An FPGA-Based Data Flow Engine For Gaussian Copula Model

Huabin Ruan*, Xiaomeng Huang†, Haohuan Fu†, Guangwen Yang*,
Wayne Luk‡, Sebastien Racaniere§, Oliver Pell§ and Wenjing Han¶
*Department of Computer Science and Technology,Tsinghua University, Beijing, China
†Center for Earth System Science, Tsinghua University, Beijing, China
‡Department of Computer Engineering, Imperial College London, London, Britain
§Maxeler Technologies, London, Britain
¶School of Computer Science and Technology, Harbin Institute of Technology, Harbin, China
*Email:ruanhuabin@gmail.com, ygw@tsinghua.edu.cn

***Abstract*—The Gaussian Copula Model (GCM) plays an important role in the state-of-the-art financial analysis field for modeling the dependence of financial assets. However, the existing implementations of GCM are all computationally-demanding and time-consuming. In this paper, we propose a Dataflow Engine (DFE) design to accelerate the GCM computation. Specifically, a commonly used CPU-friendly GCM algorithm is converted into a fully-pipelined dataflow graph through four steps of optimization: recomposing the algorithm to be pipeline-friendly, removing unnecessary computation, sharing common computing results, and reducing the computing precision while maintaining the same level of accuracy for the computation results. The performance of the proposed DFE design is compared with three CPU-based implementations that are well-optimized. Experimental results show that our DFE solution not only generates fairly accurate result, but also achieves a maximum of 467x speedup over a single-thread CPU-based solution, 120x speedup over a multi-thread CPU-based solution, and 47x speedup over an MPI-based solution.**

***Keywords*-Gaussian Copula Model; DFE; FPGA;**

I. Introduction

Gaussian Copula Model (GCM) is a kind of distribution function for modeling the dependence between random variables [1]. It has been widely used for modeling the dependence between financial assets in the community of finance analysis [2], [3]. As reported by Y. Malevergne *et al.* [3], most pairs of currencies and pairs of major stocks are compatible with the Gaussian Copula hypothesis, while this hypothesis can be rejected for the dependence between pairs of commodities. In the process of modeling the dependence between assets, GCM always shows excellent ability on the separation between the marginal distributions and the dependence, since it allows for testing several scenarios with different kinds of dependence between assets while the marginals can be set to their well-calibrated empirical estimates [3]. And as we know, this ability is potentially very useful for risk management, option pricing, and sensitivity analysis, or other financial analysis topics. Until now, a lot of work has been done on the base of GCM due to its above ability. Embrechts *et al.* [4] provided various bounds for the value-at-risk of a portfolio made of dependent risk using GCM. In addition, Rosenberg and Cherubini *et al.* [5] employed GCM to price and analyze the pricing sensitivity of binary digital options or options on the minimum of a basket assets as well.

However, the implementations of the GCM algorithm proposed in previous work are all considerably computationally-intensive and time-consuming. In general, based on those implementations, before getting the final output for every input instance (i.e., a vector of random variables), hundreds of arithmetic operations have to be performed. Moreover, with the amount of input data increases, the computing cost will also rise proportionally. In recent years, how to reduce the high computational intensity and accelerate the GCM computation has become to an urgent problem due to two demands: first, with the development of internet technology and the increasing focus on financial simulation research, a demand for applying the GCM to process the massive data collected from the internet arises. Most of the existing implementations are too computationally demanding to meet this demand. Second, in order to meet the fast response requirement in some special scenarios (e.g. real-time decision making on financial assets portfolio), the GCM algorithm will be asked to be executed within a short period of time on a large amount of input data.

To deal with the demands mentioned above, this paper focuses on a GCM design that has relatively low resource costs and high performance at the same time. Specifically, an FPGA-based Dataflow Engine (DFE) design is proposed. It is a fast GCM calculation engine running on a Maxeler Technologies MAX3 DFE, and capable of handling all the time-consuming computation operations in a fully-pipelined manner. Our major contributions are:

- We propose four effective optimization strategies used for optimizing computationally-intensive algorithm on the DFE so as to reduce the hardware resource consumption, and to enable the mapping of the algorithm into the available chip logic.
- We design and implement the GCM algorithm on the DFE platform by applying our proposed optimization strategies, and achieve significant acceleration over

well-optimized CPU versions.

- We present a systematic comparison of the performance of our DFE GCM solution with three other highly optimized CPU-based GCM solutions. Our solution shows significantly improved performance.

The rest of this paper is organized as follows. Section II presents a background introduction of GCM. Section III introduces four effective optimization strategies for the DFE algorithm implementations. Section IV describes the process of our DFE implementation for GCM algorithm. Then Section V presents our experimental results and discussions before conclusion introduced in Section VI.

II. Background

A. Algorithm implementation of Gaussian Copula Model

According to Sklar's theorem [6], for every $(x_1, x_2, \cdots, x_d) \in \mathbb{R}^d$, a cumulative distribution

$$H(x_1, x_2, \cdots, x_d) = \mathbb{P}[X_1 \leq x_1, X_2 \leq x_2, \cdots, X_d \leq x_d] \tag{1}$$

of a random vector $(X_1, X_2, \cdots, X_d)$ with marginals $F_i(x_i) = \mathbb{P}(X_i \leq x_i), 1 \leq i \leq d$ can be written as

$$H(x_1, x_2, \cdots, x_d) = C\big(F_1(x_1), F_2(x_2), \cdots, F_d(x_d)\big), \tag{2}$$

where C is a copula. The "copula" was named for its resemblance to grammatical copulas in linguistics. It contains all information on the dependence structure between the components of $(X_1, X_2, \cdots, X_d)$.

