arXiv:2603.04782v1 [cs.DC] 5 Mar 2026

解锁 Python 的核心：移除 GIL 的硬件使用与能耗影响

José Daniel Montoya Salazar[0009-0003-5902-7312]montoyasalazarjosedaniel@gmail.com

独立研究者

摘要。*Python* 的全局解释器锁（GIL）会阻止*Python* 字节码同时在多个 CPU 核心上执行，即使使用了多个线程也是如此。不过，从*Python* 3.13 开始，实验性构建版本允许禁用 GIL。尽管此前的研究探讨了禁用 GIL 对加速效果的影响，但其对能耗和硬件利用率的影响受到的关注较少。本研究测量了执行时间、CPU 利用率、内存使用量和能耗，涵盖四类工作负载：基于 NumPy 的计算、顺序内核、多线程数值工作负载和多线程对象工作负载，并对比了启用 GIL 的构建版本与自由线程构建版本的*Python* 3.14.2。

结果凸显了一种权衡。对于处理独立数据且可并行化的工作负载，自由线程构建版本可将执行时间缩短最多 $4\times$，能耗也相应降低，同时实现有效的多核利用率，但代价是内存使用量增加。相比之下，顺序工作负载无法从移除 GIL 中获益，能耗反而增加 13–43%。同样，对于线程频繁访问并修改相同对象的工作负载，由于锁竞争，性能提升会减弱，甚至出现退化。在所有工作负载中，能耗都与执行时间成正比，这表明即使 CPU 利用率提高，禁用 GIL 也不会显著影响功耗。就内存而言，无 GIL 构建版本的内存使用量普遍增加，虚拟内存的增幅比物理内存更明显。这一增长主要归因于逐对象加锁、运行时中额外的线程安全机制，以及新内存分配器的采用。

这些发现表明，*Python*的无 GIL 构建并非在所有情况下都能带来改进。开发者在采用之前，应评估自己的工作负载能否有效受益于并行执行。

1 引言

多年来，由于能效提升，数据中心的电力需求一直相对稳定。然而，根据法国国际关系研究所 （IFRI）的预测，这家法国国际关系智库指出， [20], 自2019 年后，随着云计算和

2 何塞 丹尼尔Montoya萨拉萨尔 montoyasalazarjosedaniel@gmail.com

人工智能（人工智能）迅速发展，这一趋势发生了逆转。目前，信息与通信技术行业约占全球电力消耗的9%，其中数据中心约占1%–1.3%。在美国，到2030年，数据中心的电力消耗可能占总电力消耗的13%，而2024年约为4%。在欧洲，仅人工智能相关计算就可能占总电力需求的4%–5% [20]。这些预测表明，软件基础设施正日益成为全球能源需求的重要来源。

编程语言会影响这一能耗足迹，因为它们决定了计算资源的使用方式。$Python$是使用最广泛的编程语言 [21, 23], ，与低级语言相比，它更注重开发者的生产力，而非执行效率。Pereira等人 [14]报告称，$Python$ 程序的能耗可能远高于等效的C 语言实现，例如，观察到的差异接近 76×。在他们的研究中，在所考察的语言中，$Python$ 的能效排名倒数第二，其中包括$C++$、$Rust$、$Java$等。Kempen 等人 [6]对这一结果的因果解释及其差异幅度提出了质疑。该研究认为，观察到的能耗差异主要源于执行时间、运行时行为和实验设置的差异，而非语言本身的固有属性。虽然这些论点正确且有充分依据，但它们也表明，如果$Python$ 运行得更快，自然就会消耗更少的能量。这就引出了一个问题：在实际环境中，$Python$ 是否真的能运行得更快。

在某些领域，例如人工智能，设计选择已经缓解了高能耗操作带来的影响。该领域常用的$Python$ 库主要将$Python$ 用作高级接口，并把计算密集型任务交给C、$C++$或$Rust$ 代码处理，且通常借助图形处理器加速。Kempen 等人 [6], 也强调了这种方法，它能减少$Python$的运行时开销对这些工作负载的影响。然而，人工智能并非$Python$的主要应用场景。根据 2024年 Python开发者调查 [17]„ 以$Python$ 为主要语言的开发者从事的领域十分广泛。按报告在各类别中使用$Python$ 的开发者占比计算，这些领域包括数据分析（49%）、Web开发（48%）、机器学习（42%）、数据工程（33%）、网页抓取（28%）和学术研究（28%）。其中有些领域并非以扩展库为主导。

在这些领域中，性能取决于$Python$自身的解释器，而全局解释器锁（GIL）是其关键架构组成部分。它会阻止多个线程在单个进程中同时执行$Python$ 字节码，从而限制多核CPU的有效利用率。从$Python\ 3.13$ 开始，并延续至$Python\ 3.14$，$CPython$ 提供了一种可选的自由线程构建版本，其中禁用了全局解释器锁，从而支持$Python$线程并行执行 [3]。

鉴于此，并受提升软件效率的环境重要性所驱动，本研究探讨移除全局解释器锁能否提升硬件利用率和能效。虽然现有研究（如 [5,9,10] ）侧重于比较启用全局解释器锁的构建与自由线程构建的执行时间，

但对硬件利用率或能耗的关注较少。本研究旨在弥补这一空白。此外，本文还回顾了影响软件能耗的因素，以说明为何移除全局解释器锁会让人们期待这方面有所改善。

2 文献综述

本节回顾决定软件能耗的主要因素。它也为分析移除全局解释器锁可能如何影响$Python$的硬件使用与能量行为奠定基础。

2.1 是什么使软件消耗能量?

执行时间 Pereira等人 [14]表明，执行时间是软件能耗的主导因素，但同时也证明它不能被视为唯一因素。他们的结果挑战了“更快就总是更环保”这一直观假设，展示了执行时间更长的程序仍能实现更低总体能耗的案例。Kempen 等人 [6] 认为，能效并非编程语言的固有属性，并强调了编译器选择、运行时系统以及包括电压和频率调节在内的硬件机制等因素的影响。当这些因素受到严格控制时，他们的实验表明能耗与执行时间成正比。这一观察直接源于能量的物理定义，如公式 1 所示。

$$Energy(J) = Power(W) \cdot Time(s) \tag{1}$$

然而，在现实场景中，公式 1 中的功耗项既不能假定为恒定，也无法完全由开发者控制。语言实现会引入运行时开销：解释执行、垃圾回收或内存管理，这些因素可能会独立于执行时间增加平均功耗。因此，尽管执行时间足以用于受控的实验比较，但在评估真实软件系统的能耗表现时可能并不够。

CPU 使用率 Pereira等人 [14]报告的一项观察结果是，与 CPU 相关的能耗占软件总能耗的大部分，约为 88%，其余部分则归因于动态随机存取存储器。这表明，CPU活动是能耗的主要来源。

这种消耗随 CPU 利用率增加。Dargie [2] 和 Ou等人 [13]报告了 CPU 使用率与所消耗能量之间的明确关系：在其实验条件下，双核系统近似呈线性行为，单核系统则呈二次行为。图 1 展示了某移动处理器的这一关系。

