计 算 机 研 究 与 发 展
Journal of Computer Research and Development
https://crad.ict.ac.cn
62(9): 2382−2395, 2025

一种异构系统下计算软件性能数据采集方法

顾蓓蓓[1,2] 邱霁岩[1,2] 王 宁[3] 陈 健[3] 迟学斌[1]

[1]（中国科学院计算机网络信息中心 北京 100083）

[2]（中国科学院大学 北京 101408）

[3]（北京并行科技股份有限公司 北京 100094）

（gbb@sccas.cn）

A Performance Data Collection Method for Computing Software in Heterogeneous Systems

Gu Beibei[1,2], Qiu Jiyan[1,2], Wang Ning[3], Chen Jian[3], and Chi Xuebin[1]

[1]（*Computer Network Information Center*, *Chinese Academy of Sciences*, *Beijing* 100083）

[2]（*University of Chinese Academy of Sciences*, *Beijing* 101408）

[3]（*Beijing Paratera Co.*, *Ltd.*, *Beijing* 100094）

Abstract Supercomputing has rapidly developed from traditional CPU clusters to heterogeneous platforms. With the type conversion of hardware platforms, it faces significant challenges in optimizing computing software programs and performance evaluation. Currently, some international mainstream parallel program performance analysis tools and software generally have low compatibility with domestic supercomputing heterogeneous system processor products, often requiring instrumentation and recompilation of code, and low accuracy in single node performance data collection. To improve these shortcomings, we propose a floating-point performance data collection method for heterogeneous system computing software. This method is based on the domestic supercomputing system verification platform to develop and verify the floating-point performance collection prototype. At present, effective collection of single node and multi node performance indicator data has been achieved, and it is non-invasive to the original program. There is no need to modify the code of the monitored program for monitoring in a plug-in manner, making it highly versatile. Finally, we conduct comparative experimental analysis with three types of programs: rocHPL, Cannon, and mixbench, and conduct performance data collection monitoring research on ResNet (residual network, ResNet) program for AI computing. We demonstrate that the collection method proposed in this article has high accuracy, achieves the expected collection effect in experiments, and has good reference value for program optimization, verifying the effectiveness of the proposed method.

Key words heterogeneous systems; performance indicators; floating-point data; collection program; performance evaluation

摘 要 超级计算已从传统CPU集群向异构平台快速发展，随着硬件平台的类型转换，对于计算软件程序调优及性能测评等都面临着重大挑战. 当前一些国际主流并行程序性能分析工具及软件普遍存在与国产超算异构系统处理器产品兼容性低、往往需要进行插桩及重编译代码的方式，且单节点性能数据采集准确度不高等问题. 为了改进这些不足，提出了一种异构系统计算软件浮点性能数据采集方法. 该方法基于

收稿日期：2024-06-17；**修回日期**：2025-01-15

基金项目：国家自然科学基金项目 (62372428)

This work was supported by the National Natural Science Foundation of China (62372428).

通信作者：迟学斌（chi@sccas.cn）

国产超算系统验证平台对浮点性能采集原型进行开发及验证. 目前已实现单节点和多节点性能指标数据的有效采集，且对原程序无侵入性，该方法无需修改需要被监控程序的代码，且无需采用插桩的方式进行监控，通用性强. 最后，与 rocHPL, Cannon, mixbench 这 3 类程序进行对比实验分析，并针对人工智能（artificial intelligence, AI）计算，在残差网络（residual network, ResNet）程序上开展了性能数据采集方面的监测研究. 证明提出的采集方法准确度较高，采集效果达到实验预期，且对程序调优具有较好的参考价值，验证了该方法的有效性.

关键词 异构系统；性能指标；浮点数据；采集程序；性能测评

中图法分类号 TP311.11

DOI: 10.7544/issn1000-1239.202440512 **CSTR:** 32373.14.issn1000-1239.202440512

当前信息技术的迅猛发展使得超级计算机系统架构日益复杂. 随着以 GPU 为代表的加速器技术的发展，加速器浮点性能越来越高，CPU 与加速器的浮点性能差距越来越大. 异构系统架构通过整合不同类型的处理器（如 CPU, GPU, FPGA 等），实现了资源的共享和协同工作，从而提高了系统的整体性能，成为当前计算机系统发展的整体趋势. 根据 2025 年 6 月最新公布的 TOP 500 排名，前 10 位中全部采用了异构系统架构，同时有 235 个上榜系统使用了加速器/协处理器技术，且呈现逐年增长的趋势. 这些充分表明超级计算已从传统 CPU 集群向异构平台快速发展，随着硬件平台的类型转换，无论从各种类型软件（计算软件、管理软件）到服务模式都面临着重大挑战.

由于超级计算机系统研制的巨大成本以及科学家们希望通过高性能计算实现新研究成果的目标，在系统上执行的应用程序应该尽可能高效. 因而，为此类超级计算机系统开发应用程序的一个重要步骤就是性能分析和调优. 但因为此类机器的复杂架构，人们在设计阶段就对应用程序进行优化以获得最佳性能几乎是不可能的. 因此，有必要对超级计算机系统上运行的计算软件性能进行科学评测，并不断改进和优化程序，以此提升软件的真实性能.

为了真实反映计算软件在异构系统架构下发挥的实际性能，科研人员需要研究如何准确构建浮点性能特征指标，这也成为目前计算软件性能指标采集方法研究最为核心的问题之一. 浮点性能作为衡量计算软件性能的核心指标，直接影响到计算任务的执行效率、系统资源的利用率以及整体计算成本等.

当前一些国际主流并行程序性能分析工具及软件普遍存在与国产超算异构系统处理器产品兼容性低、往往需要进行插桩及重编译代码的方式，且单节点性能数据采集准确度不高等问题. 譬如，当前的程序性能分析工具主要面向异构系统设计和研发，普遍支持 CPU, GPU, CPU+GPU 等硬件环境，但对于国产超算异构系统的协处理器等还没有体系化的研究成果. 再譬如，用于评估 GPU 混合操作强度的基准测试工具 mixbench[1]，目前已经开发了 CUDA, HIP, OpenCL, SYCL 等多种实现，可以对系统性能数据进行准确采集，但在适用于 GPGPU 架构的计算密集型任务加速的协处理器上还没有开发实现；像 Timemory[2], PAPI 等并行程序需要进行插桩，同时需对被评测计算软件程序代码进行重编译，因而对于这类性能分析工具往往不具备通用性及易用性；此外，像基准测试（Linpack Benchmark）的扩展版本 HPL 程序，主要用于解决大规模的线性方程组，由于它更多地专注于并行计算，因此在某些情况下可能无法准确评估单节点性能.

为了改进上述问题和不足，我们提出了一种异构系统下计算软件的性能数据采集方法，目前已实现单节点及多节点性能指标数据的有效采集，且对原程序无侵入性，无需修改需要被监控程序的代码，并无需采用插桩的方式进行监控，通用性强，且在国产超算异构系统平台上得到了该方法的有效性实验验证.

本文工作的主要贡献包括 3 方面：

1）针对异构系统架构的复杂性，在国产超算系统上提出一种新的浮点性能数据采集方法，可为有效构建国产系统计算软件性能评测体系提供准确的分析依据；

2）开发了异构系统下浮点性能数据采集程序，实现了对并行计算软件浮点性能数据的收集和分析，且对原程序无侵入性，无需修改需要被监控程序的代码，并无需采用插桩的方式进行监控，通用性强；

3）通过与 rocHPL, Cannon, mixbench 这 3 种典型基准测试程序的对比实验，验证了本文所提出方法的有效性和实用性，并针对人工智能（AI）计算，在残

差网络(residual network, ResNet)程序上开展了性能数据采集方面的监测研究,为科学计算软件应用生态的良性发展提供了数据支撑.

1 相关工作

自20世纪70年代以来,研究人员相继开展了有关计算系统软件及工具性能测试和分析的研究工作,包括早期超算环境上的单一资源维度到多维度性能分析和测评工具的研发(如PerfExpert[3], Score-P[4], AutoTune[5], PAPI, Periscope[6]等),以及近年来在超大规模和异构超算系统上的性能测评和分析(如Timemory[2], HPC-MixPBench[7], rocHPL[8]等).以下是相关研究工作的详细介绍.