If we denote $u_i = F_i(x_i), 1 \leq i \leq d$, we can then represent x_i as:

$$x_i = F_i^{-1}(u_i), \tag{3}$$

where F_i^{-1} is the inverse of F_i. Consequently, the copula C can also be written as

$$\begin{aligned} C\big(u_1, u_2, \cdots, u_d\big) = \mathbb{P}[X_1 \leq F_1^{-1}(u_1), \\ X_2 \leq F_2^{-1}(u_2), \cdots, X_d \leq F_d^{-1}(u_d)]. \end{aligned} \tag{4}$$

The Gaussian Copula Model (GCM) used in this work is a distribution over the unit cube $[0,\ 1]^d$. It is constructed from a multivariate normal distribution over $\mathbb{R}^d$ by using the probability integral transform. For a given correlation matrix $\Sigma \in \mathbb{R}^{d*d}$, the GCM $C_\Sigma^{Gaussian}$, $[0,\ 1]^d \rightarrow [0,\ 1]$ for random vector $(X_1, X_2, \cdots, X_d)$ can be written as

$$\begin{aligned} C_\Sigma^{Gaussian}(u_1, u_2, \cdots, u_d) = \\ \mathbb{P}[X_1 \leq \Phi^{-1}(u_1), X_2 \leq \Phi^{-1}(u_2), \cdots, X_d \leq \Phi^{-1}(u_d)] \\ = \Phi_\Sigma\big(\Phi^{-1}(u_1), \Phi^{-1}(u_2), ..., \Phi^{-1}(u_d)\big), \end{aligned} \tag{5}$$

where Φ^{-1} is the inverse cumulative distribution function (ICDF) of a standard normal, Φ_Σ is the cumulative distribution function (CDF) of a multivariate (e.g. bivariate in our study) normal distribution with mean vector zero and covariance matrix equal to the correlation matrix Σ.

As shown in Eq. (5), the algorithm of GCM is governed by the ICDF $\Phi^{-1}(\cdot)$ and the CDF $\Phi(\cdot)$. Their classic algorithm implementations are shown as follow, respectively:

1) Inverse Cumulative Distribution Function (ICDF) Algorithm: As mentioned above, Φ^{-1} is the ICDF of a standard normal. For a given $u_i \in [0,\ 1]$, the goal of our ICDF Algorithm is to calculate the value of

$$x_i = \Phi^{-1}(u_i). \tag{6}$$

Here we present the pseudocode of a commonly used ICDF algorithm in Algorithm 1 [7]. Our pseudocode exposes sufficient details of the algorithm's implementation, including the patterns in memory access and computations. From Algorithm 1, we can see clearly that from line 3 to line 13, it contains massive basic arithmetic operations, such as $+$, $-$, $\times$, $\div$. In addition, extremely time-consuming basic math function procedures are also called, like exponent function $\exp(\cdot)$, logarithmic function $\log(\cdot)$, square root function $\mathrm{sqrt}(\cdot)$. All these basic arithmetic operations and math functions result in a computation-intensive ICDF algorithm.

Algorithm 1 The ICDF algorithm of random variable X_i, with u_i as input.

```
Require: Input $u_i$ should fall in $[-1,\ 1]$.
1:  $d = dd = 0.0$, $u_i = 1.0 - u_i$;
2:  $pp = (u_i \geq 0)\ ?\ u_i : 2. - u_i$;
3:  $t_0 = \mathrm{sqrt}(-2.0 \times \log(pp/2.0))$;
4:  $x_i = -0.70711 \times ((2.30753 + t \times 0.27061)/(1.0 + t_0 \times (0.99229 + t \times 0.04481)) - t_0)$;
5:  for (Int $j = 0$; $j < 2$; $j++$) do
6:      for (Int $k = 27$; $k > 0$; $k--$) do
7:          $tmp = d$, $d = ty \times d - dd + cof[k]$, $dd = tmp$;
8:      end for
9:      $t = 2.0/(2.0 + x_i)$;
10:     $erfchebval = t \times \exp(-x_i \times x_i + 0.5 \times (cof[0] + ty \times d) - dd)$;
11:     $erfvalue = x_i > 0.0\ ?\ erfchebval : 2.0 - erfchebval$;
12:     $err = erfvalue - pp$;
13:     $x_i += err/(1.12837916709551257 \times \exp(-\mathrm{sqrt}(x_i)) - x_i \times err)$;
14: end for
15: return $u_i \geq 0.0\ ?\ x_i : -x_i$;
```

