使用 Fortran 标准并行加速生产级太阳磁流体动力学代码：从 OpenACC 到 'do concurrent'

Ronald M. Caplan、Miko M. Stulajter 和 Jon A. Linker*PredictiveScience Inc.，9990 Mesa Rim Road，170 室，圣迭戈，CA 92121，caplanr@predsci.com*

arXiv:2303.03398v2 [cs.MS] 8 Mar 2023

摘要—**人们越来越关注使用标准语言结构进行加速计算，以避免依赖（通常由供应商特定的）外部 API。这些结构有望具备更好的可移植性，并更加"面向未来"。对于 Fortran 代码，目前的重点是** `doconcurrent` **（并发循环，DC）。尽管已有一些使用 DC 对基准代码和/或小型代码进行 GPU加速的成功案例，但要广泛采用该方法，还需要展示其在完整规模应用程序中的使用效果。本文考察了在一个名为球外磁流体动力学算法（Magnetohydrodynamic Algorithm outside a Sphere，MAS）的生产级应用程序中使用 DC 的当前能力和性能。MAS 是用于研究日冕和日球层动力学的先进模型，代码超过 70,000 行，此前已通过 MPI+OpenACC 移植到图形处理器。我们尝试尽可能消除其中的 OpenACC 指令，改用 DC。我们展示了借助英伟达**`nvfortran`**编译器对 Fortran 202X 预览版实现、统一托管内存以及经过修改的 MPI 启动方法，无需使用任何 OpenACC 指令，即可实现跨多个图形处理器的 GPU加速。然而，这样做会导致速度降低 1.25 倍至 3 倍。我们讨论了避免这一性能损失所需的未来改进，并展示了如何在将 OpenACC 指令数量减少五倍的同时，仍保持接近原始代码的性能。**

关键词-**加速计算；Fortran；OpenMP；OpenACC；do concurrent；标准语言并行**

I. 引言

加速器（例如图形处理器）的使用已在高性能计算领域普及。这得益于其能效高、性能密集（即一台加速器的性能可媲美多个 CPU 计算节点）。然而，开发代码和/或将代码移植到加速器上运行仍然是一项挑战，尤其是在保持可移植性的同时。由于历史上 C、C++ 和 Fortran 等基础标准语言缺乏对加速器的直接支持，多年来涌现了大量（通常为特定厂商提供的）加速器 API、库和语言扩展，其支持程度、可移植性和软件层级各不相同（例如，从高级前端到低级直接设备编程）。其中包括 CUDA、ROCm、OpenCL、SYCL、Kokkos、RAJA、DPC++、PROTO、OpenMP 和 OpenACC [1]–[4]。对于遗留代码，以及追求最大可移植性、长期可维护性和代码可读性的新代码而言，基于指令的解决方案（如 OpenMP 和 OpenACC）往往是首选。这是因为这些指令在代码中以特殊注释的形式出现，因此代码仍可在较早的硬件/软件环境中编译和运行，因为这些指令会被直接忽略。尽管基于指令的编程模型功能丰富，但它们仍可能降低源代码的可读性，而且需要领域科学家接受过多培训，才能在不破坏加速功能的前提下开发和修改代码（这往往导致维护多个代码库）。此外，基于指令的编程模型规范可能经常变化，跨厂商支持也难以预测。

近年来，使用标准语言并行为加速计算提供支持日益普及。这包括 C++并行算法 [5]，[6]，Fortran 的 do concurrent [7]，[8],以及可直接替换 Python NumPy 的实现[1]。这些新的语言特性有望消除（或大幅减少）对外部加速 API 的需求，并且具有更好的可移植性和"面向未来"的特性。未来的可移植性得以保证，因为任何支持基础语言版本的编译器都必须能够正确编译和运行代码（尽管不一定高效）。

本文专门关注 Fortran 及其用于 GPU 加速的 do concurrent（DC）结构。DC 是标准 do 循环的一种替代结构，表明该循环不存在数据依赖，因此可以乱序计算[2]。这可以作为提示，告知编译器该循环很可能（但不一定）可并行化。低层级上如何映射这种并行性由编译器决定，许多编译器会沿用已有的基于指令的映射方式（将 DC 视为折叠后的 OpenACC/OpenMP 并行循环）。

