arXiv:2304.02712v1 [cs.PL] 5 Apr 2023

借助Cppyy 与 Cling 实现高效且准确的自动 Python 绑定

Baidyanath Kundu[*†], **Vassil Vassilev**[*†], **Wim Lavrijsen**[§]
[*]欧洲核子研究组织，Espl. des Particules 1, 1211 Meyrin，瑞士
[†]美国新泽西州普林斯顿市普林斯顿大学物理系，邮编 08544[§]美国加利福尼亚州伯克利市
赛克洛特龙路 1 号，邮编 94720-8147，劳伦斯伯克利国家实验室，MS 50B-6222
电子邮件：baidyanath.kundu@cern.ch, vassil.vassilev@cern.ch, wlavrijsen@lbl.gov

摘要。 Python的简洁性和 C++ 的强大功能迫使科学软件栈面临艰难抉择。人们已经进行了多项开发工作，通过实现语言绑定来缓解语言之间的边界问题，但C++ 的静态特性与 Python 的动态特性之间的不匹配阻碍了这些绑定的实现；示例包括在模板化 C++ 中使用用户定义的 Python 类型，以及高级内存管理。

C++ 解释器Cling 的开发改变了我们对语言绑定的看法，因为它提供了一种可在运行时使用的增量编译基础设施。也就是说，Python 可以按需查询 C++ ，并可在运行时惰性地构建绑定。这种自动绑定机制无需库作者提供直接支持，其性能也优于PyBind11等替代方案。ROOT 通过 PyROOT开创了这种方法，后来又由其后继者 cppyy加以改进。然而，迄今为止，cppyy一直依赖 ROOT 的反射层，而该反射层在功能和性能方面均有限。

本文介绍了使用 cppyy 实现语言互操作性的下一步，从而能够研究统一的跨语言执行环境，并提升跨语言边界的优化机会。我们通过 cppyy 展示了在 Numba 加速的 Python 中使用高级 C++ 。我们概述了重新设计 cppyy 部分以使用上游 LLVM 组件来提高性能和可持续性的前进路径。我们演示了完全基于 C++反射库 $InterOp$ 的 cppyy，该库提供了基于 Cling 和 Clang-Repl 的互操作性原语。

1. Introduction

C++ 编程语言在20世纪90年代中期被高能物理（HEP）领域采用，以提供更加现代且高性能的编程环境。当时，HEP拥有世界上最大的数据集（以及最高的数据处理速率）之一。HEP对Python的使用始于21世纪初，最初仅用于交互式访问实验框架和进行分析。与此同时，世界其他地区赶上并甚至超越了HEP的数据处理需求，并围绕Python发展出了一个完整的生态系统，以支持数据科学。许多用于HEP物理分析的现代软件，例如AwkwardArray [1],Uproot [2], 和Coffea [3], , 都建立在这一Python生态系统之上。C++ 仍然受到青睐，因为它具有出色的性能，能够访问异构计算环境中的加速器，并且HEP拥有ROOT [4], GEANT4 [5], 等基础库，以及大多数实验所使用的处理和生产软件框架。由于两种语言都发挥着重要作用，Python-

C++ 集成已变得至关重要。然而，对于高性能代码以及在异构环境中运行的代码而言，还需要更高级的集成。例如，由运行用户提供的 Python函数或内核的C++ 框架执行数据处理任务时，由于大量的语言切换，会产生显著的开销。同样，如果该任务作为 GPU 工作流的一部分运行，由于需要卸载执行，也会遭受性能损失，因为 Python 代码只能在 CPU 上运行。

Numba [6], a 一款用于 Python 代码的即时编译器（JIT），解决了这一问题：它能够编译 Python 代码，目标平台可以是 CPU 或 GPU，并提供可调用接口，以便从底层库中使用经过即时编译的闭包。因此，Numba 不仅通过将 Python 代码编译为机器码来大幅提升其性能，还能消除内层循环中不必要的语言（或设备）边界切换。不过，Numba 有两个缺点：要让外部代码（例如用 C 编写的函数）能够用于 Numba 即时编译，需要经过繁琐的手动处理；而且 Numba 不直接支持 C++ 。