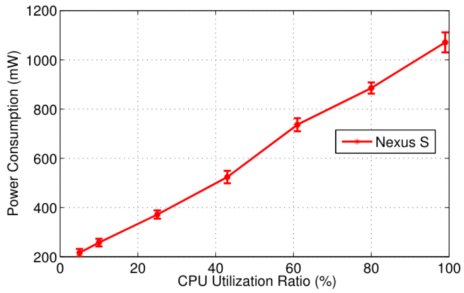

图1：移动处理器的能耗与 CPU 使用率的关系。摘自 [13]。

那么温度呢？既然它与能量耗散直接相关，是否应将其视为一个需要考虑的独立因素？De Vogeleer 等人 [22] 表明，在某些处理器中，功耗对温度呈指数依赖。此外，随着温度升高，动态电压或频率调节等机制可能被触发，从而对执行时间产生负面影响，并可能增加总能耗。

但实证测量证实，在更高的工作负载下，CPU 利用率、功耗和温度会同步上升，这表明现代处理器中利用率与热行为之间存在很强的关联 [4]。因此，控制 CPU 使用率也意味着控制温度和功耗，并通过激活热机制来减缓执行时间的恶化。

内存使用量 关于内存，有两个主要方面可能影响其能耗：峰值内存使用量，以及随时间推移的连续内存访问。Pereira等人 [14] 报告称，峰值内存使用量与动态随机存取存储器能耗之间不存在稳定的相关性，这表明内存峰值并不能可靠地预测能源需求。

相反，内存对能量的影响主要与执行期间访问内存的方式有关。CPU 缓存能够以最快的速度访问数据； [15],如果所需数据未在缓存中找到，就会发生缓存未命中，处理器必须访问速度较慢的内存层级，从而延长执行时间。此外，如果程序的内存需求超出可用随机存取存储器，系统可能会使用内存交换，在随机存取存储器与磁盘存储之间移动数据 [1]。由于磁盘访问比随机存取存储器慢几个数量级，交换会增加执行时间和能耗。

这就是为什么 Kempen 等人 [6] 指出，内存相关能耗主要取决于内存活动（执行时间和 CPU 使用率），而非仅由内存占用决定：经过良好优化的程序即使占用大量内存，也能保持出色的缓存局部性，从而降低内存活动和执行时间。

与 CPU 活动相比，内存对功耗的影响较小，因此研究能耗时，关注执行时间和 CPU 功耗是合理的。内存行为是次要因素，其对总体能耗的影响取决于访问局部性。

并行处理 现代中央处理器依靠多个核心来利用并行性，这在能耗方面既带来了机遇，也带来了挑战。

Konopik 等人 [7] 在一项分析性研究中表明，在理想化条件下，提高并行程度可以通过缩短执行时间来降低总能耗；即使考虑到在工作线程之间分配负载所需的开销也是如此。尽管实际系统可能偏离这一理想情况，但开发者已广泛接受这一权衡。

然而，决定使用多少个核心并非易事。在扩展公式 1 并忽略内存功耗贡献的情况下，多核系统中的总能耗可以表示为：

$$\text{Energy} = (\text{Power}_{\text{CPU-0}} + \ldots + \text{Power}_{\text{CPU-N}}) \times \text{Time} \tag{2}$$

这一表述凸显了一个根本性的权衡：增加活跃核心的数量会影响系统的整体功耗，而减少活跃核心的数量通常会增加执行时间。正如 Kempen 等人 [6], 所指出的，活跃核心的数量是决定功率行为的一个相关因素，因此核心利用率的变化会影响总能耗。此外，空闲核心并非不消耗能量，因为即使在零计算负载下，漏电流和功率耗散依然存在 [7]。

因此，并不存在普遍适用的最佳并行性水平，最节能的配置需要针对具体情况进行评估。但显而易见，并行化可以通过缩短执行时间来降低能耗。**这正是从 *Python* 中移除全局解释器锁令人期待能效提升的原因**。

软件能耗主要取决于执行时间，同时还受到 CPU 利用率、内存访问行为和并行执行程度的影响。其他因素，例如中央处理器温度和电源管理机制，也会影响能耗，但它们主要反映的是 CPU 活动，而非软件层面的独立成因。

这些因素由编程语言实现及其运行时系统决定。正如 Kempen 等人所示, [6], 在控制处理器频率和核心利用率等外部因素后，能耗在很大程度上与执行时间成正比，且没有哪种编程语言实现天生比另一种更节能。然而，在实际场景中，不同语言实现的运行时行为各不相同，例如内存管理策略、并行执行限制等。这些因素会影响执行时间、CPU 利用率、对可用硬件的使用以及能耗。

2.2 Python 全局解释器锁

在标准$CPython$ 构建中，全局解释器锁会阻止单个进程中的多个操作系统线程同时执行$Python$ 字节码，使多线程程序无法利用中央处理器中的可用核心。但从$Python\ 3.13$开始，$CPython$ 提供了可选的自由线程构建，可以禁用全局解释器锁 [3,16]。这种模式能够充分利用多核硬件，但与默认构建相比会引入运行时开销。

为了实现自由线程，PEP 703 [3]描述了多项运行时变更。本文将介绍其中的主要变更，以指出移除全局解释器锁可能带来的开销来源和加速效果。

首先，为了在没有全局解释器锁的情况下安全运行，$CPython$ 必须为先前受全局锁保护的操作添加同步机制和线程安全数据管理。PEP 703 描述了引用计数和内存管理方面的变更，以确保多个线程同时运行时对象生命周期和内存分配仍然正确；它还要求对循环垃圾回收进行额外协调 [3]。其次，为了让线程能够并发推进，自由线程构建将单一全局锁改为局部机制，例如为共享容器使用对象级锁，使互不相关的操作可以在不同的 CPU 核心上执行，同时确保对同一对象的并发修改安全 [3]。

自由线程构建的一项重要变更是将 mimalloc 用作默认内存分配器 [11]„ 它旨在扩展到支持大量并发线程。Leijen 等人 [8] 报告称，与其他主流分配器相比，mimalloc 的延迟性能具有竞争力，甚至更胜一筹。Rosner [19] 指出，这些优势主要体现在小型内存分配上（约 1 KB 以下），而这类分配在典型的 $Python$ 工作负载中占主导地位。在 Linux 上，自由线程构建会启用积极的内存区域预留和提交。正如 mimalloc 文档 [11], 所述，这可能会为内存区域管理预留一个较大的虚拟内存区域（约 1 GB）。这可能会增加虚拟内存大小（VMS），但并不意味着常驻集大小（RSS）会相应增加，除非实际访问了预留的内存。

这些设计选择带来了一些预期。当工作负载受中央处理器限制，并且可以在共享有限的情况下分配到多个线程中时，禁用全局解释器锁应能提高整体 CPU 利用率，并可能缩短执行时间。然而，由于自由线程执行会增加开销，那些无法从线程级并行中受益，或高度依赖共享对象的工作负载，可能只能获得较小的加速，甚至出现减速。自由线程 HOWTO 以 pyperformance 测试套件的平均减速幅度报告了这项开销：从 macOS aarch64 上约 1%，到 x86-64 Linux 系统上约 8% [16]。从能量角度来看，禁用全局解释器锁可能会增加瞬时功率，因为可以同时使用更多 CPU 核心。然而，如果执行时间的缩短幅度足够大，总能耗仍可能下降。当并行加速受到同步开销限制时，这种缩短可能不会发生，反而可能导致总能耗增加。

这种权衡引出了一个问题：移除全局解释器锁所带来的中央处理器和内存开销，是否足以抵消并行性带来的执行时间收益。由于这种平衡预计取决于工作负载，因此有必要开展本研究所呈现的实验比较。

3 研究问题

– **RQ1：** 与启用 GIL 的构建相比，Python 的自由线程构建是否能在不同工作负载和执行配置下减少能耗？– **RQ2：** 当启用 GIL 的构建与自由线程构建之间的执行时间存在差异时，这些差异是否会转化为成比例的能耗变化？– **RQ3：** 启用 GIL 的构建与自由线程构建在硬件资源利用率方面有何差异，具体体现为中央处理器和内存使用量的差异?