尽管使用并发循环进行加速计算已取得一些令人鼓舞的成果（见下一节），但这种方法能否扩展到规模大得多的代码中，仍是一个悬而未决的问题。为此，本文介绍我们如何将正在生产环境中运行的大型太阳磁流体动力学代码从 OpenACC 移植到并发循环。

[1]https://developer.nvidia.com/cunumeric[2]https://flang.llvm.org/docs/DoConcurrent.html

© 2023 IEEE。允许将本材料用于个人用途。任何其他用途均须获得 IEEE 的许可，包括在任何当前或未来媒体中将本材料再版/重新出版用于广告或促销目的、创建新的汇编作品、转售或重新分发至服务器或列表，或在其他作品中重复使用本作品的任何受版权保护的组成部分。

本文结构如下：第二节介绍使用 Fortran 标准并行进行加速计算的相关工作。第三节介绍 MAS 太阳磁流体动力学代码。第四节说明我们如何逐步使用并发循环减少 OpenACC 指令的数量，以及由此得到的六个用于测试的代码版本。第五节介绍测试问题、计算环境和性能结果。第六节总结研究结果，并据此评估 Fortran 标准并行在大型高性能计算代码中的当前状况和未来潜力。

II. 相关工作

将并发循环用于加速计算是一项非常新的能力。首个支持它的编译器是英伟达于2020年11月推出的高性能计算软件开发套件 [3]，而最新的是2022年的英特尔 IFX 编译器。

由于图形处理器对并发循环的支持较新，文献中仅有少数使用示例（例如名为 DIFFUSE 的球面扩散工具 [9],以及 BabelStream 基准测试的一种实现 [10]）。此外，还有一些正在进行的工作，包括一个名为 CCSD(T) 的化学迷你应用[4]，一个名为 MiniWeather 的计算流体力学天气迷你应用[5], 一个名为 CloverLeaf 的流体动力学迷你应用[6]，以及一个名为 POT3D、用于太阳物理学的共轭梯度求解器[7]。

尽管这些实现很有前景（其中许多实现的性能与其他加速方法相当），但它们都是小型到中型的代码/迷你应用/基准测试。本文探讨将这些初步研究中获得的经验应用于一个名为 MAS 的大型生产应用程序。

III. MAS 太阳磁流体动力学模型

球外磁流体动力学算法（MAS）代码是一款投入生产使用的磁流体动力学模型，已持续开发 20 多年，并广泛用于太阳物理学研究 [11]–[17]。该代码包含在日冕—日球层（CORHEL）软件套件中 [18], [19], 由美国国家航空航天局社区协同建模中心（CCMC）[8] 托管，用户可借此生成日冕和日球层的准稳态磁流体动力学解，也可模拟以日冕物质抛射形式从太阳传播至地球的太阳风暴 [20], [21]。MAS 以 Fortran 编写（≈ 70,000 行），并使用消息传递接口+和 OpenACC [22]实现并行。它能够运行包含超过三亿个网格单元的模拟 [23], [24] ，并展现出良好的扩展性，可扩展至数千个 CPU 核心或数十个图形处理器 [22], [25]。MAS 代码采用逻辑矩形的非均匀交错球面网格和有限差分离散化，并结合显式与隐式时间步进方法[9]。该代码高度受内存限制，其性能通常与硬件的内存带宽成正比。

[3] https://developer.nvidia.com/blog/accelerating-fortran-do-concurrent-with-gpus-and-the-nvidia-hpc-sdk/ [4] https://www.youtube.com/watch?v=DrvI2gw3tnI [5] https://github.com/mrnorman/miniWeather/tree/main/fortran [6] https://github.com/UoB-HPC/cloverleaf/tree/doconcurrent [7] https://www.nvidia.com/en-us/on-demand/session/gtcspring22413188 [8] https://ccmc.gsfc.nasa.gov