美国Argonne国家实验室应用数学所主任Pool[9]在1974年一系列非正式的讨论会中评估建立一套专门解线性系统问题之数学软件的可能性.时至今日,Linpack已成为国际上最流行的用于测试高性能计算机系统浮点性能的benchmark.利用高性能计算机,通过高斯消元法求解N元1次稠密线性代数方程组的测试,以此评价高性能计算机的浮点性能.其中Linpack测试包括3类:Linpack100, Linpack-1000, HPL.前2种测试运行规模较小,已不适合现代计算机的发展,HPL评测程序报告的每秒浮点运算次数统计结果仍然成为如今国际TOP 500组织排名的重要依据[10].Eustace等人[11]开发了一种用于构建高性能程序分析工具的灵活接口ATOM(analysis tools for object modification).该接口将OM通用对象修改库与特定于工具的插装文件相链接,以产生定制插装工具.通过读入用户应用程序,并通过添加对特定于工具的分析过程的调用来修改它,以此实现高性能程序的性能分析.为了使用户能够轻松访问计数器,以帮助进行性能分析、建模和调优,Browne等人[12]提出一种可扩展的跨平台硬件计数器应用性能调优架构PAPI.这些功能为源代码级别性能分析软件提供了必要的基础.慕尼黑工业大学Gerndt等人[6]基于APART工作组开发的一套自动性能分析工具Periscope,可以完成从性能特性规范到与2个监视系统的集成,通过搜索过程以实现应用分布式自动在线分析,可扩展到数千个处理器.Miceli等人[5]提出一种插件驱动的并行应用程序自动调优方法AutoTune.AutoTune可以扩展到Periscope,由此产生的Periscope调优框架可为多核和众核架构调优串行和并行代码,并返回可以集成到生产版本代码中的调优建议.为了使应用程序开发人员和用户更容易进行性能优化,Martin等人[3]开发了一个将简单的用户界面与复杂的分析引擎相结合的工具PerfExpert.此外,对于每个瓶颈PerfExpert都提供了简明的性能评估,并建议程序员可以采取哪些步骤来提高性能.Dieter等人[4]设计和实现了一个面向超级计算应用的联合测量基础设施Score-P.该基础设施具有高可扩展性和易用性,且可作为一些性能工具(如Periscope, Scalasca, Vampir, TAU等)的通用基础,目标是增强可伸缩性、改进互操作性和降低维护成本.Madsen等人[2]提出了一个名为Timemory的通用性能测量和分析框架.该框架为现有的性能测量和分析工具提供了一个通用的接口,解决了用户级扩展的限制,使得现有的分析工具和库可以轻松地与Timemory框架集成.同时支持多种并行模型、数据采集技术,可以满足多种应用场景下的性能测量和分析需求.同年,Parasyris等人[7]开发了一套用于混合精度分析的高性能计算基准套件HPC-MixPBench,由一组具有代表性的内核和基准测试组成,广泛应用于高性能计算领域.该基准套件配有一个测试工具框架,可以在其中插入不同的工具,并在基准集上进行评估.Chalmers等人[8]对AMD的高性能Linpack(HPL)基准测试的开源实现rocHPL进行了性能优化,通过高度优化的线性代数库利用节点上的高通量GPU加速器,以及整个CPU插槽来执行延迟敏感的因子分解阶段,并在橡树岭国家实验室的Frontier系统早期接入集群的单个节点上,以及扩展到多个节点上,展示了HPL基准测试的一些性能结果.

上述相关工作可以帮助人们更好地了解系统的运行情况和性能瓶颈,针对性地进行优化和改进,提高系统的整体性能和效率.同时,也为本文的异构系统性能指标采集方法研究提供了重要的理论和实践基础.

2 采集方法

本文提出了一种异构系统计算软件性能指标数据采集方法,如图1所示.该方法基于国产超算系统验证平台对浮点性能采集原型进行开发.通过使用AMD基于异构系统架构实现的rocm runtime技术的API接口完成.其中,2.1节主要介绍性能数据采集程序,2.2节主要介绍rocHPL, Cannon, mixbench这3类验证程序及算法的设计,以及ResNet应用程序的原理分析.

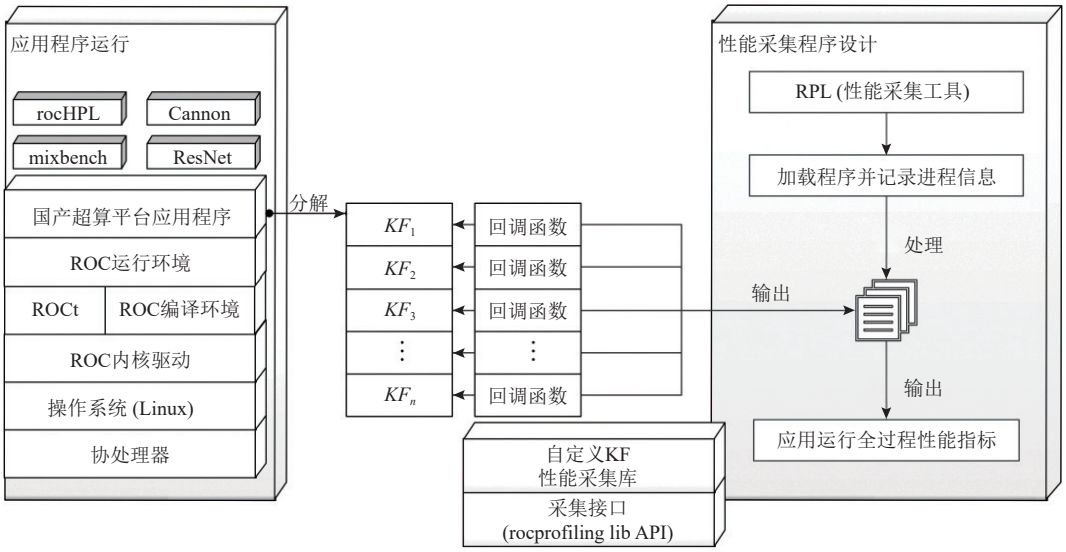


Fig. 1 Framework of performance data collection methods

图 1 性能数据采集方法框架

2.1 性能指标数据采集程序介绍

本文提出了一种性能指标数据采集程序 ROC-Profiling Library(简称 RPL)，可以实现单节点和多节点性能指标数据的有效采集. 我们自定义了一个浮点性能数据采集库(单位为 FlOPS)，并研发了一个采集接口 rocprofiling lib API，用以采集以核函数(kernel function, KF)为单位的性能文件. 此外，我们还设计了一个单机版性能收集工具 RPL，用以记录测试程序进程信息(如 PID、开始和结束时间等)，并对所收集的性能指标在时间维度上进行序列化，最终输出性能指标文件. 我们主要考虑到 rocprofiling lib API 是 AMD 提供的一个性能分析库接口，提供了一系列函数和数据结构，可用于收集和分析处理器在运行时的各种性能指标. 由于 rocprofiling lib API 是 AMD 官方提供的库，因此与 AMD 处理器的兼容性通常很好，能够确保数据的准确性和可靠性，无需修改需要被监控程序的代码，并无需采用插桩的方式进行监控，该采集程序的通用性较强. 性能指标数据采集程序流程如图 2 所示.

rocprofiling lib API 依赖 AMD rocm 生态的核心组成部分 HSA Runtime. 在 rocm 中，HSA Runtime 负责在 CPU, GPU 等异构计算单元提供底层的硬件控

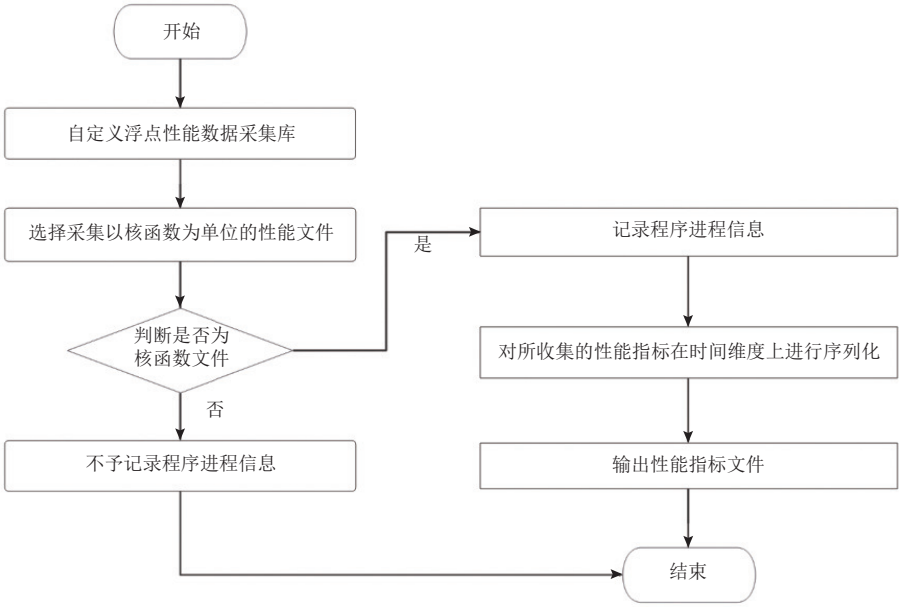


Fig. 2 Flowchart for data collection program of performance metrics

图 2 性能指标数据采集程序流程图

制和调度(如内核函数的加载、执行)、内存管理和硬件抽象层. 同时 HSA Runtime 提供一个环境变量 HSA_TOOLS_LIB, 通过此环境变量, 可以在执行基于 HSA Runtime 的 rocm 程序时动态加载特定的工具库实现对 HSA Runtime 的功能扩展, 如监控或调试. 基于此, 我们开发了 rocprofiling lib API 库, 在程序启动前, 通过设置环境变量 HSA_TOOLS_LIB, 实现在程序启动时对 rocprofiling lib API 库的动态加载, 通过事件拦截和 API 来获取 GPU 任务的执行状态、内存使用、API 调用时序等. 例如, 当 HSA Runtime 提交一个 GPU 内核执行时, rocprofiling lib API 库可以记录下该内核的提交时间、执行开始时间、结束时间、使用的 GPU 卡, 以及在这段时间内的硬件性能数据. rocprofiling lib API 库在加载后, 会实时监控 HSA Runtime 核函数调度和执行的性能指标数据, 在程序运行的整个生命周期内, 会持续按特定的格式输出性能指标数据到指定文件中, 基于这个特性, 可以实现准实时的 GPU 性能分析, 或在程序运行完毕后进行综合的性能分析. 我们选择采集以核函数为单位的性能文件, 记录和分析核函数对算法性能影响的文件. 通过分析和比较性能文件中的数据, 更好地理解核函数在算法中的作用, 并优化核函数以提高算法的性能. 同时, 这些性能文件也可以为后续研究提供有价值的参考数据.