4 方法

本节详细介绍测量设置、待测试的场景，以及用于报告无 GIL/GIL 比率的统计分析。所有源代码、分析器插桩和基准测试场景均已公开提供 [12]。

4.1 硬件和配置

所有实验均在一台运行 Ubuntu 24.04.3 LTS 和 Linux 内核 6.14.0-37-generic、未安装图形界面的 x86_64系统上进行。该机器配备英特尔酷睿 i7-8750H 处理器（6 个物理核心，通过超线程提供 12 个硬件线程，最高频率为 4.1GHz）、16 GiB 随机存取存储器和 1 TB 机械硬盘存储。该系统采用混合显卡，配有英特尔 UHD Graphics 630 集成图形处理器和英伟达 GeForce GTX 1050 Ti Mobile 独立图形处理器。

Python 3.14.2 用于两种配置：默认的启用 GIL 的构建，以及使用 –disable-gil 编译的自由线程版本。未对任何实现进行修改。对于无 GIL*Python* ，一项启用 $MIMALLOC_SHOW_STATS$和 $MIMALLOC_VERBOSE$的快速测试证实，默认情况下 $MIMALLOC_ARENA_EAGER_COMMIT$设置为 2。

4.2 将性能分析器作为提取数据的工具

我们开发了一个自定义的 *Python* 性能分析器，采用基于采样的方法，以尽量减少测量干扰。它每隔 50 ms 采集一次样本，这一间隔既能提供足够的采样频率，又能限制测量开销。实际上，每次数据收集约增加 2ms，使有效间隔达到 $\sim$52ms。所有数据收集机制的设计都旨在尽量减少对被测程序执行的影响；一项使用

10s $sleep$ 进行的测试确认，性能分析器不会给目标进程带来可测量的噪声开销。[1] 这是因为性能分析器负责启动待分析的进程，随后获取其标识符，并使用 psutil 库（版本 7.1.3 [18]）收集仅与该进程相关的统计数据。

下面将介绍本研究中收集和分析的指标：

– **执行时间：** 每次采样迭代时，都会记录一个纪元时间戳。在后处理中，这些时间戳会转换为以秒为单位的已用执行时间。– **CPU 使用率：** 通过 psutil 获取目标进程的 CPU 利用率，并按可用 CPU 核心数进行归一化。采用 $interval = 0.0$ ，以确保测量不会阻塞。– **虚拟内存大小（VMS）：** 进程预留的虚拟内存峰值。– **常驻集大小（RSS）：** 进程当前使用的物理内存（随机存取存储器）峰值。– **交换内存：** 被换出到磁盘的进程内存峰值。– **能耗：** 通过 Linux sysfs 接口（/sys/class/powercap/intel-rapl:*/energy_uj）使用 Intel RAPL 测量，因此这是一个系统级统计值。为避免 Kempen 等人 [6],讨论的一些问题， μJ 中的读数会以整数形式保留，只有在相邻读数相减后才会转换为浮点数；计数器溢出则通过 max_energy_range_uj 处理。此外，性能分析器的设计目标是在不使用 sudo 的情况下运行，因此会预先配置文件权限，使非特权用户能够读取能量计数器，避免产生额外开销。

4.3 工作负载场景

设计的基准测试套件包含原生扩展工作负载（*NumPy*）、单线程纯 $Python$ 内核以及多线程程序。所有场景均采用相同的结构。输入使用固定种子确定性生成，初始化与测量区域之间间隔 3 秒。使用 set_tag 辅助函数，将设置和数据生成阶段排除在分析之外。示例如下。

```
# 预先–构 建 输 入，再 进 行 性 能 分 析数字
 =生成_数字（num_项）时间。sleep（3）

s et_ta g ( " s t a r t _ b u b b l e _ s o r t "
) 冒泡 _ 排序 ( 数字 ) s et_ta g ( " f i n i s
h _ b u b b l e _ s o r t " )
```

[1] 休眠验证测试的结果可通过第 5 节提供的链接，在 results/preprocessed/sleep_*.csv 中查看。

NumPy 场景这些工作负载代表了数据分析模式，其中$Python$ 负责协调经过优化的原生代码。启用全局解释器锁和禁用全局解释器锁之间的差异预计较小，反映了自由线程开销或与原生库执行的交互。分析的测试用例如下：

– numpy_vectorized：对向量执行算术运算。– numpy_blas：对方阵执行矩阵乘法 A.dot(B)。– numpy_fft：对信号执行快速傅里叶变换，随后进行幅度归约。

顺序场景（单线程 Python 内核）这些 CPU 密集型程序在单个线程中运行，旨在测量无 GIL 开销基线。

– mandelbrot：在预先计算的网格上执行嵌套循环曼德博集合计算。– bubble_sort：对确定性的降序列表执行原地冒泡排序。– prime_sieve：使用埃拉托斯特尼筛法处理不同限制值以内的数。

线程化数值场景这些场景使用 ThreadPoolExecutor，并改变工作线程数量。

– factorial：计算输入范围为 0..10000 的阶乘。– matmul：基于行块执行纯 $Python$ 矩阵乘法，矩阵大小固定为 768×768。– nbody：N 体模拟，包含 num_个粒子=2000 和 num_个步长=10。

多线程对象场景 这些工作负载会对内存分配、引用计数和容器操作造成压力，其中包括共享可变状态。之所以值得关注，是因为对象密集型工作负载的扩展方式可能与数值计算内核不同。

– json_parse：使用 json.loads 解析并处理 2,000,000 个负载。– object_lists_nocopy（共享修改）：对共享列表中的 55,000,000 条记录就地将值转为大写。– object_lists_copy（减少共享修改）：使用每个线程的*C*语言级切片复制和列表推导式，将 55,000,000 条记录中的值转为大写。– object_lists：对 8,000,000 条记录执行基于数据类的对象／列表转换。

4.4 重复次数与执行计数

每个参数点（例如矩阵大小或工作线程数）在每个**Python** 构建版本下执行**十次**，从而为每个参数点生成 $n = 10$ 组无 GIL／全局解释器锁对比。后处理中会将每次无 GIL 运行与对应的全局解释器锁运行配对。

