3 we made several assumptions on the rounding errors δx_i that we check here. First of all we found numerically that the correlation matrix $Corr(\bar{x}_i \delta x_i, \bar{x}_j \delta x_j)$

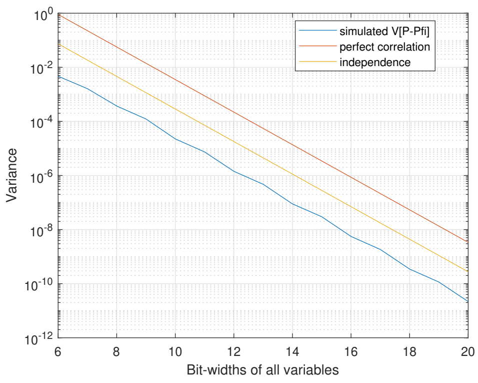

Figure 2.1: Simulated variance of the error vs variance estimates that assume independence or perfect correlation of errors, for $N = 4$ time steps.

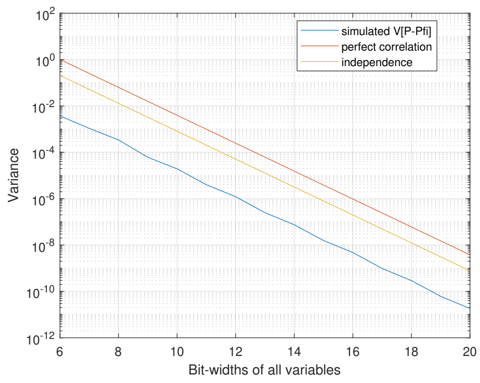

Figure 2.2: Simulated variance of the error vs variance estimates that assume independence or perfect correlation of errors, for $N = 16$ time steps.

does not have any coefficients close to 1 ; in fact most were smaller than 0.3. This implies that the optimistic bound (2.6) is closer to the overall variance than the pessimistic bound (2.7). To check this we also computed the variance $\mathbb{V}[P - \tilde{P}]$ and compared it to the variance estimates V_{indep} and V_{corr} from Section 2.3 where all variables had the same bit-width. We used 10^5 paths of $P - \tilde{P}$ to compute the variance. The results are presented in Figures 2.1 and 2.2 and confirm that V_{indep} is a better approximate of $\mathbb{V}[P - \tilde{P}]$ than V_{corr} since the blue curve is closer to the yellow bound than to the red one.

In the experiment from Figures 2.1 and 2.2 the exponents e_i were fixed by running 10^5 paths and looking at the maximal value attained by each variable. Note that the range of dW we obtained was $e_W = 3$ but more extreme values of the normal increment would have occured if we ran more paths. Additionally, the probability of the random normal increment being larger that 2^3 is approximately 1.3×10^{-15} so the increment is likely to be out of the range $[-2^{e_W}, 2^{e_W}]$ when 10^{15} paths are generated on a level, which in practice rarely happens.

Chapter 3

Bit-width optimisation

In this chapter we begin by introducing the cost model allowing to write the sample costs $C_l, \tilde{C}_l$ as functions of the level l and the bit-widths d_i. Then we focus on the single level case and introduce several ways of optimising the bit-widths. We compare them numerically and argue that the method based on a Lagrange multiplier is the most general and produces results that are very close to integer programming methods, therefore we chose this method for the rest of the thesis.

Then we treat the global optimisation problem in which the number of samples are also determined in addition to optimising the bit-widths. We show that the levels can be treated independently so that first the bit-widths of each level are optimised, then the number of samples required per level are calculated analytically.

3.1 Computational cost model and global problem formulation

We want to compute the estimated expected payoff $\mathbb{E}[P_L]$ by splitting the computational work between low precision calculations performed entirely on the FPGA and high precision ones which are done on the CPU. With the notation defined in Section 1.4.3 this corresponds to optimising the total cost of the nested framework under the constraint that the overall desired variance $\varepsilon^2/2$ is achieved, that is to say

$$\begin{aligned} \min_{d_{i,l},\tilde{N}_l,N_l} \quad & \sum_{l=0}^{L} \tilde{N}_l \times \tilde{C}_l(d_{1,l},\ldots,d_{m_l,l}) + N_l \times C_l(d_{1,l},\ldots,d_{m_l,l}) \\ \text{s.t.} \quad & \tilde{N}_l^{-1}\tilde{V}_l + N_l^{-1}V_l(d_{1,l},\ldots,d_{m_l,l}) \le \varepsilon^2/2. \end{aligned} \tag{3.1}$$

The decision variables are the bit-widths $d_{i,l}$ of the fixed-point variables and the number of samples $\tilde{N}_l$ and N_l. We make the assumption that the variance $\tilde{V}_l$ of the FPGA sample is approximately equal to $\mathbb{V}[\Delta P_l]$. Therefore it will be possible to compute it on the CPU at the same stage as when the exponents $e_{i,l}$ are calculated, namely before optimising the bit-widths. The variance V_l of the correction term depends on the precision used in the FPGA calculation and, as justified in Chapter 2, it is equal to

$$V_l = V_{indep}(d_{1,l}, \ldots, d_{m_l,l}) = \sum_i \mathbb{E}[\bar{x}_{i,l}^2] 4^{e_{i,l}-d_{i,l}-1}. \tag{3.2}$$

Now we define the sample costs $C_l, \tilde{C}_l$ along with its justification. On the CPU most of the computational cost comes from generating a full precision random normal increment dW_i at each time step ; the other operations performed to compute ΔP_l have a negligible cost. Then to obtain a sample of $\Delta P_l - \widetilde{\Delta P_l}$ one generates a full precision and a fixed-point normal increment used to compute ΔP_l and $\widetilde{\Delta P_l}$ respectively. This procedure is detailed in Chapter 5 but here the key point is that the fixed-point increment used in the correction term is computed with negligible cost compared to the cost of random number generation (RNG) on the CPU. Hence noting C_{RNG} the computational cost of one full precision normal number we get $C_l \approx 2^l C_{RNG} + \tilde{C}_l$.

On the other hand, when computing samples of $\widetilde{\Delta P_l}$ alone on the FPGA we make the assumption that the random normals are generated with a relatively low cost and that the cost $\tilde{C}_l$ of the sample computed only on the FPGA is mostly due to the other operations performed to compute $\widetilde{\Delta P_l}$. Combining it with the previous paragraph it means that we assume

$$Cost(\text{RNG on CPU}) \geq Cost(\text{computing } \widetilde{\Delta P_l} \text{ on FPGA}) \gg Cost(\text{RNG on FPGA}). \tag{3.3}$$

The cost comparison of RNG on the FPGA and the CPU is provided in Chapter 5 and the first inequality holds because if the nested framework is more expensive than the standard MLMC, we switch to standard MLMC (see Chapter 4).

Neglecting the cost of RNG on the FPGA, the cost of computing $\widetilde{\Delta P_l}$ is equals the sum of the cost of the elementary operations involved. In particular for a classical European option based on a Geometric Brownian Motion there are only additions and multiplications between two variables. Noting $\mathcal{S}$ the index set of all couples of variables that are involved in an addition and $\mathcal{M}$ the analogue for multiplications, then using Equation (1.3) the cost of computing a path of $\tilde{P}$ is

$$\tilde{C}_l(d_1, \ldots, d_{m_l}) = \sum_{(i,j)\in\mathcal{M}} d_{i,l} d_{j,l} + \sum_{(i,j)\in\mathcal{S}} (d_{i,l} + d_{j,l}). \tag{3.4}$$

However in practice we will instead use the cost

$$\tilde{C}_l(d_1, \dots, d_{m_l}) = \frac{1}{2} \sum_{i=1}^{m_l} M_{i,l} d_{i,l}^2 \tag{3.5}$$

where $M_{i,l}$ denotes the number of elementary operations in which the variable i is involved. This is an over-estimation since, assuming all bit-widths are larger than 2, $d_a + d_b \leq \frac{1}{2}(d_a^2 + d_b^2)$ and $d_a \times d_b \leq \frac{1}{2}(d_a^2 + d_b^2)$.

We have formulated the global optimisation problem (3.1) that we are looking to solve to determine the global parameters in the MLMC algorithm. In the two following sections we provide a method to solve this problem.

3.2 Bit-width optimisation methods

In this section we separate the bit-width optimisation problem from the global problem (3.1). We focus on a single level and note $\tilde{P}$ the path computed on the FPGA and P the path computed on the CPU. The bit-widths are defined such that the cost $\tilde{C}$ of computing $\tilde{P}$, given by (3.4) or (3.5), is minimised and the variance (2.6) is smaller than a tolerance $\tilde{\varepsilon}^2$, ie. a problem of the following form is solved :

$$\begin{aligned} min_{d_i,\dots,d_m} \quad & \tilde{C}(d_1, \dots, d_m) \\ s.t. \quad & V_{indep}(d_1, \dots, d_m) \leq \tilde{\varepsilon}^2. \end{aligned} \tag{3.6}$$

In this section we present several methods for solving this problem, highlighting their similarities, then we compare them numerically in Section 3.2.4. We also compare them to the uniform bit-width heuristic proposed by [6] and summarised in Appendix D.

3.2.1 Analogy with the 0-1 knapsack problem and greedy algorithm

The 0-1 knapsack problem is a classical problem (see [24], Lecture 1) in integer programming that is defined as follows : say we have n objects that we would like to pack into a suitcase, each having a value v_i and weight w_i. We would like to maximise the total value of the objects we take while keeping the total weight under a fixed threshold. Therefore the problem consists of selecting the right objects by setting some binary decision variables x_i to 0 or 1.

This section builds the analogy between problem (3.6) and a slightly modified version of the 0-1 knapsack problem where the total weight is minimised subject to a lower bound on the total value. This allows to to solve (3.6) using a well-known greedy algorithm that finds a near-optimal solution to the classical 0-1 knapsack problem.

The greedy algorithm applies only to the case where the cost and constraint are decomposable, meaning that there are no mixed terms like $d_a \times d_b$. Hence we focus on the problem (3.6) with the expression (3.5) of the cost and the variance expression (2.6). Considering that all variables in the FPGA implementation have at least 4 bits we decompose the variance as

$$V_{indep}(d_1, \dots, d_m) = \sum_{i=1}^{m} 4^{e_i - 1}\mathbb{E}[\bar{x}_i^2]\left(4^{-4} - \sum_{j=5}^{d_i}(4^{-j} - 4^{-j+1})\right) \tag{3.7}$$

and the cost as

$$\tilde{C}(d_1, \dots, d_m) = \sum_{i=1}^{m} \frac{1}{2}M_i\left(16 + \sum_{j=5}^{d_i}(j^2 - (j-1)^2)\right). \tag{3.8}$$

Now define

$$w_{i,j} = \frac{1}{2}M_i(2j-1), \qquad F = 8\sum_{i=1}^{m} M_i, \tag{3.9}$$

$$v_{i,j} = 4^{e_i-1}\mathbb{E}[\bar{x}_i^2](4^{-j} - 4^{-j+1}), \qquad G = \sum_{i=1}^{m} 4^{e_i-5}\mathbb{E}[\bar{x}_i^2]. \tag{3.10}$$

Then with this notation the coefficients $w_{i,j}$ correspond to the increase in cost due to adding an extra bit to a variable that already has $j-1$ bits, and $v_{i,j}$ is the corresponding reduction in variance. Allowing the bit-widths to go up to $d_\infty = 32$ and defining binary decision variables $x_{i,j} \in \{0, 1\}$, the problem (3.6) is reformulated as :

$$min_{x_{i,j}\in\{0,1\}} \quad \sum_{i=1}^{m}\sum_{j=5}^{d_\infty} x_{i,j}w_{i,j} \tag{3.11}$$

$$s.t. \quad \sum_{i=1}^{m}\sum_{j=5}^{d_\infty} x_{i,j}v_{i,j} \geq G - \tilde{\varepsilon}^2. \tag{3.12}$$

This problem can be solved as an Integer Linear Program, for example with the Matlab solver *intlinprog*. Another method is to use a greedy algorithm which gives a near optimal solution that is close to the real solution when the number of variables is large. In practice for a given level we have at least $7 \times (32 - 4) = 196$ variables so we consider that the greedy algorithm gives a good solution. The idea of the greedy algorithm is to define ratios

$$r_{i,j} = \frac{v_{i,j+1}}{w_{i,j+1}} \tag{3.13}$$

and sort them in decreasing order then set $x_{i,j} = 1$ for the variables with highest ratio until the variance bound is respected. The constraints that $x_{i,j+1} \leq x_{i,j}$ (which ensure that $d_i = j$ if and only if $x_{i,k} = 1$ for $k \leq j$ and $x_{i,j} = 0$ for $k > j$) is automatically satisfied since $r_{i,j+1} < r_{i,j}$.

3.2.2 Lagrange multiplier and priority ratio

An alternative method is to first consider the bit-widths as real numbers and optimise them with a Lagrange multiplier approach. The variance constraint is fixed as the equality $V_{indep} = \tilde{\varepsilon}^2$ and introducing a Lagrange multiplier $\lambda \in \mathbb{R}$ the Lagrangian is written as

$$\mathcal{L}(d_1, \ldots, d_m) = \tilde{C}(d_1, \ldots, d_m) + \lambda(V_{indep}(d_1, \ldots, d_m) - \tilde{\varepsilon}^2). \tag{3.14}$$

Then the Lagrangian is minimised by solving the (nonlinear) system $\nabla\mathcal{L} = 0$, which gives the set of equations

$$M_i d_i - 2\lambda \log 2\, \mathbb{E}[\bar{x}_i^2]\, 4^{e_i - d_i - 1} = 0, \qquad \text{for} \quad i = 1, \ldots, m. \tag{3.15}$$

We solved this system in Matlab using the solver *vpasolve* from the Symbolic Math Toolbox. This is a numerical solver using a combination of root-finding methods like the Newton method, interval arithmetic and bisection, as well as symbolic manipulations and simplifications.

Since the solution is a vector of real numbers it needs to be rounded to define the bit-widths of the variables for the FPGA path simulation. A straightforward idea is to round the solution to the nearest integer but this might not lead to the optimal solution. A better method was suggested in a preprint by M. Giles' collaborators. The idea is to round down the solution (to d^*) and define for each variable i the ratio

$$\bar{r}_i = \frac{|V_{indep}(d_1^*, \ldots, d_i^* + 1, \ldots) - V_{indep}(d_1^*, \ldots, d_i^*, \ldots)|}{\tilde{C}(d_1^*, \ldots, d_i^* + 1, \ldots) - \tilde{C}(d_1^*, \ldots, d_i^*, \ldots)}. \tag{3.16}$$

Variables with a high marginal variance over marginal cost ratio offer a good improvement in the accuracy (ie. variance) for a relatively small increase of the computational cost, so an extra bit is added to their word length in the order of priority defined by the ratios until the desired variance is achieved.