Cppyy，一种基于C++ 解释器 Cling [8],的自动运行时绑定生成器 [7]，可以解决这两个缺点。我们工作的目标之一，是以完全自动化的方式，将高级 C++（尤其是高度优化的数值库）引入由 Numba 加速的 Python，同时将语言跨越开销降至最低。在本文中，我们介绍了一个完全通用的原型，它通过 cppyy为绑定的C++ 在运行时按需提供 Numba 扩展。我们的原型支持在由 Numba加速的 Python中使用重载、C++模板、数据成员访问和实例方法调用。我们概述了重新设计 cppyy后端的未来路径，使其更直接地使用 LLVM 组件，从而提升性能和可持续性。我们在InterOp 上展示了 cppyy；InterOp是一个基于 Cling 和 Clang-Repl 实现互操作原语的新库。

本文的组织结构如下：第 2 节阐述我们采用这些技术和设计方案的动机；第 3 节概述实现设计；第 4 节展示当前实现的计时结果；第 5 节讨论新反射库 InterOp 带来的改进，该库将在重新面向其他代码时实现更高的性能和更大的灵活性；第 6 节讨论相关工作；最后，第 7 节总结我们的研究结果。

2. 动机

加速Python 应用的典型方法，是将其中对性能要求较高的部分用较低级的语言（通常是 C）编写，然后通过扩展模块在 Python解释器中访问这些代码。一种更加优雅的解决方案是使用即时编译将原始 Python 代码降低为本机代码，这样开发者就可以始终在单一环境 Python中工作，并且只需编写和调试代码一次。在HEP 中，这种即时编译器能够很好地处理绑定的 C++代码至关重要，同时即时编译的 Python 代码也必须能够从C++ 中使用，因为任何完整的 HEP 软件栈都严重依赖这两种语言。在本文中，我们使用 Numba作为 Python 即时编译器，并通过 Cppyy 将 C++ 集成其中。

Numba 提供扩展应用程序编程接口（API），用于添加即时编译器可识别、注释并降低为 LLVM IR （中间表示，随后会被汇编为本机代码）的函数和类型。手动编写此类扩展既繁琐又耗时，因此自动化至关重要，尤其是为了实现透明使用即时编译器这一目标。将 C++ 与 Numba 集成，还能让用户轻松地在 C++代码中使用 Python 内核，同时不损失性能，从而充分利用持续使用的大量现有 C++代码。为此，我们使用 cppyy 为绑定的 C++生成一流的 Python 对象，使 Numba 能够通过常规 Python 内省查询反射信息，并让 cppyy 在运行时按需提供必要的 Numba 扩展。最终实现具备通用、紧凑且高效的特点。

动机 n C++尤其适合自动化：后续的 C++ 标准已将 d

将语言从底层代码引向更具表达性的抽象，以提升代码质量并服务于优化编译器。在现代 C++ 中，许多以往难以分析的边界情况如今变得更加清晰；例如，可以比较原始指针与智能指针的所有权规则。此外，cppyy基于 Cling（C++ 解释器），从而使 C++ 能够匹配Python的动态性、交互性和运行时行为。Cling 本身基于Clang/LLVM，这使其能够保持最新状态，即使 C++ 语言仍在快速演进。使用 Cling 还意味着，尚未解决的边界情况可以在运行时根据需要由C++ 或 Python 处理，而无需引入中间桥接语言或映射语言。

3. 实现

我们通过 cppyy 之上的全新反射接口，将 C++ 引入 Numba。该接口会在 Numba 的类型注解阶段解析重载函数并实例化 C++ 模板；同时提供底层描述，以便将其降低为 LLVM IR，供后续优化并将汇编代码转换为机器码。整个过程完全自动且透明，并可与其他手动提供的 Numba 扩展集成。性能不相上下，仅增加了适度的预热开销。因此，Python 程序员可以继续完全使用 Python 开发和调试代码，只需为选定的性能关键闭包启用 Numba 即时编译即可。

Python 是动态类型语言。它通过一种称为装箱 的方法将对象封装为所谓的 PyObject对象，并通过一种称为拆箱的方法取回其原始值。由于 Numba 需要消除这些 Python装箱 和拆箱 步骤，因此必须预先获得准确的类型信息。为了向 Numba 提供详细、精确且底层的 C++ 类型信息——这些信息是 Python 自省功能已提供的类型信息之外所必需的——我们为 cppyy 引入了全新的反射 *API* 。

