场外衍生品定价中的蒙特卡洛模拟算法优化方法研究

羊隽婷* 辛霄* 陆雨† 蔡天舒†

2022 年 12 月 6 日

概述

蒙特卡洛方法 (Monte Carlo method)，也称统计模拟方法，是指使用随机数（或者伪随机数）来解决复杂计算问题的方法，在许多领域有着非常广泛的应用。

特别是在金融工程学中，自 Myron Scholes 与 Fischer Black 提出 Black Scholes 模型，以几何布朗运动 (geometric Brownian motion) 描述资产价格的随机过程，蒙特卡洛方法就成为衍生品定价的基本方法。尽管近几十年来在 BS 模型的框架下，学者与量化分析师们已经为数十种常见的复杂衍生品推导出了闭式解或近似解，如各类美式障碍期权、美式香草期权、亚式期权等，但随着场外衍生品市场的快速发展，高度异质化、定制化的结构层出不穷，以致目前大部分活跃交易的场外衍生品结构只能采用数值方法（二叉树，三叉树，有限差分法等）或蒙特卡洛方法进行相关计算。再者，在 BS 模型的基础上，Heston model、CEV model、SABR model、Jump Diffusion model 等随机过程模型也被广泛应用于金融工程领域中。这些模型引入了更多的风险因子，能更精细地刻画资产价格的随机过程，但同时也为衍生品的定价带来极大的挑战。对于某些问题，采用蒙特卡洛算法模拟随机过程进行求解几乎是唯一可行的方法。

近年来，随着国内场外衍生品市场的飞速发展，业务量和业务复杂度也相应地极速提升，对相关从业机构与从业人员都提出了更高的要求。当前，

*安信证券股份有限公司
†上海锰链科技有限公司

蒙特卡洛方法作为金融工程领域最为基础与通用的算法，虽然得到普遍运用，但运算效率上差强人意, 难以满足业务, 交易以及风险管理的要求。

因此, 以满足国内场外衍生品市场潜在发展为目标，通过完善随机过程模型、优化设计框架，结合计算机软硬件相关先进算法，打造高效通用的蒙特卡洛模拟定价引擎，成为能否在未来场外衍生品市场竞争中处于优势地位的关键一步。

本文以场外衍生品定价中的蒙特卡洛模拟算法优化为目标，在镜像法的基础上提出了基于旋转变换的快速高斯随机数算法 (FASTNORM)，并运用单一指令多数据流（SIMD）CPU 向量化计算技术以及基于 OpenMP 的跨平台多线程计算技术，将蒙特卡洛模拟方法的定价效率提升超 20 倍。最后，本文也研究了基于 GPU 的蒙特卡洛模拟算法，利用 CUDA 开发套件（CUDA toolkit）充分释放 GPU 的并行数据处理能力，实现了对蒙特卡洛模拟算法定价效率的跨越式优化。

关键词: 蒙特卡洛模拟, FASTNORM, SIMD, OpenMP, GPU, CUDA

1 几何布朗运动及其离散化

1.1 几何布朗运动

几何布朗运动（Geometric Brownian Motion，GBM）是一个简单连续随机过程，对随时间演变的随机行为建模，该模型在物理和金融领域中被广泛应用。通常情况下，资产价格 $S(t)$ 的时间演化模型能够通过几何布朗运动这一数学模型进行模拟，并由以下随机微分方程（Stochastic Differential Equation，SDE）表示:

$$dS(t) = \mu S(t)dt + \sigma S(t)dW_t \tag{1}$$

其中 W_t 是一个维纳过程 (Wiener process) 。μ 表示漂移，σ 表示随机波动的幅度（即波动率），在 Black-Scholes-Merton 的模型假设下，它们都是常数。

1.2 随机过程的离散化

根据伊藤引理（Itô's lemma）

$$d\ln S(t) = (\mu - \frac{1}{2}\sigma^2)dt + \sigma dW_t \tag{2}$$

将（2）式两边从 t 到 $t+\Delta t$ 积分，可得

$$S(t+\Delta t) = S(t)\exp((\mu - \frac{1}{2}\sigma^2)\Delta t + \sigma W_{\Delta t}) \tag{3}$$

其中，$W_{\Delta t} \sim \mathcal{N}(0, \sqrt{\Delta t})$。

因此，资产价格从 t 到 $t+\Delta t$ 的离散化表达是

$$S(t+\Delta t) = S(t)\exp((\mu - \frac{1}{2}\sigma^2)\Delta t + \sigma\sqrt{\Delta t}\epsilon) \tag{4}$$

或

$$\ln S(t+\Delta t) = \ln S(t) + (\mu - \frac{1}{2}\sigma^2)\Delta t + \sigma\sqrt{\Delta t}\epsilon \tag{5}$$

其中，$\epsilon \sim \mathcal{N}(0,1)$。

上述即为 Black-Scholes 模型下的欧拉离散化（Euler discretization），其中（5）式为其对数形式的表达方法。采用对数形式计算可以避免复杂的幂运算，仅需进行简单的加法运算，因此可以有效地节省计算机资源从而提升计算效率。在后续的蒙特卡洛计算中，均模拟资产的对数价格路径进行期权现值的计算。

2 关于正态分布随机数生成的优化方法

伪随机数生成器生成随机数耗时最长，有比较大的优化空间。由正态随机变量的性质，可以对伪随机数生成器产生的随机数应用更快速的线性变换，得到更多的随机数，从而减少生成随机数的时间，提升蒙特卡洛模拟的运算速度。

基于这一点，本节将介绍并使用镜像法和 FASTNORM 【加引用】方法提升生成随机数的效率。

2.1 镜像法

在生成符合正态分布的随机数时，伪随机数生成器首先依均匀分布产生一个随机数，随后根据这个均匀分布的随机数来计算相应的正态随机数，这个过程往往比较耗时。如果能够根据一组已经产生的正态随机数生成更多的正态随机数，那么就可减少伪随机数生成器的使用，从而提升蒙特卡洛模拟的运算速度。针对这一点，一种简单可行的方法是将原本随机数的相反数作为另一个随机数使用。

由标准正态随机变量的性质，如果 $X \sim N(0,1)$，那么 $-X \sim N(0,1)$ 。这种方法的另一个好处是生成的随机数样本均值为 0 ，这正是 X 的均值。与传统方法相比，达到了一阶动量匹配以及归一化的效果。

显然，通过线性变换的正态分布随机数依旧服从正态分布。利用正态随机变量的性质，可以将镜像法扩展到更一般 FASTNORM 方法，通过线性变换和镜像法来成倍地产生随机数，从而减少运算的时间。

2.2 FASTNORM

如果 $\mathbf{X}$ 是一个由 n 个独立标准正态随机变量组成的向量，$\mathbf{R}$ 是一个 $m \times n$ 矩阵，且 $\mathbf{R}\mathbf{R}' = I$，那么 $\mathbf{Y} = \mathbf{R}\mathbf{X}$ 是一个由 m 个独立标准正态随机变量组成的向量，这是因为随机向量 $\mathbf{Y}$ 的期望和协方差矩阵分别为

$$\begin{aligned} \mathrm{E}(\mathbf{Y}) &= \mathrm{E}(\mathbf{R}\mathbf{X}) = \mathbf{R}\mathrm{E}(\mathbf{X}) = \mathbf{R}\mathbf{0} = \mathbf{0}, \\ \mathrm{V}(\mathbf{Y}) &= \mathbf{R}\mathrm{V}(\mathbf{X})\mathbf{R}' = \mathbf{R}R' = \mathbf{I}. \end{aligned} \tag{6}$$

注意，如果 $\mathbf{R}\mathbf{R}' = I$，那么 $m \leq n$。

令 $\mathbf{R} = -\mathbf{I}$ ，可以看出镜像法实际上是 FASTNORM 的特例。

二维旋转矩阵

$$\begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \tag{7}$$

就是一类符合以上条件的 $\mathbf{R}$ 。二维空间中向量的左乘旋转矩阵就能得到逆时针旋转 θ 之后的向量，如下图所示：

应特别注意，在二维情形中，θ 的取值应满足 $\theta \neq \pi + k\pi, \theta \neq \frac{\pi}{2} + k\pi, k = 0, \pm 1, \pm 2, \ldots$，以避免与原向量或镜像后的向量重复。