2) Cumulative Distribution Function (CDF) Algorithm : As shown in Eq. (5), we can obtain CDF of any number of random variable (i.e., the number of dimensions of random vector) in theory, but the algorithm implementation of CDF with normal distribution of more than two random variables is not generally available in practice [8]. Therefore, in this paper we focus on the design and implementation of CDF algorithm of two random variables, namely cumulative bivariate normal distribution function (CBNDF). The CBNDF for random variables $(X_1,\ X_2)$ is defined as follow:

$$\Phi(x_1, x_2, \rho) = \mathbb{P}(X_1 \leq x_1, X_2 \leq x_2, \rho) = \frac{1}{2\pi\sqrt{1-\rho^2}} * \int_{-\infty}^{x_1}\int_{-\infty}^{x_2} \exp\left(\frac{-(X_1^2 - 2\rho X_1 X_2 + X_2^2)}{2(1-\rho^2)}\right) \mathrm{d}X_2 \mathrm{d}X_1 \quad (7)$$

However, a closed form solution does not exist for this integral, so a numerical approximation is required [8]. Here we implemented the CDF algorithm according to Genz *et al.*'s work [9] (See Algorithm 2). As can been seen in Algorithm 2, line 11, 17, 19, 29 all contain a procedure call named *Cumnorm*$(\cdot)$. It is a cumulative univariate normal distribute function (CUNDF) for random variable X:

$$\Phi(x) = \mathbb{P}(X \leq x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{x} \exp\left(\frac{-(X^2)}{2}\right) \mathrm{d}X. \quad (8)$$

Again, an approximation is required for Eq. (8) as well [8]. In our case, in order to reach the double precision, we implemented the approximate algorithm according to Hart's [10] solution, which can be found in Fig.2 of literature[8] .

By now, based on the Algorithm 1 for Φ^{-1} and Algorithm 2 for Φ, we can then start the work on mapping the GCM into FPGA hardware logic. And how to overcome the resource limitation of FPGA hardware architecture becomes the key issue during the mapping.

B. Related Work

There have been a number of attempts to accelerate financial applications on the heterogeneous system where a typical processor is paired with a hardware accelerator device like FPGA and Graphical Processing Unit (GPU). One of the most targeted classes of applications in the financial world is option pricing, which gives the holder the right to either buy or sell an asset by a certain date for a set price. For instance, Baxter *et al.* [11] present Monte-Carlo Asian option pricing on a 64 Xilinx Virtex 4 FX100 FPGA supercomputer with 32 fully interconnected 2.8Ghz Xeon processors, and achieve 322 times faster than their corresponding software implementation. McCool *et al.* [12] implemented single precision floating point European option pricing on an NVIDIA 7900 GTX GPU, and obtain 120 times faster than their software implementation. Morris *et al.* [13] present a comprehensive comparison for European option pricing between an FPGA, a GPU and a IBM acceralor Cell BE, and found that the single-precision floating point implementations displayed similar acceleration across the three different platforms: 41-fold on an FPGA, 32-fold on an NVIDIA 7900 GTX GPU, and 29-fold on a Cell BE. Due to the use of different processor architecture and the lack of a standardized reference code or benchmarks, here we do not compare above financial applications' merit in terms of performance, we just report the results.

Algorithm 2 The CDF algorithm of random variables X_1 and X_2, with X_1 and X_2's ICDF results x_1 and x_2, and determinant value r of correlation matrix Σ as input.

```
1:  $h1 = x_1$, $h2 = x_2$, $h12 = (h1 \times h1 + h2 \times h2)/2$;
2:  if abs$(r) \geq 0.7$ then
3:    $r2 = 1 - r \times r$, $r3 = \mathrm{sqrt}(r2)$;
4:    if $r < 0$ then
5:      $h2 = -h2$;
6:    end if
7:    $h3 = h1 \times h2$, $h7 = \exp(-h3/2)$, ;
8:    if abs$(r) < 1$ then
9:      $h6$ =abs$(h1 - h2)$, $h5 = h6 \times h6/2$, $h6 = h6/r3$;
10:     $AA = 0.5 - h3/8$, $ab = 3 - 2 \times AA \times h5$;
11:     $LH = 0.13298076 \times h6 \times ab \times (1 - Cumnorm(h6)) - \exp(-h5/r2) \times (ab + AA \times r2) \times 0.053051647$;
12:     for $i = 1$ To $5$ do
13:       $r1 = r3 \times x(i)$, $rr = r1 \times r1$, $r2 = \mathrm{sqrt}(1 - rr)$;
14:       $LH = LH - W(i) \times \exp(-h5/rr) \times (\exp(-h3/(1 + r2))/r2/h7 - 1 - AA \times rr)$;
15:     end for
16:   end if
17:   $biCDF = LH \times r3 \times h7 + Cumnorm(Min(h1, h2))$;
18:   if r $< 0$ then
19:     $biCDF = Cumnorm(h1) - biCDF$;
20:   end if
21: else
22:   $h3 = h1 \times h2$;
23:   if $r \neq 0$ then
24:     for $i = 1$ To $5$ do
25:       $r1 = r \times x(i)$, $r2 = 1 - r1 \times r1$;
26:       $LH = LH + W(i) \times \exp((r1 \times h3 - h12)/r2)/\mathrm{sqrt}(r2)$;
27:     end for
28:   end if
29:   $biCDF = Cumnorm(h1) \times Cumnorm(h2) + r \times LH$;
30: end if
31: return $biCDF$
```

III. Four Effective Optimization Strategies towards DFE Implementations

We utilize a Maxeler Technologies Dataflow Engine which contains a large FPGA to perform computation. The FPGA is a reconfigurable architecture featured by plenty of functional units. Different from CPU who processes computation tasks by reusing a limited number of functional units over time, the DFE works in a fully pipelined manner over space. Specifically, for a specific computation task, the chip's functional units are used to form a hardware logic to organize data into streams which flow through those functional units. Therefore, given this working mechanism difference between CPU and DFE, if we want to map CPU-friendly algorithms to a DFE, some adjustments are naturally expected. In this section, four optimization strategies are proposed to guide the mapping, and their detail usage will be presented in Section IV.

