随机游走模型计算优化实验报告

Napreth

2025年10月23日

摘要： 本文针对二维随机游走模型的计算性能进行了系统优化。依次采用 NumPy 向量化、CuPy GPU 加速、C 语言重写及汇编 SIMD 并行，并在最终版本中加入多线程优化。实验结果表明，运行时间由 28.7s 降至 0.0075s，整体加速超过 3800 倍。本实验使用了从 Python 到底层指令的多层次优化方式，展示了硬件并行在随机模拟计算中的显著性能提升。

Abstract: This report presents a performance optimization study of a two-dimensional random walk model. Sequential improvements include NumPy vectorization, CuPy GPU acceleration, C reimplementation, assembly-level SIMD, and multithreading. The final version reduced runtime from 28.7 s to 0.0075 s, achieving over 3800× speedup. The experiment demonstrates a hierarchical optimization process from Python to low-level instructions, highlighting the effectiveness of hardware parallelism in random simulation computation.

1 实验目的

掌握随机游走模型的计算原理，理解蒙特卡洛法在随机过程模拟中的应用，并通过多种性能优化技术提升计算效率。

2 实验原理

随机游走模型是一种典型的 **蒙特卡洛模拟（Monte Carlo Simulation）**，通过大量独立样本的随机运动统计整体系统的稳定分布。在本实验中，假设有 N 个粒子在 $L \times L$ 的二维周期性网格中独立运动，每一步粒子随机选择上下左右四个方向之一移动一格。通过重复 T 步的模拟，可以近似估计粒子在特定区域（例如中心区域）的平均占据概率。

这种方法不追求解析解，而是通过随机采样逼近概率分布。当 N 和 T 足够大时，模拟结果可视为平衡分布的数值近似。本实验的目标是高效地完成这一随机过程模拟，并通过优化手段显著降低计算时间。

其统计指标定义如下：

$$R = \frac{\text{中心区域步数总和}}{N \times T}$$

其中 R 表示粒子在中心区域的平均停留比例。

3 实验环境

系统版本：WSL 2.6.1.0，Ubuntu 22.04
开发环境：Python 3.10.12，GCC 11.4.0，GNU assembler 2.38, CMake 3.22.1
处理器：Intel(R) Core(TM) Ultra 9 275HX, x86-64 架构, 24核24线程
GPU：NVIDIA GeForce RTX 5060 Laptop GPU

4 实验过程

4.1 性能测试数据

实现版本	重复次数	平均停留比例	平均模拟时间 (s)	加速倍数
Python baseline	10	0.25015	28.6868	1.00x
Python NumPy	1000	0.24999	0.6703	42.80x
Python CuPy	1000	0.24999	0.3049	94.09x
C baseline	1000	0.25003	0.5334	53.78x
C ASM	1000	0.25003	0.5173	55.46x
C ASM SIMD	1000	0.25003	0.0796	360.17x
C ASM SIMD 24 Thread	1000	0.24998	0.0075	3814.27x

Table 1: 不同实现版本的性能对比结果

可以看到，经过多阶段优化后，运行时间从 28.7s 降至 0.0075s， 其中 SIMD 向量化与多线程并行带来了最大幅度的性能提升。

4.2 优化过程

本实验从 Python 基线版本开始逐步进行了多层次的性能优化。 首先采用 Python 科学计算中最常用的库 NumPy，通过向量化操作替代显式循环，从而显著降低了解释器执行开销。 在此阶段，运行速度较基线版本提升约四十倍。

随后，为进一步利用硬件并行性，引入了 GPU 加速方案，并从 PyTorch、Numba 与 CuPy 等框架中选择 CuPy，其优势在于与 NumPy 高度兼容，只需极少量代码修改即可实现大规模并行计算。 使用 GPU 加速后，模拟运行时间进一步缩短，展现出显著的并行计算潜力。