使用这种方法，能由 n 个随机数 $\mathbf{X}$ 立即得到 m 个新的随机数 $\mathbf{Y}$。在此基础上再应用镜像法，能够得到 $-\mathbf{X}$ 和 $-\mathbf{Y}$。因此，只需要生成 m 个伪随机数，就能得到 $2(m+n)$ 个随机数。

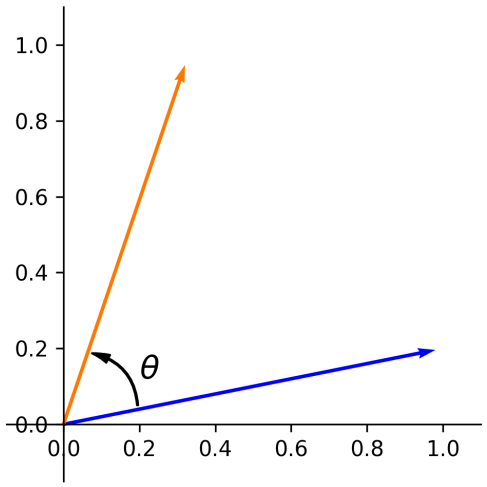

图 1: 以二维空间为例的向量旋转：蓝色向量逆时针旋转得到橙色向量

尽管 FASTNORM 能大幅提升随机数的产生速度，但这种方法的在一定程度上牺牲了随机数的质量，这是由于 $\mathbf{X}$、$-\mathbf{X}$、$\mathbf{Y}$，以及 $-\mathbf{Y}$ 并不是相互独立的。有

$$\begin{aligned} &\mathrm{Cov}(X_j, Y_i) = r_{ij}, \\ &\mathrm{Cov}(X_j, -X_j) = -1, \ \ 1 \le j \le n, 1 \le i \le m. \end{aligned} \tag{8}$$

从第一个式子中也可以看出，选取较大的 m 和 n 能够降低新随机数与原本随机数之间的相关性，从而降低 FASTNORM 在相关性上产生的影响。

下一小节将进一步讨论镜像法和 FASTNORM 在蒙特卡洛算法中的实现。

2.3 算法实现与分析

使用 FASTNORM 时，矩阵 $\mathbf{R}$ 的选取是相当重要的。理想的线性变换矩阵 $\mathbf{R}$ 应满足以下条件：1) $\mathbf{R}\mathbf{R}' = I$（这意味着 $m \le n$）；2) $\mathbf{R}$ 中的所有元素都不能为 1 或 -1，以确保生成的新随机数及其相反数不与原随机数重复。基于这两点，可以选取矩阵

$$\mathbf{R} = \begin{bmatrix} 1/2 & 1/2 & -1/2 & 1/2 \\ 1/2 & -1/2 & 1/2 & 1/2 \\ 1/2 & -1/2 & -1/2 & -1/2 \\ -1/2 & -1/2 & -1/2 & 1/2 \end{bmatrix}. \tag{9}$$

这也是 Wallace 推荐的方法。容易验证 $\mathbf{R}$ 是规范正交的。因此由线性变换产生的每组随机数是独立的正态变量。

确定了 $\mathbf{R}$ 之后，FASTNORM 的实现大致可分为三步：

- 用伪随机数生成器产生一组随机数（以下将称为原随机数）；
- 依镜像法，取原随机数的相反数，得到另一组随机数；
- 对原随机数进行线性变换，并取线性变换的相反数，得到另外两组随机数。具体实现方式如下：

假设每条路径的生成需要 N 个随机数。用伪随机数生成器生成 4 个随机数，记为 $\mathbf{x}_1$，然后计算 $\mathbf{y}_1 = \mathbf{R}\mathbf{x}_1$；然后用为随机数再次生成 4 个随机数 $\mathbf{x}_2$，并计算得到 $\mathbf{y}_2$……直到 $\mathbf{x}_N$ 和 $\mathbf{y}_N$。然后，用 $\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_N$ 的第 i 个元素生成一条标的价格的路径，这里 $i = 1, 2, 3, 4$。对 $\mathbf{y}_1, ..., \mathbf{y}_N$、$-\mathbf{x}_1, ..., -\mathbf{x}_N$、$-\mathbf{y}_1, ..., -\mathbf{y}_N$ 也用类似方式生成路径。

这种方法每循环一次就能得到总共 16 条路径，而伪随机数生成器只生成了其中 4 条路径所需要的随机数，因此这种方法能够提升随机数产生的效率。

下一节将展示镜像法和 FASTNORM 带来的性能优化，以及讨论随机数的非独立性对蒙特卡洛估计的影响。

2.4 分析对比

FASTNORM 能够大幅提升计算效率，但同时会造成路径之间的相关性。本节将从效率和误差两个角度来分析对比 FASTRNORM 和原始方法。

2.4.1 效率比较

图 3 对比了运行一次原始方法、镜像法和 FASTNORM 方法的所需的时间，模拟次数为 100 万次。如图所示，运行一次原始蒙特卡洛模拟算法需要 1398 毫秒。使用镜像法之后耗时缩短至 820 毫秒，是原始算法的 58.7%；

而使用 FASTNORM 之后，耗时仅为 418 毫秒，是原始算法的 29.9%。由此可见，FASTNORM 能将蒙特卡洛模拟的运算速度提升超过三倍。

前文提到镜像法和 FASTNORM 方法会导致随机数之间的相关性的提升，可能会对蒙特卡洛的误差带来影响。本节将从误差的角度比较原始方法和 FASTNORM 方法。

2.4.2 收敛情况比较

镜像法和 FASTNORM 方法使用的随机数（不论是由伪随机数算法产生的，还是通过线性组合得到的）都是符合标准正态分布的，因此它们的蒙特卡洛估计量仍然是无偏的。如图 4所示，可以看出在当路径数量增大时，两种方法的结果都向 1.3588 附近收敛。

估计量的无偏性确保了镜像法和 FASTNORM 的可行性。然而如前文所述，FASTNORM 方法对蒙特卡洛估计量的影响主要体现在有效性上，即估计量的标准差（或方差）上，这种影响可以通过比较估计量的标准差来讨论。

2.4.3 误差的计算

原始蒙特卡洛方和和 FASTNORM 蒙特卡洛方法的标准误差都可以根据中心极限定理推导出来。

FASTNORM 方法可以看作是每次循环生成 m 条非独立的路径，在本文讨论的方法中，$m = 16$。如果循环了 n 次，那么模拟的路径总数为 mn。用 $\mathbf{X}_{ij}, 1 \le j \le n, 1 \le i \le m$ 表示第 j 次循环生成的第 i 条路径，用 f 表示给定路径下的贴现收益，那么 FASTNORM 蒙特卡洛估计的结果可以表示为

$$\bar{V} = \frac{1}{n}\sum_{j=1}^{n}\frac{1}{m}\sum_{i=1}^{m} f(\mathbf{X}_{ij}). \tag{10}$$

记 $y_j = \sum_{i=1}^{m} f(\mathbf{X}_{ij})$，则 $y_1, ..., y_n$ 是独立同分布的，因此

$$\bar{V} = \frac{1}{n}\sum_{j=1}^{n} y_j. \tag{11}$$

根据中心极限定理，如果 $y_1, ..., y_n$ 的样本标准差是 s，那么可以用 $\frac{s}{\sqrt{n}}$ 来估计 $\mathrm{Var}(\bar{V})$。