A. Recomposing Algorithm To Pipeline-Friendly Manner

Two conditions need to be guaranteed at the same time in order to map an algorithm onto a FPGA platform: (1) The number of required functional units of the algorithm to construct pipeline should not exceed the number of available functional units of FPGA, and (2) no data dependency exists to block the pipeline process. Once an algorithm is too computation intensive to meet above conditions, two means of solutions are proposed here: (1) Reselect an equivalent algorithm consuming rather fewer functional units, even at the cost of precision reducing within the acceptable range. (2) Restructure the work flow of original algorithm by using the methods like reordering the loop and input data to meet the minimum FPGA timing schedule requirement, changing the number representation format like substituting floating point with fixed point to reduce computation latency, so that data dependency can be eliminated, then be possible mapped to a pipeline manner in FPGA architecture. In certain extent, algorithm recomposing is the first step in considering mapping an algorithm to FPGA hardware architecture.

B. Removing Unnecessary Computation

As mentioned before, completely different from the CPU architecture which computes calculations in time, the FPGA computes calculations in pipeline manner in space. However, although this pipeline working style can result in high computation efficiency, it will additionally lead to redundant computation in some cases. For instance, if the same function call with different input arguments is deployed in each selection path of a IF-ELSE conditional structure, this function will be executed twice simultaneously for pipelined implementation on FPGA, though only one output is valid actually. In other words, a waste of functional units for producing a invalid output is caused. Therefore, for mapping code segments with above feature on FPGA architecture, what we can do is to calculate input arguments or operands in advance according to context of current condition, storing all possible input arguments or operands in advance for similar code segments, then using multiplexer (MUX) to select one of those possible arguments or operands according to value of current condition combination as code segments input, so that we can just use one copy of functional units instead of multiple copies to implement the similar code segments in conditional structure in FPGA logic. We name this method as removing unnecessary computation. It can help to save a considerable number of functional units in the case massive similar calculations appear in conditional structures, which is very common in certain applications such as financial simulation and machine learning.

C. Sharing Common Computing Result

In CPU architecture, in order to make implementation of algorithm which looks more logical and clarity, some exactly same code segments producing same output are often deployed in multiple conditional structures. There is no performance degradation for implementation under CPU. But in the context of FPGA-based implementation, deploying those same code segments in hardware means performance loss because multiple copies of functional units for deploying the same code segment can be used for adding more pipeline and achieving further acceleration . Therefore, in FPGA architecture, what we should do is to extract those same code segments, so that we can just use one copy of functional units for those same computations, and then share the output result of this one copy in left conditional structures. We name this optimization method as sharing common computing result, which is also frequently seen in many algorithm implementations.

D. Reducing Algorithm Precision

In most cases, the use of optimization strategies described above can save a great number of FPGA functional units, and make the originally unsatisfying algorithms possible to be implemented in a pipeline manner. However, there always exists some fairly computation intensive algorithms which require a considerable number of FPGA functional units, so that even above optimization strategies can not help to meet the pipeline construction requirement. In this case, we propose to reduce the algorithm's precision into an acceptable range by the way of changing float number representation. For instance, changing floating point numbers to another lower resource cost format like using fixed point numbers, or reducing the mantissa bitwidth. We name this method as reducing algorithm precision. For computational intensive algorithms, it can reduce a significant number of FPGA functional units in most cases, making the algorithm possible to be implemented in a more pipeline friendly manner or further achieve performance.

We can use any combination of the above optimization strategies to reduce algorithms' functional units usage on FPGA, so that computational intensive applications can be mapped into a pipeline-friendly manner to achieve acceleration. Even if we have richer functional units in future FPGA chips, these strategies also makes sense since saved functional units can be used for deploying multiple pipelines, making applications achieving further acceleration.

IV. HARDWARE IMPLEMENTATION OF GAUSSIAN COPULA MODEL

In this section, we focus on the implementation of GCM algorithm which will be deployed on a Maxeler Technologies MAX3 DFE with one Virtex-6 SX475T FPGA. Hardware resource provided by the Virtex-6 SX475T FPGA can be found from the second column of Table I. We use

Maxeler's MaxCompiler tool suite to program the GCM algorithm in Java, and MaxCompiler compiles and builds the Java code into a DFE configuration,. including FPGA bitstream. The generated configuration file (`.max` file) can be loaded into the Maxeler DFE card at runtime.

From the description in Section II, we understand clearly that the commonly used GCM algorithms nowadays are mainly designed towards CPU-based implementation, and featured by a large number of complicated arithmetic operations (e.g., $\times$, $\div$, $\exp(\cdot)$, $\log(\cdot)$). A comparison of the available resources of FPGA and the required resources of the algorithm described in Section II is shown in Table I. Obviously, if we simply migrate the original GCM algorithm to our FPGA chip, it will fail because of the overranging resources consumption. In order to overcome above resource limitation and then convert the GCM into a pipeline-friendly hardware logic, here we apply four optimization strategies proposed in previous section step by step as follow.