IV. 从 OpenACC 到 do concurrent

本文介绍了我们将 OpenACC 替换为并发循环的几种实现方案。我们的策略形成了六个不同的代码版本，始于当前的 OpenACC MAS 实现。这些版本源于多方面的考量，包括是否避免重构代码、是否遵循当前 Fortran 规范、是否使用实验性编译器功能，以及减少 OpenACC 指令与提升性能之间的权衡。在这些版本中，我们成功实现了一个不含任何 OpenACC 指令的可运行代码（代码 5）。

用于加速计算的并发循环支持尚处于起步阶段。数家供应商已宣布未来计划提供支持，但目前只有英伟达和 Intel 提供了可用的实现，其中英伟达 HPCSDK 的 nvfortran 更为成熟。由于我们当前的 MAS 图形处理器实现使用 OpenACC，且针对英伟达图形处理器进行了测试，因此本研究仅使用 nvfortran 编译器和英伟达图形处理器。

将当前 MAS 的 OpenACC 实现（代码 1）移植到并发循环的过程分为几个步骤。我们首先尽量避免修改基础代码，并遵循当前的 Fortran 2018 规范，逐步将 OpenACC 指令替换为并发循环，最终得到两个版本：一个采用优化的手动数据管理（代码 2），另一个采用自动数据管理（代码 3）。随后，我们使用编译器的 Fortran 202X 预览特性，进一步减少 OpenACC 指令的数量，并仅采用自动数据管理（代码 4）。通过使用特殊的编译器标志、替代代码启动脚本和最少的基础代码修改，我们得以实现一个不含任何 OpenACC 指令的可用版本（代码 5）。为提升性能，我们还创建了代码 5 的一个版本，并重新加入 OpenACC 手动数据管理指令（代码 6）。

表 I 总结了本节将讨论的全部六个代码版本。

A. 代码 *1 [A]*：当前 *OpenACC* 实现

我们从 MAS 当前的图形处理器生产分支开始。这是对参考文献 [22], 中开发的代码的扩展，其中添加了 OpenACC 指令，以加速 CORHEL 软件套件中使用的所有物理模型。由于没有使用任何 OpenACC API 调用，该代码可完全移植到 CPU 编译器。代码共有 73,842 行

[9] https://www.predsci.com/mas

表 I 所开发和测试的所有 MAS 代码版本汇总。

Code 版本	代码描述和 nvfortran GPU 编译器标志	总计行数	$acc 行
0：CPU	原始纯 CPU 版本	69874	∅
1：A	原始 OpenACC 实现 -acc=图形处理器 -gpu=cc80	73865	1458
2：AD	针对与 DC 不兼容的循环使用 OpenACC 以及数据管理，其余循环使用 DC -acc=gpu -stdpar=gpu -gpu=cc80,nomanaged	71661	540
3：ADU	OpenACC 用于与并发循环不兼容的循环，其余循环使用并发循环，统一内存 -acc=gpu -stdpar=gpu -gpu=cc80,managed	71269	162
4：AD2XU	OpenACC 用于 for 功能，剩余循环使用 DC2X，统一内存 -acc=gpu -stdpar=gpu -gpu=cc80，统一内存	70868	55
5：D2XU	所有循环均使用 DC2X，对部分代码进行修改，统一内存 -stdpar=图形处理器 -gpu=cc80 -Minline=reshape,name:s2c,boost,interp,c2s,sv2cv	68994	∅
6：D2XAd	所有循环的 DC2X，一些代码修改，用于数据管理的 OpenACC -acc=图形处理器 -stdpar=图形处理器 -gpu=cc80,nomanaged -Minline=reshape,name:s2c,boost,interp,c2s,sv2cv	71623	277

其中包括 1,458 条 !$acc 指令注释。OpenACC 指令类型的分布汇总见表 II。

表 II MAS 原始 GPU 分支中的 OpenACC 指令（代码 1）。

OpenACC 指令类型	行数
并行，循环	997
数据管理： 进入，退出，更新， 主机_数据，声明	320
原子操作	34
例程	12
内核	6
wait	6
设置设备 _num	1
续行（!$acc&） （分布于所有指令类型中）	82
总计	1458