反射API遵循请求/回复模式，并为所有由 cppyy 绑定的对象添加一个名为 __cpp_reflex__ 的新函数。对于所有 cppyy 对象类型，该函数都具有固定的签名，但其功能取决于调用它的对象。它接受两个参数：反射信息的类型及其格式，并返回所请求的信息。请求内容可以是询问对象是否表示一个命名空间或类，也可以是请求
对象的 C++ 类型。format 参数用于指定我们希望以字符串格式、C++ 类型，还是以 API 认为最优的方式获取信息。

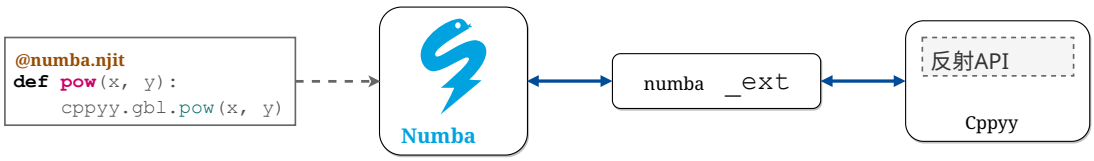


图1。Numba、Numba扩展和cppyy的交互示意图

图1展示了Numba为分离关注点所采用的分层方法。Numba按指示分析Python闭包，在遇到cppyy类型时，会向cppyy预注册的*Numba*扩展 模块查询精确的类型信息。该模块进而在遇到未见过的类型时，会查询cppyy的新反射API。然后，该模块使用Numba的扩展和核心模块生成必要的类型类，以便在此类型步骤中返回给Numba；以及降低方法，Numba将在下一步中期望这些方法来编写LLVM IR。每种核心语言构造（自由函数、命名空间、类、方法、数据成员等）都有其自己的实现。例如，为支持C++自由函数，我们扩展Numba的*Callable* 类型，并将其注册为cppyy函数对象的类型提供者。当Numba遇到由cppyy绑定的C++ 函数时，它会查询

此已注册类型以获取函数签名（即返回类型和参数类型），基于迄今追踪到的实际参数类型。Cppyy的*Numba*扩展 使用参数类型来选择适当的重载，借助新的反射API，并将此重载包装在唯一的类型类中。最后，*Numba*扩展 为所选C++函数重载的地址注册一个唯一生成的降低方法，该方法将在Numba降低步骤中被调用。此过程为Numba提供了将函数调用转换为LLVM IR所需的全部信息。

4. 结果

我们对使用 cppyy对象 的 Numba JIT函数 与其对应的 Python 函数的运行时间进行基准测试，以获取以下耗时：Numba 对函数进行即时编译所需的时间（Numba JIT时间）、cppyy 创建类型信息并可能执行查找和实例化模板所需的时间（cppyy JIT时间）、函数完成即时编译后的运行时间（热运行时间），以及运行等效Python 函数所需的时间。结果是在配备 32G RAM的 3.1 GHzIntel NUC Core i7-8809G CPU 上获得的。

C++ 代码在 cppyy 中

```
cppyy.cppdef(r"""
template<class T>
T add42(T t) {
    return T(t+42);
}
""")
```

使用 Cppyy 的 Python/C++绑定

```
def go_slow(a):
    trace = 0.0
    for i in range(a.shape[0]):
        trace +=
            cppyy.gbl.add42(a[i, i]) +
            cppyy.gbl.add42(int(a[i, i]))
    return a + trace
```

cppyy的Numba加速

```
@numba.jit(nopython=True)
def go_fast(a):
    trace = 0.0
    for i in range(a.shape[0]):
        trace +=
            cppyy.gbl.add42(a[i, i]) +
            cppyy.gbl.add42(int(a[i, i]))
    return a + trace
```

图2。“模板化自由函数”案例的基准测试夹具。左侧在 C++中声明模板化函数。在中间，我们使用Python内核运行此 C++ 函数。在右侧，我们使用相同的内核，但添加Numba JIT装饰器以加速该函数，并比较两个内核的计时结果。其他案例使用相同的设置，仅将高亮显示的代码替换为正在进行基准测试的案例代码。

为评估对 cppyy对象进行Numba即时编译所获得的加速比，我们使用图2 中的测试装置。对于表 1 中的每个基准测试用例，向函数传入了大小为 100×100 的 NumPy数组；所示时间为 3000 次运行的平均值。Numba 即时编译函数在方法的情况下至少可实现 2.3倍的加速比，在模板化自由函数的情况下最高可实现近21倍的加速比。表 1 汇总了在我们选定的基准测试中获得的计时结果。