Table I
COMPARISON OF THE AVAILABLE RESOURCES OF THE FPGA CHIP WITH THE REQUIRED RESOURCES OF THE ORIGINAL GCM ALGORITHM.

Resource Name	Available	Required / Percent
Lookup Tables(LUTs)	297600	593772 / 199.52%
Flip-Flops(FFs)	595200	935178 / 157.12%
Block RAMs(BRAMs)	1064	220 / 20.67%
DSPs	2016	3039 / 150.74%

A. Step 1: Recomposing ICDF Algorithm

For the implementation of GCM algorithm processing 2-dimension input vector, the subfunction ICDF (see Algorithm 1) will be invoked twice, resulting in more than 400 basic arithmetic operations, and leading to a large scale of functional units consumption. More specifically, to obtain a FPGA-based implementation, twice invoking of that ICDF algorithm requires *77.81% (231563)* Lookup Tables (LUTs), *53.82% (320336)* Flip-Flops (FFs), *3.0% (32)* Block RAMs (BRAMs), and *47.62% (960)* DSPs of our FPGA chip. Here we replace current ICDF algorithm with Acklam ICDF method [14](see Algorithm 3) to reduce the computation resource spent on ICDF, so as to reduce the global computation resource of GCM. The Acklam ICDF provides a Maximum Relative Error smaller than 1.15×10^{-9}, but consumes much less functional units compared to the original one. Specifically, it consumes *26.44% (78606)* LUTs, *17.44% (105588)* FFs, *4.7% (50)* BRAMs, *18.15% (366)* DSPs of FPGA resources. Obviously, compared to the original ICDF algorithm, the Acklam ICDF algorithm saves a great number of functional unit. The resources saved by using Acklam ICDF component and the required resources for the GCM algorithm can be found in Table II.

Here we give a brief description on why Acklam can save such a lot of resources. This algorithm uses two separate rational minimax approximations (RMAs). One is used for the central region ($0.02425 \leq u_i \leq 1 - 0.02425 = 0.97575$) and another one is used for the tails ($u_i < 0.02425$ *or* $u_i > 0.97575$). Based on above two RMAs, the Acklam ICDF algorithm only contains 60 arithmetic operations, which are much less than the 200 arithmetic operations requirement by the original one.

Algorithm 3 Acklam ICDF algorithm with u_i as input and x_i as output, where a, b, c, d are arrays with constant values.

```
1: $u_i\_low = 0.02425, u_i\_high = 1 - u_i\_low$;
2: if $0 < u_i < u_i\_low$ then
3:     $q = \text{sqrt}{-2} \times \log(u_i)$;
4:     $x_i = (((((c(1) \times q + c(2)) \times q + c(3)) \times q + c(4)) \times q + c(5)) \times q + c(6))/((((d(1) \times q + d(2)) \times q + d(3)) \times q + d(4)) \times q + 1)$;
5: end if
6: if $u_i\_low \leq u_i \leq u_i\_high$ then
7:     $q = u_i - 0.5, r = q \times q$;
8:     $x_i = (((((a(1) \times r + a(2)) \times r + a(3)) \times r + a(4)) \times r + a(5)) \times r + a(6)) \times q/(((((b(1) \times r + b(2)) \times r + b(3)) \times r + b(4)) \times r + b(5)) \times r + 1$;
9: end if
10: if $u_i\_high < u_i < 1$ then
11:     $q = \text{sqrt}(-2 \times \log(1 - u_i))$;
12:     $x_i = -(((((c(1) \times q + c(2)) \times q + c(3)) \times q + c(4)) \times q + c(5)) \times + c(6))/((((d(1) \times q + d(2)) \times q + d(3)) \times q + d(4)) \times q + 1)$;
13: end if
```

B. Step 2: Removing Unnecessary Computations

Though the Acklam ICDF algorithm saves a great number of FPGA functional units, it still contains unnecessary computations inside. If we look deeply inside of Algorithm 3, we can observe that the computation processes in central and tails regions are almost the same except the different coefficients. Here we use multiplexer (MUX) to select corresponding coefficients, then those six polynomial calculations and three division calculations are reduced to two and one, respectively. After multiplex these coefficients, the new calculation process of Acklam ICDF algorithm is shown in Algorithm 4, which contains only one division and two polynomials, leading to further reducing on the FPGA functional units usage. To be specific, the optimized Acklam ICDF algorithm only consumes *12.4% (36902)*

Table II
RESOURCES SAVED BY USING ACKLAM ICDF COMPONENT AND STILL REQUIRED BY THE GCM ALGORITHM AFTER STEP 1.

Resource	Available	Saved / Percent	Required / Percent
LUTs	297600	152957 / 51.40%	440815 / 148.12%
FFs	595200	214748 / 36.08%	720430 / 121.04%
BRAMs	1064	0 / 0%	238 / 22.37%
DSPs	2016	594 / 29.46%	2445 / 121.28%

LUTs, *8.04% (47854)* FFs, *0.75% (8)* BRAMs, *7.74% (156)* DSPs of FPGA resources, and saves much more functional units than unoptimized one.