B. 代码 *2 [AD]*：对与并发循环不兼容的循环和数据管理使用 *OpenACC*，其余循环使用并发循环）

在这次首次尝试中，我们将 OpenACC 替换为并发循环，力求在尽可能少改动代码（除了将 do 循环转换为并发循环）的前提下，尽可能保持性能和可移植性。

MAS 中使用的典型嵌套 do 循环及其 OpenACC 指令如清单 1 所示。使用并发循环的相同循环如清单 2 所示。除了省去 OpenACC 指令，并发循环还更加紧凑，从而减少了代码总行数。

```
!$acc parallel default(present)!
```

清单 1。 MAS 代码中使用 OpenACC 的标准 do 循环

并发循环 (k=1,n3,j=1:n2,i=1:n1) 使用数组

清单 2。使用并发循环实现的清单 1 的等效循环

为确保修改后的代码在使用目前能够运行该代码的任意编译器厂商/版本时，仍可用于仅在 CPU 上运行的情况，我们遵循 Fortran 2018 标准。因此，我们不使用即将推出的 Fortran 202X 中用于 DC 循环的 reduce 子句，而是让所有归约循环继续使用 OpenACC。此外，由于一些主流编译器尚不支持诸如 private 和 shared 这样的 DC 亲和性子句（而且有些编译器能够很好地自动检测正确的亲和性），我们不使用这些子句。如清单 3 所示，当前 MAS 代码中的数组归约使用 OpenACC 原子指令，以实现完全并行化。

```
!$acc parallel
```

清单 3. MAS 中数组归约循环示例

为了对循环使用 DC 并符合规范，我们需要重写循环（将其改为外部并行 DC 循环，内部为顺序循环）。鉴于此时我们希望尽可能避免更改代码，而且这种更改可能会影响性能，因此数组归约循环也继续使用 OpenACC 循环。

我们的一些并行循环中包含对函数/例程的调用。在 OpenACC 中，必须使用 routine 指令在指定的

并行级别上声明这些函数/例程。Fortran 规范要求，在 DC 循环中调用的函数/例程必须是“纯”的。我们循环中的函数/例程都是纯函数，但目前英伟达编译器尚不支持在 DC GPU 卸载循环中使用它们，除非保留 OpenACC routine 指令。

我们遇到的另一个问题是使用 OpenACC 的 kernels 构造。kernels 不仅可用于常规 do 循环，也可用于包含 Fortran 数组语法操作和 MINVAL 等 Fortran 内置函数。在这些情况下，若要使用并发循环，就必须将这些操作展开为显式循环。在更新后的 MAS OpenACC 移植版本中，我们已在大多数情况下完成了这项工作。在这个并发循环版本中，我们希望避免额外修改代码，因此目前保留剩余的少数 kernels 区域。

OpenACC 中有两项并发循环不具备的功能，可能会导致性能损失。第一项是“内核融合”。在 OpenACC 中，可以在一个并行区域内包含多个彼此独立的数据循环，编译器随后可将它们编译为单个 GPU 计算内核。将这些循环转换为并发循环时，编译器必须为每个循环分别创建 GPU 内核。由于内核启动存在开销，这可能会降低性能（尤其是频繁调用的小型内核）。第二项功能是异步内核启动。在 OpenACC 中，可以使用 async 子句启动内核，同时让代码继续执行（此时可在 CPU 上计算，和/或启动其他异步内核）。目前，并发循环没有办法告知编译器使用异步内核启动。因此，如果代码大量依赖 async，使用并发循环可能会降低性能。对于 MAS 代码，我们并未大量使用 async，因此预计这不会造成太大问题。