在性能指标的选择上, 我们选择了 *SQ_INSTS_VALU* 这个指标作为基准采集指标, *SQ_INSTS_VALU* 代表在 GPU 内核中由矢量算术逻辑单元(vector arithmetic logic unit, VALU)执行的指令数, 这个指标数据会侧面反映程序的浮点性能 F(单位为 GFLOPS)的执行情况. 目前采用的计算公式为:

$$F=\frac{SQ_INSTS_VALU\times gpi}{10^9\times t},\tag{1}$$

其中, *gpi* 为实验测得的经验值, 针对本实验平台(AMD Vega20), 取值为 125.038 8, 针对不同的平台, *gpi* 值可能需要进行相应的调整; t 为计算时长, 在 *SQ_INSTS_VALU* 按秒进行归一化后, t 取值为 1.

此外, 对所收集的性能指标在时间维度上进行序列化, 主要涉及将性能指标数据按照时间顺序进行编码和转换, 以便于存储、传输和后续的分析处理. 序列化是将数据结构或对象状态转换为可以存储或传输的格式的过程. 通过对性能指标在时间维度上进行序列化, 我们可以方便地存储、传输和分析大量的性能数据, 从而更好地了解系统的性能状况, 发现潜在的性能问题, 并进行优化和改进.

该采集程序的伪代码如算法 1 所示.

算法 1. 采集程序.

① /*注册核函数运行态数据采集*/
② *OnLoadToolProp* (*rocprofiler_settings_t** *settings*);
③ /* 加载配置文件及环境变量 */
④ *LoadEnv*();
⑤ /*向 rocm 内核注册监视指标信息*/
⑥ *registerMonitorMetrics*();
⑦ /*获取 profiling 队列*/
⑧ *getProfilingVec*();
⑨ /*获取可用 GPU agents 信息*/
⑩ *HsaRsrcFactory*::*Instance*(). *GetGpuAgent Info*();
⑪ /* 打开 GPU profiling 池接口*/
⑫ for i = 0 to n do
⑬ *rocprofiler_pool_open*();
⑭ end for
⑮ /* 向 GPU 注册 dispatch 回调接口*/
⑯ *rocprofiler_set_queue_callbacks*();
⑰ /* 回调接口处理*/
⑱ *dispatch_callback_opt*();
⑲ /* 获取核函数运行 GPU agent 信息*/
⑳ *GetAgentInfo*();
㉑ /* 获取 profiling 上下文信息*/
㉒ *rocprofiler_pool_fetch*();
㉓ /* 通过上下文获取指标计数*/
㉔ *rocprofiler_get_group*.

基于 2.1 节性能指标数据采集程序的设计与描述, 我们将在 2.2 节中设计 4 类程序及算法, 对上述采集程序的准确性及可行性进行验证分析.

2.2 验证程序及算法设计

在应用程序运行设计环节, 我们主要在国产超算平台上进行采集方法的验证. 具体配置参数包括: Linux 操作系统、ROC Kernel Driver, rocm 库、Tool-lib Interface, 以及基于 HSA 架构实现的 rocm runtime 技术的 API 接口完成, 并同时选择使用 rocHPL, Cannon, mixbench, ResNet 这 4 类程序对采集方法的有效性进行测试验证. 以下是这 4 类程序及算法原理的相关介绍.

2.2.1 rocHPL 程序

我们使用 rocHPL 验证采集程序的准确性和可行性. rocHPL[8] 是基于 HPL 基准应用程序的基准测试工具, 在 AMD 的 Radeon 开放计算 ROCm 平台、运行时和工具链之上进行实现. 其根据 HIP 编程语言创建, 并针对 AMD 最新的独立 GPU 进行优化, 能充分利

用 AMD ROCm 平台计算卡性能. rocHPL 采用精细的并行化策略，充分利用 AMD GPU 的并行计算能力，优化数据布局和内存访问模式，最大程度地减少内存访问延迟和提高内存带宽利用率.

此外，ROCBLAS 作为 AMD Radeon 开放计算平台(ROCm)中的基本线性代数子程序(BLAS)库，提供了一组高效的基本线性代数操作函数，如矩阵乘法、矩阵向量乘法等. rocHPL 可以利用 ROCBLAS 中优化的 BLAS 函数来执行一些基本的线性代数操作，从而加速 LU 分解过程和其他计算操作，获得更好的性能和效率. rocHPL 程序的伪代码如算法 2 所示.

算法 2. rocHPL 程序.

① *generate_random_matrix* (***A***, *n*);
② *allocate_and_initialize_memory* (***A***, *n*);
③ /* 执行 LU 分解*/
④ for *k* = 1 to *n* do
⑤ *synchronize_threads*();
⑥ /* 发送数据到计算设备 */
⑦ *send_to_device*();
⑧ /* 找到主元素并行广播 */
⑨ *find_and_broadcast_pivot* (***A***, *k*, *n*);
⑩ /* 执行部分选主消元 */
⑪ *partial_pivot_elimination* (***A***, *k*, *n*);
⑫ /* 对于每个列，计算乘法因子并更新下一列，基于 ROCBLAS 进行求解 */
⑬ for *j* = *k*+1 to *n* do
⑭ ***A***[*j*][*k*] = ***A***[*j*][*k*]/***A***[*k*][*k*];
⑮ for *i* = *k*+1 to *n* do
⑯ ***A***[*j*][*i*] = ***A***[*j*][*i*] − ***A***[*j*][*k*] × ***A***[*k*][*i*];
⑰ end for
⑱ end for
⑲ /* 同步数据 */
⑳ *device_synchronize*;
㉑ end for
㉒ *free_memory* (***A***).

为了保证数据的可靠性，我们对 rocHPL 程序特别编译了双精度和单精度版本，分别进行测试，通过对比 rocHPL 自身的输出和采集程序输出来验证采集程序的准确性和可行性. 在 2.2.2 节中，我们将展示异构系统下的国产超算系统在单个节点及多个节点上的 rocHPL 基准测试的性能采集分析结果.

2.2.2 Cannon 算法

我们在异构系统架构下实现了一个高性能的 Cannon 算法来验证采集程序的准确性和可行性. Cannon 算法专为分布式计算环境中的矩阵乘法任务设计，优化了多处理器系统中大规模矩阵的计算，因此在通信效率上具有显著优势. 该算法采用分块矩阵乘法策略，将矩阵细分成更小的子矩阵或块，并按网格状布局分配到各个处理器上. 通过对数据块执行有序轮换操作，Cannon 算法确保每个处理器能有效减少跨处理器通信，同时获取完成其计算任务所需的准确数据.

关于 Cannon 算法的原理具体介绍如下：

假设矩阵 $\boldsymbol{A}$, $\boldsymbol{B}$, $\boldsymbol{C}$ 可以分成 $m\times m$ 块矩阵，即 $\boldsymbol{A}=(\boldsymbol{A}_{i,j})_{m\times m}$, $\boldsymbol{B}=(\boldsymbol{B}_{i,j})_{m\times m}$, $\boldsymbol{C}=(\boldsymbol{C}_{i,j})_{m\times m}$. 其中 $\boldsymbol{A}_{i,j}$, $\boldsymbol{B}_{i,j}$, $\boldsymbol{C}_{i,j}$ 是 $n\times n$ 矩阵，进一步假定有 $p=m\times m$ 个处理机. 为了研究 Cannon 算法，我们引入块置换矩阵 $\boldsymbol{Q}=(\boldsymbol{Q}_{i,j})$，使

$$\boldsymbol{Q}_{i,j}=\begin{cases}\boldsymbol{I}_n, & j\equiv i+1 \bmod m,\\ \boldsymbol{O}_n, & \text{其他情况},\end{cases} \tag{2}$$

其中 $\boldsymbol{I}_n$ 和 $\boldsymbol{O}_n$ 分别是 n 阶单位矩阵和零矩阵. 定义对角块矩阵 $\boldsymbol{D}_A^{(l)}=\mathrm{diag}(\boldsymbol{D}_A^{(l)})=\mathrm{diag}(\boldsymbol{A}_{i,i+1} \bmod m)$，容易推导出：

$$\boldsymbol{A}=\sum_{l=0}^{m-1}\boldsymbol{D}_A^{(l)}\times\boldsymbol{Q}^l. \tag{3}$$

因此，

$$\boldsymbol{C}=\boldsymbol{A}\times\boldsymbol{B}=\sum_{l=0}^{m-1}\boldsymbol{D}_A^{(l)}\times\boldsymbol{Q}^l\times\boldsymbol{B}=\sum_{l=0}^{m-1}\boldsymbol{D}_A^{(l)}\times\boldsymbol{B}^{(l)}, \tag{4}$$

其中 $\boldsymbol{B}^{(l)}=\boldsymbol{Q}^l\times\boldsymbol{B}=\boldsymbol{Q}^l\times\boldsymbol{B}^{(l-1)}$，利用这个递推关系式，把处理器编号从 1 维映射到 2 维，即有 $\boldsymbol{P}_{\mathrm{myid}}=\boldsymbol{P}_{\mathrm{myrow,mycol}}$，数据 $\boldsymbol{A}_{i,j}$, $\boldsymbol{B}_{i,j}$, $\boldsymbol{C}_{i,j}$ 存放在 $\boldsymbol{P}_{i,j}$ 中，得到在处理器 $\boldsymbol{P}_{\mathrm{myid}}$ 上的算法，如图 3 所示.