Therefore the knapsack method and the Lagrange multiplier with the priority criterion (3.16) approach are similar. Indeed,

$$v_{i,j+1} = V_{indep}(\ldots, j, \ldots) - V_{indep}(\ldots, j+1, \ldots) \approx -\partial V_{indep}/\partial d_i(\ldots, j, \ldots) \tag{3.17}$$

and a similar relation holds for the cost vector $w_{i,j}$, which means that

$$r_{i,j} \approx -\frac{\frac{\partial V_{indep}}{\partial d_i}}{\frac{\partial \tilde{C}}{\partial d_i}}(d_i = j). \tag{3.18}$$

Therefore when evaluated at the optimal bit-widths the ratio used in the greedy approach is equal to the Lagrange multiplier λ. This indicates that there is an equivalence between the Lagrange multiplier approach and the greedy algorithm : in the knapsack approach we use the ratio $r_{i,j}$ to add bits to variables until, in practice, this ratio drops to a value that satisfies the variance constraint and does not depend on the variable i. This analogy justifies the validity of using the Lagrange multiplier and (3.16) to obtain the right bit-widths.

3.2.3 Integer quadratic programming for the cost (3.4)

If we want to use the cost expression (3.4) with an integer programming approach, the problem formulated in Equation (3.11) must be modified because of the mixed terms $d_i d_j$. To fix the notation, the decision variables are concatenated into a vector $x = (x_{1,1}, x_{1,2}, \ldots)$. Since

$$d_i = 4 + \sum_{j=5}^{d_\infty} x_{i,j}, \quad \text{for } i = 1, \ldots, m \tag{3.19}$$

the terms $d_i d_j$ can be written as a quadratic term plus a linear term :

$$d_i d_j = 16 + \sum_{k=5}^{d_\infty} \sum_{l=5}^{d_\infty} x_{i,k} x_{j,l} + 4 \left(\sum_{k=5}^{d_\infty} x_{i,k} + \sum_{l=5}^{d_\infty} x_{j,l} \right). \tag{3.20}$$

Therefore we define a matrix Q by blocks, where the block (i,j) is a $(d_\infty - 4) \times (d_\infty - 4)$ all ones square matrix and the other blocks are set to 0. The terms $d_i + d_j$ are also linear, so the overall cost is of the form $\frac{1}{2} x^T Q x + w^T x$.

Since the problem is quadratic, we cannot directly use the Matlab solver *intlinprog* which can only solve mixed integer linear programs, instead we use a cutting planes method as presented in [37]. The idea is to approximate the quadratic term $x^T Q x$ by a slack variable z such that the problem can be rewritten as

$$\begin{aligned} min_x \quad & z + w^T x \\ s.t. \quad & Ax \leq b \\ & x^T Q x - z \leq 0 \\ & z \geq 0. \end{aligned} \tag{3.21}$$

Then we iteratively solve a MILP and add new linear constraints that approximate the quadratic constraint locally : suppose at iteration k our guess of the solution is $x^{(k)} \in \{0,1\}^{m(d_\infty - 4)}$ then linearising the quadratic term about $x^{(k)} + \delta$ gives

$$x^T Q x - z = x^{(k)T} Q x^{(k)} + 2x^{(k)T} Q \delta - z + O(|\delta|^2) \tag{3.22}$$

Then replacing δ by $x - x^{(k)}$ gives

$$x^T Q x - z = -x^{(k)T} Q x_k + 2x^{(k)T} Q x - z + O(|x - x_k|^2). \tag{3.23}$$

Hence we update the matrix of linear constraints A and the vector b to include the constraint $2x_k^T Q x - z \leq x^{(k)T} Q x_k$ and repeat the process. The algorithm stops when the relative error in the approximation of the quadratic constraint is lower than a predefined tolerance. In our numerical tests we used the value 10^{-8} for this tolerance.

3.2.4 Numerical comparison of the bit-width optimisation methods

In this section we compare the different optimisation methods suggested in the previous sections.

We consider that the variables $mult1_i, mult2_i, sum1_i, S_i, dW_i$ used to compute $\tilde{P}$ have the same bit-width at every time step i. The benefit is that it limits significantly the number of unknowns in the system from the Lagrange multiplier and the number of decision variables in the integer programs presented above. The expected squared sensitivities of each variable over time steps shown in Figure C.1 imply that this assumption is particularly reasonable for $mult1_i, sum1_i$ and dW_i as there is no clear trend but for S_i and $mult2$ the sensitivities as we go backwards in time. Maybe an explanation is that S and $mult2$ $(= S_{i+1} - S_i)$ have a drift term due to $rh \times S_i$ while the other variables behave like random walks (see definition in Algorithm 1). It would be useless to adapt only the bit-widths of S and $mult2$ over time if the other intermediate variables are not more accurate. Therefore to adapt more tightly to the sensitivity of S over time, it might be relevant to adapt the word lengths of all variables at each time step in future experiments. However it may increase the overall cost and for robustness the sensitivities with respect to $mult1_i, sum1_i$ and dW_i would need to be bounded or determined with more paths.

Then we optimised the bit-widths with the above methods and looked at the resulting variance $\mathbb{V}[P - \tilde{P}]$ shown in Figure 3.1. There are no significant differences in the variance nor the computational cost (see Figures 3.2 and 3.3) or in the bit-widths (Figure C.2) obtained with the suggested optimisation methods. This shows

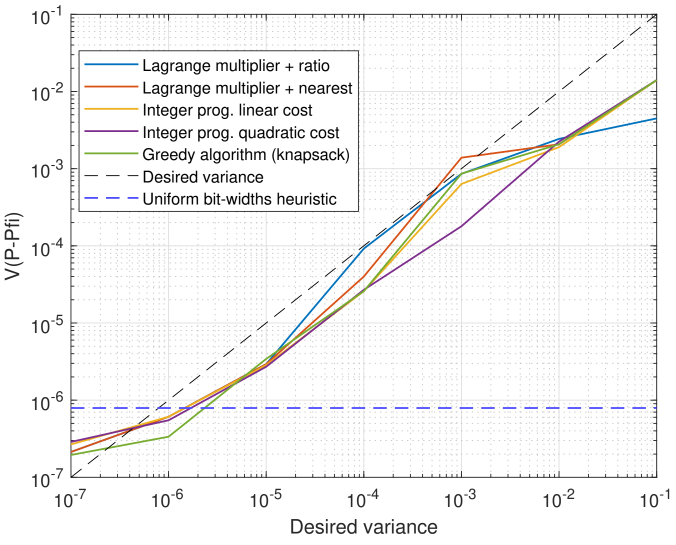

Figure 3.1: Variance obtained with different bit width optimisation methods vs. desired variance and variance obtaind with uniform bit-widths set using [6].

that the Lagrangian multiplier approach with the ratio (3.16) gives results that are very close to the integer programming approaches. The advantage of the Lagrange multiplier is that it is a very general technique, therefore we chose to use this method in the next experiments.

Another argument that would lead to choosing one optimisation method over another method would be the complexity of the optimisation method itself and its guarantee of finding the optimum. For the greedy algorithm, there are at most 9x(32-4)=252 binary decision variables in the integer programming formulation. This means the greedy algorithm requires to compute the 252 $v_{i,j}$, $w_{i,j}$ and $r_{i,j}$ coefficients, then sort the $r_{i,j}$ and find the solution by dichotomy. The IP and MIQP implementations are more expensive because they are usually solved with successive linear relaxations and a branch-and-bound framework [24]. In Matlab the function *intlinprog* also appeals to a set of heuristics and cuts when they are relevant. Although in happy cases the solver could take only a few iterations to find the solution the overall cost is harder to estimate than for the greedy algorithm. Furthermore in our test, the constraint was violated at the desired variance value $\tilde{\varepsilon}^2 = 10^{-3}$, we believe is due to a constraint tolerance that was too loose. Finally, the Lagrange multiplier approach boils down to solving a non-linear system. With the assumption presented above, the system has at most 9 unknowns (7 at level 0 and 9 at other levels) so it is easily solved in Matlab with *vpasolve*. This method is also more realistic in case we would like to vary the number of bits for variables S and $mult2$ in future work.

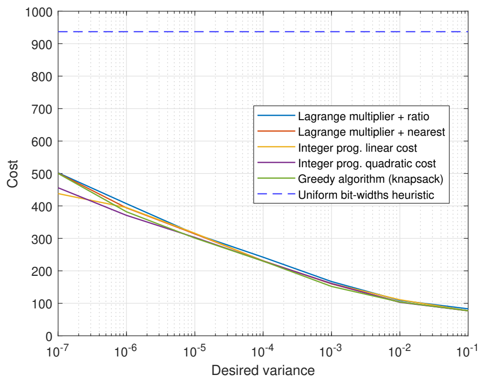

Figure 3.2: Cost vs desired variance for the cost (3.4).

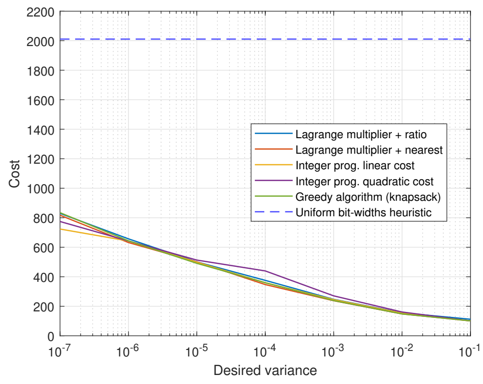

Figure 3.3: Cost vs desired variance for the cost (3.5).

On Figures 3.1 to 3.3 and C.2 we also plotted the corresponding quantities obtained with the uniform bit-width heuristic from [6] (blue dashed curve). This heuristic method is summarised in Appendix D. Clearly the uniform bit-width heuristic is too conservative as its cost per time step is above the cost obtained with any of the optimisation methods. To define the bit-width, the heuristic does not take into account the desired overall variance but instead attempts to make the variance $\tilde{V}$ as small as the CPU variance $\mathbb{V}[P]$, which partly explains the large cost. Indeed the uniform bit-width heuristic gives a variance of about 10^{-6} while in our method for $N = 16$ time steps the optimal variance of $P - \tilde{P}$ is about 10^{-4}.

Finally we compare the cost formulations (3.4) and (3.5) : even though the MIQP optimisation used the more realistic cost (3.4) the cost saving is negligible compared to the other methods based on (3.5). Also comparing Figures 3.2 and 3.3 we see that the resulting cost is of comparable size independently of the cost expression that we choose. This is because the multiplications, in particular $S_i \times mult2_i$, are the most expensive operations. Therefore in the remainder of the thesis we consider that the FPGA cost is defined as in (3.5).

3.3 Global optimisation for the nested Monte Carlo estimator

In this section we detail how the bit-widths are optimised along with the number of samples (denoted $N, \tilde{N}$ in the single level case) so that all necessary parameters for path simulations are determined. Starting with the single level case, the total cost in the nested Monte Carlo framework is

$$\varepsilon^{-2}\left(\sqrt{VC}+\sqrt{\tilde{V}\tilde{C}}\right)^2. \tag{3.24}$$

Note that this formulation of the cost allows to avoid variables $N, \tilde{N}$ in the optimisation problem.

One can minimise the total computational cost (3.24) as a function of the bit-widths using the Lagrange multiplier approach from Section 3.2. To round the bit-widths we used (3.16) with the desired variance $\tilde{\varepsilon}^2$ equal to the variance V of the correction that is obtained from solving $grad(\mathcal{L}) = 0$. Then the number of samples is $N = 2\varepsilon^{-2}\sqrt{V/C}\left(\sqrt{VC}+\sqrt{\tilde{V}\tilde{C}}\right)$ and $\tilde{N} = 2\varepsilon^{-2}\sqrt{\tilde{V}/\tilde{C}}\left(\sqrt{VC}+\sqrt{\tilde{V}\tilde{C}}\right)$, where ε is the desired RMS error of the MLMC estimation.

The generalisation to the multilevel case follows naturally. The overall cost of the nested framework is

$$2\varepsilon^{-2}\left(\sum_{l=0}^{L}\sqrt{\tilde{V}_l\tilde{C}_l}+\sqrt{V_lC_l}\right) \tag{3.25}$$

so for level l the following objective is minimised to determine the bit-widths :

$$obj_l = \sqrt{\tilde{V}_l\tilde{C}_l}+\sqrt{V_lC_l}. \tag{3.26}$$

Then the number of samples $\tilde{N}_l, N_l$ are calculated analytically and rounded up.

To ensure the existence of optimal bit-widths at each level we could show that obj_l is convex, at least locally. In practice we simply provided $d_i = 4$ as an initial guess to *vpasolve*, and managed to find the initial solution up to level $l = 15$.

Chapter 4

Nested Multilevel Monte Carlo and rounding error analysis

In this chapter we include the global optimisation of the bit-widths into the nested MLMC algorithm and show the resulting computational savings. Further we discuss the necessity of adapting the bit-widths to prevent rounding errors from accumulating and restraining the accuracy of the nested framework as the level increases [3, 39].

4.1 Mixed precision MLMC algorithm

In this section we detail how the global optimisation is included in the MLMC framework. We consider as in Algorithm 4 that the user specifies a minimal number of levels $L_{min} \geq 2$. At initialisation the algorithm computes the sensitivity terms $\mathbb{E}[\bar{x}_{i,l}^2]$, the exponents $e_{i,l}$ and estimates the variances $\tilde{V}_l$ (which are approximately equal to $\mathbb{V}[\Delta P_l]$) using N_{sens} paths generated in full precision on the CPU. In our numerical experiments we take $N_{sens} = 10^3$. Then the bit-widths are optimised as described in Chapter 3 for all levels $l = 0, \ldots, L_{min}$, which allows to determine the number of samples that are required on each level $N_l, \tilde{N}_l$ to achieve the desired variance. The samples of $\widetilde{\Delta P}_l$ are computed on the FPGA while the samples of $\Delta P_l - \widetilde{\Delta P}_l$ require a CPU and an FPGA calculation that is based on the same random normal increments. As detailed in Chapter 5, the FPGA path generation will actually use an approximate random normal variable, but in the experiments from this chapter the approximate random numbers are simply obtained by rounding (to nearest) the full precision ones. After computing paths, their mean and variance are calculated in full precision on the CPU to avoid losing accuracy when all sample outputs are summed together.