连续执行之间增加了 60 秒的冷却时间，以减少残留效应。

4.5 结果的解释执行

本节介绍了用于比较和总结性能分析结果的数学分析方法。

每次运行的效应量 对于每次运行，使用在匹配运行上计算的比率，对比启用全局解释器锁和禁用全局解释器锁两种配置的指标。对于某项指标 X （例如执行时间、CPU 使用率或能耗），每次运行的效应量定义如下：

$$R = \frac{X_{\text{noGIL}}}{X_{\text{GIL}}} \tag{3}$$

跨运行聚合由于比率表示乘法效应，因此使用对数变换和几何平均数进行聚合。对于运行 i，对数比率定义如下：

$$L_i = \ln(R_i) \tag{4}$$

n 次运行的平均对数比率为：

$$\bar{L} = \frac{1}{n}\sum_{i=1}^{n} L_i \tag{5}$$

聚合效应量通过对平均对数比率取指数得到：

$$R_{\text{geo}} = \exp(\bar{L}) \tag{6}$$

置信区间 每次运行之间的变异性以对数空间量化。使用s_L 作为对数比值的样本标准差，均值标准误的计算方式如下：

$$SE = \frac{s_L}{\sqrt{n}} \tag{7}$$

使用具有 $n-1$ 个自由度的 Student's t 分布，构造均值对数比值的双侧 95% 置信区间：

$$\bar{L} \pm t_{0.975,\,n-1} \cdot SE \tag{8}$$

对区间上下限取指数，即可得到比值空间中的置信区间。

解释执行置信区间完全低于（高于）1，表示无 GIL 比全局解释器锁更快（更慢）或使用的资源更少（更多）。当置信区间包含 1 时，在 95% 的置信水平下，观察到的差异在统计上无法区分。

5 结果

本节通过表格总结刚才所述的分析。为简洁起见，本文仅展示部分案例；可以放心地认为，省略的案例也遵循与所报告结果相同的趋势。完整结果可在线查阅：https://drive.google.com/drive/folders/1PDxkxgaz8Fz0YLwIXghI0AnufRDyTDs5。选择这些工作负载是为了确保无需交换，从而避免引入磁盘访问这一额外变量。

表 1：NumPy 向量化。

	Time		CPU		能量		VMS		RAM	
size	R	CI	R	CI	R	CI	R	CI	R	CI
150M	1.013	1.001–1.024	0.994	0.978–1.010	1.028	1.017–1.040	1.135	1.135–1.135	1.001	1.001–1.001
170M	0.997	0.988–1.007	0.999	0.991–1.008	1.008	0.994–1.021	1.121	1.121–1.121	1.001	1.001–1.001
190M	1.004	0.995–1.013	0.993	0.984–1.003	1.006	0.992–1.020	1.109	1.109–1.109	1.001	1.001–1.001
210M	1.005	0.997–1.013	1.003	0.998–1.008	1.006	0.993–1.018	1.099	1.099–1.099	1.001	1.001–1.001
230M	0.999	0.988–1.011	1.004	0.998–1.010	0.992	0.973–1.010	1.091	1.091–1.091	1.001	1.001–1.001
250M	0.997	0.989–1.006	1.008	1.000–1.017	0.997	0.987–1.008	1.084	1.084–1.084	1.001	1.001–1.001
275M	1.006	0.993–1.019	1.005	1.000–1.009	1.004	0.989–1.020	1.077	1.077–1.077	1.001	1.001–1.001

表 2：顺序冒泡排序。

	Time		CPU		能量		VMS		RAM	
size	R	CI	R	CI	R	CI	R	CI	R	CI
5k	1.334	1.316–1.351	1.010	1.007–1.014	1.346	1.324–1.368	40.332	40.332–40.332	1.232	1.231–1.233
8k	1.329	1.313–1.345	1.005	1.003–1.007	1.324	1.308–1.341	40.332	40.332–40.332	1.235	1.233–1.236
11k	1.330	1.316–1.345	1.002	1.001–1.002	1.329	1.310–1.349	40.332	40.332–40.332	1.238	1.236–1.239
14k	1.342	1.322–1.363	1.001	1.001–1.002	1.344	1.323–1.365	40.332	40.332–40.332	1.239	1.238–1.241
17k	1.351	1.328–1.375	1.001	1.000–1.001	1.353	1.328–1.379	40.332	40.332–40.332	1.240	1.239–1.242
21k	1.335	1.322–1.349	1.001	1.000–1.001	1.335	1.319–1.351	40.179	39.835–40.526	1.242	1.241–1.244
25k	1.352	1.330–1.374	1.000	1.000–1.001	1.354	1.334–1.375	38.827	38.827–38.827	1.242	1.241–1.243

表3：顺序素数筛。

	Time		CPU		能量		VMS		RAM	
限制	R	CI	R	CI	R	CI	R	CI	R	CI
16M	1.167	1.138–1.198	1.004	1.000–1.008	1.170	1.139–1.202	5.558	5.537–5.578	1.186	1.181–1.192
20M	1.173	1.139–1.209	1.005	1.000–1.009	1.174	1.141–1.208	4.576	4.570–4.582	1.123	1.111–1.136
24M	1.134	1.118–1.151	1.003	1.000–1.007	1.139	1.123–1.155	3.885	3.876–3.895	1.128	1.109–1.147
28M	1.145	1.117–1.174	1.003	0.999–1.006	1.144	1.118–1.171	3.380	3.377–3.383	1.151	1.149–1.153
32M	1.136	1.111–1.162	1.003	1.000–1.005	1.140	1.117–1.164	2.997	2.995–2.999	1.146	1.144–1.148
36M	1.132	1.122–1.143	1.006	1.003–1.008	1.138	1.125–1.150	2.692	2.689–2.694	1.172	1.171–1.173
40M	1.146	1.129–1.164	1.004	1.001–1.006	1.151	1.132–1.171	2.439	2.437–2.440	1.146	1.137–1.154

表4：多线程数值阶乘。

	Time		CPU		能量		VMS		RAM	
w	*R*	CI	*R*	CI	*R*	CI	*R*	CI	*R*	CI
1	1.212	1.209–1.216	1.050	1.048–1.052	1.215	1.212–1.218	6.053	6.053–6.053	1.058	1.057–1.058
2	0.610	0.600–0.619	2.062	2.058–2.067	0.614	0.605–0.624	6.178	6.178–6.178	1.018	1.016–1.020
4	0.307	0.303–0.310	4.050	4.032–4.069	0.307	0.303–0.311	3.958	3.958–3.959	0.953	0.947–0.958
6	0.246	0.244–0.248	6.001	5.940–6.064	0.250	0.247–0.253	3.073	3.070–3.076	0.907	0.884–0.932
8	0.247	0.245–0.250	7.913	7.855–7.970	0.251	0.247–0.255	2.596	2.595–2.596	0.885	0.842–0.930
12	0.249	0.247–0.250	11.309	11.163–11.457	0.254	0.253–0.255	2.091	2.091–2.092	0.899	0.841–0.962

表 5：线程化数值运算 N 体问题。

	Time		CPU		能量		VMS		RAM	
w	*R*	CI	*R*	CI	*R*	CI	*R*	CI	*R*	CI
1	1.174	1.167–1.180	1.001	1.000–1.002	1.174	1.168–1.181	11.131	11.131–11.131	1.293	1.291–1.294
2	0.615	0.608–0.622	1.983	1.979–1.988	0.623	0.616–0.631	6.889	6.889–6.889	1.305	1.303–1.308
4	0.316	0.313–0.319	3.921	3.909–3.933	0.319	0.316–0.322	4.205	4.205–4.205	1.339	1.335–1.344
6	0.231	0.229–0.233	5.783	5.760–5.805	0.232	0.230–0.235	3.257	3.134–3.385	1.347	1.343–1.351
8	0.251	0.248–0.254	6.580	6.564–6.597	0.253	0.250–0.256	2.781	2.661–2.905	1.372	1.369–1.374
12	0.223	0.222–0.224	11.410	11.371–11.448	0.226	0.224–0.227	2.286	2.198–2.377	1.428	1.425–1.430