考虑到 Python 语言本身存在的解释与类型管理开销，其在高性能计算场景中仍具有结构性瓶颈。 由于随机游走模型的逻辑结构相对简单，C 语言版本的实现难度与 Python 相差不大，且性能相较 Python 方案有明显提升。 因此，后续阶段将主要计算部分改写为 C 语言实现。 C 语言基线版本的执行速度超过 NumPy 实现，但仍略低于基于 CuPy 的 GPU 版本。

在此基础上，优化方向被分为两个主要路径：多线程并行化与更底层的指令级优化。 为消除编译器抽象层带来的性能开销，实验引入汇编语言进行底层实现。 编写了 `gen_rand_arr.s` 与 `simulate.s` 两个核心模块，用于随机数生成与模拟计算。 初始的汇编基线版本性能略低于 C 语言实现，分析认为主要原因在于编译器自动优化带来的优势。

为此，进一步采用 SIMD（Single Instruction Multiple Data）技术，基于 AVX2 指令集实现数据级并行计算。 SIMD 向量化显著提升了计算吞吐量，使汇编版本的性能超越所有先前方案。

最终在 C 端引入无锁多线程并行机制，结合预先在汇编模块中预留的线程扩展接口，形成了完整的并行优化方案。综合优化后，运行时间由最初的 28.7 秒降至约 0.0075 秒，实现了超过 3800 倍的性能提升。

4.3 多线程性能分析

根据实验数据，24 线程仅比单线程快约 10 倍，而非理想情况下的约 24 倍。为进一步分析原因，对 C ASM SIMD 多线程版本在不同线程数下进行了详细测试，记录了单线程与多线程的运行时间、CPU 占用率以及并行效率。测试结果如图 1 所示：第一行左图为运行时间随线程数的变化，右图为 CPU 占用率；第二行为并行效率。