As in the classical MLMC framework, new levels are added when a weak convergence condition fails to be satisfied. Using that $\mathbb{E}[\Delta P_L] = \mathbb{E}[\widetilde{\Delta P}_L] + \mathbb{E}[\Delta P_L - \widetilde{\Delta P}_L]$, the same condition (1.23) as in the classical MLMC is used to determine when a new level is needed. We also introduce a condition to decide when adding a nested level becomes computationally more expensive than adding a standard Monte Carlo level. With the notation from before and noting $C_l^{CPU} = 2^l C_{RNG}$ the cost of generating a sample of ΔP_l on the CPU and $V_l^{CPU} = \mathbb{V}[\Delta P_l]$, adding a nested level is preferred unless

$$\sqrt{V_l C_l} + \sqrt{\tilde{V}_l \tilde{C}_l} > \sqrt{C_l^{CPU} V_l^{CPU}}. \tag{4.1}$$

If this condition is satisfied for one level, we consider that we switch to standard levels without coming back to nested ones. Then noting L_{tot} the total number of levels and L_{nes} the number of nested levels, the overall cost and variance are

$$\begin{aligned} Cost &= \sum_{l=0}^{L_{nes}} \left(\tilde{N}_l \tilde{C}_l + N_l C_l \right) + \sum_{l=L_{nes}+1}^{L_{tot}} N_l C_l^{CPU} \\ Variance &= \sum_{l=0}^{L_{nes}} \left(\tilde{N}_l^{-1} \tilde{V}_l + N_l^{-1} V_l \right) + \sum_{l=L_{nes}+1}^{L_{tot}} N_l^{-1} V_l^{CPU}. \end{aligned} \tag{4.2}$$

Since the cost and variance parameters are all known using a Lagrange multiplier approach to minimise the overall cost subject to $Variance = \varepsilon^2/2$ gives the following value of the Lagrange multiplier :

$$\lambda = \sum_{l=0}^{L_{nes}} \left(\sqrt{\tilde{V}_l \tilde{C}_l} \sqrt{V_l C_l} + \sqrt{V_l C_l} \right) + \sum_{l=L_{nes}+1}^{L_{tot}} \sqrt{V_l^{CPU} C_l^{CPU}} \tag{4.3}$$

and the number of samples are simply $\tilde{N}_l = \lambda \sqrt{\tilde{V}_l / \tilde{C}_l}$ and $N_l = \lambda \sqrt{V_l / C_l}$ for $l \leq L_{nes}$, and $N_l = \lambda \sqrt{V_l^{CPU} / C_l^{CPU}}$ for $l > L_{nes}$.

The nested MLMC algorithm is summarised in Algorithm 3.

Algorithm 3 MLMC with optimised precision

```
1: Inputs: the desired RMS error, the initial number of levels ($\geq 2$), the number
   of samples $N_{sens}$ generated to initialise each level.
2: generate $N_{sens}$ full precision paths and use them to determine the exponents $e_{i,l}$
   and variances $\tilde{V}_l$ and initialise the path sums ;
3: optimise the bit-widths for all nested levels ;
4: determine the number of samples needed on each level ;
5: compute extra samples and update the cost, variance and path sums estimates,
   and add new levels if the weak convergence condition fails, until no extra samples
   are required anymore ;
6: Output: the expected payoff.
```

In practice, in financial applications the drift and volatility of the asset evolve slowly so the bit-width optimisation stage would probably not be required every time the algorithm is used to price an option. For example, the parameters could be tuned and the optimal bit-widths determined to configure the FPGA(s) every month. Therefore the bit-width optimisation is performed off-line and the its cost and time is not included in the measure of the online performance of the algorithm.

4.2 Numerical experiments : global optimisation and nested framework

In this section we discuss numerical results obtained with the global optimisation method and our proposed nested MLMC framework. In the following subsections, we will first demonstrate the cost savings achieved by the nested framework then discuss the behaviour of the bit-widths and the variance. The latter are tightly related as the bit-widths and the order of the operations determine the size of individual rounding errors, which in turn influences the variance.

4.2.1 Cost savings with the nested framework

To begin with, we comment on the cost per sample and the cost of the overall simulation compared to the standard MLMC framework. We call *level cost* (or "cost per level") the quantity

$$\sqrt{\tilde{V}_l \tilde{C}_l} + \sqrt{V_l C_l}, \tag{4.4}$$

and the total cost of the nested framework is the square of the sum of the level costs. We multiplied the overall cost by the overall variance so that the figure is not dependent on the desired variance $\varepsilon^2/2$ which does not change the interpretation of the curves since this factor would only shift the curves upwards on the logarithmic plots.

In most of our tests we used the value $C_{RNG} = 10^4$. The corresponding cost per sample $(\tilde{C}_l, C_l)$, cost per level and estimated total cost of the nested MLMC framework are shown in Figures 4.1 to 4.4 respectively. For this value, in the nested framework the sample cost is considerably reduced on the first levels which leads to the corresponding cost per level being smaller than that of the standard MLMC (see Figures 4.1 and 4.2). At level 0, the sample computed on the FPGA is 28 times cheaper than the one computed on a CPU and the FPGA 4.1. Then the level cost for the nested framework slowly gets closer to the cost per level of the standard MLMC, but for $C_{RNG} = 10^4$ the two costs still did not intersect at level $L = 15$.

This result is very different when we take a smaller value of C_{RNG}, for example in Figure 4.5 we took $C_{RNG} = 10^3$ and obtained that the level costs intersect at level $l = 4$ (see Figure 4.5). Therefore the next levels should be standard MC levels. Despite this, on the Figure 4.6 we see that the nested framework is more expensive overall than standard MLMC only starting from $L_{tot} = 14$. This implies that for $L_{tot} = 5, \ldots, 14$ only a few paths are computed on levels $l > 4$ so most of the cost is still on the first levels where the nested calculations are considerably cheaper.

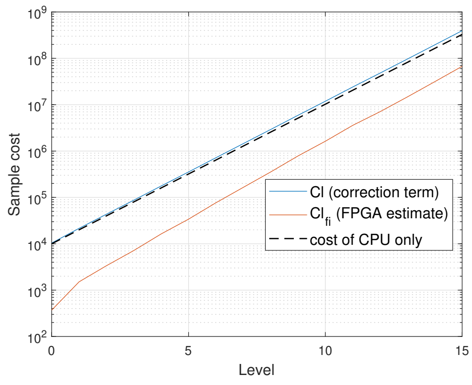

Figure 4.1: Cost per sample C_l and $\tilde{C}_l$ for $C_{RNG} = 10^4$.

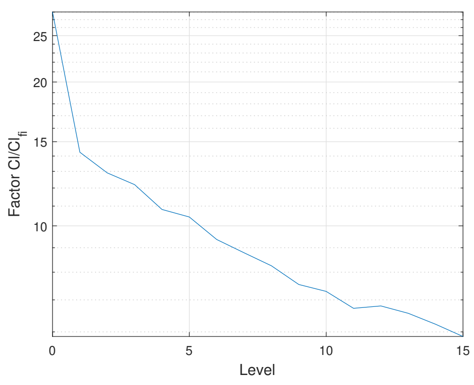

Figure 4.2: Factor $C_l/\tilde{C}_l$ for $C_{RNG} = 10^4$.

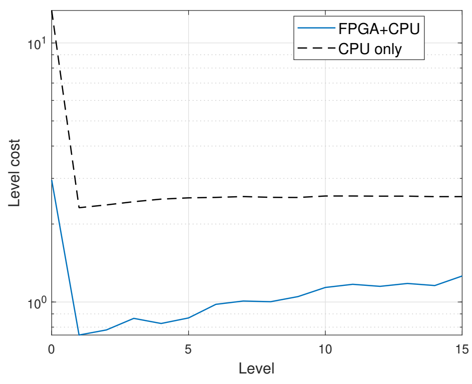

Figure 4.3: Level cost for $C_{RNG} = 10^4$. Comparison with classical MLMC.

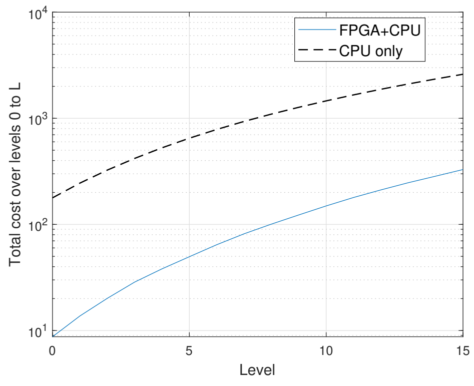

Figure 4.4: Total cost $(\sum_{l=0}^{L} \sqrt{\tilde{V}_l \tilde{C}_l} + \sqrt{V_l C_l})^2$ for $C_{RNG} = 10^4$ (blue). Comparison with classical MLMC.

Another key observation from Figure 4.3 is that after the very few first levels, the level cost in the standard MLMC method does not vary significantly with level. This

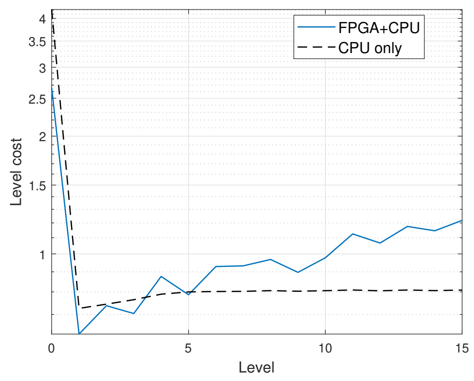

Figure 4.5: Level cost for $C_{RNG} = 10^3$. Comparison with classical MLMC.

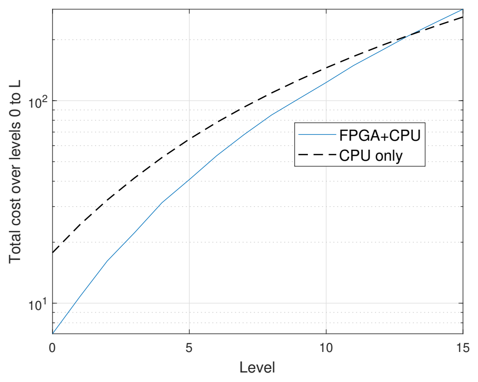

Figure 4.6: Total cost $(\sum_{l=0}^{L} \sqrt{\tilde{V}_l \tilde{C}_l} + \sqrt{V_l C_l})^2$ for $C_{RNG} = 10^3$ (blue). Comparison with classical MLMC.

is not surprising as in the Euler-Maruyama scheme, for the European vanilla option, the sample cost is $C_l^{CPU} \approx \mathcal{O}(2^l)$ and the variance is $V_l^{CPU} \approx \mathcal{O}(2^{-l})$, which leads to $V_l^{CPU} C_l^{CPU} = \mathcal{O}(1)$).

For the same option but using the Milstein scheme we would have $V_l^{CPU} \approx \mathcal{O}(2^{-2l})$ instead, so the squared level cost $V_l^{CPU} C_l^{CPU}$ would decrease exponentially with level. In that case using a nested framework would be even more efficient as it would tackle an even greater proportion of the total computational cost.

4.2.2 Bit-widths and variance over levels

Next we look at the optimal bit-widths obtained for different levels 4.7. We observe that the bit-widths of variables $S, Sc, mult2, mult2c$ increase monotonically over levels, with S and Sc having one additional bit per level and $mult2$ and $mult2c$ having one additional bit every two levels, while the over bit-widths also increase globally but have a less well defined trend. We believe this is because the other variables are influenced by the sample normal random variables that were used to compute the sensitivities, similarly to the remark on Figure C.1 made in Chapter 3. Using the definition of the variables in Algorithm 1 as well as the assumption $h \ll \sqrt{h}$, σ and $r, S_0, dW_i = \mathcal{O}(1)$, we obtain the scale of the variables :

$$sh2,\ mult1,\ sum1,\ mult2,\ mult2c \sim \sqrt{h} \tag{4.5}$$

$$rh \sim h \tag{4.6}$$

$$S,\ Sc \sim 1. \tag{4.7}$$

This explains perfectly the evolution of the exponents from Figure 4.10 : the exponents of variables $sh2$, $mult1$, $sum1$, $mult2$, $mult2c$ decrease by one every two levels, the exponent of rh decreases by one every level and the exponents of S and Sc remain constant. The same scaling argument does not help determine the trends of the sensitivities, which are presented in Figure 4.10 though. Therefore we defined and plotted the factor

$$\alpha_{i,l} = 4^{e_{i,l}}\mathbb{E}[\bar{x}_{i,l}^2] \tag{4.8}$$

in Figure 4.11, which shows that the trend of the exponent and the sensitivity partly compensate each other, in particular for variable rh for example. This particular case is sensible since rh is involved in an addition with $mult1$ which has a larger scale than rh, so to increase the precision of the result $sum1 = rh + mult1$, we usually need to increase the precision of $mult1$ as illustrated by the difference in the fraction lengths of rh and $mult1$ in Figure 4.8. For variables $S, Sc, mult2$ and $mult2c$ we notice that the factor α is approximately multiplied by 2 at every level. Therefore, using $d_{S,l+1} = d_{S,l} + 1$, the error coming from these variables, which is $\alpha \times 4^{-d_{S,l}-1}$, is divided by 2 at every level. In fact a similar argument can be used for the error coming from $mult2, mult2c$ and numerically we see in Figure 4.12 that the same is (almost) true for all variables. This is a very important observation because it shows that the portion of the overall error attributed to a certain variable is constant over levels. In other words, to leading order, all errors are roughly of the same size. This is intuitive because if an error was "disproportionately" small, there would be potential for cost savings with a small increase in the error.

Note that the order of the curves from down to up in Figure 4.7 is consistent with the order of the operations in the path calculation. Indeed for the variables of scale $\sqrt{h}$, the last variables need more precision to limit the rounding errors that could then propagate. The large difference between the bit-widths of variables S, Sc and the other variables might come from the difference in scale : since the other variables are decreasing in scale (unlike S and Sc), the fraction length of S, Sc needs to increase fast to capture the results from the previous operations with enough accuracy.

We have shown and explained the consistency and meaning of the numerical results on the evolution of the bit-widths and the differences between variables.

Finally, Figure 4.13 shows the variance V_l and the variance $\tilde{V}_l = \mathbb{V}[\widetilde{\Delta P_l}]$ obtained for $C_{RNG} = 10^4$. They have the same slope as the larger one is assumed approximately equal to its full precision analogue $\mathbb{V}[\Delta P_l]$ which is $= \mathcal{O}(2^{-l})$, and the slope of the latter was exhibited above. However this is not as natural as it seems : [3, 39] show that, if the bit-widths were the same for every level, starting a certain level $\tilde{V}_l$ would

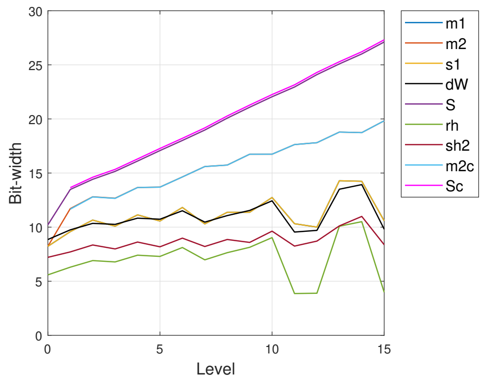

Figure 4.7: Optimal bit-widths obtained over levels 0 to 8 for $C_{RNG} = 10^4$.