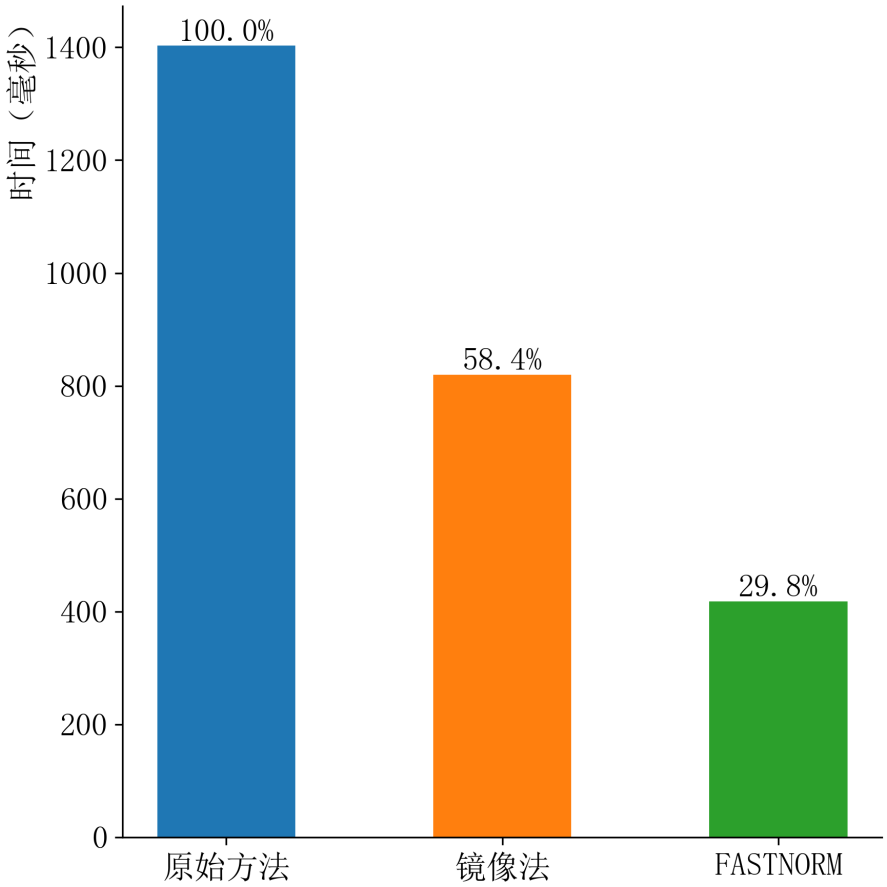

图 2: 运行 100 万次蒙特卡洛模拟的时间：蓝色部分为原始蒙特卡洛模拟算法，橙色部分为镜像法，绿色部分为 FASTNORM 方法；数据通过运行 100 次模拟取平均值得到

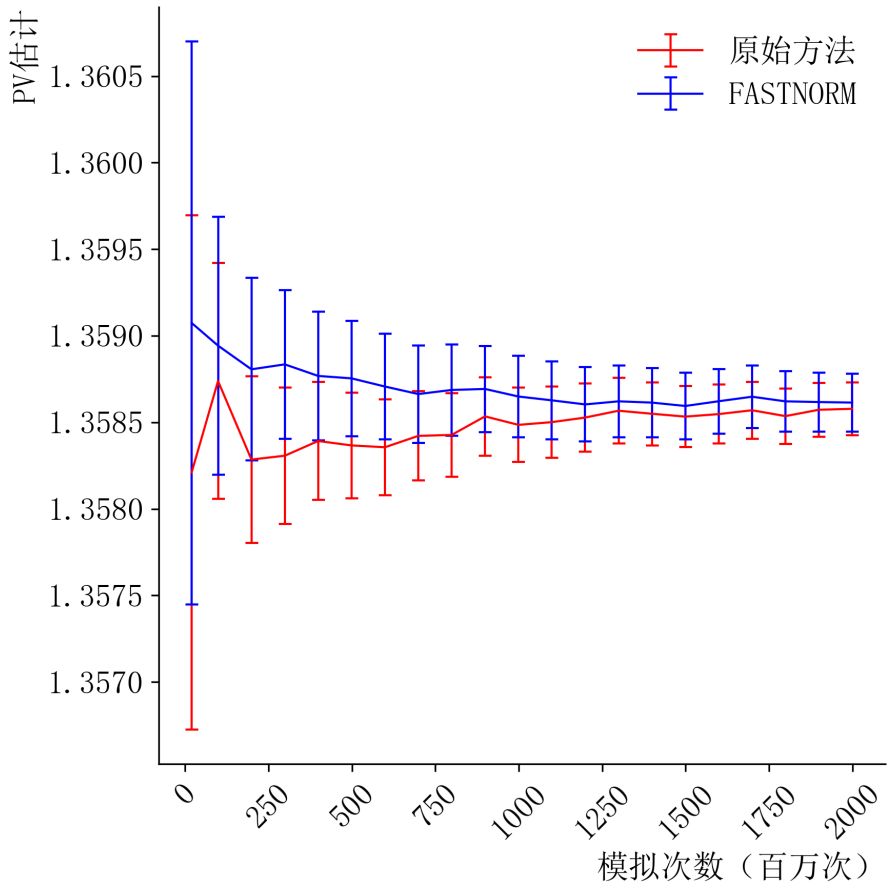

图 3: 随着模拟次数增加，原始方法和 FASTNORM 向 1.3588 附近的值收敛。

方法名称	标准差
原始方法	0.00350160089209261
镜像法	0.0034882495542142957
FASTNORM	0.003828353638954014

表 1: 各方法的标准差比较

	F 统计量	P 值
原始方法与镜像法	1.0071655989622432	0.8731991106943031
原始方法与 FASTNORM	0.836164800985319	6.419816278691966e-05

表 2: F 检验：原始方法、镜像法，以及 FASTNORM 方法得到的估计量的标准差及 F 检验结果，模拟的路径数均为 100 万条.

2.4.4 误差比较

表 5 展示了原始方法、镜像法，以及 FASTNORM 方法的标准差，用以比较估计量的有效性，表 6 展示了 F 检验的结果。

可以看出，镜像法和原始方法的标准差没有显著差异，然而对比原始方法，FASTNORM 的标准误差上升了 9.3%。因此，FASTNORM 下 100 万条路径的模拟精度大约相当于原始方法下 80 万条路径的模拟精度，然而前者的耗时却不到后者的 1/4，对比镜像法，FASTNORM 也只需要一半的时间。该结果表明 FASTNORM 是比原始方法更高效的算法。

3 基于硬件平台的进一步优化

本文不仅从算法层面上对随机数生成进行了优化，也考量了现代 CPU（central processing unit，中央处理器）在提升蒙特卡洛模拟计算效率上的可能性。随着 CPU 的更新迭代，现代 CPU 都能够支持多核多线程，并且大多数 CPU 支持单指令多数据流（Single Instruction Multiple Data, SIMD）的向量化（vectorization）运算以及多线程（multiprocessing，MP）并行运算。基于此基础上，下文将从 CPU 层面上对提升随机数生成速率进行相关研究。

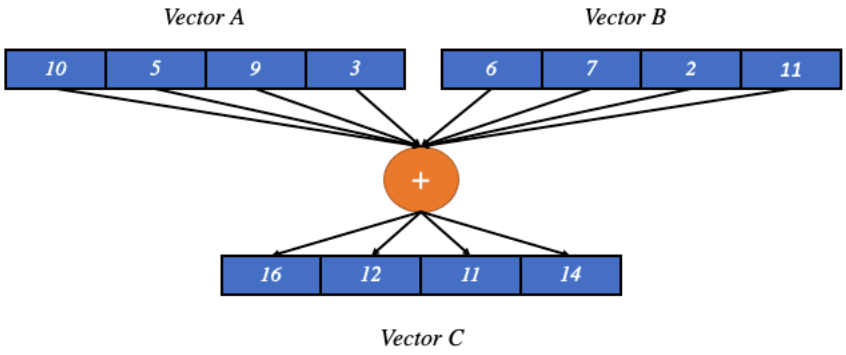

图 4: SIMD 向量加法示例。将输入的两个向量分别命名为 $Vector\,A$ 和 $Vector\,B$ ，这两个向量均包含四个操作数，对这两组数并行执行四个相同的加法运算，得到结果向量 $Vector\,C$ 。

3.1 基于 SIMD 的 CPU 向量化

向量化是提高现代 CPU 性能的关键工具，而 SIMD 可以实现 CPU 的向量化运算。SIMD 认为计算机具有多个处理元素，并且这些处理元素在多个数据点上能够同时执行相同的运算。向量化能够实现从一次操作多个值转换为一次操作一组值，并且现代 CPU 能够直接支持 SIMD 向量运算。例如，一个 512 位寄存器的 CPU 可以容纳 16 个 32 位单精度浮点数并进行单次运算，比单次执行一条指令快 16 倍。因此，将向量化与多核多线程 CPU 相结合，可以带来数量级的性能提升。