基准测试案例	Cppyy 即时编译时间 (s)	Numba 即时编译时间（秒）	热运行时间（秒）	运行 Python 时间（秒）	加速比
无参数函数	1.72e-03	3.33e-01	3.58e-06	1.73e-05	4.84×
重载函数	1.05e-03	1.35e-01	4.51e-06	3.47e-05	7.70×
模板化自由函数	8.92e-04	1.45e-01	3.48e-06	7.18e-05	20.66×
类数据成员	1.43e-06	1.33e-01	5.87e-06	1.82e-05	3.10×
类方法	2.16e-03	1.39e-01	6.06e-06	1.43e-05	2.36×

表 1. 性能比较。 各列**cppyy JIT时间** 和 **Numba JIT时间**分别表示 cppyy 为 C++ 实体创建 Python 对象所需的时间，以及 Numba 编译 Python 函数所需的时间。两者都不会执行编译后的函数。列 **热运行时间** 显示运行 Numba 即时编译函数所需的时间，而列 **Python 运行时间**显示使用 cppyy 代码运行未编译 Python 函数所需的时间。列**加速比** 比较热运行时间 与 *Python* 运行时间。

5. 进一步改进

一个很有前景的研究方向，是将C++ 代码的 LLVM IR直接整合到Numba 生成的 LLVMIR 中，实际上就是将 C++函数内联到 Python 中。这样可以为编译器带来更多优化机会，并简化对其他目标平台（例如GPU）的支持。在实践中，这要求在Cling 与 Numba之间双向传递 IR。这里的挑战之一在于，Numba 使用 llvmlite [9] 包来合成 LLVM IR，而Cling则直接操作底层 IR。由于LLVM IR 不保证向后或向前兼容，这可能会形成相当脆弱的依赖关系。此外，Cling 当前的设计没有提供适当的抽象，也没有必要的底层基础设施，无法从交互式 C++ 环境中提取 IR 或将 IR 注入其中。

因此，我们创建了一个名为InterOp 的新型专用库，以缓解这些问题。该库充当C++ 与 Python 之间的高性能接口，抽象掉底层编译器API 的细节。InterOp 库构建于 Clang-Repl 之上，这是 LLVM中一项新功能，并提供以反射为核心的 API。InterOp 将遵循 LLVM 的发布周期，使其客户端能够快速采用新功能。

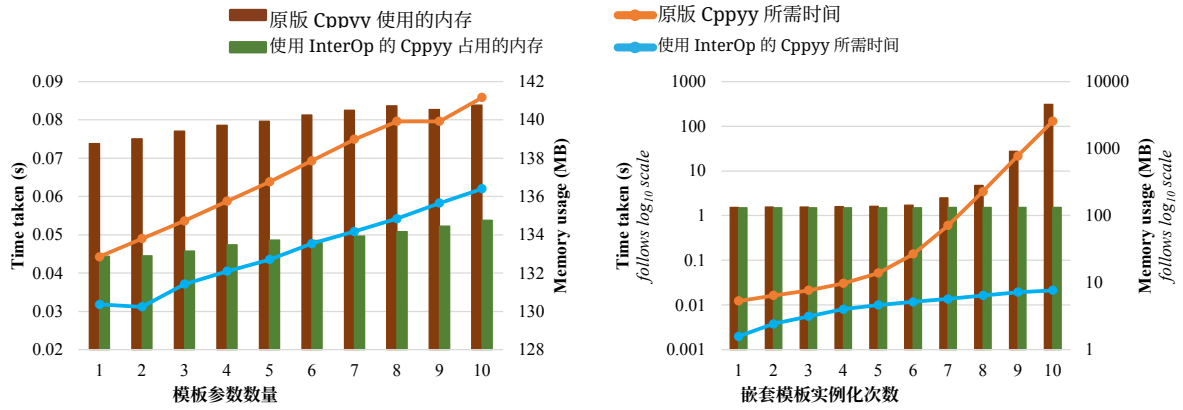


图 3. 类模板实例化期间所耗费的时间和使用的内存。左侧比较了使用 `std::tuple<double,double,...>`的模板实例化，其中C++ 解释器所执行的模板实例化次数会随着模板参数数量的增加而增加。右侧比较了嵌套模板的实例化，例如 `std::vector<...<std::vector<double> >`，其中 cppyy必须从最内层到最外层，分别实例化每一层嵌套。这些是利用模板表达式的高性能模板化数值库的常见特性。