表 6：线程对象 JSON 解析。

	Time		CPU		能量		VMS		RAM	
w	*R*	CI	*R*	CI	*R*	CI	*R*	CI	*R*	CI
1	1.183	1.171–1.194	1.000	0.998–1.001	1.186	1.175–1.197	1.873	1.871–1.875	1.077	1.076–1.077
2	0.607	0.599–0.615	1.983	1.977–1.988	0.615	0.607–0.624	1.778	1.776–1.780	1.077	1.077–1.077
4	0.314	0.312–0.316	3.898	3.885–3.911	0.317	0.315–0.318	1.640	1.638–1.641	1.078	1.078–1.078
6	0.274	0.271–0.277	5.438	5.409–5.468	0.277	0.274–0.281	1.543	1.542–1.544	1.079	1.079–1.079
8	0.268	0.261–0.274	6.634	6.605–6.663	0.271	0.264–0.278	1.472	1.471–1.473	1.080	1.080–1.080
12	0.275	0.272–0.279	10.912	10.839–10.986	0.279	0.274–0.284	1.374	1.373–1.375	1.083	1.083–1.083

表 7：线程化对象 对象列表 复制。

	Time		CPU		能量		VMS		RAM	
w	*R*	CI	*R*	CI	*R*	CI	*R*	CI	*R*	CI
1	0.875	0.863–0.888	1.714	1.706–1.721	0.879	0.866–0.891	1.197	1.197–1.197	1.147	1.147–1.148
2	0.521	0.518–0.524	2.901	2.885–2.917	0.526	0.519–0.532	1.216	1.178–1.254	1.151	1.148–1.155
4	0.355	0.351–0.359	4.294	4.255–4.334	0.360	0.355–0.366	1.383	1.277–1.498	1.169	1.164–1.175
6	0.320	0.315–0.325	5.044	4.958–5.132	0.322	0.317–0.326	1.567	1.418–1.731	1.168	1.161–1.175
8	0.301	0.297–0.305	5.938	5.852–6.025	0.300	0.295–0.305	1.452	1.306–1.615	1.163	1.160–1.167
12	0.292	0.288–0.296	7.104	7.018–7.190	0.296	0.290–0.302	1.652	1.412–1.932	1.169	1.165–1.173

表 8：线程化对象对象列表，无复制。

	Time		CPU		能量		VMS		RAM	
w	R	CI	R	CI	R	CI	R	CI	R	CI
1	5.222	5.142–5.303	0.476	0.473–0.478	5.269	5.198–5.340	2.256	2.255–2.258	2.308	2.307–2.309
2	4.845	4.618–5.084	0.906	0.886–0.926	4.958	4.715–5.213	2.584	2.574–2.595	2.296	2.286–2.307
4	4.519	4.239–4.819	1.761	1.727–1.796	4.622	4.341–4.921	2.574	2.496–2.654	2.278	2.266–2.290
6	4.817	4.610–5.032	2.624	2.584–2.665	4.886	4.682–5.098	2.518	2.497–2.540	2.282	2.267–2.297
8	7.889	7.574–8.216	3.592	3.522–3.663	7.967	7.640–8.308	2.669	2.485–2.868	2.254	2.234–2.273
12	12.184	11.876–12.501	5.003	4.952–5.055	12.308	11.970–12.657	2.474	2.442–2.506	2.308	2.308–2.309

表 9：线程化对象 对象列表。

	Time		CPU		能量		VMS		RAM	
w	R	CI	R	CI	R	CI	R	CI	R	CI
1	1.277	1.269–1.285	1.207	1.200–1.213	1.281	1.271–1.292	1.284	1.283–1.285	1.150	1.150–1.150
2	0.810	0.804–0.817	2.114	2.101–2.127	0.812	0.803–0.821	1.364	1.291–1.441	1.174	1.167–1.180
4	0.565	0.556–0.575	3.457	3.432–3.482	0.568	0.559–0.578	1.506	1.363–1.664	1.300	1.259–1.343
6	0.545	0.535–0.555	4.062	3.986–4.139	0.550	0.541–0.561	1.593	1.495–1.698	1.368	1.335–1.401
8	0.514	0.505–0.523	5.090	5.023–5.158	0.517	0.507–0.527	1.631	1.538–1.730	1.364	1.321–1.409
12	0.563	0.553–0.573	6.302	6.205–6.399	0.569	0.557–0.581	1.795	1.653–1.949	1.627	1.578–1.678

5.1 讨论

本节分析各组场景的结果，以回答研究问题。

5.2 NumPy 场景

对于 NumPy 场景，三项基准测试的表现均一致。禁用全局解释器锁并未带来有意义的加速。在 21 个参数点中，所有执行时间比均落在 0.997–1.015 之间，且 21 个置信区间中有 20 个包含中性值 1.0。剩余情况仅显示出可忽略的减速，幅度为 1.3%，对应 ∼86ms 的差异。这一结果符合预期，也体现了 Kempen 等人 [6] 强调的方法：将 $Python$ 作为高级协调器，同时把密集型工作交由原生库处理，从而抵消$Python$的运行时开销。

CPU 利用率比同样处于中性水平，证实两个构建版本对这一资源的使用情况完全相同。此外，预处理后的数据表明，两种实现都使用了中央处理器的所有核心，这说明 NumPy 在默认构建版本中成功释放了全局解释器锁。

RQ1 和 RQ2： 能耗几乎完全反映了执行时间。所有能耗比均在 0.992–1.028 之间，置信区间通常

包含 1.0。由于执行时间和 CPU 利用率均未发生变化，能耗差异可以忽略不计。

RQ3： 唯一显著的硬件差异在于虚拟内存大小（VMS）。无 GIL 构建的 VMS 比率为 1.070–1.714（即高出 7.0%–71.4%）。按绝对值计算，这相当于额外约 1GB 的虚拟空间。不过，常驻集大小（RSS）比率仍接近 1.0，表明在这些场景中，额外的虚拟内存并未由物理内存支撑。这反映了 $mimalloc$所预期的内存行为，此处可将其视为基线，因为 NumPy 使用标准的 *C*语言 内存分配机制。

总之，对于由会释放 GIL 的原生扩展主导的工作负载，移除解释器锁对执行时间、CPU 利用率或能耗没有实际影响。自由线程构建确实会带来相当大的虚拟内存开销，但除此之外，其表现与启用 GIL 的解释器相当。

5.3 顺序场景

对于单线程工作负载，移除全局解释器锁会一贯地降低性能。冒泡排序在所有输入大小下的执行时间比为 1.329–1.352，这表明无 GIL 版本比启用全局解释器锁的解释器慢 33–35%。曼德博集合也呈现出相同趋势，减速幅度为 40–43 %。素数筛的开销较小，但仍然显著，执行时间比为 1.132–1.173（慢 13–17%）。所有置信区间均完全高于 1.0，证实这些减速在统计上显著。这种开销在预期之中，不过高于 $Python$ 开发者 HOWTO [16], 报告的减速幅度；该文档显示平均性能损失为 8%。