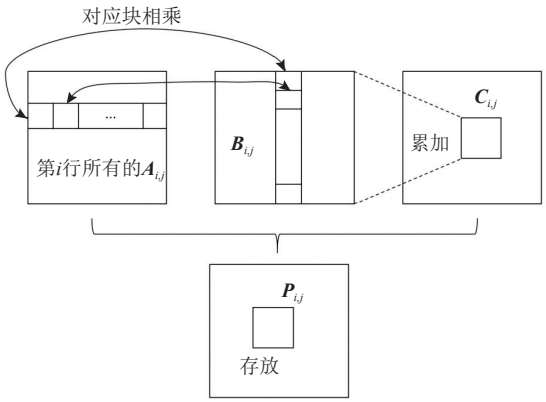


Fig. 3 Illustration of the algorithm on processor $\boldsymbol{P}_{\mathrm{myid}}$

图 3 处理器 $\boldsymbol{P}_{\mathrm{myid}}$ 上的算法示意图

基于上述有关 Cannon 算法的原理介绍，我们设计了一种 Cannon 算法的并行策略程序，用以与本文提出的采集程序 RPL 进行验证分析.

Cannon 算法的并行策略如图 4 所示.

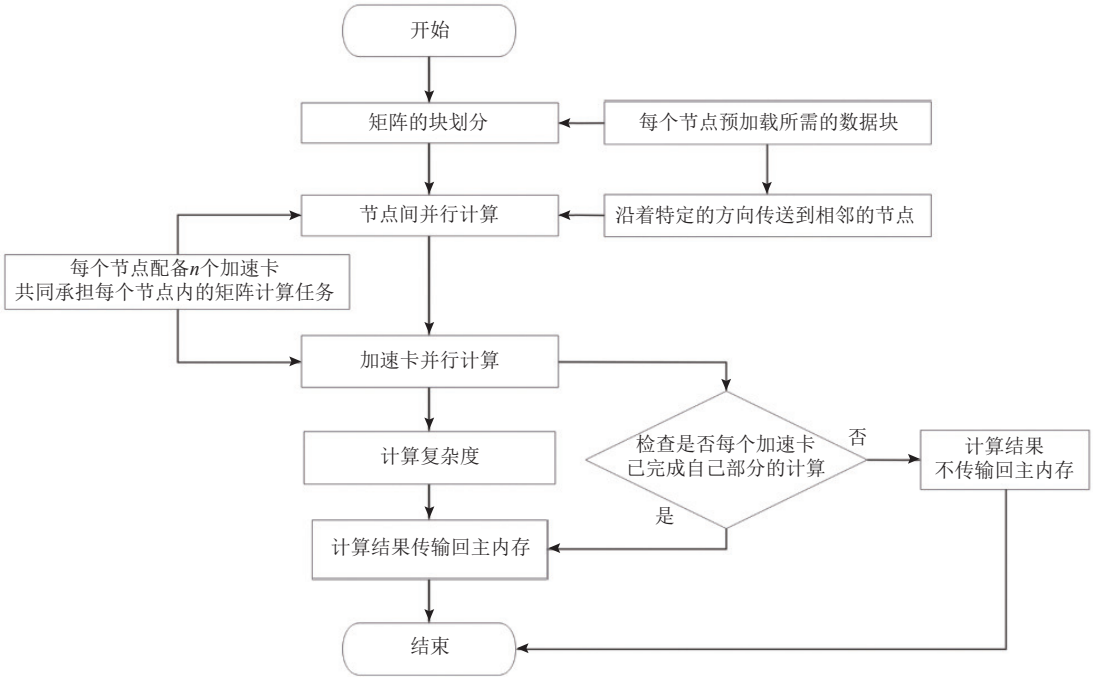


Fig. 4 Illustration for the design of parallel strategies in Cannon algorithm

图 4 Cannon 算法的并行策略设计示意图

将 Cannon 算法的并行策略设计主要分为以下部分：

1）使用 MPI 实现节点间并行 Cannon 算法，在节点间并行方面，通过 MPI 实现矩阵的分块处理. 计算整个矩阵$N\times N$被分割成$m\times m$个小块，每个节点负责计算矩阵的一个子块，子块的大小为$\frac{N}{m}\times\frac{N}{m}$，共计使用$m\times m$个节点. 算法初始阶段，每个节点会预加载所需的数据块. 然后，随着计算的进行，每个节点在处理完当前的数据块后，会将其沿着特定的方向传送到相邻的节点,每个节点的通信代价为$O\left(\frac{2N^2}{m^2}\right)$，计算复杂度为$O\left(\frac{N^3}{m^2}\right)$. 这种方式有效减少了节点间的通信需求，并允许每个节点同时进行计算和数据传输，最大限度地利用了网络带宽.

2）充分考虑并行策略在加速卡并行方面的扩展性，考虑到每个节点配备了n个加速卡，这些加速卡将共同承担每个节点内的矩阵计算任务. 在这种设置下，每个加速卡将负责计算其分配到的子矩阵块的一部分. 每个节点上的矩阵子块被进一步划分为n个更小的部分，每个加速卡负责处理其中的一个部分. 在进行加速卡计算时，每个加速卡需要处理的数据量是整个矩阵的$\frac{1}{nm^2}$. 以一个$N\times N$的矩阵为例，每个加速卡处理的通信代价为$O\left(\frac{N^2}{nm^2}\right)$，计算复杂度为$O\left(\frac{N^3}{nm^2}\right)$.

在每个加速卡完成自己部分的计算后，我们需要将计算结果传输回其所属节点的主内存中. 这一步骤涉及的通信代价也是$O\left(\frac{N^2}{nm^2}\right)$.

该 Cannon 算法程序伪代码如算法 3 所示.

算法 3. 基于 Cannon 算法的分布式和加速卡矩阵.

① procedure *CannonMatrixMultiply* (***A***, ***B***, ***C***, *m*, *n*)；
② $N\leftarrow$ 矩阵的总维度；
③ 每个节点负责的子矩阵大小 *block_size* ← *N*/*m*；
④ 每个加速卡负责的子矩阵大小 *sub_block_size* ← *block_size*/*n*；
⑤ 分配内存和显存资源；
⑥ 初始化本地矩阵 $\boldsymbol{A}_{\text{local}}$, $\boldsymbol{B}_{\text{local}}$, $\boldsymbol{C}_{\text{local}}$；
⑦ 将 ***A***, ***B*** 分块广播到各个节点；
⑧ for *node_id* = 0 to m^2-1 do
⑨ for *gpu_id* = 0 to $n-1$ do
⑩ 在 GPU 上分配 $\boldsymbol{A}_{\text{gpu}}$, $\boldsymbol{B}_{\text{gpu}}$, $\boldsymbol{C}_{\text{gpu}}$ 及双缓冲区；
⑪ end for
⑫ end for
⑬ 对 $\boldsymbol{A}_{\text{local}}$, $\boldsymbol{B}_{\text{local}}$ 进行初始行列对齐；
⑭ for *step* = 0 to $m-1$ do
⑮ 并行对每个节点的每个 GPU 执行：
⑯ 双缓冲异步复制 $\boldsymbol{A}_{\text{local}}$, $\boldsymbol{B}_{\text{local}}$ 的 1/2 到 $\boldsymbol{A}_{\text{gpu}}$,

$\boldsymbol{B}_{gpu}$;

⑰ 使用矩阵乘法核心计算 $\boldsymbol{C}_{gpu} \leftarrow \boldsymbol{C}_{gpu} + \boldsymbol{A}_{gpu} \times \boldsymbol{B}_{gpu}$;

⑱ 计算出的 $\boldsymbol{C}_{gpu}$ 是 $\boldsymbol{C}_{local}$ 的 1/4;

⑲ 对 $\boldsymbol{A}_{local}$, $\boldsymbol{B}_{local}$ 进行环形移位，与下一个节点的数据交换同时进行;

⑳ 同步所有 GPU 处理器;

㉑ end for

㉒ 收集所有点上的 $\boldsymbol{C}_{local}$ 到 $\boldsymbol{C}$;

㉓ end procedure

2.2.3 mixbench 程序

我们使用 mixbench 程序验证采集程序 RPL 的准确性和可行性. mixbench[1] 是一个 GPU 基准测试工具，旨在评估混合操作强度内核上 GPU 的性能范围. 它允许用户在一系列不同的操作强度值上对执行的内核进行自定义，以深入了解 GPU 在不同工作负载下的性能表现.