通常情况下，SIMD 单元中的操作包括基本的数学运算（比如加、减、乘、除）及其他常见的数学运算，包括绝对值（abs）和平方根（sqrt）的运算。比如，SIMD 接收两个向量作为输入（每个向量都有一组运算对象），对这两组运算对象（每个向量的每个操作数）执行相同的运算，并输出一个带结果的向量，如图 4 所示。

在现代 CPU 的发展中，指令集架构（Instruction Set Architecture，ISA）也在不断地更新，实现了更高效地进行与 SIMD 相关的数据处理的可能性。指令集架构是能够同时在多个数据对象上执行相同操作并且提升性能的额外指令，其中包括 SIMD、流 SIMD 拓展（Streaming SIMD Extensions，SSE）、高级矢量拓展（Advanced Vector Extensions，AVX）。SSE 是对现

有的 32 位 (x86) CPU 架构的 SIMD 指令集扩展，用于取代最初的、性能较差的 MMX（MultiMedia EXtensions）架构。随着技术的更新，AVX 的出现提供了宽度更高的向量、新的拓展性语法以及更加丰富的功能，并且在 SSE 基础上实现更高效的数据管理功能以及性能提升。

目前，最新的处理器都具有多核，在日益增加的大型数据集上使用 SIMD 执行单指令，能够在提升性能的同时降低能耗。

3.1.1 SIMD 在蒙特卡洛模拟中的具体应用

在蒙特卡洛模拟中，使用基于 CPU 的 SIMD 向量化计算技术能够提升矩阵运算和随机数生成的速度。例如，FASTNORM 中矩阵的旋转变换、以及利用向量化进行快速随机数生成等。

（1）矩阵运算

在上文中提及的 FASTNORM 算法中所运用的矩阵旋转变换可以通过基于 SIMD 优化的矩阵运算实现，进而提升计算速率。

在实际运算中，使用 SIMD 实现常见的线性代数底层程序，即 BLAS。BLAS 规范了常见的线性代数运算的内核底层程序，是线性代数库的标准底层程序。BLAS 功能分为三个被称为级别（levels）的程序，分别对应于相关定义和发布时间的顺序，以及多项式次数对应的算法复杂度。第一级别的 BLAS 操作通常耗时为线性的，第二级别操作为二次方时间，第三级别操作为立方时间。现代化的 BLAS 实现通常提供所有的级别。由于矩阵乘法在大部分数学应用中至关重要，使用 BLAS 能够高效、便捷、广泛地应用在数学运算中。

尽管 BLAS 是通用的规范，但是 BLAS 的实现通常针对特殊的机器进行优化。例如，Intel oneMKL 的 BLAS 针对 Intel 体系结构进行了优化，因此使用该数学库的 BLAS 接口进行矩阵运算能够在 Intel 芯片上实现优异的性能提升。

（2）随机数生成

为了充分利用现代 CPU 对 SIMD 向量化计算的优势，随机数生成器能够通过调用高度优化的基本随机数生成器和向量数学函数进行开发，从而实现大幅提升随机数生成速率的目标。例如，面向 SIMD 的快速梅森旋转 (SIMD-oriented Fast Mersenne Twister，SFMT) 是一个线性反馈移位寄存器（Linear Feedbacked Shift Register, LFSR）生成器，可以一次生成 128 位伪随机整数，其随机数生成速度是使用普通梅森生成速度的两倍。

3.2 CPU 多线程

除了基于 SIMD 的向量化运算之外，利用现代 CPU 的多核多线程也能够大幅提升计算效率。目前，几乎所有的现代 CPU 都支持多线程。不仅如此，多核 CPU 上的多线程能够一次处理不同的任务，从而使得计算机性能更加高效、耗能更低。

线程（Thread）是并发编程（concurrent programming）中的一个执行单元，而多线程则允许 CPU 同时执行一个进程中的多个任务。在计算机操作系统中，并发编程指的是在重叠时间段内执行多个任务，并且没有特定的执行顺序，而线程编程（threaded programming）则最常用于共享内存并行 (shared memory parallel) 计算机中。线程与进程（processes）类似，不同之处在于线程之间可以彼此共享内存 (以及拥有私有内存)。在实际操作中，线程之间必须能够交换数据以便进行有用的并行程序，也可以通过读写共享数据进行通讯。例如，线程 1 向共享变量 a 写入一个值，线程 2 可以从共享变量 a 中读取这个值。

多线程编程（Multithreaded programming）是指对多个并发执行的线程进行编程从而实现高性能的机制，在单核 CPU 或者多核 CPU 上均能够运行多线程。值得注意的是，单核 CPU 上的多线程并不是并行运行的。因为在实际运用中，单核 CPU 是通过调度算法（scheduling algorithm），或者根据外部输入（或干扰）的组合以及线程间的优先级进行切换。然而，多核 CPU 的多线程则是并行的，因为多核 CPU 是通过多个微处理器的共同协作从而提升性能。

并行化（Parallelism）是指在多核 CPU 等具有多种计算资源的硬件上同时运行多个任务或同一任务。当 CPU 内部运行的时钟频率（clock speed）已经最大化时，并行化则是在多核 CPU 上提升性能的唯一办法。由于多核 CPU 的内部运行的时钟频率是单核 CPU 的多倍速，因此，采用多核 CPU 不仅能在单核 CPU 的基础上有效地处理指令速度，还能够降低计算机的耗能。

3.2.1 具体应用:OpenMP

CPU 与多线程之间有多种交互方式，为了计算的通用型及代码的可移植性，本文采用 OpenMP 简化多线程，进而提升蒙特卡洛模拟的计算速度。OpenMP（Open Multi-Processing）是一个应用程序编程接口（API），它支持在各不同的平台、指令集架构和操作系统（包括 Solaris、AIX、FreeBSD、

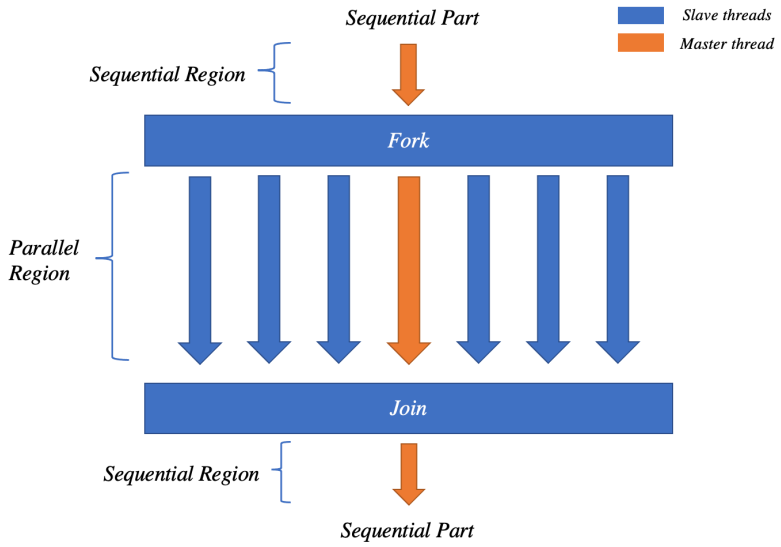

图 5: Fork-join 模型。每个线程执行并行区域内的命令，在并行区域结束后，主线程等待其他线程结束后，继续执行下一个命令。

HP-UX、Linux、macOS 和 Windows）上针对 C，C++ 和 Fortran 的一组拓展，进行多平台共享内存的多线程编程。

OpenMP 通过指令（directive）和标记（sentinel）简化多线程。一个 OpenMP 指令是指一行只针对某些编译器有意义的源代码，通常在行首由一个特殊标记进行区分。

OpenMP 使用并行执行的 Fork-join 模型从而实现并行化。其中，并行区域（parallel region）是 OpenMP 中的基本并行结构，用于定义程序的一部分。当程序开始时，在一个单独的线程（主线程，即 master thread）上开始执行。当遇到第一个并行区域时，主线程创建一个线程组（fork-join 模型，如图 3 所示）。

（1）并行循环

循环是大部分应用程序中进行并行的主要来源，OpenMP 对并行循环（parallel loops）提供了广泛的支持。当一个循环的迭代是独立的，即可以以任何顺序完成时，那么其能够在不同的线程中共享迭代，并认为这一个迭代或一组迭代是一个任务。在 OpenMP 中，仅需利用一行编译指令指示编译器生成代码，进而对多线程之间的循环迭代进行拆分，如 Algorithm 1 所示。