CPU 利用率比基本持平，证实两个版本对单个核心的使用情况相同。这是预期的，因为没有多线程时，工作负载仍受限于一个 CPU 核心。

回答 RQ1 和 RQ2：能耗与执行时间高度一致。由于 CPU 利用率没有变化，两个版本的功耗保持不变，因此能量差异完全由执行时间决定。这进一步为 Kempen 等人 [6] 的发现提供了实证支持：在功耗表现保持恒定时，能量与执行时间成正比。因此，对于顺序工作负载，无 GIL 会使能耗的增加幅度与执行时间的增加幅度相同。

回答 RQ3：内存开销相当显著。无 GIL 构建中，冒泡排序的虚拟内存比率约高出 $40\times$，而素数筛的比率为 2.4–$5.5\times$。这种差异主要由工作负载规模所致：随着素数筛自身的内存占用增加，其规模逐渐与无 GIL 运行时引入的基础开销相当。

更重要的是，与 NumPy 场景不同，RAM 使用量也有所增加。冒泡排序的 RSS 比率为 1.23–$1.24\times$，而素数筛则为 1.12–$1.18\times$。这表明，当执行主要由 $Python$ 解释器支配时，自由线程分配机制会消耗实际的物理内存。

总而言之，对于顺序执行的纯$Python$ 工作负载，移除全局解释器锁并无益处，反而会带来可衡量的代价：执行时间更长、能耗按比例增加，以及内存使用量上升。这些结果确立了基准开销；无 GIL 必须通过并行性弥补这一开销，才能体现优势。

5.4 多线程数值计算场景

使用单个工作线程时，无 GIL 构建会产生预期的顺序执行开销：时间比为 1.17–1.21，与顺序执行场景的结果一致。然而，随着工作线程数量增加，无 GIL 构建实现了显著加速。当工作线程数达到 6 个时，三个基准测试的时间比均降至 0.23–0.25，执行速度提升 $\sim 4\times$ 。矩阵乘法呈现出与 N 体问题相同的趋势。

CPU 比率显示，无 GIL 构建对这一资源的使用量最多增加 $\sim 11\times$ ，这表明线程可在多个核心上并发执行。相比之下，启用 GIL 的构建在增加线程后并无改善，说明执行仍受限于单个核心。

工作线程数超过 6 个后，收益递减是意料之中的。测试机器有 6 个物理核心（启用超线程后有 12 个逻辑核心），因此将工作线程数增加到 6 个以上只能带来有限的提升。这说明了潜在并行化能力与可用核心实际数量之间的权衡。

回答 RQ1 和 RQ2：能耗与执行时间高度吻合。在 6 个工作线程下，能耗比为 0.23–0.26，与时间比几乎完全相同。这意味着能耗减少了 $\sim 4\times$。尽管启用了更多 CPU 核心（这会增加功耗），但执行时间的缩短对能耗的影响更大。这印证了 Kempen 等人的发现 [6]：激进的并行化几乎总能提高能效。在他们的平台上，CPU 核心数量翻倍仅增加了 ~ 31W 的功耗，却仍带来了更显著的时间缩短和节能效果。

回答 RQ3：在无 GIL 模式下，CPU 利用率会随着工作线程数量增加而提升，在 12 个工作线程时达到 $\sim 11\times$ 。这证实，自由线程构建能够让 $Python$ 字节码在多个核心上实现真正的并行执行。在这些场景中也观察到了虚拟内存大小开销（最高增加 $11\times$ ），但随着线程数增加，其相对影响会减小。这表明，GIL 构建在这方面的扩展性较差，因为随着线程数增加，其内存使用量往往会逐渐接近无 GIL 构建。常驻集大小开销仍然不大；在 12 个线程运行 N 体问题场景时，观察到的最大绝对增量为 8 MB。有趣的是，阶乘场景在 4 个线程之后，无 GIL 模式下的常驻集大小略低。其原因可能是工作负载特有的行为：此基准测试共享一个只读输入列表，并在 math.factorial 中执行算术运算，因此每个线程的 $Python$ 容器内存分配少于 matmul 和 nbody。此外，mimalloc 在小型内存分配方面表现良好 [8, 19]。因此，应将常驻集大小的这一下降视为此特定场景的结果，而非无 GIL 模式普遍具有内存优势的证据。无论如何，移除全局解释器锁后的虚拟内存大小开销十分显著。

总之，多线程数值工作负载是自由线程构建的理想用例。当任务可以在线程间划分，且共享状态较少时，无 GIL 模式能够带来明显加速，最高可扩展至物理核心数，并实现相应的节能。

5.5 线程化对象场景

这些结果表明，行为在很大程度上取决于线程是否共享可变状态。

竞争极少的工作负载表现良好。json_parse 和 object_lists_copy 基准测试的扩展表现与数值场景相当。在 6–8 个工作线程时，json_parse 的时间比达到 ~ 0.27，object_lists_copy 则达到 0.30–0.32。这些工作负载有一个共同特点：线程处理的数据基本相互独立。在 json_parse 中，线程从共享列表中读取不同片段的 JSON 负载，但将结果累积到线程局部结构中。在 object_lists_copy 中，每个线程先将分配给自己的片段复制到局部列表，再进行转换。由于线程不会修改共享对象，对象级锁很少发生竞争，因此并行处理能够高效进行。[2]

object_lists 基准测试显示出中等程度的提升。在 8 个工作线程时，时间比为 0.51，低于先前工作负载实现的 ~ 3–$4\times$ 加速。该基准测试会创建类实例，并在线程各自处理的片段中执行列表拼接，因此带来了更大的内存分配压力和引用计数开销。不过，由于每个线程都处理自己的数据片段，并构建独立的输出结构，竞争仍在可控范围内。

共享可变状态会导致严重的性能下降。object_lists_nocopy 基准测试呈现出鲜明对比。在单个工作线程下，无 GIL 慢了 $5.2\times$。随着工作线程增加，性能变得更差：在 12 个工作线程时，无 GIL 慢了$12.18\times$ 。这与在所有其他多线程场景中观察到的扩展性相反。

该工作负载反复创建新字符串并将其写入共享列表。在 PEP 703 自由线程下，这些操作会产生额外的运行时开销（对象管理和同步），并且在工作线程数量较高时，对同一容器的并发修改会增加锁竞争 [3]。这些影响共同掩盖了任何并行收益。

CPU 利用率数据支持这一解释。对于 object_lists_nocopy，在 12 个工作线程时的中央处理器比率仅为 5.0，低于其他基准测试在相同工作线程数下达到的 6–$11\times$ 。在较低线程数（一个和两个）时，object_lists_nocopy 的表现甚至不如启用全局解释器锁的 $Python$。巨大的时间差（高达 20 秒）以及较低的中央处理器使用率表明，执行时间主要由内存管理开销主导，而非中央处理器占用。