Algorithm 4 Optimized Acklam ICDF algorithm with input u_i and output x_i.

```
1: $islow = u_i < u_i\_low, isupper = u_i > u_i\_high;$
2: $isbnd = islow || isupper, iscentral = \sim isbnd;$
3: $pp = (isupper)?(1.0 - u_i) : u_i;$
4: $q = (iscentral)?(u_i - 0.5) : sqrt(-2.0log(pp));$
5: $r = q \times q, x_i = iscentral?r : q; mult = iscentral?q : 1;$
6: for $inti = 1; i \leq 6; ++i$ do
7:     $num\_c[i] = iscentral?a[i] : c[i];$
8: end for
9: $numerator = mult \times (num\_c[6] + x_i \times (num\_c[5] + x_i \times (num\_c[4] + x_i \times (num\_c[3] + x_i \times (num\_c[2] + x_i \times num\_c[1])))));$
10: for $inti = 5; i \geq 1; --i$ do
11:     $denom\_c[i] = iscentral?b[i] : d[i-1];$
12: end for
13: $denominator = ((((denom\_c[1] \times x_i + denom\_c[2]) \times x + denom\_c[3]) \times x_i + denom\_c[4]) \times x_i + denom\_c[5]) \times x_i + 1;$
14: $result = numerator/denominator;$
15: $x_i = isupper? - result : result;$
```

Unnecessary computations can also be found during the calling of univariate normal distribution function $Cumnorm(\cdot)$ in the IF-ELSE structure of Algorithm 2: $Cumnorm(Min(h1, h2))$ in Line 17 and $Cumnorm(h2)$ in Line 29. The only difference between two functions is their input parameter. If we simply project these two callings to FPGA hardware logic, two copies of hardware resource for $Cumnorm(\cdot)$ will be consumed simultaneously, though for every input only one output of above two functions is valid. In our case, we are able to determine the input parameter for function $Cumnorm(\cdot)$ in advance by the combination of following conditions: (1) $|r| \geq 0.7$, (2) $r < 0$, (3) $h1 < h2$, (4) $h1 < -h2$. Specifically, this can be done by saving all possible values in advance according to result of these conditions combination, and then using MUX to select a valid input value. Table III shows the boolean combination of above four conditions and the corresponding input value for $Cumnorm(\cdot)$. Through calculating input parameters for $Cumnorm(\cdot)$ in advance, we finally save a copy of FPGA resources spent on $Cumnorm(\cdot)$.

Some other unnecessary computations can still be found in Algorithm 2 such as calculating result for r1 in Line 13 and Line 25, which also can be removed by using MUX. After removing all these unnecessary computations described above in GCM algorithm, we save a great number of functional units again. We show the total hardware resource saved by this step in Table IV.

C. Step 3: Sharing Common Computing Result

It can be seen from Algorithm 2 that function call $Cumnorm(h1)$ distributed in conditional structure in Line 19 and Line 29 is executed twice, but producing same result. The same case is happened in $r1 \times r1$ in Line 13 and Line 25. Because those code segments produce same result, we can extract those same code segments out of conditional structure so as to avoid the repeated calculation in FPGA and share the result in needed places. After sharing the result of $Cumnorm(h1)$ and $r1 \times r1$, we further reduce the resource usage recorded in Table V.

D. Step 4: Reducing Float-Point Number Precision

So far, we have reduced a great number of functional units for hardware implementation of GCM algorithm, but we still do not have enough FPGA resources for GCM algorithm. For instance, we still need 123.68% LUTs which exceeds the

Table III
POSSIBLE INPUT VALUES FOR CUMNORM(MIN(H1,H2)) AND CUMNORM(H2) .

Conditions				Input Value
|r| ≥ 0.7	r < 0	h1 < h2	h1 < -h2	
0	0	0	0	h2
0	0	0	1	h2
0	0	1	0	h2
0	0	1	1	h2
0	1	0	0	h2
0	1	0	1	h2
0	1	1	0	h2
0	1	1	1	h2
1	0	0	0	h2
1	0	0	1	h2
1	0	1	0	h1
1	0	1	1	h1
1	1	0	0	-h2
1	1	0	1	h1
1	1	1	0	-h2
1	1	1	1	h1

Table IV
RESOURCES SAVED BY REMOVING UNNECESSARY COMPUTATIONS AND STILL REQUIRED BY THE GCM ALGORITHM AFTER STEP 2.

Resource	Available	Saved / Percent	Required / Percent
LUTs	297600	59312 / 19.93%	381503 / 128.19%
FFs	595200	94935 / 15.95%	625495 / 105.09%
BRAMs	1064	63 / 5.92%	175 / 16.45%
DSPs	2016	379 / 18.80%	2066 / 102.48%

Table V
RESOURCES SAVED BY SHARING COMMON COMPUTING RESULT AND STILL REQUIRED BY THE GCM ALGORITHM AFTER STEP 3.