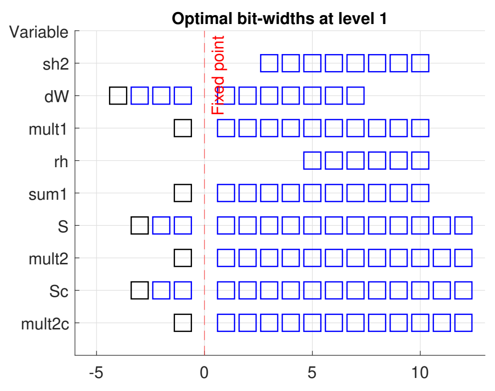

Figure 4.8: Optimal bit-widths at level 1 (rounded with Equation (3.16)) for $C_{RNG} = 10^4$, represented in the order in which they are used in the path generation in each time step.

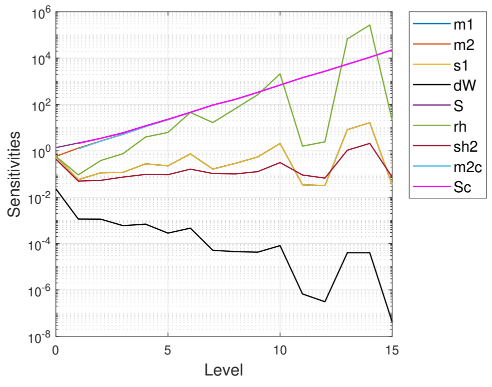

Figure 4.9: Sensitivities for $C_{RNG} = 10^4$ over levels.

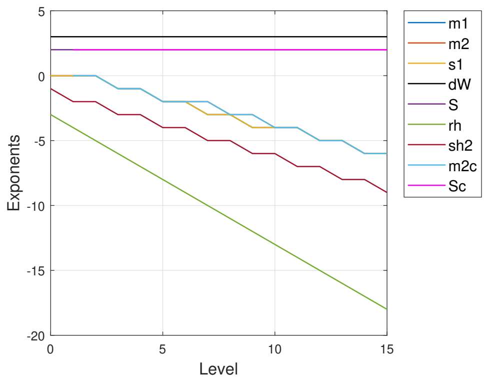

Figure 4.10: Exponents for $C_{RNG} = 10^4$ over levels.

not decrease with level anymore because the rounding errors would be too large and accumulate. In fact we think that the error caused by each variable would then have the same trend as its factor α (see Figure 4.11), so the error caused by S, Sc as well as (with a smaller magnitude) the error from $mult2, mult2c$ would increase with level, making the variance increasingly large. That is why increasing the bit-widths is necessary to keep decreasing $\tilde{V}_l$.

In the next section we discuss in more detail the results from [3, 39].

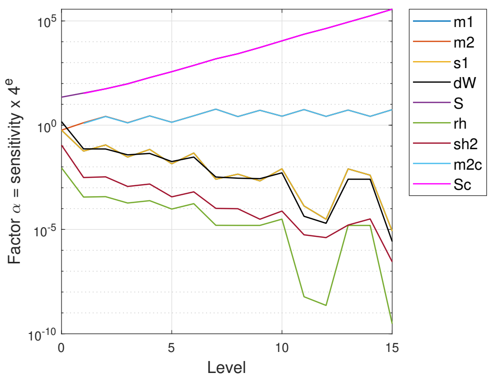

Figure 4.11: Factor α for $C_{RNG} = 10^4$ over levels.

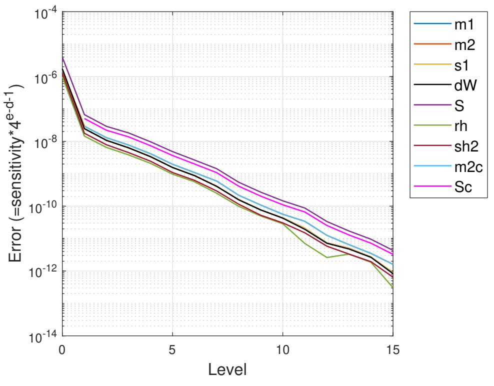

Figure 4.12: $\mathbb{E}[\bar{x}_{i,l}^2]4^{e_{i,l}-d_{i,l}-1}$ for each variable i for $C_{RNG} = 10^4$.

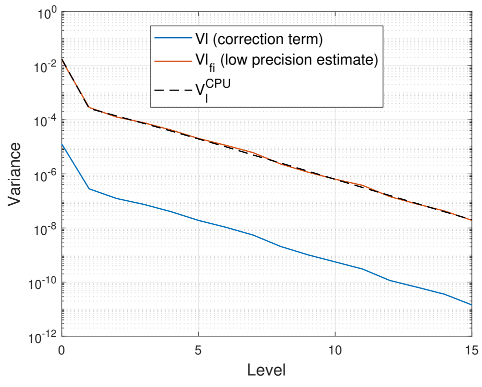

Figure 4.13: Variance V_l and $\tilde{V}_l$ over levels 0 to 8 for $C_{RNG} = 10^4$.

4.3 A leading order model of the error

As mentioned in the literature review (Section 1.5), a leading order model of the net error due to finite precision computations was previously proposed in [3] and further developed in [39]. This model allowed to show that the error $\mathbb{E}[|\Delta S_l - \widetilde{\Delta S_l}|^2]$ behaved like $\mathcal{O}(h^{-1}\rho^2)$ as h tends to 0, where ρ is the unit round off. In other words and as the figure 4 from [39] shows, when computing in finite precision there are two phases as we decrease the time step h : first the variance V_l decreases as we go to higher levels, but starting a certain level the calculations become contaminated by rounding errors leading variance V_l to increase as $\mathcal{O}(h^{-1})$. In [39] it was shown that increasing the precision or using Kahan compensated summation [28] to compensate for errors delays this increase in error but no discussion on the effect of gradually changing the precision as l increases was proposed.

In this section we show that the gradual increase of the bit-widths obtained with our optimisation method is consistent with the analysis from [3, 39] and allows to avoid the asymptotic increase in error that was observed when the precision is fixed [39].

In [39] the Euler-Maruyama scheme is written in a more general form than the specific example we used in this thesis. The asset price S_t satisfies the SDE $dS_t = a(t, S_t)dt + b(t, S_t)dW_t$, where the *drift function* a and the *volatility function* b are assumed to be Lipschitz continuous.

The error model formulated in [39] is :

Model 3. *In the Euler-Maruyama scheme the composite effects of rounding error introduce two dominant sources of error, η and η', where at each step we have*

$$\tilde{S}_{i+1} = \tilde{S}_i + a(t_i, \tilde{S}_i)h + b(t_i, \tilde{S}_i)\sqrt{h}\tilde{Z}_i + \eta_i + \eta_i'. \tag{4.9}$$

Noting ρ the unit roundoff, the larger of these is $\eta_i = \mathcal{O}(\rho(1 + |\tilde{S}_i|))$ which is a martingale increment and the smaller of these is $\eta_i' = \mathcal{O}(\rho\sqrt{h}(1 + |\tilde{S}_i|))$ which is a possibly martingale increment.

To fix the notation in this section we use a tilde to note the low precision (fixed-point) variables. We kept the approximate random variables $\tilde{Z}_i$ as in the original paper to be consistent with the fact that in our numerical tests we limited the precision of the random normal increments too. In the original paper [39] the normal increment $\tilde{Z}_i$ refers to approximate random variables obtained using an inversion method as detailed in Chapter 5.

Using this model [39] then prove the following lemma, which states a leading order model of the error committed on the asset prices at level l :

Lemma 4. *For fine and coarse path simulations constructed using approximate random variables as described by [20] [and in Chapter 5], then with rounding errors described by model 3 we have*

$$\mathbb{E}[|\Delta S_l - \widetilde{\Delta S_l}|^2] \approx \mathcal{O}(h\mathbb{E}[|Z - \tilde{Z}|]^{2+\epsilon}) + \mathcal{O}(h^{-1}\rho^2) \tag{4.10}$$

as the discretisation h decreases for some $\epsilon \in (0, \infty)$.

In [3, 39], all variables were in floating point and had the same precision and roundoff ρ. In our project the roundoff in the operation that results in variable x is $\rho_x = 2^{-f_x-1}$, where f_x is the fraction length of variable x. In practice the models Theorems 3 and 4 still apply, with the roundoff ρ being replaced by ρ_S. To justify this we will go through the derivation of the model 3, changing the notation and making a few remarks.

Noting $\oplus, \otimes$ the addition and multiplication operations performed in finite precision, we will write the Euler-Maruyama scheme :

$$\tilde{S}_{i+1} = \tilde{S}_i \oplus (A_i \oplus B_i), \tag{4.11}$$

as in [39]. Since we used the Geometric Brownian Motion, in our case $A_i = a(\tilde{S}_i, t_i) = r\tilde{S}_i h$ and $B_i = b(\tilde{S}_i, t_i) = \sigma \tilde{S}_i \sqrt{h} \tilde{Z}_i$, although as long as $a(\cdot), b(\cdot)$ are Lipschitz continuous it does not affect the results. Note that to make our experiments consistent with this order of the operations we would need to slightly change our forward algorithm 1, but it is easy to show that variable B_i and $mult2$ have the same size (so the same exponents) and the same sensitivity therefore they also have the same precision. In addition, this does not affect our final point.

Then considering $a(\cdot), b(\cdot), \tilde{S}_0 = \mathcal{O}(1)$ [39] argue that $\tilde{S}_i, \tilde{Z}_i, A_i, B_i = \mathcal{O}(1)$. Therefore assuming $h \ll \sqrt{h} \ll 1$ gives the size ordering

$$\mathbb{E}(|A_i|) \ll \mathbb{E}(|B_i|) \ll \mathbb{E}(|\tilde{S}_i|). \tag{4.12}$$

Then note η_i' the error in the first addition, ie. $A_i \oplus B_i = A_i + B_i + \eta_i'$. Since the error η_i' is of the size of the roundoff of the larger of A_i and B_i, we have $\eta_i' = \mathcal{O}(\rho_B \sqrt{h})$. Similarly defining the error η_i in the second addition by $\tilde{S}_{i+1} = \tilde{S}_i + B_i + A_i + \eta_i + \eta_i'$ gives $\eta_i' = \mathcal{O}(\rho_S)$. Since $\rho_B = \rho_{mult2}$, we obtained the model

$$\tilde{S}_{i+1} = \tilde{S}_i + \eta_i + \eta_i' \tag{4.13}$$

$$\text{with} \quad \eta_i = \mathcal{O}(2^{e_S - d_S - 1}) \tag{4.14}$$

$$\text{and} \quad \eta_i' = \mathcal{O}(\sqrt{h} 2^{e_{mult2} - d_{mult2} - 1}) = \mathcal{O}(2^{e_{mult2} - d_{mult2} - 1 - l/2}) \tag{4.15}$$

Now the dominant error is η_i so we argue that $V_l \approx \mathbb{E}[|\Delta S_l - \widetilde{\Delta S}_l|^2] = \mathcal{O}(h^{-1}\rho_S^2) = \mathcal{O}(2^l 4^{e_S - d_S - 1})$. Also note that in our experiments the size of variables $S, Sc, mul2$ and $mult2c$ do not vary with level (see exponents on Figure 4.10). Furthermore in practice we see that V_l is divided by 2 at each level therefore we conclude that d_S must increase by 1 bit at every level. This way, we have shown a link between the observed evolution of the bit-width of variables S, Sc and the leading order error model from [39].

4.3.1 Numerical experiment

In Figures 4.14 and 4.16 we illustrate numerically that adapting the bit-widths allows to decrease the variance while if the bit-widths in the low precision calculation were maintained fixed the variance would increase with l.

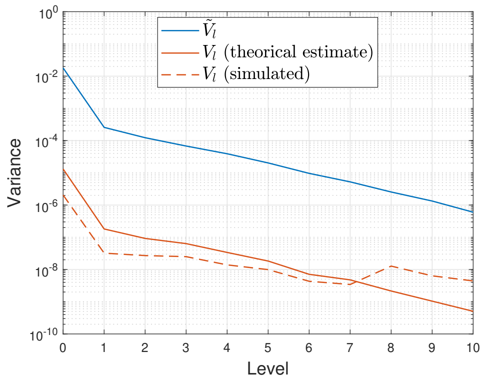

Figure 4.14: Variance for path generation in fixed point with *optimised* precision at each level.

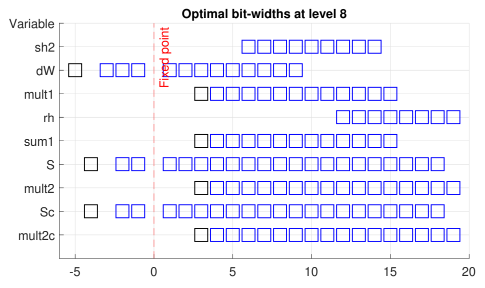

Figure 4.15: Optimal bit-widths at level 8.

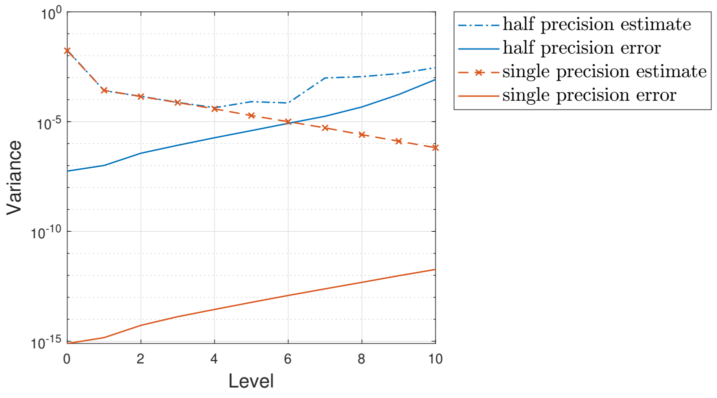

Figure 4.16: Variance for path generation in floating point with *fixed* precision.

This experiment is similar to the figure 4 from [39] : the continuous curves from Figure 4.16 are analogues of the curves with the downward triangles and squares markers from [39] respectively. The difference in our experiment is that to produce the figure we rounded the random numbers instead of taking approximate random numbers, therefore we preferred to do the test again to consistently compare them with the variance we obtained with optimised bit-widths 4.14. In Figure 4.16 the estimate $\widetilde{\Delta P_l}$ and error $\Delta P_l - \widetilde{\Delta P_l}$ samples were computed entirely in floating point arithmetic, were the low precision path was computed in single precision for the blue curves and in half precision for the red ones. In Figure 4.16 we used our proposed framework, namely we computed in fixed-point arithmetic with optimised bit-widths for the low precision paths and in double precision floating point for the accurate path.