[2]对于 object_lists_copy 中不存在单线程性能下降的一个合理解释是，它主要由针对线程局部数据的 *C*语言 级批量操作（切片复制和列表推导式）主导，因此锁竞争可以忽略不计。这种加速是特定场景下的，而非无 GIL 的普遍优势。

回答 RQ1 与 RQ2：能耗会因共享模式而显著分化。对于 json_parse 和 object_lists_copy，在 6–8 个工作线程时的能耗比为 0.27–0.32，相当于 $\sim 3\times$ 节能，与本研究中观察到的时间-能耗比例一致。然而，对于 object_lists_nocopy，无 GIL 在相同计算上消耗的能量是全局解释器锁的 12 倍以上。这是自由线程构建的最坏情况：不仅并行性未能实现，锁争用的开销还同时增加了执行时间和能耗。

回答 RQ3：内存行为因工作负载而异。无 GIL 情况下的 VMS 一直较高，其中观测到的最大 VMS 比率为 $2.67\times$，出现在使用 8 个线程的 object_lists_nocopy 基准测试中，对应的最大绝对差异为 12。71 GB。该组中的 RSS 开销也最高，达到 $2.3\times$，绝对差异最高可达 7 GB。总体而言，这些结果证实，无 GIL 通常会更大量地使用内存，而且无 GIL 情况下的 CPU 使用率通常也更高，前文讨论的 object_lists_nocopy 例外情况除外。

总之，当线程处理独立数据时，对象密集型工作负载在无 GIL 环境下表现良好。然而，并发修改共享容器会因每个对象的锁竞争而导致性能和能耗严重恶化。这体现了自由线程式 $Python$ 中一项重要的权衡：移除全局锁能够实现并行性，但也会使应用程序面临此前被 GIL 隐藏的竞争。采用自由线程式 $Python$ 的开发者必须合理组织代码，以尽量减少共享可变状态。

5.6 研究结果总结

本节综合所有场景下的结果，为各项研究问题提供综合答案。

RQ1：不使用全局解释器锁的$Python$ 与使用全局解释器锁的$Python$ 相比，能否降低能耗?

答案是：有条件地成立。能耗降低完全取决于工作负载能否在不承受竞争开销的情况下利用并行性。表 10 总结了各场景类别中观测到的能耗比。

表10：各场景类别在最佳工作线程数量下的能耗比总结（$6+$）。

场景类别	能耗比解释	
原生扩展	~ 1.0	没有差异
单线程	1.13–1.43	13–43% 更多
线程化数值运算	0.23–0.26	74–77% 更少
线程化对象（低竞争）	0.27–0.55	45–73% 更少
线程化对象（高竞争）	4.8	380 %更多

RQ2：当全局解释器锁与无 GIL 之间的执行时间存在差异时，这些差异是否会转化为能耗的成比例变化?

是的。在本研究的全部 84 个参数点中，能量与时间高度吻合。所有参数点的时间比与能耗比之间的平均绝对差异小于 1%。对于所展示的硬件和场景，可以肯定地说，GIL 与无 GIL $Python$ 之间的能耗差异主要由执行时间决定，而非功耗变化。

值得注意的是，即使不同构建版本的 CPU 利用率相差很大，这种比例关系依然成立。在涉及线程化数值运算的场景中，无 GIL 版本的 CPU 利用率最高可增加 $11\times$（程序执行期间的平均占用率最高达到 95%），从而增加功耗。然而，执行时间的相应缩短弥补了这一点，最终实现节能。这与 Kempen 等人 [6],的研究结果一致：功耗随活跃核心数量的增长呈次线性关系，这意味着只要能带来显著加速，并行化几乎总能提高能效。

实际影响十分显著：**对于 $Python$ 3.14 应用而言，优化执行时间实际上等同于优化能耗**。开发者不应将能耗视为独立的优化目标：无论是通过改进算法、并行化还是选择运行时来缩短执行时间，都能按比例节能。

RQ3：全局解释器锁与无 GIL 在硬件资源利用方面有何不同?

– **CPU 利用率：** 在无 GIL 模式下，CPU 使用率会随线程数增加，最高可提升 $11\times$。– **虚拟内存大小（VMS）：** 无 GIL 构建始终会占用更多虚拟地址空间（根据工作负载，为 1.1–$40\times$），这反映了对象级锁结构、线程局部存储以及 $mimalloc$ 行为带来的影响。对于内存密集型工作负载，这项开销所占比例较小，证实了无 GIL$Python$存在基线内存使用量。– **常驻集大小（RSS）：** 对于大多数工作负载，物理内存开销较小（1.0–$1.6\times$），但在竞争密集型场景中会显著增加（$2.3\times$）。

5.7 自由线程的代价

尽管结果表明，自由线程能够为并行工作负载显著节能，但坦诚地审视后会发现，对于通用场景的采用而言，前景并不乐观。

顺序执行开销不可避免，而且无论是否使用线程，在自由线程构建下运行的所有代码都会受到影响。现实应用从来都不是 100% 并行的；如果顺序代码会消耗更多能量，那么总运行时间中就必须有相当大一部分用于处理并行部分（使用独立数据），才能实现净节能。

内存开销加剧了这一顾虑。虚拟内存大小和常驻集大小的增量会贯穿整个执行过程，而不仅仅出现在并行阶段。对于嵌入式系统，或限制严格的容器化微服务等内存受限环境，即使能耗权衡有利，这种开销也可能仍然过高。

这些观察结果并不会削弱自由线程对能够利用它的工作负载的价值。在并行数值场景中观察到的 $4\times$ 能耗降低是显著且有意义的。然而，对于更广泛的 *Python* 生态系统而言，大多数应用程序混合了顺序阶段与并行阶段，并且许多依赖原生扩展，自由线程构建不太可能成为一种普遍的改进。

6 对有效性的威胁

– **不断演进的运行时实现**。 实验使用了*Python 3.14.2*。尽管这代表一个正式发布版，但该自由线程构建是第一个未被标记为实验性的版本，其性能可能会在未来版本中得到优化。

– **工作负载规模的选择**。 一个潜在的担忧是，实验中使用的最小工作负载值可能不足以揭示执行时间更短却因瞬时功率增加而导致能耗更高的情况。尽管这一担忧是合理的，但执行时间大幅缩短会使CPU活动测量因采样分辨率有限而变得困难。此外，对于极短时运行程序，即使比率很大，也只会转化为绝对值差异极小的量级，从而限制其实际意义。

– **单机测量**。所有实验均在一台笔记本电脑上进行，结果可能无法推广到其他平台。

– **未与多进程进行比较**。本研究有意排除基于多进程的并行性，因为线程与进程代表不同的编程模型，具有不同的开销特征。本研究的目标是评估自由线程是否兑现了其实现真正共享内存并行的承诺，而非将多线程与多进程进行比较。

7 结论

本研究评估了*Python*的自由线程（无 GIL）构建相较于传统启用 GIL 的构建，在四类工作负载上的能耗、执行时间和硬件利用率：基于 NumPy 的计算、顺序纯*Python* 内核、线程化数值运算场景以及线程化对象场景。

结果揭示了一个明显的权衡。对于能够在独立数据上实现有效并行化的 CPU 密集型工作负载，移除全局解释器锁的构建版本可带来

显著收益：执行时间最多缩短 $4\times$，能量消耗也按比例降低；在线程数较高且处理线程局部数据时，结果最佳。这些收益证实，移除全局解释器锁能够让 $Python$ 线程有效利用多核硬件。