InterOp 的设计显著改善了 cppyy 实例化模板所需的时间和内存。得益于新的建模方式，它展现出令人鼓舞的初步结果。在图 3（左）中，我们展示了解包模板方面的改进：我们递归实例化一个教科书中的多类型数组示例，其底层实现基于 `std::tuples`。InterOp 版本的速度约快 40%，内存效率提高 4.5%。在图 3（右）中，我们展示了实例化深层嵌套模板时更好的扩展性。最初 6.2 倍的加速比在嵌套约 4 层时降至 3.8 倍，但此后迅速提升。此外，我们还减少了总内存用量。

除了性能提升之外，在 Cppyy 中使用Clang-Repl 和 InterOp 的工程优势还包括：(a) 更快地采用新特性，包括潜在的 CUDA支持；(b) 由于直接使用底层编译器中的不透明数据结构，而不是它们的字符串表示，因此具有更好的运行时性能和内存性能；以及 (c) 更广泛的支持范围

标准 LLVM 版本。

6. 相关工作

自动提升 Python 代码的性能，即无需重写代码，一直是许多项目的研究及/或工程主题。所选方法包括即时编译 [10][11], 转译后进行提前编译 [12], 或将 Python 用作领域特定语言 [13][14]。即时编译提供了最符合 Python 风格的体验，因为它可以交互使用，并且在当今非常相关，因为它最适合异构计算环境。

PyPy 项目实现了一种替代的 Python 解释器，它使用跟踪式即时编译器来提升性能。PyPy 与标准 CPython 实现完全兼容，并且能够提供显著的加速比，尤其是在数值代码方面。Cppyy 在解释器级别集成到 PyPy 中，这使得 C++ 代码可以直接从即时编译的跟踪中调用，而无需函数包装器，并且能够内联对 C++数据的访问。与所有其他选项相反，PyPy 的跟踪式即时编译器不基于 LLVM。这减少了依赖和开销，但也意味着该即时编译器可以针对的平台更少，提供的优化也更少。PyPy 的一个弱点是对现有 CPython 扩展模块的支持，由于内存模型不同，访问这些模块可能会相当缓慢。此外，由于需要映射内存模型，它在引用计数错误方面不如 CPython 宽容，从而导致不兼容性。

JAX 是基于 XLA 构建的 NumPy 替代实现。XLA 是一种领域专用编译器，用于优化机器学习中的线性代数。它可以对任何基于 NumPy 的代码进行即时编译，面向 CPU、GPU 和 TPU。它会利用实际使用的数据类型和尺寸等运行时信息，实现专门的优化。通过底层的 XLA，JAX 可以扩展用户定义函数和类型。不过，截至本文撰写时，JAX 尚无官方支持的扩展接口，而推荐的实现方式会使用未公开且不受支持的 API，这些 API 可能会发生变化。

我们发现，PyPy和JAX 都是实现既定目标的良好替代方案，但 Numba 提供了更好的平台：JAX 能够很好地支持 GPU，而PyPy 支持使用用户定义函数和类型进行扩展，Numba 则两者兼具。不过，我们将继续跟踪这些替代技术的发展；如果未来情况需要，我们可能会重新考虑这一选择。

7. 结论

编程语言之间的无缝集成对于包括高能物理在内的许多科学领域至关重要。Python和C++ 在该领域得到广泛使用，集成这两种语言对于实现高效且易用的工作流程非常重要。即时编译器Numba能够将Python 代码编译为本机代码，但不支持C++。为了解决这一问题，我们通过基于 cppyy 的新型反射接口，将C++引入Numba。cppyy是一个基于 Cling（C++解释器）的自动运行时绑定生成器。该方法提供了一个完全自动且透明的流程，包括与其他Numba扩展的集成，同时性能可与传统方法相媲美。这项技术使高能物理程序员能够使用Python开发和调试代码，并利用C++ 库，同时为选定的性能关键闭包启用 Numba 即时编译。