Regarding Figure 4.16, the first term from the leading order model 4 is apparently negligible and we see that even on the first levels the variance V_l (see continuous curves) already increases. Continuous curves have slope h^{-1} as predicted by the leading order model 4. We also observe a similar behaviour for the variance of the estimate $\widetilde{\Delta P_l}$ when half precision is used (dashed blue curve), showing that the precision needs to increase for the accuracy of the low precision simulations to increase after level $l = 4$.

In contrast, when using optimised precision (Figure 4.14) the theoretical V_l (continuous red curve), the corresponding simulated variance (dashed red curve) and the variance $\tilde{V}_l$ of the low precision estimate continue to decrease with level with a constant slope, which proves the efficiency of our optimisation procedure.

Note however that for the simulated V_l we observe that the optimistic theoretical bound on V_l is violated starting from level 8 (Figure 4.14). We believe that this is because either the linear model ceases to be valid at this level of accuracy, meaning we should include higher order errors in the model, or the correlation between the errors is higher so we should switch to using the pessimistic estimate of the variance V_l.

Hence, the key takeaway from this experiment is that up to level 8 our proposed optimised precisions improve the accuracy of the correction term (comparing red lines from both Figures 4.14 and 4.16), although the performance of our model encounters a limitation after level 8.

4.4 Conclusion

In conclusion, the leading order error model provides another motivation for using optimised precisions instead of taking a fixed low precision to decrease the cost and relying on the fine levels to increase the accuracy in the MLMC simulation.

In cases where the drift and volatility terms describing the asset dynamics are more complex to evaluate or the numerical scheme that is employed makes path simulations more complicated, the FPGA path calculation could be more expensive relative to C_{RNG} so switching to standard Monte Carlo levels could happen "early" due to level cost considerations, as illustrated in Section 4.2.1 by simply varying the value of C_{RNG}. Despite this, using optimised bit-widths allows to exploit cheap lower precision calculations on more levels than a fixed precision approach as the variance of the correction term keeps decreasing with level.

Moreover the optimisation stage could be moved online to avoid extra cost and latency.

Chapter 5

Gaussian random number generation on CPUs and FPGAs

An important assumption in our nested framework is that the samples generated on the FPGA use low precision random variables that are very cheap to compute. In this chapter we discuss several possible methods to generate approximate Gaussian Random Numbers (GRNs) on the FPGA very efficiently. In MLMC applications, to generate samples of the correction terms we require that ΔP_l and $\widetilde{\Delta P_l}$ use the same random normal increments. Therefore among the wide range of existing GRN generators (GRNG) we chose to focus on inversion methods because they allow to compute very cheaply a couple of GRNs $(Z, \tilde{Z})$ used on the CPU and the FPGA respectively that are approximately equal.

Using low precision approximate GRNs on the FPGA is the second key improvement over [6], where the low precision paths used single precision normally distributed increments.

5.1 Background on Gaussian Random Number Generators

Due to their importance in a large number of applications, a number of GRNG algorithms that are suitable for hardware implementation have been developed, and a good survey of the literature on efficient hardware architectures implementing the main algorithms is available in [35]. In essence, nearly all GRNGs produce their output by taking as input one or several uniformly distributed numbers and converting them to Gaussian random numbers. The uniform random numbers (URNs) are seen as a chain of random bits that are usually generated by applying bit-wise operations on an initial seed number [40]. Fast methods for generating URNs are available in the literature [38] and a comparison of the generation speed for different hardware devices, including CPUs and FPGAs, is provided in [40]. A key takeaway from [40] (2009) is that FPGAs are much faster and power efficient for URN generation than CPUs and even GPUs, giving a number of generated samples per second that is 15 (resp. 50) times higher than on GPUs (resp. on CPUs). The power efficiency also improves by a factor 60 compared to GPUs. We believe that this factor is still relevant on modern devices, which motivates again the use of FPGAs in MLMC.

For conversion to Gaussian (or more specifically to normal) random numbers, an important class of methods relies on the CDF inversion. According to [40], inversion methods are cheap and produce approximate GRNs with arbitrary precision which makes them very competitive. These methods are suitable and efficient for both CPU and FPGA implementations, as shown in [35] and references therein and in the Intel description of RNGs available at [27].

The idea of inversion methods is to apply an approximation of the inverse CDF Φ^{-1} of the Gaussian distribution to an input URN to obtain a GRN. The inverse CDF is approximated on the interval $[0, 1]$ by a piecewise polynomial function whose coefficients are stored in a LUT and the hardware design includes a process to identify the segment of $[0, 1]$ that contains the input uniform variable U. In practice, since the Gaussian CDF is symmetric around the point $(1/2, 0)$ the function is usually approximated only on $[0, 1/2]$ and a random bit is used to invert the sign of the GRN. Several possibilities for the segmentation of the interval $[0, 1/2]$ have been studied in the literature. The simplest way is to split $[0, 1/2]$ uniformly as illustrated in Figure 5.1, but since the function Φ^{-1} has a singularity near 0 defining a partition that is fine enough to obtain a desired approximation accuracy can lead to very large LUTs. To avoid this, [7, 31] proposed a hierarchical partitioning of the input interval and an efficient address finding hardware architecture that is adapted to the shape of the inverse CDF. For their baseline segmentation they used what we will call *dyadic* intervals, which are split by two as we get closer to the singularity, as illustrated in Figure 5.2. Similarly, in [16] a piecewise linear approximation on geometric intervals is presented and the corresponding RMS error is analysed in the L_p norm.

In the next sections we focus on three approximations of Φ^{-1} : a piecewise constant approximation on uniform intervals, a piecewise linear approximation on dyadic intervals, and a variant of the first method that takes advantage of the Central Limit Theorem to produce more accurate GRNs by summing several low accuracy GRNs. These three methods were suggested in the notes that summarised the ideas for this project provided by Prof. Mike Giles.

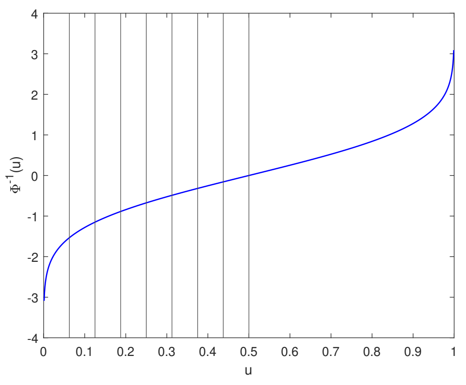

Figure 5.1: Function Φ^{-1} on $[0, 1]$ and uniform intervals on $[0, 1/2]$.

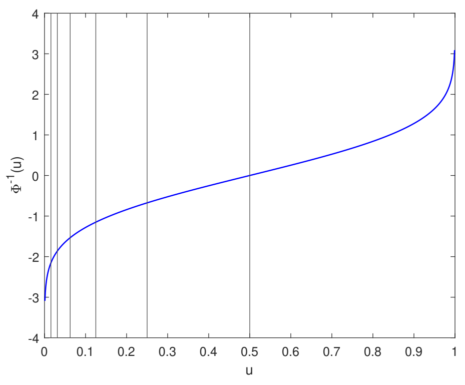

Figure 5.2: Function Φ^{-1} on $[0, 1]$ and dyadic intervals on $[0, 1/2]$ as in [16].

5.2 Generation of gaussian variables using the inversion method

In this section we detail three CDF inversion methods for the generation of an approximate GRN $\tilde{Z}$ on the FPGA that comes from the same uniform variable as the full precision GRN Z computed on the CPU.

In all three approaches below, the coupled GRN computed on the CPU is obtained with a piecewise approximation of Φ^{-1} with polynomials of degree 5 on the hierarchical interval segmentation described in [7]. Therefore for each of the following methods to obtain $\tilde{Z}$, we mention how to obtain the corresponding full precision uniform variable, then the full precision GRN Z follows.

To fix the notation, we consider that the double precision uniform variable U corresponds to a D-bit integer that we denote by J and the associated low precision uniform variable corresponds to a d-bit integer denoted by j. The leading bit of j gives the sign of the RNG and the next $d-1$ are used to generate an approximate GRN with one of the following methods.

5.2.1 Piecewise constant approximation on uniform intervals (Method 1)

In the first approach for the low precision GRNG we simply construct a LUT containing the constant values Z_j that the approximate GRN takes when the input uniform variable is inside the interval $\mathcal{I}_j = [u_j, u_{j+1}] \subset [0, 1/2]$. The intervals are uniform with $u_j = 2^{-d}j, j < 2^{d-1}$. Locating the interval corresponding to the input uniform variable is trivial since the integer j is the index of the interval.

To determine the optimal constants Z_j the MSE

$$\int_{u_j}^{u_{j+1}} (Z_j - \Phi^{-1}(u))^2 du \tag{5.1}$$

is minimised with respect to Z_j, which leads to Z_j being the mean of Φ^{-1} over the interval $\mathcal{I}_j$:

$$Z_j = 2^d \int_{u_j}^{u_{j+1}} \Phi^{-1}(u) du. \tag{5.2}$$

Then the corresponding full precision uniform variable is defined as $U = 2^{-D}\left(J + \frac{1}{2}\right)$ and the full precision GRN follows.

The issue with this approach is that the LUT is of size 2^{d-1}, which may not fit on the FPGA or may take up too many hardware resources, for example for $d = 10$ the LUT stores $2^9 = 512$ values.

The resulting approximation of Φ^{-1} and the approximation error are shown in Figures 5.3 and 5.4 for $d = 10$. Since the strongest variations of Φ^{-1} are near $U = 0$ the error is increasingly large as we get close to that point. The piecewise constant approximation on uniform segments is good enough on parts of the function were the slope is mild (near $U = 0.5$).

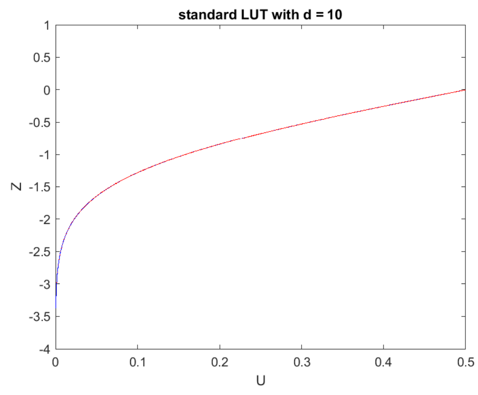

Figure 5.3: Piecewise constant approximation with uniform intervals for $d = 10$.

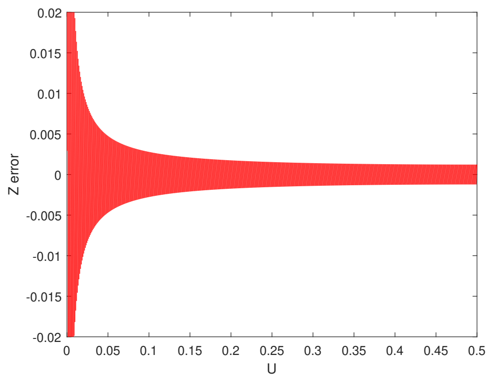

Figure 5.4: Error in piecewise constant approximation with uniform intervals for $d = 10$.

5.2.2 Piecewise linear approximation on dyadic intervals (Method 2)

In order to reduce the size of the LUT, we now use dyadic intervals as mentioned in Section 5.1 and illustrated in Figure 5.2. The interval containing the point indexed by j is identified by the leading bit i of j and contains the points indexed by the integers

$[\![2^{i-1}, 2^i - 1]\!]$. In each of these intervals the GRN is approximated by $\bar{Z}_j = a + bj$, with a separate pair (a, b) for each interval. The coefficients are stored in a LUT of size $d - 1$ and the values (a, b) are again obtained by minimising the MSE. A simple calculation shows that

$$\int_{u_j}^{u_{j+1}} (\bar{Z}_j - \Phi^{-1}(u))^2 du = 2^d (\bar{Z}_j - Z_j)^2 + \int_{u_j}^{u_{j+1}} (Z_j - \Phi^{-1}(u))^2 du \tag{5.3}$$

where Z_j is defined as in the previous section. Therefore to calculate the pairs (a, b) we only need to minimise

$$\sum_{j=1}^{2^{d-1}} (\bar{Z}_j - Z_j)^2. \tag{5.4}$$

Equation (5.3) also shows that for the same value of d this approximation cannot be as good as the Z_j approximation, but it is important to keep in mind that here the LUT is much smaller and the resulting MSE will be shown in Table 5.2. The uniform CPU variable U is defined in the same way as in method 1 so the coupling $(\tilde{Z}, Z)$ follows naturally.

For $d = 10$, the resulting piecewise linear function and approximation error are shown in Figures 5.5 and 5.6 respectively. The error plot shows that this structure of the segments allows to have an error of the same magnitude across most of the intervals.

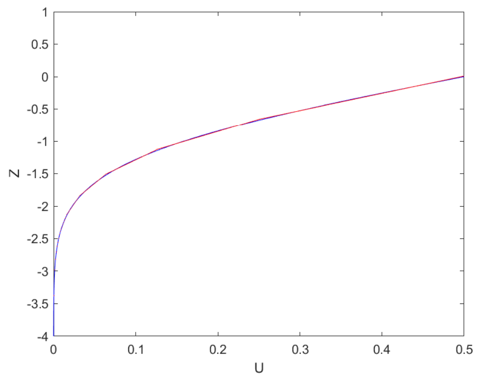

Figure 5.5: Piecewise linear approximation with dyadic intervals for $d = 10$.

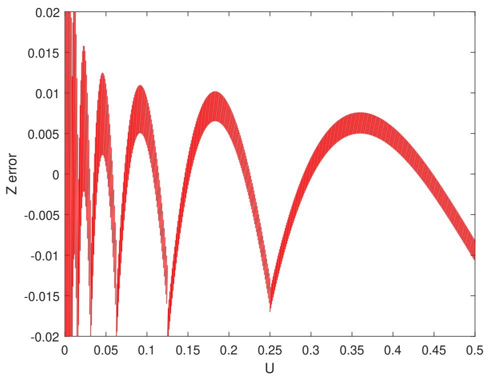

Figure 5.6: Error in piecewise linear approximation with dyadic intervals for $d = 10$.

5.2.3 Combination of CDF inversion and Central Limit Theorem (Method 3)

The idea of the last approach is to reduce further the precision of the approximate normal random variables generated with the LUT and use the Central Limit Theorem (CLT) to improve the statistical quality of the final approximate Gaussian variables. For example take an integer n that divides d and produce one GRN $X^{(1)}$ from the first d/n bits of j, then $X^{(2)}$ from the next d/n bits and so on using the CDF inversion method from one of the previous subsections. Then by the *Central Limit Theorem* the variable

$$\frac{1}{\sqrt{n}} \sum_{i=1}^{n} X^{(i)} \tag{5.5}$$

also follows an approximate Gaussian distribution. According to [35] (section 2.11), "when n uncorrelated random numbers are added, the resulting [probability distribution function] (PDF) can be computed by convolving the individual PDFs". In the same paper they show experimentally that the PDF of the GRNs resulting from 5.5 is more accurate than the PDF of the initial $X^{(i)}$ variables (see figures 20 and 22 from [35]). Hence this method allows to reduce significantly the size of the required LUT without sacrificing accuracy.