结合全局内存访问执行单精度浮点乘加运算、双精度浮点乘加运算、半精度浮点乘加运算（仅适用于 GPU）和整数乘加运算这 4 种类型进行实验. 当前的 GPU 能够通过将执行切换到能够执行计算操作的线程来隐藏内存延迟. 使用该工具可以评估计算设备在这 2 种类型的操作中的实际最优平衡.

本文使用 mixbench 程序对国产超算系统单个节点上开展性能指标采集，通过测试程序进行实验验证.

该采集程序的部分关键代码如算法 4 所示.

算法 4. mixbench 程序.

① /* 设置计算矩阵大小*/

② const long *reduced_grid_size* = *size*/(*ELEMENTS_PER_THREAD*)/128;

③ const int *BLOCK_SIZE* = 256;

④ const int *TOTAL_REDUCED_BLOCKS* = *reduced_grid_size* / *BLOCK_SIZE*;

⑤ /* 拆分矩阵计算网格*/

⑥ dim3 *dimReducedGrid*();

⑦ /* 定义计算函数*/

⑧ *benchmark_func*();

⑨ /* 加载到加速卡进行计算*/

⑩ *hipExtLaunchKernelGGL*().

2.2.4 ResNet 程序

我们选择对一种深度卷积神经网络模型 ResNet 进行监控测试，尝试针对包括训练和推理的 AI 计算做一些性能数据采集方面的研究. ResNet 由 He 等人[13] 在 2015 年提出，是一种深度神经网络架构. 通过引入残差连接解决了深度神经网络在训练过程中遇到的梯度消失或梯度爆炸问题，从而允许网络层数显著增加，性能也随之提升. 经典神经网络 ResNet 作为一种重要的深度学习模型，成为许多深度学习模型的基础架构，在人工智能领域有着较广泛的应用[1].

下面是有关 ResNet 程序算法的原理，主要分为残差学习、残差块、网络结构、瓶颈设计、训练与性能 5 个方面.

1）残差学习

传统的神经网络直接学习输入到输出的映射函数 $H(x)$. 而 ResNet 程序通过引入残差学习这一方法原理，使得网络学习的是输入 x 与输出 $H(x)$ 之间的残差 $F(x) = H(x) - x$. 这样，网络的实际输出是 $H(x) = F(x) + x$. 这种设计使得网络在训练时更容易优化，因为残差映射通常比原始映射更容易学习，如图 5 所示.

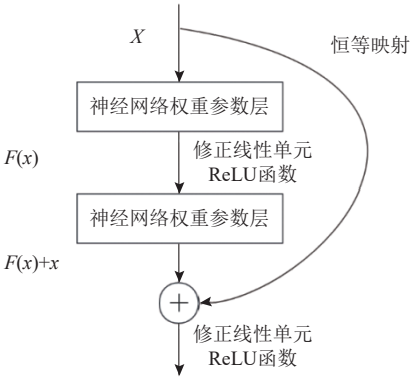


Fig. 5 Ilustration of the residual learning for building block model[13]

图 5 残差学习构建块模型示意图[13]

2）残差块

ResNet 的基本组成单元是残差块. 残差块内部通常包含多个卷积层（通常为 2～3 个），每个卷积层后面跟着批量规范化和非线性激活函数. 重要的是，残差块的输入会通过跨层连接直接加到输出上，形成残差连接. 这种连接是恒等映射，不会增加额外的参数和计算量.

3）网络结构

ResNet 的网络结构基于 VGG 网络，但去除了全连接层，改用全局平均池化层来减少参数量并提高模型的泛化能力. ResNet 的网络深度可以从几十层到几百层不等，如 ResNet-18, ResNet-34, ResNet-50, ResNet-101 等. 不同深度的 ResNet 在网络结构和参数量上有所不同，但都遵循了残差连接的基本原则.

4）瓶颈设计

在较深的 ResNet 版本中（如 ResNet-50 及以上），

为了降低计算复杂度,残差块采用了瓶颈设计.瓶颈设计首先通过一个 1×1 的卷积层来降低特征图的维度(称为降维),然后通过一个 3×3 的卷积层进行特征提取,最后再通过一个 1×1 的卷积层将特征图的维度恢复(称为升维).这种设计减少了参数量和计算量,同时保持了模型的性能.

5)训练与性能

ResNet 在训练过程中利用残差连接的优势,有效缓解了梯度消失问题,使得深层网络也能够被有效训练. ResNet 通过引入残差连接和残差块设计,解决了深度神经网络在训练过程中遇到的梯度消失问题,使得网络层数可以显著增加,性能也随之提升.

本文选择 ResNet 程序对性能数据进行监控,尝试针对包括训练和推理的 AI 计算做一些性能数据采集方面的研究.

3 实验与分析

3.1 实验环境介绍

本实验的测试环境部署在国产超算异构系统上,由 6 400 个计算节点组成,每个节点配备一个 AMD CPU(32 核 AMD EPYC 7551P CPU, 128 GB 主内存)和 4 个最新的 AMD GPUs(Vega20).系统中使用的 GPU 是 AMD 定制的固定低频(1.4 GHz)的特殊型号.

一个 CPU 有 32 个核心,提供 384 GFLOPS,而一个 GPU 提供 5.9 TFLOPS.在每个节点中,CPU 侧有 128 GB 的 DDR4 内存,GPU 侧有 64 GB 的 HBM2 内存. HBM2 内存的带宽高达 1TBps.所有节点通过具有 200 Gbps 高速网络接口的高带宽 FatTree 互连网络连接.在 1 个节点内,2 个计算组件通过 PCIe 总线(Gen 3.0)连接.总共有 25 600 个 GPU,它们一起提供了 152 PFLOPS 的峰值性能.在 2020 年之前,它是世界上最大的基于 PCIe 的 GPU 异构系统.实验的操作系统基于 CentOS 7.6, GNU 编译器套件版本为 7.3.1, OpenMPI 版本为 4.1.4, ROCm 版本为 4.0.1.

3.2 实验结果分析

在实验环节,我们分别使用 rocHPL, Cannon, mixbench 这 3 种测试程序对本文提出的性能指标数据采集方法进行实验验证和对比分析.同时,针对 AI 计算,在 ResNet 程序上开展了性能数据采集方面的监测分析.

1)rocHPL

① 单节点双精度采集分析.在这个实验中,我们首先进行了单节点 4 卡双精度的测试,在同一节点上对 rocHPL 双精度版根据不同的运算规模进行测试,在运行 rocHPL 的同时运行采集程序 RPL,输出结果如表 1 和图 6 所示.

rocHPL 的实际性能随着运算规模的增大而逐渐升高,基本呈线性关系;采集程序的输出比 rocHPL 输出稍低,并随着 rocHPL 的性能升高而进行相应的升高,两者的整体变化趋势趋于一致.经过对输出数据的运算,采集程序 RPL 的整体误差在−0.66% 左右.

Table 1 Output Results of rocHPL (Double Precision) and RPL Programs at Different Scales

表 1 不同规模下 rocHPL 程序(双精度)和 RPL 程序的输出结果

运算规模/阶数	运行时间/s	rocHPL 浮点运算速度/GFLOPS	RPL 浮点运算速度/FLOPS	误差/%
80 000	126	2 741.4	2 712.73	−1.05
81 000	129	2 763.7	2 749.63	−0.51
82 000	131	2 811.9	2 810.80	−0.04
83 000	126	2 838.4	2 806.07	−1.14
84 000	128	2 881.4	2 866.74	−0.51
85 000	142	2 912.7	2 888.28	−0.84
86 000	145	2 952.6	2 927.42	−0.85
87 000	148	2 977.9	2 970.92	−0.23
88 000	152	3 017.6	2 994.61	−0.76

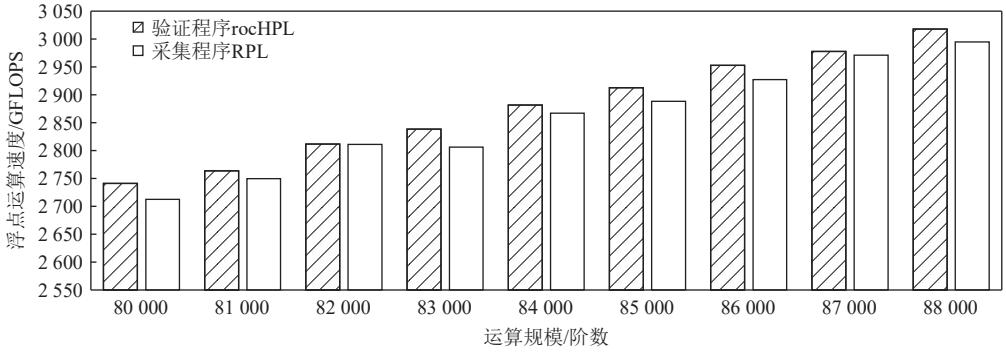


Fig. 6 Output results of double precision rocHPL and RPL programs using a single node with 4 GPUs at different scales

图 6 使用单节点 4 GPUs 的双精度 rocHPL 和 RPL 程序在不同规模下的输出结果

针对以上有关双精度浮点数计算准确性验证实验我们可以看出，在不同规模下采集程序输出的浮点性能与目标程序输出浮点性能的误差没有明显变化，采集准确性与计算规模无关.