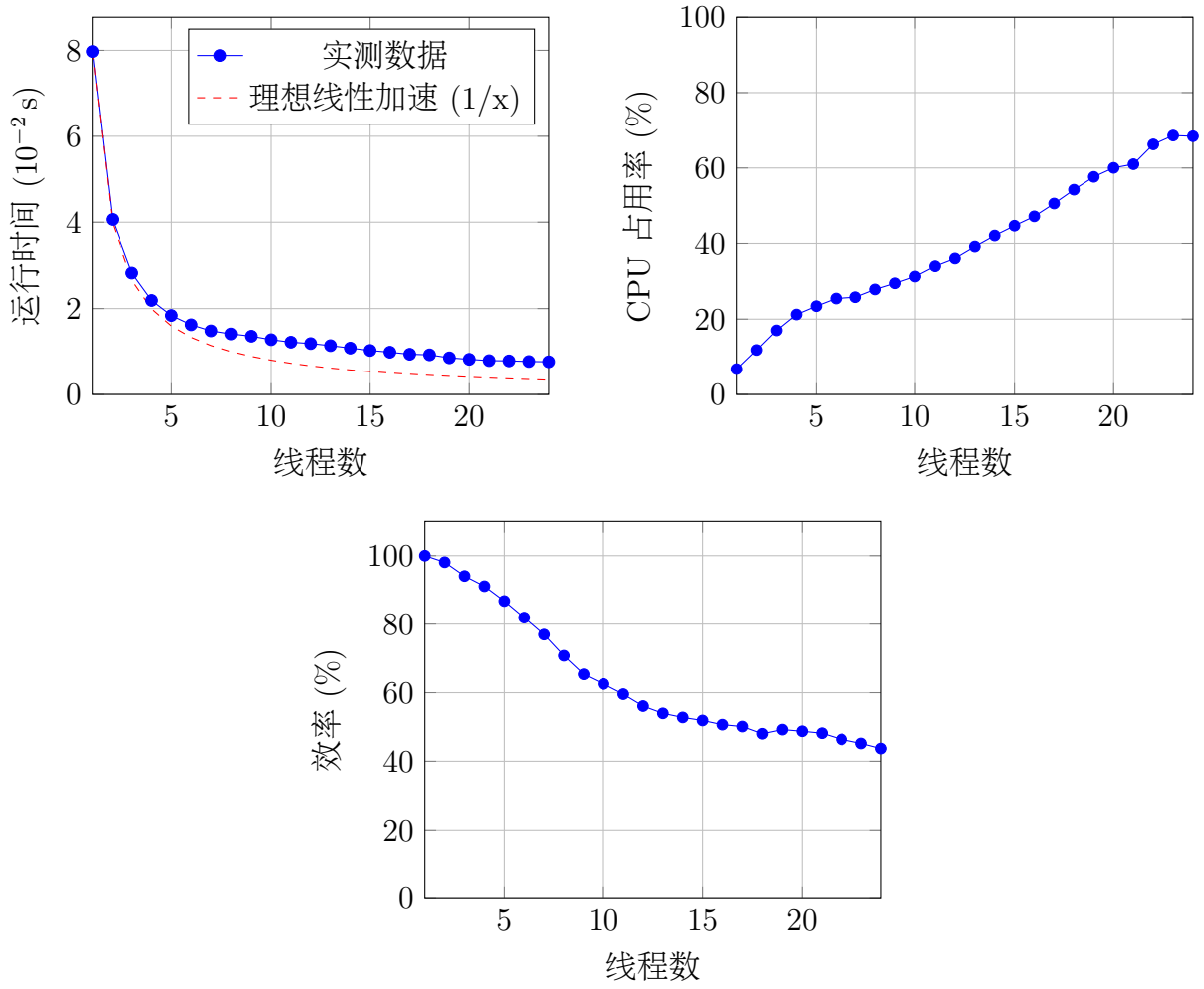

Figure 1: 多线程运行性能对比：上行为运行时间与效率，下行为 CPU 占用率

测试结果表明，实测的运行时间—线程数曲线与理想反比例曲线基本拟合，仅在线程数较多的区域存在轻微偏差。由此得出的解释是：该实现已达到较高的绝对性能水平，此时内存带宽、线程调度与缓存行为等因素的开销会在最终加速结果中被显著放大，因而出现右图所示的并行效率随线程数递减的现象。可以看到，当线程数增加至 23∼24 时，CPU 占用率已达到约 68%，说明该实现已经充分利用了大部分硬件计算资源。而在高线程区域中，CPU 占用率增幅减缓甚至略有下降，这更表明了此时性能瓶颈主要来自内存带宽、线程调度及缓存竞争等因素。

4.4 进一步优化方向

- 在 C 语言端使用 GPU 加速，C 语言 CUDA 作为原生接口，可直接编译为 GPU 机器码，避免了 CuPy 作为 Python 库所引入的解释器调用开销与抽象层封装成本；同时能通过 C 语言对显存分配、数据传输、线程块划分及共享内存使用等进行精细化控制，减少不必要的资源消耗与冗余操作，而 CuPy 为兼顾易用性，其自动管理机制往往难以实现同等程度的底层优化，因此在性能上存在天然差距。
- 使用频域幂法而非蒙特卡洛法。频域幂法（Frequency-Domain Power Iteration）结合了幂法迭代与频域计算的思想，用于高效求解具有卷积或扩散结构的线性系统。

  传统幂法通过迭代

  $$\mathbf{v}_{k+1} = \frac{\mathbf{A}\mathbf{v}_k}{\|\mathbf{A}\mathbf{v}_k\|} A$$

  逼近矩阵 $\mathbf{A}$ 的主特征向量，从而提取系统的主要模式或稳态特征 [1]。 当 $\mathbf{A}$ 含有卷积型算子时，直接在时域计算 $\mathbf{A}\mathbf{v}_k$ 代价较高。 根据卷积定理，可将该运算转至频域执行：卷积在频域中等价为逐频率点的乘法， 并可通过快速傅里叶变换（FFT）将复杂度由 $O(N^2)$ 降至 $O(N \log N)$ [2]。