Resource	Available	Saved / Percent	Required / Percent
LUTs	297600	13422 / 4.51%	368081 / 123.68%
FFs	595200	18273 / 3.07%	607222 / 102.02%
BRAMs	1064	10 / 0.94%	165 / 15.51%
DSPs	2016	51 / 2.53%	2015 / 99.95%

maximum available LUTs shown in Table V. Note that the Acklam ICDF algorithm provides a maximum relative error smaller than 1.15×10^{-9} meaning that the final relative error of output of the GCM algorithm will not smaller than 1.15×10^{-9}, so it is unnecessary to use IEEE-754 double precision float-point number representation in GCM algorithm. In our case, we change the mantissa bits of float-point number from 53 bits to 36 bits, and sufficient experimental test results show that using float-point number with 36 bits mantissa can still obtain a maximum relative error smaller than 4×10^{-9} in GCM algorithm. More important, by reducing the mantissa bits size to 36, we save a great number of FPGA resources again. The final resource usage is shown in Table VI, from which one can observe that all FPGA resource items required by GCM algrithm are under 52% of the available items, meaning that sufficient FPGA hardware resources can be offered to form the hardware logic for GCM algorithm by now.

Table VI
FINAL RESOURCE USAGE BY GCM.

Resource Name	Available	Finally Required / Percent
LUTs	297600	153234 / 51.49%
FFs	595200	249329 / 41.89%
BRAMs	1064	44 / 4.14%
DSPs	2016	399 / 19.79%

V. EXPERIMENTS AND RESULTS

A. Experimental Settings

With the purpose of performance comparison, three different CPU-based implementation versions of GCM algorithm are also performed in addition to our FPGA-based implementation, including (1) a Single-Thread CPU (S-CPU) version, (2) a Multi-Thread CPU (M-CPU) version, and (3) a MPI version. Specific settings for running each version of implementation are presented as follow: The S-CPU and M-CPU versions are deployed in a PC workstation powered by a 2.93GHz quad-core Intel i7 CPU with a 4GB DDR3 memory. The MPI version is deployed over a 16-node cluster with Intel MPI Library configured at highest optimization level, and each node powered by a 2.93 GHz quad-core Intel i7 CPU with 4GB DDR3 memory. Moreover, this version is performed with 64 MPI processes, which are the maximum physical processes can be forked in our 16-node cluster. Though better performance can be expected with more physical processes forked in other larger scale cluster, much more power will be also required at the same time compared to the FPGA-based implementation. Therefore, here we run the MPI-based implementation in a limited number of processes (64 MPI processes in our case) only for simple reference purpose, rather than discussing the performance of scalable MPI solution. Additionally, above three versions of implementation are all coded in the C program language and compiled with the Intel C Compiler configured at highest compiler optimization level. The DFE implementation is deployed onto a Maxeler DFE card with a Xilinx Virtex-6 SX475T FPGA running at 175MHz and 24GB DDR3 onboard memory.

These four GCM implementations are then performed on four data sets, which are generated following the example of data sets from real-world adopted in [15]. The size and precision of these data sets are presented in Table VII.

Table VII
AN OVERVIEW OF FOUR EXPERIMENTAL DATA SETS.

Data Set ID	Number of Instances	Precision of Float Type (bits)	
		Exponent	Mantissa
1	1.6×10^7	8	24
2	3.2×10^7	8	28
3	1.6×10^8	11	36
4	3.2×10^8	11	53

Moreover, two measures are calculated to evaluate the FPGA-based implementation's efficiency from different aspects, namely Maximum Relative Error (MRE) and Speedup: (1) As we calculate the MRE on a given data set, the same outputs produced by these three CPU-based implementations are considered as reference values. So the MSE actually describes the output difference between the CPU-related implementations and the FPGA-based implementation, and the smaller the better. (2) And the speedup represents how fast the FPGA-based implementation is in contrast to other implementations.

B. Results and Discussion

We first illustrate the MREs of the DFE implementation on four data sets in Fig 1. We can see clearly that, all these MREs are between 1.71×10^{-9} and 3.74×10^{-9}, which are rather small and acceptable for most GCM-based applications. This suggests that our optimization strategies for DFE GCM implementation is rather effective.

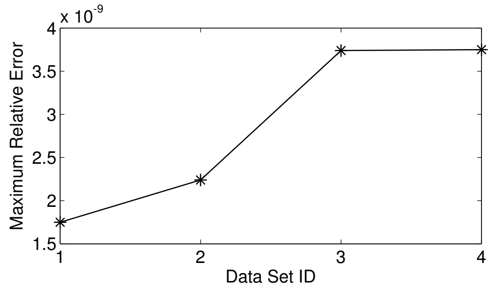

Figure 1. Maximum Relative Errors of DFE GCM implementation on four data sets with different sizes and precisions.

Next, we present the throughput (i.e., number of processed instances per second) of each version of implementation in Table VIII, and the speedups of DFE version over other versions in Table IX. As shown in Table VIII, the throughput of the DFE implementation stays at a constant level around

1.7×10^8 instances per second. As the FPGA runs at 175MHz, this performance implies that the system is able to process an instance per clock cycle without being confronted with memory bottlenecks since the bandwidth of PCIe is already enough for the data transmission in our case. And as shown in Table IX, our DFE implementation is significantly more efficient than others. Moreover, with the increasing of the size of data set, the speedup is obviously keeping rising. In our case, the best speedups are achieved on the largest data set (i.e., data set with ID 4) with 467x speedup over the S-CPU solution, 120x speedup over the M-CPU version, and 47x speedup over MPI-based solution.