Algorithm 1 OpenMP: 并行循环

Input: Vectors $\mathbf{A}^{(n)}$, $\mathbf{B}^{(n)}$

Output: Element-wise addition of $\mathbf{A}^{(n)}$, $\mathbf{B}^{(n)}$

Initialize:

$\mathbf{A}^{(i)} \leftarrow i$

$\mathbf{B}^{(i)} \leftarrow i$

$i = 1, \ldots, n$

Parallel Loops:

#pragma omp parallel for default(none)

shared(a, b)

for $i = 1 : n$ **do**

$\mathbf{A}^{(i)} \mathrel{+}= \mathbf{B}^{(i)}$

end for

return $\mathbf{A}^{(n)}$

（2）降维

除了并行循环外，降维（reduction）可以实现并行执行一些重复计算的形式：从加、乘、最大值、最小值、和、或等关联操作中产生一个值。在降维中，每个线程可以积累自己的私有副本，然后这些副本被降维以得到最终结果。如果操作的数量远远大于线程的数量，那么大多数操作可以在并行中进行，如 Algorithm 2 所示。

3.3 结果分析对比

插入图片：插图，展示使用 SIMD 带来的性能提升]

4 希腊值

衍生品或其他资产的希腊值是该资产对不同风险因子的敏感程度，因此能够描述该资产对不同风险因子的暴露程度。希腊值越大，则对应的风险因子的暴露程度越大。比较常见的风险因子有 Delta、Gamma、Vega、Theta 和 Rho。这些希腊值分别定义为：

Algorithm 2 OpenMP: 降维

Input: Vector $\mathbf{A}^{(n)}$, double **b**

Output: Sum of the elements in $\mathbf{A}^{(n)}$

Initialize:

$\mathbf{A}^{(i)} \leftarrow i$

$\mathbf{b} \leftarrow 0$

$i = 1, \ldots, n$

Reduction:

#pragma omp parallel for default(none)

shared(a)

reduction(+: b)

for $i = 1 : n$ **do**

$\mathbf{b} \mathrel{+}= \mathbf{A}^{(i)}$

end for

return b

$$
\begin{aligned}
\text{Delta} &= \frac{\partial f}{\partial S}, \\
\text{Gamma} &= \frac{\partial^2 f}{\partial S^2}, \\
\text{Vega} &= \frac{\partial f}{\partial \sigma}, \\
\text{Theta} &= \frac{\partial f}{\partial \tau}, \\
\text{Rho} &= \frac{\partial f}{\partial r}.
\end{aligned}
\tag{12}
$$

这里 f 表示衍生品价值，S 表示标的价格，σ 标的的隐含波动率，τ 表示衍生品剩余期限，r 表示无风险利率。

由于风险因子会不断变化，从而导致衍生品价值的变化，因此在风险管理中，希腊值起着非常重要的作用。希腊值不仅可以直观地描述资产的风险，也可以帮助调整整个投资组合的风险暴露程度。在对冲中，可以根据希腊值调整用于对冲的资产的持有数量，从而使整个组合的对各风险因子的暴露达到理想的状态。

4.1 有限差分法计算希腊值

即使是在 Black-Scholes 模型下，对于奇异衍生品来说，希腊值很少有解析解。因此，经常使用有限差分法来计算衍生品的希腊值。一阶导数的前向差分格式为

$$f'(x) \approx \frac{f(x+\Delta x) - f(x)}{\Delta x}. \tag{13}$$

二阶导数的差分格式为

$$f''(x) \approx \frac{f(x+\Delta x) - 2f(x) + f(x-\Delta x)}{(\Delta x)^2}. \tag{14}$$

这里 $\Delta x > 0$。因此，在已经计算出期权现值的基础上，如果要计算希腊值，需要改变 Black-Scholes 模型的参数，并根据新的参数再一次进行蒙特卡洛模拟，得到新参数对应的现值。随后按照有限差分格式 (**??**) 及式 (**??**) 近似计算希腊值。例如，期权 Delta 的计算格式为

$$\text{Delta} \approx \frac{PV(S_0 + \Delta S_0, r, q, \sigma) - PV(S_0, r, q, \sigma)}{\Delta S_0}. \tag{15}$$

这里 PV 表示蒙特卡洛模拟得到的期权现值，S_0 表示标的资产的初始价格。可以看到，计算 Delta 需要两倍计算现值的时间，但是如果重复使用生成的随机数，则能够大幅提升希腊值计算的效率，在**??**节中，将介绍这种方法。在此之前，首先希腊值计算的误差进行简略分析。

4.2 有限差分的误差

现在以 (**??**) 为例，分析有限差分法计算希腊值的误差。假设 f'' 存在，那么

$$f(x+\Delta x) = f(x) + f'(x)\Delta x + \frac{1}{2}f''(t)(\Delta x)^2 \tag{16}$$

对某个 $t \in (x, x+\Delta x)$ 成立。因此

$$\left| \frac{f(x+\Delta x) - f(x)}{\Delta x} - f'(x) \right| \leq \frac{M}{2}\Delta x. \tag{17}$$

这里 $M = \sup|f''|$。因此，(**??**) 式的误差是 $o(\Delta x)$。在蒙特卡洛模拟中，无法精确计算期权的现值，因此更小的 Δx 反而可能带来更大的误差，这是由于 (**??**) 式右端的分母是 Δx，会将误差放大。因此，在计算希腊值时，更能小的 Δx 不能保证更精确的结果。

4.3 重复使用随机数

Black-Scholes 随机微分方程的解为

$$\ln S(t) = \ln S_0 + (r - q - \frac{\sigma^2}{2})t + \sigma\sqrt{t}W(t). \tag{18}$$

要计算期权的现值，需要根据该式模拟标的资产的对数价格路径。假设已经模拟了一条标的资产的对数价格路径 $\{x(t) : t = 1, 2, ..., n\}$。

由 (**??**) 可以看出，如果标的的初始价格为 $S_0 + \Delta S_0$，那么只需要将每个 $x(t)$ 增加 $\ln(S_0 + \Delta S_0) - \ln S_0$，就能得对应的路径；类似地，从每个 $x(t)$ 减去 $\ln(S_0 + \Delta S_0) - \ln S_0$ 就能得到初始价格为 $S_0 - \Delta S_0$ 对应的标的的对数价格路径；如果无风险利率 r 增加了 Δr，那么 $\{x(t) + r\Delta t\}$ 可以作为新的路径。用这种方法，在计算 Delta、Gamma 和 Rho 时，既无需重新生成随机数，也无需根据 (**??**) 重新计算路径。而在 Vega 的计算中，虽然无法直接对整条路径平移，但是用于生成的原本路径的随机数仍然可以继续被用来生成波动率为 $\sigma + \Delta\sigma$ 时的路径。

以 Delta 为例，计算希腊值的步骤为:

- 生成对数价格路径 $\{x(t) : t = 1, 2, ..., n\}$，并计算期权现值 PV_0;
- 把 $\{x(t) + \ln(S_0 + \Delta S_0) - \ln S_0 : t = 1, 2, ..., n\}$ 作为对数路径，计算现值 PV_1;
- Delta $\approx \frac{PV_1 - PV_0}{\Delta S_0}$。

计算希腊值时重复使用随机数的另一个好处是（在 ΔS_0 和模拟路径数不变的前提下）可以减少计算出的希腊值的标准差，这是因为重复使用随机数时，PV_1 和 PV_0 是正相关的，因此两者之差的方差小于方差之和；而后者是不重复使用随机数时 $PV_1 - PV_0$ 的方差。

4.4 结果

图 **??**比较了仅计算现值和计算现值及 Delta、Gamma、Vega、Rho 和 Theta 所需的时间。计算这些希腊值至少需要生成 7 条路径，而在重复使用随机数的前提下，计算希腊值只需要计算现值 4.2 倍的时间。