② 单节点单精度采集分析. 由于单精度在运算中的内存占用比双精度小，在实际测试中，我们使用更高的规模进行了 4 卡单节点单精度的测试，如表 2 和图 7 所示. rocHPL 单精度版的浮点性能随着规模的增加而增高，基本呈线性关系，采集程序 RPL 的输出比 rocHPL 稍低，并随着 rocHPL 的性能升高而相应地升高，两者的整体变化趋势趋于一致. 经计算，整体误差为−1.65%.

Table 2 Output Results of rocHPL Programs (Single Precision) and RPL Programs at Different Scales

表 2 不同规模下 rocHPL 程序（单精度）和 RPL 程序的输出结果

运算规模/阶数	运行时间/s	rocHPL 浮点运算速度/GFLOPS	RPL 浮点运算速度/GFLOPS	误差/%
80 000	72	4 775.2	4 720.77	−1.14
90 000	91	5 415.0	5 319.97	−1.75
100 000	112	6 041.6	5 923.38	−1.96
110 000	135	6 663.6	6 535.68	−1.92
120 000	160	7 271.1	7 158.06	−1.55
127 000	179	7 708.7	7 586.57	−1.58

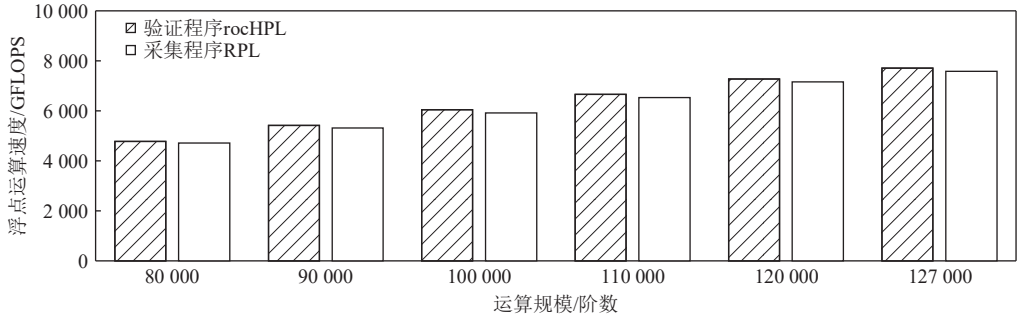


Fig. 7 Output results of single precision rocHPL and RPL program using a single node with 4 GPUs at different scales

图 7 使用单节点 4 GPUs 的单精度 rocHPL 和 RPL 程序在不同规模下的输出结果

经实验分析，在不同规模下采集程序输出的浮点性能与目标程序输出的浮点性能的误差没有明显变化，采集准确性与浮点数精度变化无关.

③ 多节点双精度采集分析. 为了验证采集程序在多节点上的扩展性，我们也对 rocHPL 在多节点上的采集进行了测试，鉴于整体内存资源的增加，进行了更大规模的测试.

表 3 给出了 rocHPL 双精度版和采集程序在不同规模下的输出结果. 从图 8 中可以看出，rocHPL 的浮点性能随着规模的增加而增高，基本呈线性关系，采集程序 RPL 的浮点运算值输出比 rocHPL 稍高，并随着 rocHPL 的性能升高而进行相应的升高，两者的整体变化趋势趋于一致. 经计算，整体的误差为 0.1%.

2）Cannon

本实验的主要目标是评估并优化 Cannon 并行矩阵乘法算法在异构系统上的执行效率. 异构系统广泛使用各种串行算法库，但为了实现高效的并行计算，特别是在处理矩阵乘法这类计算密集型任务时，理解并优化并行算法如 Cannon 是非常必要的. Cannon 算法通过优化任务分配和负载平衡，有助于提高并行处理的效率.

Table 3 Output Results of rocHPL (Single Precision) and RPL Programs at Different Scales

表 3 不同规模下 rocHPL 程序（单精度）和 RPL 程序的输出结果

运算规模/阶数	运行时间/s	rocHPL 浮点运算速度/GFLOPS	RPL 浮点运算速度/GFLOPS	误差/%
80 000	122	2 802.8	2 802.89	0.00
90 000	154	3 159.5	3 161.84	0.07
100 000	189	3 531.8	3 539.41	0.22
110 000	231	3 865.9	3 857.63	−0.21
120 000	272	4 252.3	4 255.47	0.07
130 000	319	4 602.7	4 614.90	0.27
140 000	369	4 973.6	4 981.31	0.16
150 000	423	5 330.3	5 342.72	0.23
160 000	483	5 673.7	5 681.24	0.13

因此，除了使用标准的 rocHPL 程序外，我们还实现了一个 Cannon 程序并且进行了深入测试，以验证其在实际应用中的表现，并通过采集程序监控其性能表现，以确保结果的可靠性和准确性. 这种测试不仅帮助我们评估算法的性能，还为进一步的优化提供了实验数据支持.

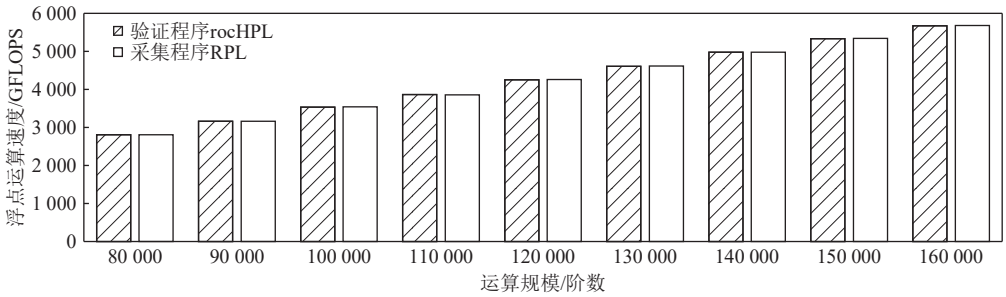


Fig. 8 Output results of single precision rocHPL and RPL program using four nodes with 16 GPUs at different scales

图 8 使用 4 节点 16 GPUs 的单精度 rocHPL 和 RPL 程序在不同规模下的输出结果

如表 4 所示，展示了 Cannon 程序和 RPL 程序在单节点相同规模但不同参数配置下的输出结果. 经测算，整体误差为−4.55%.

Table 4 Output Results of Cannon and RPL Programs with the Same Scales but Different Parameters for a Single Node

表 4 单节点相同规模不同参数下 Cannon 和 RPL 程序的输出结果

运算规模/阶数	参数配置	次数	运行时间/s	Cannon 输出浮点运算值/GFLOPS	RPL 输出浮点运算值/GFLOPS	误差/%
16 000	−g1，−p1	10	8.5	960.318	902.866	−5.89
16 000	−g4，−p1	10	20.7	395.157	382.871	−3.11

注：−g 代表每个节点的卡数. −p 代表每个方向的分块数，在 Cannon 程度中固定为节点数的平方根.

除了对单节点相同规模不同参数下 Cannon 程序和采集程序 RPL 的输出结果进行对比外，我们也针对多节点的情况对 Cannon 进行了较大规模的实验验证和测试，分别为 4 节点 16 GPU 和 16 节点 64 GPU. 如表 5 和表 6 所示，Cannon 是个弱扩展的程序，在规模相同的情况下，增加节点和 GPU 对性能几乎无影响，但更多节点使 Cannon 能够执行更大规模的算例. 如图 9 和图 10 所示，无论是 4 节点 16 GPU，还是 16 节点 64 GPU，随着规模的增加，Cannon 程序的性能也随之大幅提升，相应地，采集程序 RPL 的输出也随之增大. 经实验测算，误差在−1.72%~−3.33% 波动，误差并没有随着程序性能的提升有较大的波动. 经过计算，在 4 节点 16 GPU 的情况下，采集程序的整体误差为−2.62%，在 16 节点 64 GPU 的情况下，整体误差为−2.56%.

针对以上有关 Cannon 浮点数计算准确性验证实验我们可以看出，在单节点或多节点的各个规模下 RPL 程序输出的浮点性能与目标程序输出浮点性能的误差没有明显变化，且采集程序对我们实现的 Cannon 程序也有很好的监控效果.