The idea of combining an approximate GRNG method with the CLT was already used in [40, 35, 33] with $n = 8$, $n = 4$ and $n = 2$ respectively. In particular, [40] used the piecewise linear approximation on dyadic intervals to generate the $X^{(i)}$s.

In this section we introduce a similar method where several GRNs obtained using the piecewise constant approximation (method 1) are summed. The difference compared to the previous implementations is an optimisation stage where the coefficients stored in the LUT are adapted iteratively, which improves the quality of the estimation compared to simply taking the mean of Φ^{-1} over the intervals. The second difference compared to previous work is the coupling between the low precision and full precision uniform variables that is required in our application.

We explain the method for two variables, as it is then easy to generalise it. Take the integer j and split it into two integers with bit-widths $d/2$. Both streams of lower precision variables $X^{(1)}, X^{(2)}$ are computed with the same LUT of size $2^{d/2-1}$. This LUT is initialised using the method 1 and dividing the LUT values by $\sqrt{2}$. Therefore let's note X^{j} the values in the small LUT. Using this initial LUT we form a larger LUT of size 2^d by computing all possible sums $_j \pm X_l$ and order the resulting outputs $\tilde{Z}_k$ in ascending order. This ordering defines a permutation π such that $\pi(j)$ gives the position of the normal computed from the integer j in the ordered list. Then we perform a least-squared minimisation of

$$\sum_{k=1}^{2^d} (Z_{\pi(k)} - \tilde{Z}_k)^2 \tag{5.6}$$

where the $Z_{\pi(k)}$ are obtained with method 1. In this minimisation problem, the variables $\tilde{Z}_k$ are considered as linear variables of the decision variables, which are the values X_j from the small LUT that we want to optimise.

After this step update the permutation π by ordering the resulting $\tilde{Z}_k$ and repeat the least-squares optimisation. The algorithm stops when the permutation has converged. In practice we observed that for $d = 10$ and $d = 12$ this process takes about 20 and 100 iterations respectively, and that the optimisation stage considerably reduces the MSE (see Table 5.2).

Finally, generating the coupled full precision variable requires to take the permutation π into account. When the optimised LUT is used and the input integer is j, the output is the value $\tilde{Z}_j$, which is an approximate of $\Phi^{-1}(2^{-d}\pi(j))$. Therefore the corresponding uniform variable used on the CPU is

$$U = 2^{-d}\pi(j) + 2^{-D}\left((J - 2^{D-d}j) + \frac{1}{2}\right) = 2^{-d}(\pi(j) - j) + 2^{-D}\left(J + \frac{1}{2}\right). \tag{5.7}$$

The permutation π only needs to be stored (in a permutation table of size 2^d) and used on the CPU while the GRNG on the FPGA is simply done using only the optimised LUT of size $2^{d/2-1}$.

Generalising the method to $n > 2$ variables is straightforward. The the only change is that the values of the small LUT are divided by $\sqrt{n}$ and n low precision variables are summed to obtain the $\tilde{Z}_k$. The coupled uniform variable used on the CPU is defined with exactly the same formula as in the two variables case.

5.3 Computational cost of approximate GRNG on CPUs and FPGAs

Before diving into numerical tests to compare thoroughly the approximation methods introduced above, we add a few words on the computational cost of the random number generation specific to our application. We first estimate the cost of a single random normal variable generated on the CPU, that is to say we find the order of magnitude of the parameter C_{RNG} used in the previous chapters.

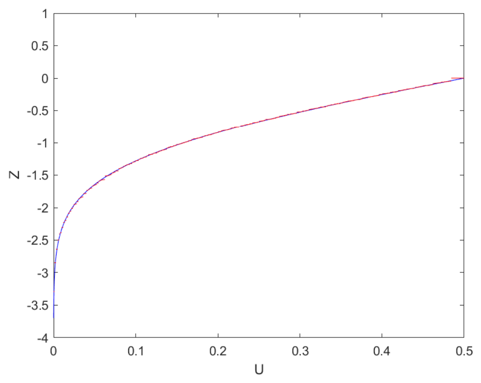

Figure 5.7: Two variables approach with uniform intervals for $d = 10$.

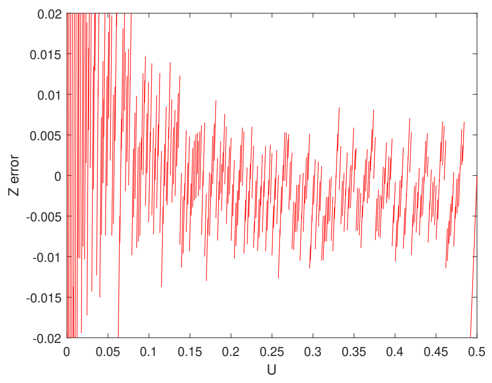

Figure 5.8: Error for the two variables approach with uniform intervals for $d = 10$.

Based on references of available RNGs for Intel CPUs [27] we will assume that normal random numbers are generated on the CPU using a polynomial approximation of Φ^{-1} with polynomials of degree five on dydic intervals. We consider that the cost of locating the interval that contains the input uniform variable x is negligible. Once the segment is identified, the output GRN is computed by interpolation as follows

$$((((a_5 \times x + a_4) \times x + a_3) \times x + a_2) \times x + a_1) \times x + a_0 \tag{5.8}$$

where a_i are the coefficients stored in the LUT. Assuming all coefficient and the uniform variable x are double precision floating-point numbers, their bit-width is $D = 53$, and we neglect the cost of the operations on the exponents. Then using Equation (1.3) leads to a total cost of the interpolation of approximately 13000. Finally, to approximate the cost of the uniform number generation, based on performance data on the available URNGs from Intel [27], we consider that a variant of the Mersenne Twister generator was used to generate x, which adds to the cost a term proportional to 64 that we neglect because it is small relative to the cost of estimating $\Phi^{-1}(x)$. In addition, if we use the multiple variable approach on the FPGA, the CPU needs to apply the permutation π to the leading d bits of x, which is either considered as a linear operation or a look-up. This stage could be quite demanding in terms of memory as it requires storing a permutation table of size 2^d but also has negligible cost.

Next we discuss the cost of the GRNG on the FPGA. We first summarised the size of the LUTs and the resulting conversion cost from uniform to gaussian variables for methods 1, 2 and 3 in Table 5.1. The piecewise constant approximation is cheap to use once the construction of the LUT is complete as the address finding (for uniform segmentation of $[0, 1/2]$) is trivial and no interpolation is required. For piecewise linear approximation we considered the same cost per operation as in the previous chapters 1.3 and that the input uniform variable, the coefficients (a, b) and the output normal variables all have the same bit-width, leading to the cost $d^2 + 2d$. Then, as we have seen in Figure 4.7 (at least) up to level 15, the optimal bit-widths used for the normal increments dW is no larger than 16, leading to a cost of the conversion to normal variables of at most 300.

Approximation method	LUT size	Evaluation cost
Piecewise constant	2^{d-1}	Look-up
Piecewise linear (PWL) + dyadic intervals	$2 \times (d-1)$	$d \times (d-1) + 2d$
Two variables | n variables	$2^{d/2-1}$ | $2^{d/n-1}$	Look-up

Table 5.1: Storage and evaluation cost on the FPGA for the approximate GRNG methods 1,2 and 3.

It remains to estimate the cost of the URNG on the FPGA as well as its coupling with the CPU calculation. The uniform variables on the FPGAs are computed in two different manners, depending on the sample path where they are used : In samples of $\Delta P_l - \widetilde{\Delta P_l}$ that are computed on both CPUs and FPGAs, for each time step the uniform variable used on the FPGA is simply a truncated version of a uniform variable generated in full precision on the CPU. On the other hand, for samples of $\widetilde{\Delta P_l}$ that are only computed on the FPGA the uniform variables can be generated with negligible cost using Bakhvalov's trick ([19], Appendix A), which uses two independent streams of URNs generated by the CPU to generate new independent uniform variables very cheaply. Therefore in both cases the URNG on the FPGA only takes a few bit-wise operations, and we can conclude that the GRNG cost on the FPGA is well approximated by the cost of converting the uniform random numbers into normal random numbers alone.

5.4 Empirical comparison of the GRNG methods 1,2 and 3

In this section we compare empirically the quality of the estimated GRNs from each approximation method. First, we compare the MSE obtained for each method for several values of the bit-width d. We chose $d = 10$ and $d = 12$ as the optimal bit-width of dW obtained in Section 4.2 was close to these values (at least for levels 0 to 15, with the value of $C_{RNG} = 10^4$). In addition to the MSE comparison, we looked

Method	d	MSE	Maximal value
Piecewise constant	9	3.32×10^{-4}	3.18
Piecewise constant	10	1.50×10^{-4}	3.37
Piecewise constant	12	3.13×10^{-5}	3.74
PWL + dyadic	10	1.91×10^{-4}	3.37
PWL + dyadic	12	7.26×10^{-5}	3.74
Two variables (init.)	10	4.26×10^{-4}	3.19
Two variables (init.)	12	1.10×10^{-4}	3.55
Two variables (optimal)	10	2.80×10^{-4}	3.29
Two variables (optimal)	12	6.00×10^{-5}	3.67

Table 5.2: Approximation error for different approximate GRNG methods.

at the maximal size of the GRNs generated by each method. For all methods above, the determination of the range is trivial : for the piecewise constant approximation it is sufficient to take $|Z_1|$ and for piecewise linear approximation taking the absolute value of the degree 0 coefficient $|b|$ from the leftmost interval gives the desired range. The results are summarised in Table 5.2 (see columns 3 and 4).

We used the code provided by Prof. Giles in our numerical tests.

5.4.1 Discussion on the MSE

We first compare roughly the three approximations methods by looking at the resulting MSE.

First note that the MSE in method 2 is necessarily larger than in method 1 (see (5.3)). Despite this, for $d = 10$ the dyadic approximation is only slightly worse, which is probably because for the first two dyadic intervals there are only two points in each so $\bar{Z}_j$ will exactly match Z_j. However for $d = 12$ there is nearly a factor 2, which illustrates that the MSE does not evolve in the same way depending on the type of intervals that is used, as will be discussed below.

Next, as suggested in Mike Giles' notes, it is also interesting to compare methods 3 with d bits against the method 1 with $d-1$ bits, since most of the values in method 3 occur in pairs ($X_i + X_j$ and $X_j + X_i$). The values for $d = 10$ are shown in Table 5.2. We see that the double variable approach gives a smaller MSE than the simple LUT approach, which confirms the accuracy of method 3. This difference is due to the extra values of the form $X_i + X_i$ that the method 3 can produce. Also it is important to note that without the optimisation the method 3 does not perform well.

Finally, comparing methods 2 and 3 is less straightforward. In Table 5.2 we see that, for $d = 10$, the dyadic approach is more accurate, while for $d = 12$ the optimised two variables approach is preferred. Therefore we made a plot of the MSE (Figure 5.9) for different values of d to visualise better the trends of the MSE with increasing bit-width.

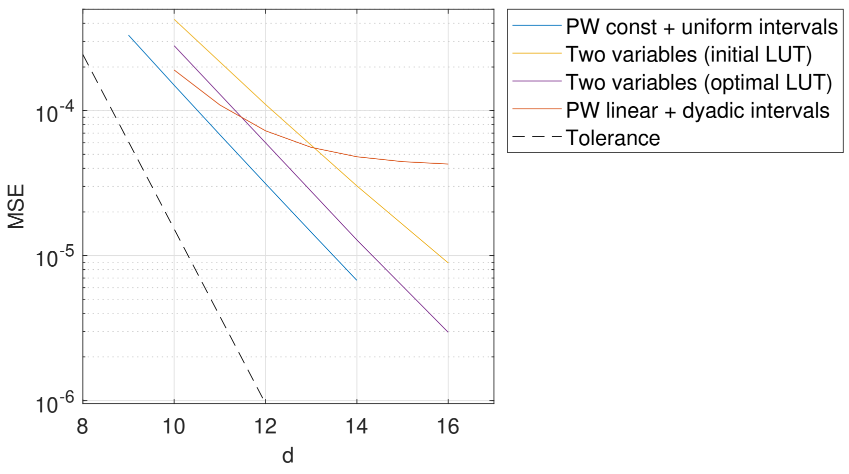

Figure 5.9: MSE for methods 1,2 and 3 for different d.

As illustrated by Figure 5.9 and the convergence analysis from [16], as d tends to infinity, the uniform intervals give $MSE \longrightarrow 0$ and the dyadic intervals give $MSE \longrightarrow C$, for some positive constant C. The latter is because with our simple dyadic segmentation, when d increases by 1 the MSE is reduced only in the interval closest to 0, so the error due to the other intervals remains the same. This is the reason why the dyadic intervals were split further into smaller uniform intervals in [31] as it allows to improve the approximation of the inverse CDF. However their address location process is too complex for our needs, so we do not consider improving the segmentation of the dyadic method further.

In addition, the Figure 5.9 illustrates that both method 1 and 3 have their MSE divided by 2 each time d increases by 1. This was theorically expected for method 1 (see [16]) and is intuitively true for method 3 after the optimisation stage as the MSE is minimised such that the $\tilde{Z}_j$ approximate the Z_k corresponding to a larger LUT, so the slope of the MSE in method 3 should be similar to that of method 1. Without optimisation, the MSE of the double variable approach has a different slope on Figure 5.9, probably because the initial $\tilde{Z}_j$ values are "irregular". Despite the factor 2 between the method 1 and method 3, the double variable approach shows good accuracy for significantly reduced LUT size. This might be because the optimisation allows some values of the table to take more "extreme" values to better

Method	d	$\mathbb{P}[\lvert error\rvert > tol]$	$Card(\lvert error_j\rvert > tol)/size(LUT)$
Piecewise constant	9	5.79%	0.39%
Piecewise constant	10	5.80%	0.39%
Piecewise constant	12	5.79%	0.15%
PWL + dyadic	10	66%	33%
PWL + dyadic	12	92%	91%
Two variables (optimal)	8	34%	31%
Two variables (optimal)	10	50%	49%
Two variables (optimal)	12	66%	66%
Two variables (optimal)	16	86%	87%

Table 5.3: Pointwise error for different approximate GRNG methods.

approximate the tail of the distribution. This is consistent with the increase in the maximal value after optimisation (see Table 5.2).