图 **??** - **??**为现值和希腊值在不同标的即期价格下的计算结果。

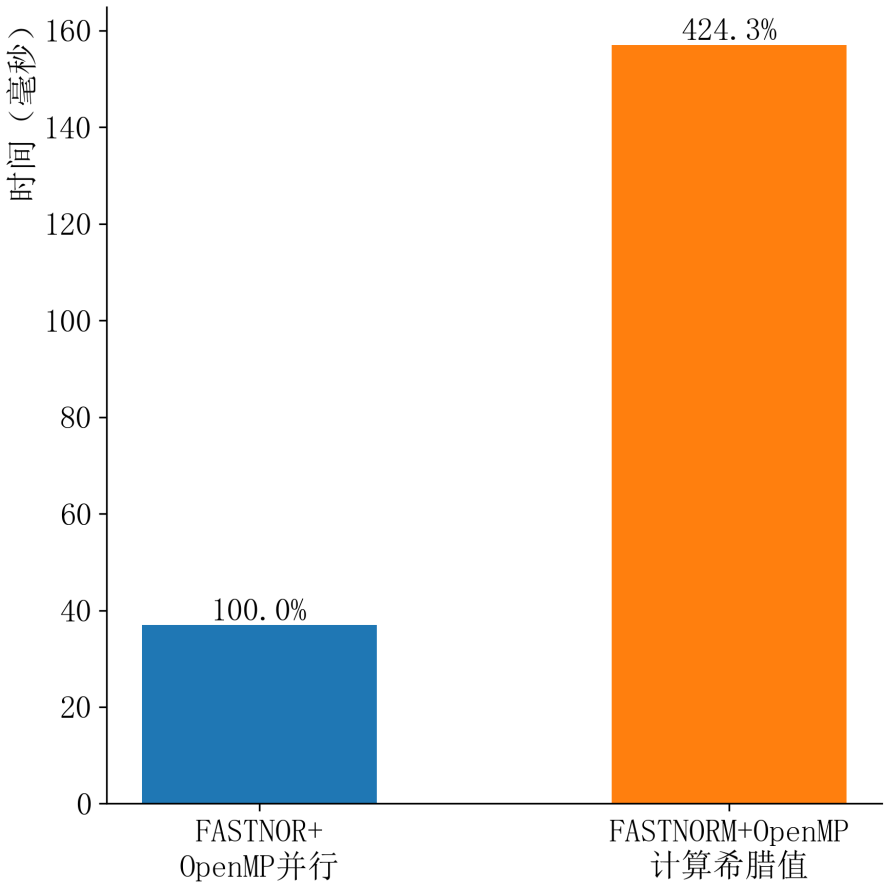

图 6: 计算希腊值所用时间

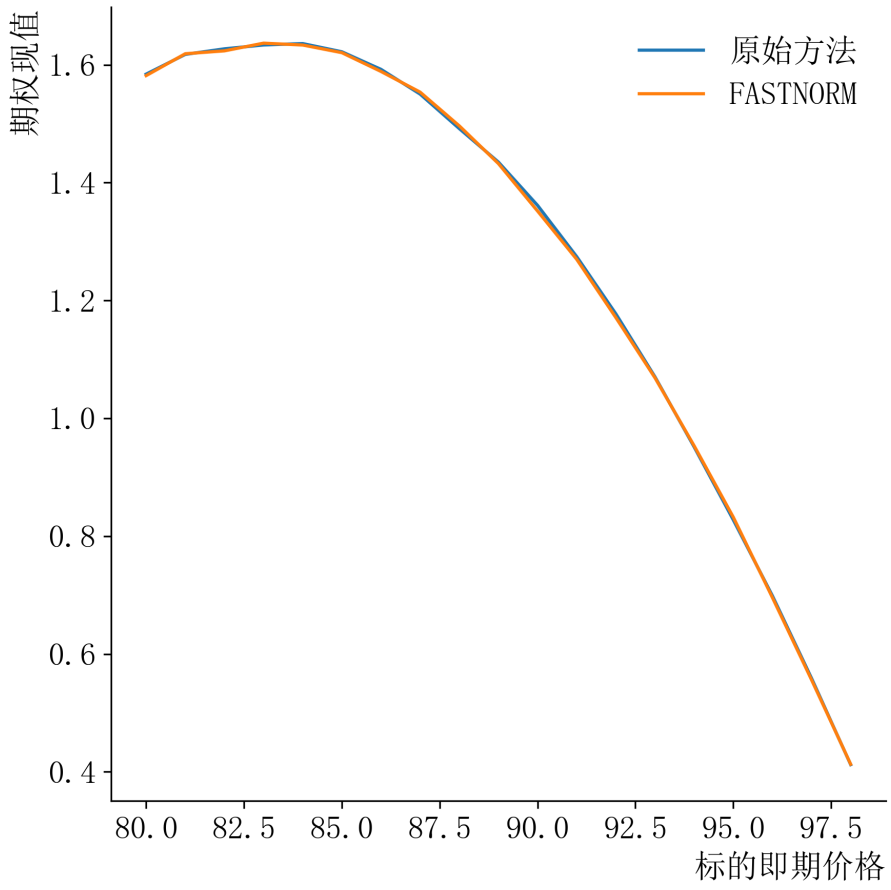
期权现值
1.6
1.4
1.2
1.0
0.8
0.6
0.4
80.0
82.5
85.0
87.5
90.0
92.5
95.0
97.5
标的即期价格
原始方法
FASTNORM

图 7: PV

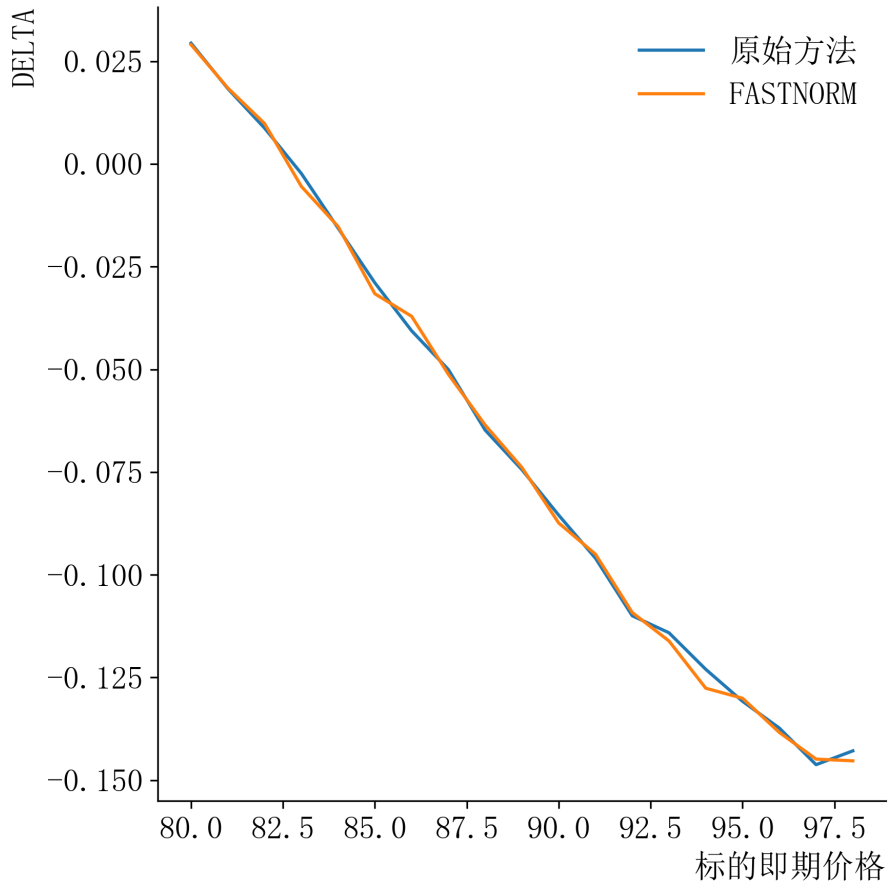
DELTA
0.025
0.000
−0.025
−0.050
−0.075
−0.100
−0.125
−0.150
80.0
82.5
85.0
87.5
90.0
92.5
95.0
97.5
标的即期价格
原始方法
FASTNORM