根据我们先前将 OpenACC 移植到并发循环的经验 [7]，，我们了解到，采用手动数据管理比依赖统一托管内存（UM）功能能获得更好的性能。统一托管内存是英伟达的一项功能，可在 GPU 加速代码中[10]自动在图形处理器和 CPU 之间传输数据。这减轻了程序员的负担，因为他们无需手动管理数据传输。不过，使用 UM 也有一些缺点，例如可能导致性能下降，以及占用过多图形处理器内存。使用 UM 带来的性能下降可能很小，但与 CUDA 感知的 MPI 调用（如 MAS 中的用法）结合使用时，性能下降可能会更明显。此外，UM 并未纳入 Fortran 或 OpenACC 规范。因此，在此版本（AD）中，我们不使用 UM，并保留 OpenACC 数据管理指令。

最终，对于代码的这一版本（AD），我们将 OpenACC 指令的数量从 1458 条减少到 540 条，减少了近三分之二。

C. 代码 *3 [ADU]*：与并发循环不兼容的循环使用 *OpenACC*，其余循环使用并发循环，统一内存

对于此版本（ADU）的代码，我们通过使用统一托管内存，大幅减少了 OpenACC 指令的数量。如上所述，统一内存并不属于 Fortran 或 OpenACC 规范；如果不手动进行数据移动，每个 OpenACC 和并发循环内核都会默认执行 GPU 与 CPU 之间的数据迁移，从而导致性能大幅下降。因此，多个厂商都创建了统一内存管理系统（英伟达采用 UM，英特尔 OneAPI 采用统一共享内存[11], , AMD 采用智能访问内存[12]）。由于统一内存管理系统已得到广泛应用，我们认为，在此版本（ADU）中依赖 UM 所带来的可移植性降低是可以接受的。

在移除 OpenACC 数据指令时，我们发现无法将其全部移除。具体而言，对于某个在 GPU 内核区域内调用的函数所使用的数据元素，仍然需要一个声明（以及后续的更新）指令。此外，对于派生类型，我们还必须保留一些进入和退出数据传输。这是因为我们仍在使用 OpenACC 处理归约循环，而这些循环使用了 default(present) 子句。该子句告诉编译器所有数据都已位于设备上，并可避免编程错误导致的性能问题，因为对于任何不在设备上的数据，程序都会失败。然而，在使用统一内存时，尽管派生类型中的数组会分页到图形处理器，派生类型结构本身却不会，因为它属于静态数据（不是统一内存分页所要求的可分配数组或自动数组）。因此，既然我们希望继续使用 default(present)，就必须手动将这些结构放置到图形处理器上。

此版本（ADU）将代码中的 OpenACC 指令数量进一步减少了三倍以上，从 540 条降至 162 条。由于此代码唯一的改动是移除了数据指令，因此其性能与启用统一内存（UM）运行代码 2 (AD) 时相当（此时所有数据指令都会被忽略）。因此，此版本（ADU）可用于测试单独启用 UM 对性能的影响。

D. 代码 *4 [AD2XU]*：用 *OpenACC* 实现功能，所有循环使用 *DC2X*，统一内存

为了了解还能在多大程度上减少 OpenACC 指令的数量，在此版本（AD2XU）的代码中，我们允许自己利用英伟达对 DC 循环中 reduce 子句的 Fortran 202X 预览实现。使用 reduce 会降低此版本（AD2XU）的可移植性，使其目前只能使用 nvfortran 编译器（即使在 CPU 上也是如此）。不过，预计该子句将在 Fortran 标准中

[10]https://www.pgroup.com/blogs/posts/openacc-unified-memory.htm

[11]https://www.intel.com/content/www/us/en/developer/articles/code-sample/dpcpp-usm-code-sample.html [12] https://www.amd.com/en/technologies/smart-access-memory

一年内成为标准，因此最终它将具有可移植性（只要所有主流编译器都能识别它）。

使用 reduce 可以将所有标量 OpenACC 归约循环转换为并发循环，但对于数组归约，nvfortran 尚不支持直接使用 reduce。不过，由于底层的编译器机制，我们发现可以在并发循环中使用 OpenACC 原子指令，方式与之前在 OpenACC 循环中使用它们相同。因此，我们也将所有数组归约 OpenACC 循环转换为并发循环（同时保留其中的原子指令），如清单 4 所示。

```
并发循环(j=1:n2,i=1:n1)
!$acc 原子操作 更新
   sum0(i)=sum0(i)+array(i,j)*...
循环结束
```