我们展示了通过我们的扩展使用 Numba 加速 Cppyy 时可实现 2–20倍的加速比。我们概述了利用 LLVM 的 Clang-Repl 组件和新开发的库 InterOp 进一步提升性能的方法。使用新基础设施获得的初步结果显示，在Cppyy 的两种不同场景中，模板化代码的处理速度提高了 1.4 到 144 倍，这将间接改进经 Numba 加速的 Python。

8. 致谢

本项工作得到美国国家科学基金会资助，资助编号为 OAC-1931408，并得到合作协议 OAC-1836650 的支持。本材料基于美国能源部科学办公室高级科学计算研究（ASCR）SC 项目 31 所支持的工作。

参考文献

[1] Pivarski J、Escott C、SmithN、Hedges M、Proffitt M、Escott C、Nandi J、Rembser J、bfis、benkrikler、Gray L、Davis D、Rodrigues E、SchreinerH、Gizdov K、Nollde 和 FackeldeyP 2021 scikit-hep/awkward-0.x：0.15.5 网址 https://doi.org/10.5281/zenodo.4520562[2] Pivarski J、Das P、Burr C、Smirnov D、Feickert M、Gal T、Kreczko L、Smith N、Biederbeck N、Shadura O、Proffitt M、benkrikler、Dembinski H、Schreiner H、RembserJ、R M、Gu C、Edoardo、RodriguesE、JMSchoeffmann、R¨ubenach J、Koch L、Peresano M、EichN、TurraR和 bfis 2021 scikit-hep/uproot3：3.14.4 网址 https://doi.org/10.5281/zenodo.4537826[3] Gray L、SmithN、Tovar B、Novak A、Chakraborty J、Fackeldey P、Hartmann N、WattsG、Thain D、Stark G、BenGalewsky、R¨ubenach J、FischerB、Taylor D、MoAly98、KondratyevD、Gessinger P、Chen YM E、PataJ、Woodard A、Potrebko A、Albert A、R M、slehti、Surma Z、Pedro K、dnoonan08 和 Hennessee A 2023 0 7 21 10 5281 7733568 Coffeateam/coffea：[4] 1997 *Nucl. Instrum. Meth. A* **389** 81 发布版本 v .. 网址 https://doi.org/ . /zenodo. Brun R 和Rademakers F 86 [5] *et al.* 4 2003 *Nucl. Instrum. Meth. A* **506** 250Agostinelli S（GEANT ）–303[6] Lam S K、stuartarchibald、Pitrou A、Florisson M、Seibert S、Markall G、esc、Anderson T A、rjenc29、Leobas G、luk-fa、Collison M、Bourque J、Meurer A、Oliphant TE、Riasanovsky N、Kaustubh、Wang M、densmirn、njwhite、Pronovost E、Totoni E、WieserE、SeefeldS、Grecco H、Peterson P、Virshup I、G M、Turner-TrauringI 和Bourbeau J 2022 numba/numba：版本 0.56.4 网址https://doi.org/10.5281/zenodo. 7289231[7] Lavrijsen W T LP 和 Dutta A 2016 *Proceedings of the 6th Workshop on Python for High-Performance and Scientific Computing* PyHPC '16（IEEE Press）第27–35 页 ISBN 9781509052202[8] Vassilev V、Canal P、Naumann A、Moneta L 和 RussoP 2012 第396 卷（IOP Publishing）第 052071 页 网址 https://iopscience.iop.org/article/10.1088/1742-6596/396/5/052071/pdf[9]llvmlite——用于编写即时编译器的轻量级 LLVM Python 绑定 网址 https://llvmlite. readthedocs.io/en/latest/[10] PyPy——Python 的快速、符合规范的替代实现 网址 https://www.pypy.org/[11] Bradbury J、FrostigR、Hawkins P、Johnson M J、LearyC、Maclaurin D、Necula G、Paszke A、VanderPlas J、Wanderman-Milne S和Zhang Q 2018JAX：可组合的 Python 变换+NumPy 程序网址 http://github.com/google/jax[12] Shed Skin——受限 Python 到 C++ 编译器 网址 https://github.com/shedskin/shedskin[13] 选择性嵌入式即时编译特化 网址 https://sejits.eecs.berkeley.edu/[14] Arkouda——超算规模的交互式数据分析：由 Chapel 驱动的 Python API网址 https://bears-r-us.github.io/arkouda/