图 8: DELTA

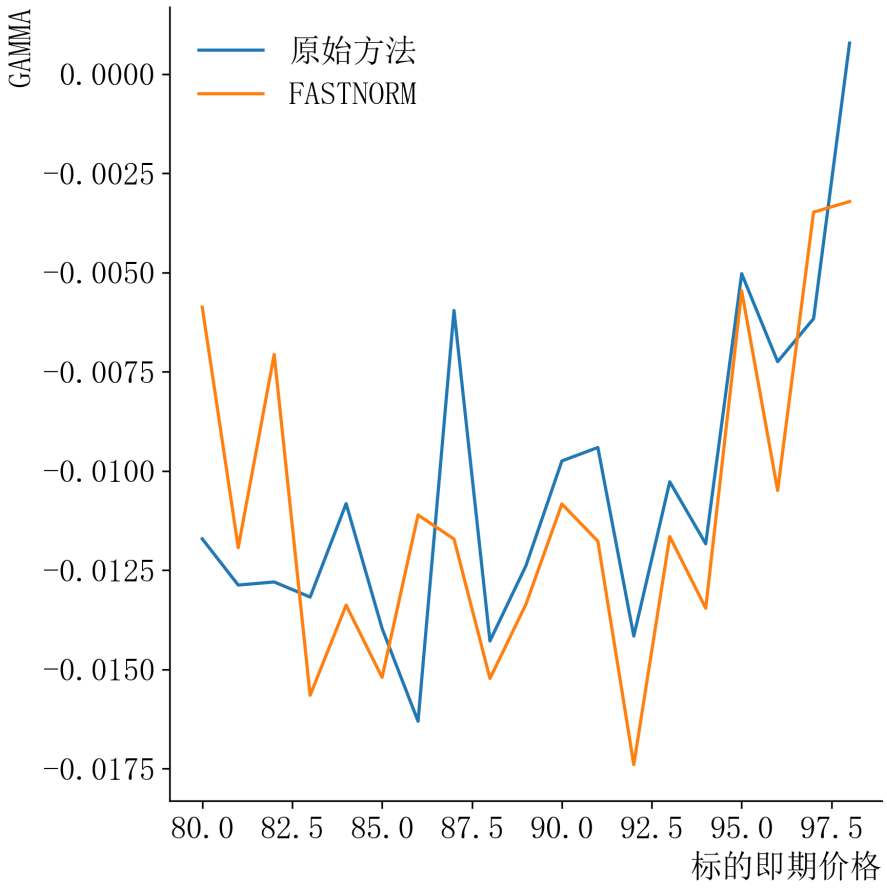
GAMMA
原始方法
FASTNORM
0.0000
−0.0025
−0.0050
−0.0075
−0.0100
−0.0125
−0.0150
−0.0175
80.0
82.5
85.0
87.5
90.0
92.5
95.0
97.5
标的即期价格

图 9: GAMMA

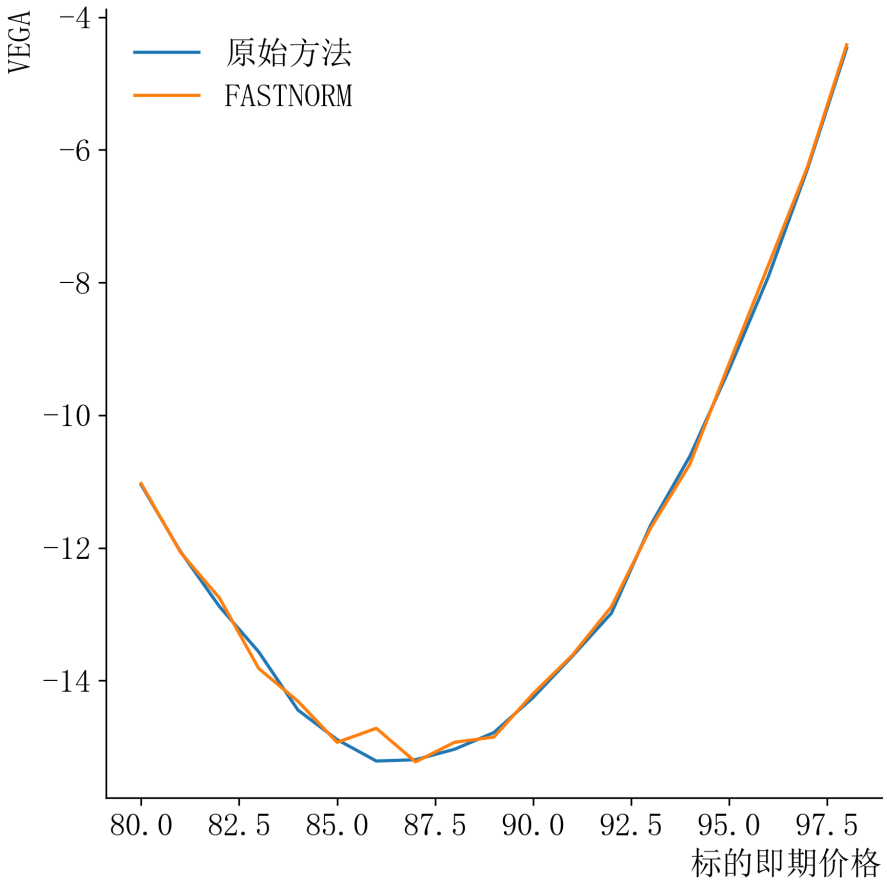
VEGA
−4
−6
−8
−10
−12
−14
原始方法
FASTNORM
80.0
82.5
85.0
87.5
90.0
92.5
95.0
97.5
标的即期价格

图 10: VEGA

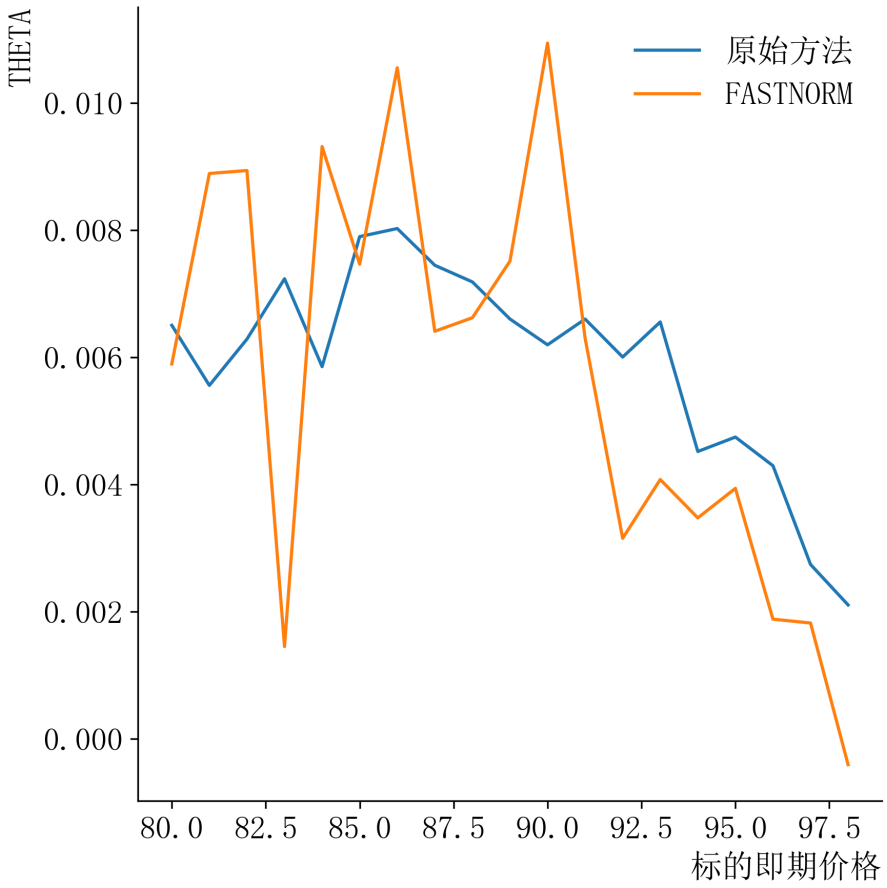
THETA
0.010
0.008
0.006
0.004
0.002
0.000
80.0
82.5
85.0
87.5
90.0
92.5
95.0
97.5
标的即期价格
原始方法
FASTNORM