然而，这些优势也有代价。在自由线程构建下，顺序工作负载的能耗会增加 13–43%。对于包含共享可变状态的工作负载，即使采用多线程执行，也可能因锁竞争而导致性能下降。此外，在所有场景中，内存使用量都会增加，只是增加比例和绝对值可能相差很大。这似乎是为了在不使用全局解释器锁的情况下安全运行而必须采用相关机制所致。

在实际采用时，开发者应考虑工作负载的并行比例以及可用内存。对于运行时大部分用于可并行化、使用独立数据进行计算的应用程序，自由线程构建将带来益处，因为它能缩短执行时间。以顺序执行为主或严重依赖共享可变容器的应用程序，则应继续使用启用 GIL 的构建。而以 NumPy 为主的工作负载无论采用哪种构建，差异都微乎其微，因为计算是在$Python$解释器之外进行的。总的来说，对于$Python$ 应用程序，优化执行时间就等同于优化能耗。

总而言之，$Python$ 的自由线程构建并非适用于所有场景的改进。它是一种专用工具，能为部分工作负载带来显著的能量和性能提升，但也会给其他工作负载增加开销。对于希望在多核环境中构建能效更高的 $Python$ 应用的开发者而言，理解这种权衡至关重要。

参考文献

1. Ashwathnarayana, S.: 交换内存——何时以及如何在生产系统或云提供的虚拟机上使用它。Netdata 博客（2023年5月），https: //www.netdata.cloud/blog/swap-memory- when- and- how- to- use- it- on- your-production-systems-or-cloud-provided-vms/，访问日期：2026年3月4日 2. Dargie, W.: 用于估算处理器功耗的随机模型。《IEEE计算机汇刊》64(5)，1316–1329（2015年）。https: //doi. .2014.2315629 3 10.1109 703 org/ /TC . Gross, S.: PEP——在 CPython 中将全局解释器锁设为可选项。Python 增强提案（2023年），https://peps.python.org/pep-0703/，访问日期：2026年3月4日 4. Haywood, A.M., Sherbeck, J., Phelan, P., Varsamopoulos, G., Gupta, S.K.S.: 面向余热利用的 CPU 利用率、温度与热功率之间的关系。《能源转换与管理》95，297–303（2015年）。https: //doi. .2015.01.088 5 10.1016 . org/ /j enconman . Hennecke, F.: 无 GIL 的 CPython 对性能与兼容性的影响。研讨报告，哥廷根大学（2025年4月），https://hps.vi4io.org/ _media/teaching/autumn_term _2024/stud/scap/frederik_hennecke.pdf 6. van Kempen, N., Kwon, H.J., Nguyen, D.T., Berger, E.D.: 绿色并不容易：论编程语言的能效。arXiv 预印本 arXiv:2410.05460（2024年），https://arxiv.org/abs/2410.05460，访问日期：2026年3月4日

7. Konopik, M., Korten, T., Lutz, E., Linke, H.: 有限时间可并行计算的基本能量成本。《自然通讯》14，447（2023年）。https: //doi.org/10.1038/s41467 2 8 023 36020- - - . Leijen, D., Zorn, B., de Moura, L.: Mimalloc：空闲链表分片的实践。技术报告 MSR-TR-2019-18，微软研究院（2019年6月），https:// www.microsoft.com/en-us/research/wp-content/uploads/2019/06/mimalloc-tr-v1.pdf，访问日期：2026年2月16日 9. Marchand, B.: 使用 –disable-gil 的 Python 3.14b2 基准测试。DEV（2025年6月），https://dev.to/basilemarchand/benchmarks- of- python- 314b2- with- disable- gil- 1 4, 2026 10 3, ml 访问日期：3月 . Margot, V.: 性能比较：Python 3.13 与 Python 3.13t（无 GIL）。GitHub 仓库（2025年），https://github.com/VMargot/gil-vs-no-gil 11. 微软公司：mimalloc 文档，https://microsoft.github.io/ mimalloc/，访问日期：2026年2月16日 12. Montoya, J.: 系统分析器。GitHub 仓库（2024年），https://github.com/ Joseda8 4, 2026 13 / profiler/tree/v0.1.0, 访问日期：3月 . Ou, Z., Dong, S., Dong, J., Nurminen, J., Ylä-Jääski, A., Wang, R.: 刻画并发网络密集型应用对移动平台的能量影响。见：ACM MobiCom 会议论文集。第23–28页（2013年）。https: //doi. 2505906.2505909 14 10.1145org/ / . Pereira, R., Couto, M., Ribeiro, F., Rua, R., Cunha, J., Fernandes, J.P., Saraiva, J.: 跨编程语言的能效：能量、时间与内存如何关联？见：ACM SIGPLAN 软件语言工程国际会议（SLE）论文集。第256–267页（2017年）。https://doi.org/ 10.1145 3136014.3136031 15 / . Phillips, G.: CPU 缓存如何工作？什么是 L1、L2 和 L3 缓存？MakeUseOf（2023年1月），https://www.makeuseof.com/tag/what-is-cpu-cache/，访问日期：2026年3月4日 16. Python 软件基金会：Python 对自由线程的支持。Python 3 文档（2026年），https://docs.python.org/3/howto/free-threading-python.html，访问日期：2026年3月4日 17. Python 软件基金会与 JetBrains：Python开发者调查 2024年结果。2024年10月至11月（2024年10月），https://lp.jetbrains.com/python-developers-survey-2024 4, 2026 18 7 1 3/, 访问日期：3月 . Rodolà, G.: psutil . . . Python 包索引（PyPI）（2026年），https://pypi.org/ project/psutil/7.1.3/，访问日期：2026年3月4日 19. Rosner, F.: libmalloc、jemalloc、tcmalloc、mimalloc——探索不同的内存分配器。DEV 社区（2023年），https://dev.to/frosnerd/libmalloc-jemalloc-tcmalloc-mimalloc-exploring-different-memory-allocators-4lp3，访问日期：2026年2月16日 20. de Roucy-Rochegonde, L., Buffard, A.: 人工智能、数据中心与能源需求：重新评估并探索相关趋势。IFRI 论文（2025年2月），https://www.ifri.org/ sites/default/files/2025-02/ifri _ _ _ _ _ _ 2025 _ 2.

buffard-rochegonde ai data centers energy pdf，访问日期：2026年3月4日 21. TIOBE Software BV：2026年2月的 TIOBE 指数。TIOBE 编程社区指数（2026年2月），https://www.tiobe.com/tiobe-index/，访问日期：2026年3月4日 22. Vogeleer, K.D., Memmi, G., Jouvelot, P., Coelho, F.: 为应用处理器中的纳米级 CPU 建立功耗的温度偏差模型。见：嵌入式计算机系统国际会议论文集：

架构、建模与仿真（SAMOS XIV）。第172–180页（2014年）。https://doi..2014。6893209 2310。1109org//SAMOS。Yigit, D.：TIOBE指数评级。GitHub 仓库（2021年），https://github。com/toUpperCase78/tiobe-index-ratings/blob/master/Tiobe_Index February2026。csv，访问日期：2026年3月4日