Table VIII
THROUGHPUT OF GCM ALGORITHM ON S-CPU, M-CPU, MPI, AND DFE.

Data Set ID	**Throughput (Num. of Processed Inst. per Sec.)**			
	S-CPU $\times 10^5$	**M-CPU** $\times 10^6$	**MPI** $\times 10^6$	**FPGA** $\times 10^8$
1	9.293	3.298	5.256	1.682
2	5.865	2.493	4.134	1.695
3	3.679	1.432	4.405	1.718
4	3.787	1.320	3.706	1.742

Table IX
SPEEDUPS OF DFE GCM IMPLEMENTATION OVER IMPLEMENTATIONS ON S-CPU, M-CPU, AND MPI.

Data Set ID	**Speedup of DFE GCM Implementation**		
	over S-CPU	**over M-CPU**	**over MPI**
1	181x	51x	32x
2	289x	68x	41x
3	460x	112x	39x
4	467x	120x	47x

We believe that the reason behind the high speedup of our DFE implementation is the use of our proposed optimization strategies, including recomposing ICDF algorithm, removing unnecessary computations, sharing common computing result , and reducing float-point number precision. Applying these optimization strategies on GCM algorithm saves a large number of hardware resources on the FPGA chip, and enable us to deploying sufficient functional units in the pipeline. However, it is not feasible to perform similar optimization on CPUs since CPUs only contain limited functional units which are hard to achieve enough pipeline in space. Furthermore, if we have richer hardware resources on the FPGA, we can deploy more than one pipeline of GCM algorithm which will result in more significant acceleration.

VI. CONCLUSION

This paper presents four effective optimization strategies to guide the implementation of the GCM algorithm on DFE. By applying these strategies step by step into the GCM design, the original computationally-intensive GCM algorithm was implemented as a deep pipeline. For comparison, we evaluate the performance of the DFE implementation and three CPU-based implementations on four data sets. Experimental results show that our DFE solution achieved better speedup with larger size of data set, and in best case, it achieves a maximum of 467x speedup over the single-thread CPU solution, 120x speedup over the multi-thread CPU solution, and 47x speedup over the MPI solution. We believe that DFEs have high potential for accelerating the GCM algorithm, and for providing close to real time solutions for financial applications.

VII. ACKNOWLEDGMENTS

The funding support from National High Technology Development Program of China (2010AA012401, 2011AA01A203), Natural Science Foundation of China (61073165) is gratefully acknowledged. We would also like to thank the IEEE reviewers for their comments and suggestions.

REFERENCES

[1] D. Li, "On default correlation: a copula function approach," *Available at SSRN 187289*, 1999.

[2] P. Embrechts, A. Höing, and A. Juri, "Using copulae to bound the value-at-risk for functions of dependent risks," *Finance and Stochastics*, vol. 7, no. 2, pp. 145–167, 2003.

[3] Y. Malevergne and D. Sornette, "Testing the gaussian copula hypothesis for financial assets dependences," *Quantitative Finance*, vol. 3, no. 4, pp. 231–250, 2003.

[4] P. Embrechts, A. McNeil, and D. Straumann, "Correlation and dependence in risk management: properties and pitfalls," *Risk management: value at risk and beyond*, pp. 176–223, 2002.

[5] U. Cherubini and E. Luciano, "Bivariate option pricing with copulas," *Applied Mathematical Finance*, vol. 9, no. 2, pp. 69–85, 2002.

[6] A. Sklar, "Fonctions de repartition an dimensions etleurs marges," *Publ Inst Statist Univ Paris*, pp. 229–231, 1995.

[7] J. Dyer and S. Dyer, "Approximations to inverse error functions," *Instrumentation & Measurement Magazine, IEEE*, vol. 11, no. 5, pp. 32–36, 2008.

[8] G. West, "Better approximations to cumulative normal functions," *Wilmott Magazine*, vol. 9, pp. 70–76, 2005.

[9] A. Genz, "Numerical computation of rectangular bivariate and trivariate normal and t probabilities," *Statistics and Computing*, vol. 14, no. 3, pp. 251–260, 2004.

[10] J. Hart, *Computer approximations*. Krieger Publishing Co., Inc., 1978.

[11] R. Baxter, S. Booth, M. Bull, G. Cawood, J. Perry, M. Parsons, A. Simpson, A. Trew, A. McCormick, G. Smart *et al.*, "Maxwell-a 64 fpga supercomputer," in *Adaptive Hardware and Systems, 2007. AHS 2007. Second NASA/ESA Conference on*. IEEE, 2007, pp. 287–294.

[12] M. McCool, K. Wadleigh, B. Henderson, and H. Lin, "Performance evaluation of gpus using the rapidmind development platform," in *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*. ACM, 2006, p. 181.

[13] G. Morris and M. Aubury, "Design space exploration of the european option benchmark using hyperstreams," in *Field Programmable Logic and Applications, 2007. FPL 2007. International Conference on*. IEEE, 2007, pp. 5–10.

[14] http://home.online.no/~pjacklam/notes/invnorm/#An_overview_of_the_algorithm.

[15] D. Wang, S. Rachev, and F. Fabozzi, "Pricing tranches of a cdo and a cds index: recent advances and future research," *Risk Assessment*, pp. 263–286, 2008.