  因此，频域幂法在迭代结构上保持幂法的收敛特性， 同时利用频域乘法实现加速，兼具特征提取与高效计算的优点， 适用于随机游走、扩散及其他大规模物理过程的求解。

5 优化技巧

在优化过程中，总结出若干具有普适性的性能提升方法：

5.1 编译器优化选项

在 C/C++ 与汇编集成构建中，启用以下编译与链接优化选项可显著降低循环与函数开销、提升指令级并行与向量化覆盖度，并减少不必要的运行期开销（如 PLT/间接符号开销）：

```
-O3 -Ofast -flto -march=native -mtune=native
-funroll-loops -fomit-frame-pointer -ffast-math
-fno-math-errno -fno-trapping-math -fno-stack-protector
-falign-functions=32 -falign-loops=32
-ftree-vectorize -funsafe-math-optimizations
-fno-plt -fno-semantic-interposition -fopenmp
```

其中关键选项作用说明：

- `-O3 -Ofast`：启用高级别优化，包括循环展开、函数内联等
- `-march=native`：针对本地 CPU 架构生成专用指令集
- `-flto`：启用链接时优化，跨编译单元进行全局优化
- `-ffast-math`：牺牲部分浮点数精度换取计算速度提升
- `-fopenmp`：启用多线程并行支持

5.2 使用 Xorshift 算法高效生成伪随机数

随机游走属于典型的蒙特卡洛模拟，对伪随机数序列质量与吞吐有较高要求。实现中采用了 **Xorshift** 算法以获得高吞吐的均匀分布伪随机数，相比调用标准库接口可显著降低函数开销并改善指令流水效率。

Xorshift 是 Marsaglia (2003) 提出的一种伪随机数生成算法， 该算法结构极为简洁，仅由异或（XOR）与移位（SHIFT）两种基本位运算组成， 通过在 GF(2) 上构造线性递推关系，实现了在保证高速性的同时具有长周期的伪随机序列生成 [2]。

5.3 通过比较临界值实现周期边界

正常思路下周期边界需要对 L 取模，但是由于每一步中每个粒子只移动一格，因此可以直接判断坐标是否为 L 或 -1，结合掩码即可高效实现。

5.4 使用位运算技巧优化计算性能

在向量化与多线程并行场景下，分支与高延迟算子（如整数除法）会放大性能损失。通过将条件分支、取模/除法等重写为 **位运算与加减运算** 的形式，可降低分支失误与长延迟指令对流水线的影响，并提升 SIMD 指令的可向量化区域。常见技巧包括:

- 对 N 取模（N 为2的幂）:

  `var & N-1;`

- 向下取整到 N 的倍数（N 为2的幂）:

  `var & ~(N-1);`

- 判断是否为 2 的幂:

  `(var & (var-1)) == 0;`

- 根据第一位构造全0/全1掩码——取负;

  `-var;`

  `1(0b00000001) -> -1(0b11111111)`

- 位运算取负（补码规则）:

  `~var + 1`

- 反转指定位——用 1 异或该位:

  反转第 2 位: `var ^ 0b00000010;`

  反转全部位: `var ^ 0b11111111`

- 向上取整到 N 的倍数（N 为2的幂）:

  `(var+N-1) >> 2;`

6 开发中遇到的问题

本实验中的问题集中在汇编优化阶段，主要包括以下几个方面:

6.1 使用汇编语言计算性能反而下降

在汇编开发初期，我首先实现了基于 64 位寄存器的串行随机数生成版本。 随后尝试通过“越界访问”方式进行向量化改写：即将粒子坐标改为 16 位寄存器存储，并利用 64 位寄存器一次生成并写入 4 个随机数。 原本预期此方法能显著提升生成效率，但结果出乎意料——首次编译后的性能甚至较 C 语言基线版（约 0.5s）更慢，达到约 0.7s。 在恢复至 64 位实现后，运行时间提升回约 0.5s。

我认为这并非算法问题， 可能是由于 64 位整数在 C 语言端的内存访问与寄存器操作更为匹配， 而 16 位实现的逻辑越界访问反而增加了寻址和数据搬运的开销。至于性能差异的根本原因尚未完全验证（若要深入，需要重写 `gen_rand_arr.s` 与 `simulate.s` 的 16 位版本并重新测试）。