5.4.2 Pointwise error and accuracy of the last bits

For all methods above, even if we look at the error in Z only in the range of values defined by the GRNG, this error can go as far as 0.2 (see Figures 5.4, 5.6 and 5.8), which is worse than our desired tolerance. Indeed we would like the approximate normal variables to have d bits of accuracy. In fixed-point arithmetic the exponent of the GRNs is e, so we want the error on Z to be smaller than $tol = 2^{e-d-1}$.

Therefore we first looked at the error at the points $u_j = j2^{-d}$ that are used to construct the LUT in all three approximation methods. Then we also compared the probability of the absolute error being larger than the tolerance 2^{e-d-1} when the input uniform random variable can take any value in $[0, 1]$. To compute this probability we split every segment $[u_j, u_{j+1}]$ into 100 intervals and counted the number of points that violate the tolerance. Then summing these across the segments and scaling by $2 \times 2^{-d}/100$ gives the desired probability. The results from these calculations are given in the two last columns of Table 5.3.

For the PWL approximation on dyadic intervals the distance between the approximate inverse CDF and the real one is very large notably in the intervals in the middle of $[0, 1]$, which obviously cannot be improved by increasing d. This explains why this method performs badly with respect to $\mathbb{P}[|error| > tol]$. Therefore it would be pointless, for instance, to take finer input uniform variables in an attempt to decrease the MSE by evaluating the approximate inverse CDF at more points, because the additional uniform points would yield a high error.

Nb. of variables $\times d/n$	MSE (optimised)	$\mathbb{P}[\lvert error\rvert > tol]$	$Card(\lvert error_j\rvert > tol)$
2×6 bits	6.00×10^{-5}	33%	67%
3×4 bits	2.80×10^{-4}	4.6%	93%
2×8 bits	2.96×10^{-6}	43%	87%
4×4 bits	1.12×10^{-4}	99%	99%

Table 5.4: Tests method 3 5.2.3 for different LUT sizes and number of combined variables (corresponding to $d = 12$ and $d = 16$).

The method 1 gives very good values of $\mathbb{P}[|error| > tol]$ and the tolerance is violated only at very few points of the mesh. Therefore for these values of d we conclude that all bits of the produced variable are accurate with probability ≈ 0.94. It is therefore surprising that the two variables approach has considerably poorer performance for $d = 10$. For method 2, we have also made tests where the tolerance was larger to see how many bits are accurate. It seems that as d increases the number of inaccurate bits increases too. This suggests we might need to sum more variables to improve the quality of the approximate random normal increments, which we did in the next subsection.

5.4.3 Summing more low precision Gaussian variables

As mentioned in Section 5.2.3, previous works [40, 33, 35] motivate summing more lower precision variables in method 3 in order to reduce even further the size of the LUT or improve the accuracy of the produced GRNs. Therefore we chose several values of d and n such that the results are comparable and summarised the results in Table 5.4. Note that we don't need more than 16 bits of accuracy since in Chapter 4 we saw that the bit-width of the random increment was usually between 8 and 15.

Contrarily to the our expectations, here we do not obtain a good accuracy in terms of tolerance on the pointwise errors, in most of the cases above. Despite this, with the 3×4 bits approximation we obtained a small $\mathbb{P}[|error| > tol]$, but the accuracy at the mesh points was not good. It is unclear whether the bit-widths of the low precision variables are too small or we don't take enough variables to obtain an improvement in the distribution, since the case with $d = 12$ and $d = 16$ show different evolutions of the accuracy as we increase n.

But we could take even more variables and use a larger bit-width for the uniform input that is split into n random integers. To mention the limitations linked to our application, for the full FPGA samples we use a stream of URNs provided by the CPU, which have 64 bits, so dividing that in two in order to use Bakhanov's trick we get $D = 32$ bits. This can be split in different ways. For example we could use $d = 24$ and $n = 4$. Then if the accuracy is not good enough we could use the sample from the second stream too, in order to add 8 low precision approximate GRNs.

5.5 Discussion and conclusion

All three methods have their strengths and weaknesses that can be summarised as follows. In the experiments, for methods 2 and 3 it appears that taking a LUT of size equal to the number of desired bits of accuracy is not sufficient to ensure that the output random normal variables have enough accuracy. Method 1 requires a lot of storage on the FPGA but is the most accurate. Method 2 is less storage demanding but the MSE is bounded so seeking higher approximation accuracy requires defining more complex segmentation. On top of this the method requires an interpolation instead of a simple look-up. The method 3 is cheap on the FPGA but it is not trivial to define the right parameters to achieve a desired accuracy on the pointwise error and the method requires storing a permutation on the CPU which can be large (2^{d-1} as for method 1, but stored on the CPU instead of the FPGA).

In our application we want to reduce as much as possible the number of operations required by the RNG on the FPGA, both for the uniform input variable and the resulting normal variable. Therefore the hybrid method which exploits the PWC evaluation and improves the accuracy using the CLT is the most efficient choice to make the RNG cost negligible. This would allow to respect the assumption Equation (3.3), which was admitted in our thesis and used in the numerical experiments. More precisely, with carefully chosen parameters d and n, the method 3 would reduce the operations on the FPGA performed for the RNG to a few bitwise operations : using the stream of URNs from the CPU and Bakhvalov's trick, the FPGA generates random integers with only d bitwise operations, then looks up for the Gaussian variables $X^{(i)}$ $(i = 1, \dots, n)$ in a LUT and sums them with $n \times d/n$ bit-wise operations.

The counterpart of the method is that we need to store and use the permutation π to generate the URNs used for path generation on the CPU, which slightly increases the cost of CPU samples. We believe that this increase in C_{RNG} is acceptable as most samples are computed on the FPGA. Indeed for the first levels of the nested MLMC we do not need much accuracy in the normal increments therefore d would be small, which would give a permutation table of size 2^{d-1} that fits well in the CPU's L2 cache. Then as more accuracy is required d and the permutation table would grow. However we think that this is not problematic on at least the 3 first levels.

Chapter 6

Conclusion

To summarise this thesis, using a model of the error based on algorithmic differentiation we formulated a problem that allows to customise the bit-widths in the MLMC framework. We have shown that optimising the bit-widths allows to compensate for the rounding errors that would otherwise accumulate and decrease the accuracy of the correction term as the time step decreases. We then compared several possible methods for generating the low precision normal random variables that could be used in our framework to reduce considerably the cost of generating paths on the FPGA.

The optimisation of the bit-widths could easily be implemented in the industry and could be performed once per week or per month, allowing to reduce considerably the daily computational workload for pricing financial options and save energy and time.

The following steps of the project would be to implement this framework on real hardware to produce data and confirm its improved speed and power efficiency over existing methods. Hardware implementation and testing would also be important to confirm the value of the parameters and the GRNG method that should be used. For example, one could seek the most relevant values of d, n or the cost of the GRNG on the CPU (C_{RNG}) which are important factors to set the number of levels that should be nested and configure the hardware devices.

Note however that real hardware tests are required to rigorously conclude on the hardware efficiency comparison, which will be the scope of a future project between Mike Giles and his collaborators (from the computer science community).

Appendix A

Main components of an FPGA

Component	Description	Example : AMD Versal Premium VP1902
Look-Up-Tables (LUTs)	Serve as basic logic elements where truth tables can be mapped onto them. They are notably used for storing values that allow to evaluate precomputed functions.	8.46M
Configurable Logic Blocs	CLBs consist of LUTs, flip-flops, and interconnectivity resources combined together.	18.5M
Input/Output Blocks (IOBs)	IOBs facilitate communication between the external world and internal modules of FPGAs.	2,328 SelectIO resources capable of operation up to 3.2 Gbps
Routing Resources	The routing matrix allows signal propagation throughout the chip.	
Clock Networks	Dedicated resources ensure efficient distribution of clock signals across the chip.	
Memory blocks (RAMs or ROMs)	(Optional)	block RAM of 238Mbit and UltraRAM of 619Mbit
Digital Signal Processors (DSP)	(Optional) Dedicated twos complement multipliers and a accumulators to accelerate multiplications and other functions.	6,864 blocks

Table A.1: Main components of an FPGA, their functions and corresponding characteristics for the largest existing FPGA : the AMD Versal Premium VP1902 [2]. Reproduced from [21] and complemented with [40, 1, 2].

Appendix B

Multilevel Monte Carlo Algorithm pseudo-code

Algorithm 4 Multilevel Monte Carlo Algorithm

```
**Input:** the desired RMS $\varepsilon$, an initial number of levels $L_{min} \geq 2$ and an initial number of samples $N_0$ for levels $l = 0, 1, \ldots, L_{min}$ + arguments for the subroutine used to compute the samples
**Output:** expected payoff, number of samples $N_l$ generated and cost of the path generation $C_l$ at each level $l$
$sums \leftarrow 0$        ▷ initialise a matrix that contains the sample sums of $\Delta P_l$ and $\Delta P_l^2$
$L \leftarrow L_{min}$        ▷ initialise the number of levels
$C_l \leftarrow 0$
$N_l \leftarrow 0$
$dN_l \leftarrow N_0$  ▷ initialise the number of additional samples that need to be generated
while $dN_l > 0$ for at least one $l$ do
    for $l = 0, 1, \ldots, L$ do
        if $dN_l(l) > 0$ then
            $sums(l) \leftarrow sums(l)+$ [result of subroutine]
            $C_l(l) \leftarrow$ [result of subroutine]
            $N_l(l) \leftarrow N_l(l) + dN_l(l)$
        end if
    end for
    compute $V_l$ from $sums$
    $N_s = \lceil 2\varepsilon^{-2}\sqrt{(}V_l/C_l)\left(\sum_{l=0}^{L}\sqrt{(}V_lC_l)\right)\rceil$
    $dNl \leftarrow max(N_s - N_l, 0)$
    if $\mathbb{E}[\Delta P_L]/(2^\alpha - 1) > \varepsilon/\sqrt{2}$ then        ▷ test weak convergence
        $L \leftarrow L + 1$        ▷ add a level (and an extra column to $N_l, dN_l, V_l, C_l, sums$)
    end if
end while
```

Appendix C

Complements referenced in Chapter 3

This appendix presents the plots that justify using only 7 or 9 different bit-widths per level and shows the bit-widths obtained with the various optimisation methods in Chapter 3. We notice a slight difference in the bit-widths Figure C.2 of variables $mult2, S$ that are obtained with the quadratic programming approach compared to the other optimisation methods (Figure C.2).

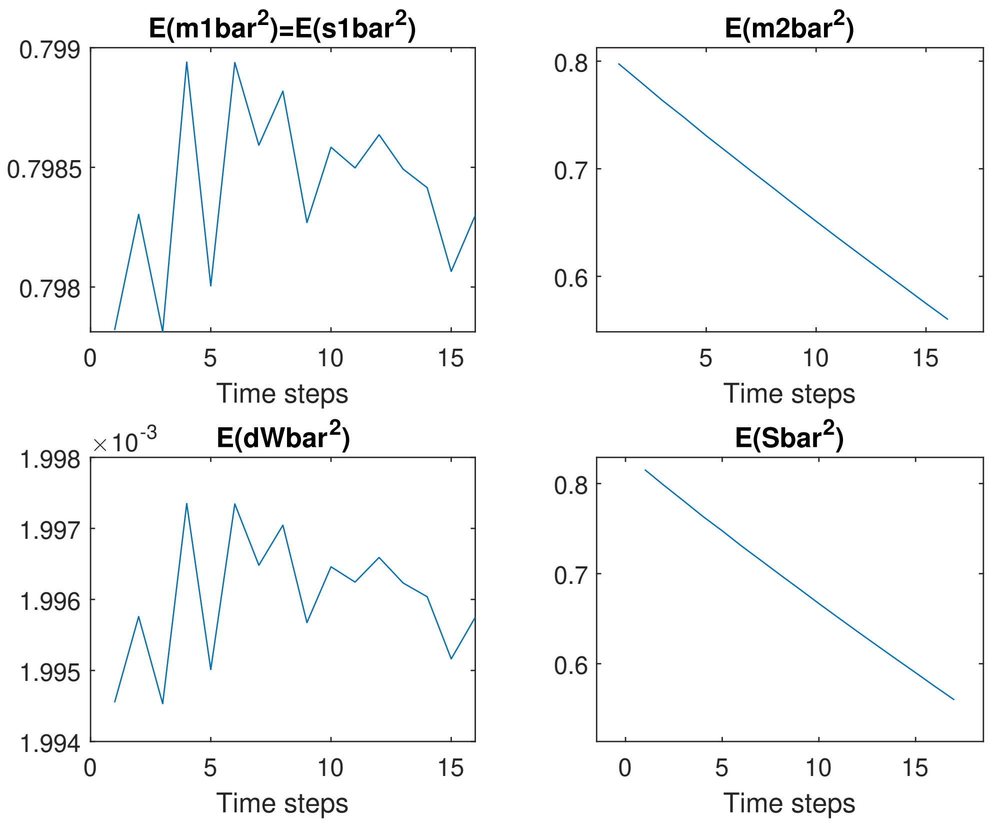

Figure C.1: $\mathbb{E}[\bar{x}^2]$ for all the intermediary variables (from Algorithm 1) over the time steps, for $N = 16$ time steps.