Table 5 Output Results of Cannon and RPL Programs Using Four Nodes with 16 GPUs Under the Same Parameters at Different Scales

表 5 4 节点 16 GPUs 下不同规模相同参数下 Cannon 程序和 RPL 程序的输出结果

运算规模/阶数	参数配置	运行时间/s	Cannon 浮点运算速度/GFLOPS	RPL 浮点运算速度/GFLOPS	误差/%
16 000	−g4，−p2	24.62	332.70	321.70	−3.31
20 000	−g4，−p2	24.53	652.23	640.60	−1.78
24 000	−g4，−p2	24.83	1 113.45	1 077.30	−3.25
28 000	−g4，−p2	25.25	1 738.82	1 680.89	−3.33
32 000	−g4，−p2	30.81	2 127.25	2 072.69	−2.56
36 000	−g4，−p2	31.13	2 997.30	2 908.19	−2.97
40 000	−g4，−p2	32.16	3 979.96	3 909.19	−1.78
44 000	−g4，−p2	33.95	5 018.20	4 893.30	−2.49
48 000	−g4，−p2	34.52	6 408.18	6 209.09	−3.11
52 000	−g4，−p2	35.46	7 931.46	7 758.24	−2.18
56 000	−g4，−p2	37.75	9 304.78	9 053.28	−2.70
60 000	−g4，−p2	39.44	10 953.08	10 742.57	−1.92

注：−g 代表每个节点的卡数. −p 代表每个方向的分块数，在 Cannon 程度中固定为节点数的平方根.

3）mixbench

我们还对 mixbench 程序进行了测试. mixbench 是一个性能基准测试工具，专门用于评估在支持 HIP（异构接口移植性）编程模型的 GPU 平台上的混合计算和内存负载的工作性能. 在本次测试中，使用的 mixbench 为单进程版，且固定使用 GPU0 来进行计算，通过设计用例来模拟程序在合理场景（1 节点 1 进程）和不合理场景（1 节点多进程）的性能表现，验证采集程序在不同场景下的输出是否符合预期.

表 7 给出了在相同规模、相同测试次数和相同迭代次数下，节点上启动 1 个进程和多个进程的 mixbench 程序和 RPL 程序的输出结果.

场景中的 1n_1p 代表 1 节点运行 1 个进程，1n_4p 代表 1 节点运行 4 个进程，依此类推.

Table 6 Output Results of Cannon and RPL Programs Using Sixteen Nodes with 64 GPUs Under the Same Parameters at Different Scales

表 6 16 节点 64 GPUs 下不同规模相同参数下 Cannon 程序和 RPL 程序的输出结果

运算规模/阶数	参数配置	运行时间/s	Cannon 浮点运算速度/GFLOPS	RPL 浮点运算速度/GFLOPS	误差/%
16 000	−g4，−p4	24.73	331.22	324.15	−2.13
24 000	−g4，−p4	24.85	1 112.42	1 084.48	−2.51
32 000	−g4，−p4	25.45	2 574.84	2 491.57	−3.23
40 000	−g4，−p4	25.15	5 089.31	5 001.53	−1.72
48 000	−g4，−p4	35.12	6 298.12	6 141.64	−2.48
56 000	−g4，−p4	30.83	11 392.37	11 076.96	−2.77
64 000	−g4，−p4	32.50	16 133.76	15 662.66	−2.92
72 000	−g4，−p4	33.67	22 168.59	21 527.30	−2.89
80 000	−g4，−p4	34.38	29 783.12	29 280.91	−1.69
88 000	−g4，−p4	37.47	36 376.88	35 481.71	−2.46
96 000	−g4，−p4	37.34	47 390.02	45 946.58	−3.05
104 000	−g4，−p4	40.19	55 981.57	54 788.45	−2.13
112 000	−g4，−p4	43.33	64 853.90	62 756.47	−3.23

注：−*g* 代表每个节点的卡数. −*p* 代表每个方向的分块数，在 Cannon 程度中固定为节点数的平方根.

在合理场景下，如 1n_1p, 4n_4p, 16n_16p 随着节点数的增加，整体性随之线性增长，这是由于 mixbench 本身为单进程版，各进程间没有依赖和通信，所以性能的增长情况是符合预期的，与之对应的采集程序 RPL 的采集输出也随之线性增长，整体误差在 5% 以内.

另外，对比 1n_1p 和 1n_4p 以及 4n_4p 和 4n_16p 的结果可以看出，在不合理场景下，1n_4p 和 4n_16p 由于单节点上出现多个进程竞争同一个 GPU 资源导致整体的性能不仅没有增加，反而出现降低，这时相对应的 RPL 的输出也出现相应的降低，误差在 3% 以内.

4）ResNet

本文通过实验环节，判断采集程序 RPL 在面对 AI 领域的计算任务能否正常提供支持. 由于 ResNet 本身不会输出自身的运算性能，暂时不能通过对比验证采集程序 RPL 的准确性，所以增加本实验，用以证明 RPL 程序在 AI 领域的拓展性. 实验结果如表 8 所示.

由上述 rocHPL, Cannon, mixbench, ResNet 四种实验测试结果可以看出，采集程序对 rocHPL 和 mixbench 的性能监控采集有较高的准确性，对于移植的

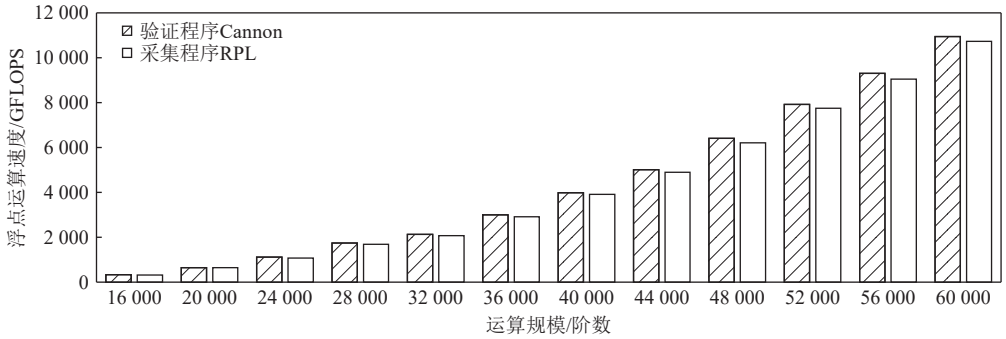


Fig. 9 Output results of single precision Cannon and RPL programs using four nodes with 16 GPUs at different scales

图 9 使用 4 节点 16 GPUs 的单精度 Cannon 和 RPL 程序在不同规模下的输出结果

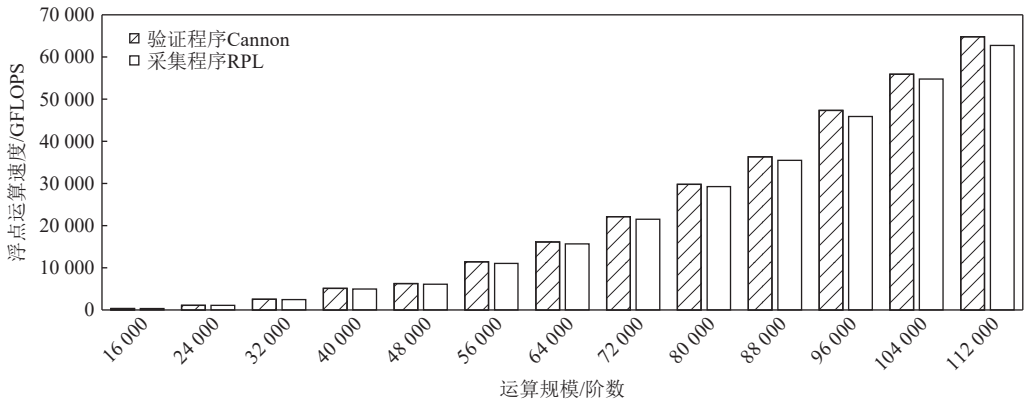


Fig. 10 Output results of Cannon and RPL programs using four nodes with 64 GPUs at different scales

图 10 使用 4 节点 64 GPUs 的 Cannon 程序和 RPL 程序在不同规模下的输出结果

Table 7 Output Results of mixbench and RPL Programs in Different Scenarios with the Same Scales and Parameters

表 7 相同规模和相同参数下不同场景 mixbench 程序和 RPL 程序的输出结果

场景	矩阵规模	测试次数	迭代次数	运行时间/s	mixbench 浮点运算速度/GFLOPS	RPL 浮点运算速度/GFLOPS	误差/%
1n_1p	320×1 024×1 024	100	512	63	5 381.88	5 252.42	−2.41
1n_4p	320×1 024×1 024	100	512	286	4 806.03	4 742.65	−1.32
4n_4p	320×1 024×1 024	100	512	64	19 866.15	20 686.44	4.13
4n_16p	320×1 024×1 024	100	512	287	19 145.15	18 741.42	−2.11
16n_16p	320×1 024×1 024	100	512	64	80 431.26	82 745.77	2.88

Cannon 的性能监控采集准确性稍差, 但对于不同规模或不同参数的测试, RPL 的输出会表现出极高的趋同性. 此外, 使用本文的采集方法, 还可以针对包括训练和推理的 AI 计算做一些性能数据采集方面的研究, 如使用 ResNet 进行监控测试, 证明 RPL 在 AI 领域的拓展性. 由此可见, 使用此种方法的软件性能指标采集具有准确性和实用性, 具有较高的数据参考价值.