6.2 SIMD 向量化导致部分逻辑失效

在将串行汇编改写为 SIMD 向量化版本的过程中， 原有的部分计算逻辑（如条件跳转、乘除运算）在 SIMD 指令集中无法直接使用。 我将这些操作重新改写为基于位运算与加减运算的等价逻辑，从而兼顾并行执行与正确性。

6.3 指令集兼容性问题

我的 CPU 最高支持 AVX2 指令集，但由于参考资料来源不一， 在初次实现时误用了部分 AVX512 指令（如 `vpcmpq`）， 编译时出现 “illegal instruction” 报错。 经查阅 Intel 官方手册后，将相关指令替换为 AVX2 可用版本后问题解决。

6.4 多线程崩溃与内存对齐问题

在完成 SIMD 版本后，程序在 8 线程下运行正常， 但当线程数设置为 6、7、16 或 24 时均出现段错误（Segmentation fault）。 检查逻辑后发现并非计算逻辑错误，而是内存对齐问题： 汇编中使用了对齐加载指令 `vmovdqa`， 但分配的内存并未保证 32 字节对齐。 针对这一问题，我考虑了两种方案： (1) 在 C 端强制调整任务分配，使每个线程处理的数组段长度为 4 的倍数，仅最后一段例外； (2) 直接改用非对齐加载指令 `vmovdqu`。 最终选择了后者，保证了多线程版本在所有线程数下稳定运行。

7 实验收获

通过本次实验，我系统地理解了 **高性能计算的分层优化思路**。 在算法层面，随机游走模型体现了蒙特卡洛方法通过大量随机样本近似统计特征的思想； 在语言层面，掌握了 NumPy 与 CuPy 的基本用法与性能差异，并学习了 PyQt 的界面开发方法（虽未在最终版本中使用）； 在体系结构层面，完成了汇编语言的入门实践，掌握了 SIMD 指令集的基本使用方式及其在数据并行中的应用。

实验过程中，我对跨语言项目的构建流程有了较为清晰的认识。 理解了从源代码到可执行文件的编译与链接过程，包括编译器如何将多语言模块分别生成目标文件（.o）、静态库（.a）与动态链接库（.so），以及这些模块在最终可执行程序中的装载与调用方式。 并熟悉了寄存器操作、位运算技巧以及数据对齐与加载方式等底层优化细节。

总体而言，本实验使我掌握了从高层语言到底层指令的性能优化路径， 并加深了对现代计算机体系结构、编译系统以及并行执行机制的认识。

8 结论

本次实验通过分层优化展示了从 Python 到底层并行优化的全过程， 性能提升近 4000 倍。通过 SIMD 向量化与多线程设计， 充分利用现代 CPU 的计算资源，为高性能随机模拟提供了可行路径。 多线程加速虽未达到理想线性比例，但已验证并行方案的正确性与性能极限。

9 参考文献

[1] F. Lin and W. W. Cohen, "Power iteration clustering", in *Proc. 27th Int. Conf. Mach. Learn. (ICML)*, Haifa, Israel, 2010, pp. 655–662. [Online]. Available: `https://dl.acm.org/doi/10.5555/3104322.3104406`.

[2] P. Costa, "A FFT-based finite-difference solver for massively-parallel direct numerical simulations of turbulent flows", *Computers and Mathematics with Applications*, vol. 76, no. 7, pp. 1853–1862, 2018, doi: 10.1016/j.camwa.2018.07.034.

[3] G. Marsaglia, "Xorshift RNGs", *Journal of Statistical Software*, vol. 8, no. 14, pp. 1–6, 2003, doi: 10.18637/jss.v008.i14.

10 附录：项目仓库

源代码与测试脚本已开源至 GitHub： `https://github.com/Napreth/RandomWalkModel/`

注：本项目的测试脚本由 AI 生成。