图 11: THETA

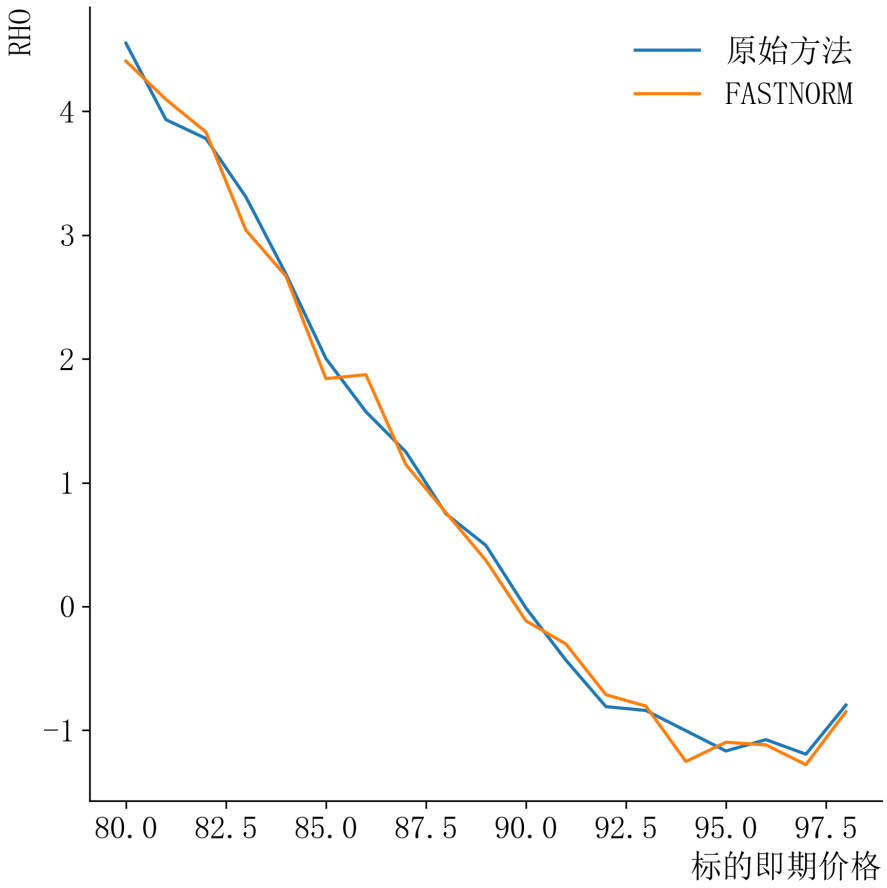
RHO
原始方法
FASTNORM
4
3
2
1
0
−1
80.0
82.5
85.0
87.5
90.0
92.5
95.0
97.5
标的即期价格

图 12: RHO

5 基于 GPU 的蒙特卡洛模拟算法

除了采用现代 CPU 的多核多线程优化蒙特卡洛模拟的计算速率外，下文将研究基于 GPU（graphics processing unit，图形处理器）的蒙特卡洛模拟算法，从而进一步提升蒙特卡洛模拟的适用性和实践价值。

5.1 GPU 计算的优势

无论是个人计算还是商业计算，GPU 已经成为最重要的计算技术之一。GPU 专为并行处理而设计，其高度并行结构能够比 CPU 更高效地并行处理大型数据块的算法，被广泛应用于图形和视频渲染等领域。GPU 的线程数量很多并且能够在线程之间快速切换，因此能够确保硬件一直处于繁忙状态，并且有效地隐藏内存延迟。不仅如此，将这些特点与现代 CPU 中可用的存储器带宽（memory bandwidth）层相结合，从而能够更高效地提升 GPU 的性能。

GPU 是一个可编程的处理器，在 GPU 上能够并行运行数千个核心。GPU 程序也被称为内核（kernel），是通过单程序多数据（Single Program Multiple Data, SPMD）模型进行编写而成。为了避免将 SPMD 和上文提及的 SIMD 混淆，此处对二者进行比较说明。SPMD 允许多个线程使用它们自身的指令指针运行相应的任务，而 SIMD 则恰好相反，SIMD 是完全相同的指令被多次执行，不能像 SPMD 那样同时执行不同的操作。因此，SPMD 的运用更为宽泛。不仅如此，SIMD 定义的是执行模型（execution model），模型中有相应的专门用于并行化的程序和硬件，但是 SPMD 实际上是一个程序描述符（program descriptor）。

SPMD 模型在 GPU 上的工作原理是启动数千个线程运行同一个内核，进而处理不同的数据。SPMD 是一种可用于实现并行的范式，其特征是将任务细分为更小的部分，并从单个程序中并行运行这些部分。经过不断的优化，GPU 为 SPMD 模型提供了非常高的浮点运算吞吐量（throughput），因此，现代 GPU 有着强大的计算能力。

随着技术的不断更新迭代以及现代科学技术的发展需求，所有现代 GPU 均为 GPGPU（General purpose computing on graphic processing units，图形处理器上的通用计算）。GPGPU 方法是用于 GPU 通用目的的计算，除了传统的计算机图形计算之外，还能广泛应用于各科学技术领域中。GPGPU 将 GPU 用于通用计算中不仅能够对部分程序进行加速，还能

在 CPU 上继续运行未加速的程序，进而增强 CPU 架构。最终 GPGPU 通过结合 CPU 和 GPU 的处理能力，创建整体上更快、高性能的应用程序。

上文提到，当采用蒙特卡洛模拟对期权进行定价时，随机数生成这一步骤占用了大部分的计算时长。采用现代 GPU 能够并行加速蒙特卡洛模拟中随机数生成和对数价格路径生成，从而提高计算效率。在标准计算机上生成百万条路径通常十分耗时，若只使用 CPU 无法满足计算高精度的需求。通过使用 GPU 的并行路径可以有效地解决这一问题，因为 GPU 能够非常高效地实现并行化，能够将每个路径分配给单个线程并并行模拟大量的独立路径，进而大幅降低计算机的耗能和计算时间，提升计算效率。

5.2 具体应用：CUDA

基于 GPGPU 对通用计算的极大程度化加速，为了更方便地使用 GPU 编程，本文采用一种在 GPGPU 上进行通用计算的硬件和软件体系架构：NIVIDA CUDA（Compute Unified Device Architecture）。CUDA 是一种在 GPU 上实现通用计算的并行计算平台和编程模型，CUDA 编程模型提供了一个连接软件应用程序与在 GPU 硬件上实现的一个抽象化 GPU 架构，并支持多种编程语言在 GPU 上进行编程，包括 C，C++，Fortran 和 Python。

GPU 并行编程是加速密集型工作负载处理的最佳方法之一。CUDA 架构利用了不同的处理方法，采用一组流多处理器（streaming multiprocessors，简称为 SM）执行相同的指令集，包括处理不同数据区域上多线程的分支条件，Nivida® 创造了单指令多线程（single instruction multiple threads，SIMT）这个术语用来描述这种架构。由于所有的指令都来自独立的指令流，SIMT 在 GPU 上取得了很大的进展。

目前，最新 NIVIDA GPU 能够每秒执行几万亿次的浮点运算，这得益于现代 GPU 的硬件架构：单个硅芯片（scilicon chip）上拥有数百个核的大规模并行架构。CUDA 平台是一个软件层，它可以直接访问 GPU 的虚拟指令集和并行计算元素，以执行计算内核。

对于 GPU 编程的随机数生成器而言，需要有良好的统计特性、高效的计算速度、较低的内存占用等特点。在 CUDA 中，cuRAND 库提供了 GPU 加速的高性能随机数生成器，该库能够使用简单并且高效的方法生成高质量的伪随机数和准随机数。在 GPU 上创建随机数的优点在于随机数是在 GPU 上创建的，不需要调用内存来加载全局内存。

操作系统	Ubuntu 22.04.1 LTS, 64-bit
处理器	12th Gen Intel(R) Core i7-12700
内存	16 GB 3200 MHz DDR4
图形处理器	Mesa Intel(R) Graphics (ADL-S GT1)

表 3: 期权定价参数

5.3 分析比较

插入图片：带有 GPU 相关结果的图]

6 结论与致谢

本文从软件和硬件对蒙特卡洛模拟算法进行了最大程度的优化，实现了计算速率上的大幅提升。

本研究由安信证券股份有限公司与上海锰链科技有限公司联合支持。

7 参考文献

附录

（1）测试机器参数

（2）具体结果