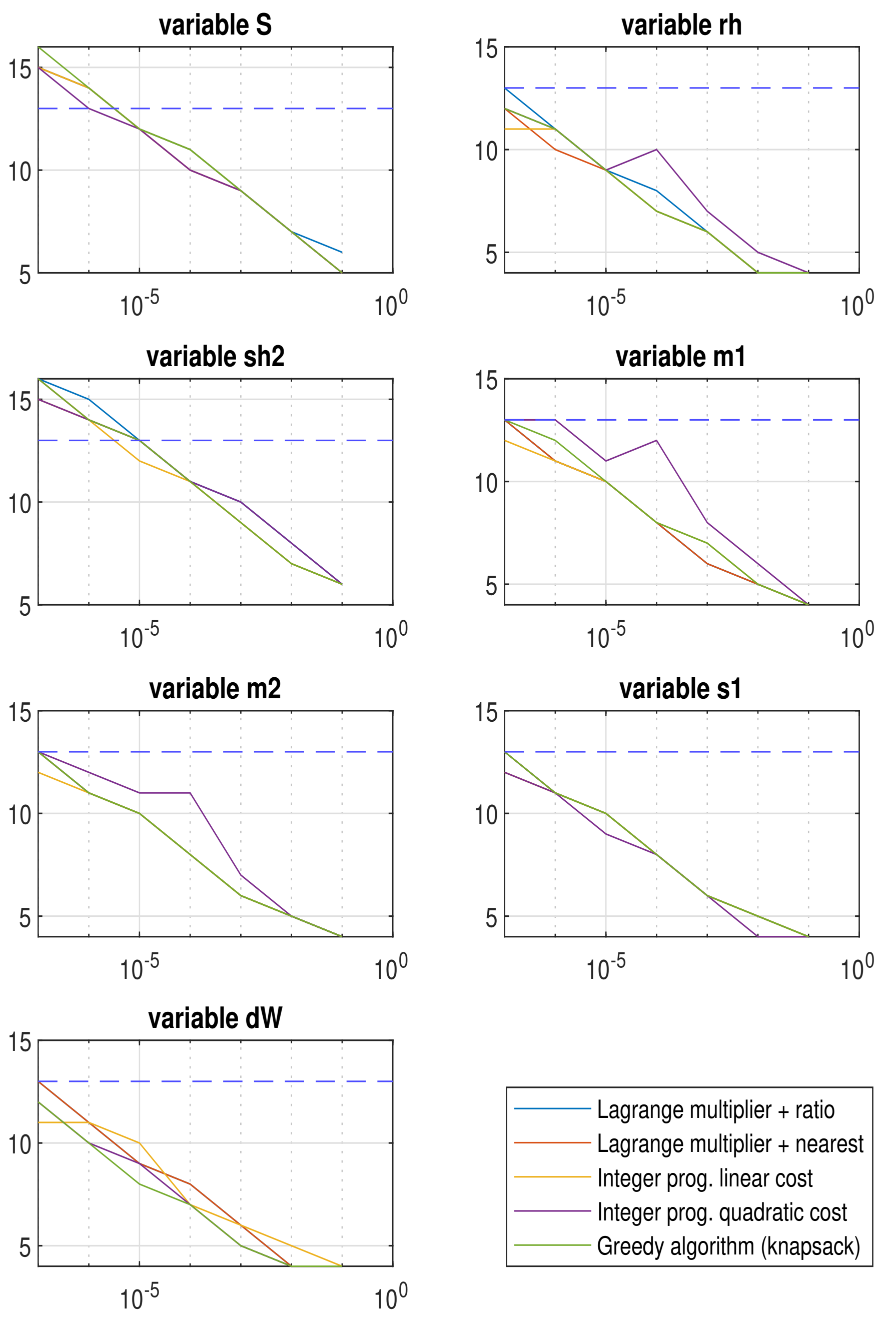

Figure C.2: Bit-widths obtained with different optimisation methods for a single level and $N = 16$ time steps.

Appendix D

Uniform precision heuristic

The most relevant existing mixed-precision MLMC framework in the literature is probably [6]. The authors chose to fix a uniform precision p_l at each level and write the telescoping sum on the expectations as

$$\mathbb{E}[P(S_L^{(p_L)})] = \mathbb{E}[P(S_0^{(p_0)})] + \sum_{l=1}^{L} \mathbb{E}[P(S_l^{(p_l)}) - P(S_{l-1}^{(p_{l-1})})]. \tag{D.1}$$

Note $V_l' = \mathbb{V}[\Delta P_l]$ the variance obtained at level l in the standard MLMC algorithm (double precision), and $V_l^p = \mathbb{V}[P(S_l^{(p_l)}) - P(S_{l-1}^{(p_{l-1})})]$ the level variance corresponding to D.1. The authors want to take p_l such that V_l^p is close to V_l', which is achieved by choosing p_l such that $\xi(p_l) < 1.1$, where the fraction ξ is defined as

$$\xi_l(p) = \frac{\mathbb{V}\left[P(S_{l+1}^{float}) - P(S_l^p)\right]}{\mathbb{V}\left[P(S_{l+1}^{float}) - P(S_l^{float})\right]}. \tag{D.2}$$

To determine p_l, for each level l the precision p_l is initialised at p_{l-1}, then the variances in D.2 are estimated with only 100 paths and p_l is incremented by one until the condition $\xi_l(p_l) < 1.1$ is met. We chose a different initialisation value for p_0 since our test example is different from what was done in [6].

Then the multilevel framework suggested in [6] proceeds as follows :

1. set the precisions $p_0, \dots, p_L$ as described previously
2. compute 10^4 paths on each level to determine the variances V_l^p
3. determine the number of paths N_l that need to be computed on each level
4. evaluate the required extra samples
5. when the algorithm has converged compute the output $\mathbb{E}[P(S_L^{(p_L)})]$.

To compare the bit-width optimisation methods introduced in Chapter 3 with this uniform bit-width heuristic we made a numerical experiment. For $N = 14$ time steps and a single level, we used the heuristic with the ratio ξ_1 to determine the uniform bit-width that all variables should have and obtained $d = 13$. Note that we considered that the normal random variables are in full precision. With this precision the variance of the error $P - \tilde{P}$ is equal to $7.9e-07$ but the FPGA cost per time step is $\tilde{C}/N = 4022$ (using the cost model that sums the squares of the bit-widths), meaning that there is a factor 2 between the cost obtained with uniform bit-widths and the optimised bit-widths (see Figure 3.3). However this is only the cost of the FPGA calculations but taking into account the fact that this heuristic uses full precision random normals, its cost per sample is actually $NC_{RNG} + \tilde{C}$. Therefore the samples computed with this framework are more expensive than the correction term samples $P - \tilde{P}$ from our framework (since in our framework $\tilde{C}$ is smaller). This remark is also valid for higher levels l.

In our framework the terms $\widetilde{\Delta P_l}$ also have variance approximately V_l' but since the variance of the correction terms is very small, the FPGA cost is smaller and the random variables used on the FPGA are generated in low precision, we believe that the total computational cost is reduced compared to the framework from [6].

Bibliography

[1] AMD. Radiation tolerant Versal AI core series data sheet (ds946), digital signal processing (dsp). Available at : `https://docs.amd.com/r/en-US/ds946-xqr-versal-ai-core/Digital-Signal-Processing-DSP`. (Accessed: 21 August 2024).

[2] AMD. Versal premium vp1902 adaptive SoC product brief. Available at : `https://www.amd.com/content/dam/amd/en/documents/products/adaptive-socs-and-fpgas/versal/2118851-versal-premium-vp1902-product-brief.pdf`. (Accessed: 4 August 2024).

[3] Armando Arciniega and Edward Allen. Rounding error in numerical solution of stochastic differential equations. *Stochastic Analysis and Applications*, 21(2):281–300, 2003.

[4] Haidar Azzam, Bayraktar Harun, Tomov Stanimire, Dongarra Jack, and Higham Nicholas J. Mixed-precision iterative refinement using tensor cores on gpus to accelerate solution of linear systems. *Proceedings of the Royal Society A*, 476(2243):20200110, 2020.

[5] Andrew Boutros, Brett Grady, Mustafa Abbas, and Paul Chow. Build fast, trade fast: FPGA-based high-frequency trading using high-level synthesis. In *2017 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, pages 1–6, 2017.

[6] C. Brugger, C. de Schryver, N. Wehn, S. Omland, M. Hefter, K. Ritter, A. Kostiuk, and R. Korn. Mixed precision Multilevel Monte Carlo on hybrid computing systems. In *Proceedings of the IEEE Conference on Computational Intelligence for Financial Engineering & Economics (CIFEr)*, pages 215–222. IEEE, 2014.

[7] Ray C. C. Cheung, Dong-U Lee, Wayne Luk, and John D. Villasenor. Hardware generation of arbitrary random number distributions from uniform distributions via the inversion method. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 15(8):952–962, 2007.

[8] Gary Chun Tak Chow, Anson Hong Tak Tse, Qiwei Jin, Wayne Luk, Philip H.W. Leong, and David B. Thomas. A mixed precision Monte Carlo methodology for reconfigurable accelerator systems. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '12, page 57–66, New York, NY, USA, 2012. Association for Computing Machinery.

[9] Richard J. Clancy, Matt Menickelly, Jan Hückelheim, Paul Hovland, Prani Nalluri, and Rebecca Gjini. Trophy: Trust region optimization using a precision hierarchy. In Derek Groen, Clélia de Mulatier, Maciej Paszynski, Valeria V. Krzhizhanovskaya, Jack J. Dongarra, and Peter M. A. Sloot, editors, *Computational Science – ICCS 2022*, pages 445–459, Cham, 2022. Springer International Publishing.

[10] K. A. Cliffe, M. B. Giles, R. Scheichl, and A. L. Teckentrup. Multilevel Monte Carlo methods and applications to elliptic PDEs with random coefficients. *Computing and Visualization in Science*, 14(1):3–15, Aug 2011.

[11] Nathan O. Collier, Abdul-Lateef Haji-Ali, Fabio Nobile, Erik von Schwerin, and Raúl Tempone. A continuation multilevel Monte Carlo algorithm. *BIT Numerical Mathematics*, 55:399 – 432, 2014.

[12] Matteo Croci, Massimiliano Fasi, Nicholas J Higham, Theo Mary, and Mantas Mikaitis. Stochastic rounding: implementation, error analysis and applications. *Royal Society Open Science*, 9(3):211631, 2022.

[13] Christian de Schryver, Pedro Torruella, and Norbert Wehn. A multi-level Monte Carlo FPGA accelerator for option pricing in the Heston model. *2013 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 248–253, 2013.

[14] A.A. Gaffar, O. Mencer, and W. Luk. Unifying bit-width optimisation for fixed-point and floating-point designs. In *12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 79–88, 2004.

[15] M. B. Giles and B. J. Waterhouse. *Multilevel quasi-Monte Carlo path simulation*, pages 165–182. De Gruyter, Berlin, New York, 2009.

[16] Michael Giles and Oliver Sheridan-Methven. Approximating inverse cumulative distribution functions to produce approximate random variables. *ACM Trans. Math. Softw.*, 49(3), sep 2023.

[17] Michael B. Giles. Multilevel Monte Carlo path simulation. *Operations Research*, 56(3):607–617, 2008.

[18] Michael B. Giles. Multilevel Monte Carlo methods. *Acta Numerica*, 24:259–328, 2015.

[19] Michael B. Giles, Mario Hefter, Lukas Mayer, and Klaus Ritter. Random bit multilevel algorithms for stochastic differential equations. *Journal of Complexity*, 54:101395, 2019.

[20] Michael B. Giles and Oliver Sheridan-Methven. Analysis of nested multilevel Monte Carlo using approximate normal random variables. *SIAM/ASA J. Uncertain. Quantification*, 10:200–226, 2021.

[21] Piyush Gupta. Introduction to FPGA basics [2023]. Available at : `https://fpgainsights.com/fpga/fpga-basics/`. (Accessed: 4 August 2024).

[22] Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. Deep learning with limited numerical precision. In *Proceedings of the 32nd International Conference on International Conference on Machine Learning - Volume 37*, ICML'15, page 1737–1746. JMLR.org, 2015.

[23] Abdul-Lateef Haji-Ali, Fabio Nobile, Erik von Schwerin, and Raúl Tempone. Optimization of mesh hierarchies in multilevel Monte Carlo samplers. *Stochastics and Partial Differential Equations Analysis and Computations*, 4(1):76–112, Mar 2016.

[24] R. Hauser. Lecture notes for the course B6.3 Integer programming. *Mathematical Institute, University of Oxford*, October 2023.

[25] Desmond John Higham, Xuerong Mao, and Andrew M. Stuart. Strong convergence of Euler-type methods for Nonlinear Stochastic Differential Equations. *SIAM J. Numer. Anal.*, 40:1041–1063, 2002.

[26] Nicholas J. Higham and Theo Mary. Mixed precision algorithms in numerical linear algebra. *Acta Numerica*, 31:347–414, 2022.

[27] Intel. Developer reference for intel® oneapi math kernel library for c. distribution generators. Available at : `https://www.intel.com/content/www/us/en/docs/onemkl/developer-reference-c/2024-1/distribution-generators.html#GUID-38330C50-A45E-403A-9ADB-7BA5D102C3E9`. (Accessed: 16 August 2024).

[28] William Morton Kahan. Further remarks on reducing truncation errors. *Communications of the Association for Computing Machinery (ACM)*, 8:40, 1965.

[29] Robert Keim. What is an FPGA? An introduction to programmable logic. Available at : `https://www.allaboutcircuits.com/technical-articles/what-is-an-fpga-introduction-to-programmable-logic-fpga-vs-microcontroller/`. (Accessed: 4 August 2024).

[30] Christian Leber, Benjamin Geib, and Heiner Litz. High frequency trading acceleration using fpgas. In *2011 21st International Conference on Field Programmable Logic and Applications*, pages 317–322, 2011.

[31] Dong-U Lee, Ray C. C. Cheung, Wayne Luk, and John D. Villasenor. Hierarchical segmentation for hardware function evaluation. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 17(1):103–116, 2009.

[32] Dong-U. Lee, Altaf Abdul Gaffar, Ray C. C. Cheung, Oskar Mencer, Wayne Luk, and George A. Constantinides. Accuracy-guaranteed bit-width optimization. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 25(10):1990–2000, October 2006.

[33] Dong-U Lee, Wayne Luk, John D. Villasenor, and Peter Y. K. Cheung. A Gaussian noise generator for hardware-based simulations. *IEEE Trans. Comput.*, 53(12):1523–1534, dec 2004.

[34] B. Lindsey, M. Leslie, and W. Luk. A Domain Specific Language for accelerated Multilevel Monte Carlo simulations. In *Proceedings of the IEEE 27th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. IEEE, 2016.

[35] Jamshaid Sarwar Malik and Ahmed Hemani. Gaussian random number generation. *ACM Computing Surveys (CSUR)*, 49:1 – 37, 2016.

[36] MathWorks. Rounding mode : Convergent. Available at : `https://de.mathworks.com/help/fixedpoint/ug/rounding-mode-convergent.html`. (Accessed: June 2024).

[37] Optimisation Toolbox example MathWorks. Mixed-integer quadratic programming portfolio optimization: Problem-based. Available at : `https://de.mathworks.com/help/releases/R2020b/optim/ug/miqp-portfolio-problem-based.html`. (Accessed: May 2024).

[38] Mutsuo Saito and Makoto Matsumoto. Simd-oriented fast mersenne twister: a 128-bit pseudorandom number generator. In Alexander Keller, Stefan Heinrich, and Harald Niederreiter, editors, *Monte Carlo and Quasi-Monte Carlo Methods 2006*, pages 607–622, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[39] Oliver Sheridan-Methven and Michael Giles. Rounding error using low precision approximate random variables. *SIAM Journal on Scientific Computing*, 2024.

[40] David B. Thomas, Lee W. Howes, and Wayne W. C. Luk. A comparison of CPUs, GPUs, FPGAs, and massively parallel processor arrays for random number generation. In *Symposium on Field Programmable Gate Arrays*, 2009.

[41] AMD Xilinx. Adaptive computing technology overview. Available at : `https://www.xilinx.com/content/dam/xilinx/publications/technology-briefs/adaptive-computing-technology-overview.pdf`. (Accessed: 4 August 2024).