4 总 结

本文针对异构系统架构的复杂性, 在国产超算系统上提出了一种新的浮点性能数据采集方法. 该方法基于国产超算系统验证平台对浮点性能采集原型进行开发及验证, 并基于此方法开发了一种异构系统下浮点性能数据采集程序. 该程序实现了对并行计算软件浮点性能数据的收集和分析, 且对原程序无侵入性, 无需修改需要被监控程序的代码, 并无需采用插桩的方式进行监控, 通用性强. 目前已实现单节点及多节点性能指标数据的采集, 并在 rocHPL, Cannon, mixbench, ResNet 这 4 类程序中完成测试验证, 采集程序准确度较高, 采集效果达到实验预期, 对程序调优具有较好的参考价值, 由此可证明该方法的有效性和实用性, 可为有效构建国产计算软件性能评测体系提供准确的分析依据.

下一步我们将继续在多个计算软件应用程序上对本文采集程序进行测试验证, 同时进一步开展 AI 计算方向的测试程序实验验证, 并对采集方法加以优化完善. 我们还将在 HSA 架构下的其他平台上进行实验分析和应用推广, 希望可以为国产科学计算软件应用生态的良性发展提供数据和理论支持.

Table 8 Output Results of RPL Program Under the Multiple Parameters in ResNet Program

表 8 RPL 程序在 ResNet 程序中多个参数下的输出结果

精度	模型	任务性质	参数大小	批量大小	Class 数量	GPU 数量	RPL 浮点运算速度/GFLOPS
float	resnet34	训练	21 797 672	32	1 000	1	87.52
float	resnet34	训练	21 797 672	32	1 000	4	115.10
float	resnet34	测试	21 797 672	32	1 000	1	39.96
float	resnet34	测试	21 797 672	32	1 000	4	53.62
float	resnet50	训练	25 557 032	32	1 000	1	100.31
float	resnet50	训练	25 557 032	32	1 000	4	124.94
float	resnet50	测试	25 557 032	32	1 000	1	72.64
float	resnet50	测试	25 557 032	32	1 000	4	110.18
float	resnet101	训练	44 549 160	32	1 000	1	162.12
float	resnet101	训练	44 549 160	32	1 000	4	209.64
float	resnet101	测试	44 549 160	32	1 000	1	117.02
float	resnet101	测试	44 549 160	32	1 000	4	188.67
float	resnet152	训练	60 192 808	32	1 000	1	233.12
float	resnet152	训练	60 192 808	32	1 000	4	292.06
float	resnet152	测试	60 192 808	32	1 000	1	183.30
float	resnet152	测试	60 192 808	32	1 000	4	258.90
double	resnet34	训练	21 797 672	32	1 000	1	306.95
double	resnet34	训练	21 797 672	32	1 000	4	494.84
double	resnet34	测试	21 797 672	32	1 000	1	194.79
double	resnet34	测试	21 797 672	32	1 000	4	259.68
double	resnet50	训练	25 557 032	32	1 000	1	370.14
double	resnet50	训练	25 557 032	32	1 000	4	517.37
double	resnet50	测试	25 557 032	32	1 000	1	163.06
double	resnet50	测试	25 557 032	32	1 000	4	272.99
double	resnet101	训练	44 549 160	32	1 000	1	417.67
double	resnet101	训练	44 549 160	32	1 000	4	500.17
double	resnet101	测试	44 549 160	32	1 000	1	218.62
double	resnet101	测试	44 549 160	32	1 000	4	305.26
double	resnet152	训练	60 192 808	32	1 000	1	455.73
double	resnet152	训练	60 192 808	32	1 000	4	551.28
double	resnet152	测试	60 192 808	32	1 000	1	323.48
double	resnet152	测试	60 192 808	32	1 000	4	322.89

作者贡献声明: 顾蓓蓓主要负责方法研究、实验设计、论文写作与格式校正; 邱霁岩、王宁、陈健主要负责实验分析和数据处理; 迟学斌主要负责论文框架设计和实验指导.

参 考 文 献

[1] Szegedy C, Liu Wei, Jia Yangqing, et al. Going deeper with

convolutions[C/OL]//Proc of the 28th Conf on Computer Vision and Pattern Recognition. Piscataway, NJ: IEEE, 2015[2025-01-09]. https://ieeexplore.ieee.org/document/7298594

[2] Madsen J R, Awan M G, Brunie H, et al. Timemory: Modular performance analysis for HPC[C]//Proc of the 35th Conf on ISC High Performance 2020 (ISC 2020). Berlin: Springer, 2020: 434–452

[3] Martin B, Kim B D, Jeff D, et al. PerfExpert: An easy-to-use performance diagnosis tool for HPC applications[C/OL]//Proc of the 24th Int Conf for High Performance Computing, Networking, Storage and Analysis. Piscataway, NJ: EEE, 2010[2025-01-09]. https://ieeexplore.ieee.org/document/5644905

[4] Dieter M, Scott B. Bischof C, et al. Score-P: A unified performance measurement system for petascale applications[C]//Proc of Competence in High Performance Computing. Berlin: Springer, 2012: 85–97

[5] Miceli R, Civario G, Sikora A, et al. AutoTune: A plugin-driven approach to the automatic tuning of parallel applications[C]//Proc of the 11th Int Conf on Applied Parallel and Scientic Computing. Berlin: Springer, 2013: 328−342

[6] Gerndt M, Kereku E. Periscope: Advanced techniques for performance analysis[C]//Proc of the Int Conf on Parallel Computing: Current & Future Issues of High-End Computing 2005[2024-10-15]. https://www.researchgate.net/publication/221552437

[7] Parasyris K, Lna I, Menon H, et al. HPC-MixPBench: An HPC benchmark suite for mixed-precision analysis[C]//Proc of the 17th IEEE Int Symp on Workload Characterization. Piscataway, NJ: IEEE, 2020: 25−36

[8] Chalmers N, Kurzak J, McDougall D, et al. Optimizing high-performance linpack for exascale accelerated architectures[J]. arXiv preprint, arXiv: 2304.10397v1, 2023

[9] Dongarra J, Luszczek P, Petitet A. The LINPACK benchmark: Past, present and future[J]. Concurrency and Computation: Practice and Experience, 2003, 15(9): 803−820

[10] Li Leisheng, Yang Wenhao, Ma Wenjing, et al. Optimization of HPL on complex heterogeneous computing system[J]. Journal of Software, 2021, 32(8): 2307−2318 (in Chinese)
(黎雷生, 杨文浩, 马文静, 等. 复杂异构计算系统 HPL 的优化[J]. 软件学报, 2021, 32(8): 2307−2318)

[11] Eustace A, Srivastava A. ATOM: A flexible interface for building high performance program analysis tools[C/OL]//Proc of the Winter 1995 USENIX Conf. Berkeley, CA: USENIX Association, 1995[2025-01-09]. https://dl.acm.org/doi/abs/10.5555/1267411.1267436

[12] Browne S, Dongarra J, Garner N, et al. A scalable cross-platform infrastructure for application performance tuning using hardware counters[C]//Proc of the 12th Int Conf for High Performance Computing, Networking, Storage and Analysis. Piscataway, NJ: IEEE, 2000: 42−55

[13] He Kaiming, Zhang Xiangyu, Ren Shaoqing, et al. Deep residual learning for image recognition[C]//Proc of the 29th Conf on Computer Vision and Pattern Recognition. Piscataway, NJ: IEEE, 2016: 770−778



Gu Beibei, born in 1982. PhD, associate professor. Member of CCF. Her main research interest includes performance evaluation and analysis of high performance computing software applications.
顾蓓蓓, 1982 年生. 博士, 高级工程师. CCF 会员. 主要研究方向为高性能计算软件应用性能测评分析.



Qiu Jiyan, born in 1998. PhD candidate. His main research interests include AI for science and high performance computing.
邱雾岩, 1998 年生. 博士研究生. 主要研究方向为人工智能驱动的科学研究、高性能计算.



Wang Ning, born in 1988. Bachelor. His main research interest includes performance analysis and optimization of general-purpose processors.
王　宁, 1988 年生. 学士. 主要研究方向为通用处理器性能分析和优化.



Chen Jian, born in 1977. PhD. Vice chairman of CCF. His main research interests include AI and high performance computing.
陈　健, 1977 年生. 博士. CCF 副理事长. 主要研究方向为人工智能、高性能计算.



Chi Xuebin, born in 1963. PhD, professor, PhD supervisor. His main research interest includes parallel computing.
迟学斌, 1963 年生. 博士, 研究员, 博士生导师. 主要研究方向为并行计算.