清单 4. 清单所示数组归约的并发循环版本 3

我们还成功将少数使用原子指令的非归约循环转换为并发循环。经过这项修改，所有使用派生类型的循环现在都是 DC 循环，因此剩余的少数数据指令也得以移除。

这个代码版本（AD2XU）进一步将 OpenACC 指令的数量减少了三倍以上，从 162 个降至 55 个。剩余的 OpenACC 指令包括原子操作、声明、更新、设置设备_数量、例程和内核。

E. 代码 *5 [D2XU]*：所有循环均采用 *DC2X*，进行了一些代码修改，统一内存

在此，我们允许自己使用 nvfortran 专用编译器标志，并对基础代码做一些修改。这是为了了解目前要实现“理想”的代码（即消除 OpenACC 指令）需要做些什么。

首先，我们通过将所用的 Fortran 内在函数展开为显式的 DC 归约循环，消除了剩余的少数内核指令。需要指出的是，对于大量使用内核的代码，这一步可能相当复杂。

接下来，我们通过调换循环顺序，并在外层循环中使用标准的并发循环、在内层循环中使用并发循环归约，重写了所有数组归约循环，如清单 5 所示。

```
do concurrent (i=1:n1)
   tmp=0.
   do concurrent (j=1:n2) reduce(+:tmp)
      tmp=tmp+array(i,j)*...
   enddo
   sum0(i)=tmp
enddo
```

清单 5。对清单 4 中数组归约的修改，从而可以移除原子操作指令

这使得这些循环不再需要 OpenACC 原子指令。我们也可以将内层循环替换为串行 do 循环或 Fortran 内置函数（例如 SUM）。事实上，nvfortran 编译器的输出表明，它会将内层并发循环归约串行化（因为这样做可能比启动大量微型内核更快）。我们还在少数几个地方对代码做了小幅修改，以便移除最后几条原子指令。

剩余的一组 OpenACC 指令是例程指令，用于声明在 GPU 内核循环中调用的函数和例程。由于 MAS 中的所有此类函数都是“纯”函数，因此它们应当能够在并发循环中得到支持，而 nvfortran 编译器的进一步开发也应当能够消除对例程指令的需求。在此期间，我们使用编译器标志 -Minline,name 来指定这些例程需要内联。对于其中一个例程，我们还需要指定 reshape 内联选项。编译器拒绝内联其中一个例程，迫使我们手动将其内联。幸运的是，这只有一个例程，而且调用次数并不多；但对于其他代码而言，这一步可能难以承受。这一步还使我们能够移除内联函数中所使用数据元素的 declare 指令。

最后剩下的 OpenACC 指令是 set device num 指令，用于在多 GPU 运行中根据本地 MPI 秩选择图形处理器设备。为了移除此指令，我们修改了代码的启动方式。我们不再直接启动代码（例如 mpirun ... ./mas ...），而是启动一个 Bash 脚本（如清单 6 所示），该脚本使用 MPI 运行时环境变量来设置 NVIDIA 环境变量，使 MPI 进程只能看到正确的图形处理器设备。运行 mpirun ... ./launch.sh ./mas ... 即可调用该脚本。

```
#!/bin/bash
# 假设每个 MPI 本地秩使用 1 个图形处理器
```

清单 6. 用于启动代码 5（D2XU）和 6（D2XAd）的 Bash 脚本（launch.sh）。

虽然此示例专用于 OpenMPI 库（该库随 NV HPC SDK 一同提供），但其他消息传递接口库中也存在类似的环境变量。

完成上一步后，我们实现了目标：获得一个不含任何 零 OpenACC 指令、且可在多个 GPU 上运行的代码版本。此外，通过移除 OpenACC 实现中所需的一系列重复例程，我们还可以进一步缩减代码规模。MAS 中有许多例程既会在设置阶段调用，也会在代码的计算部分调用。由于设置阶段包含大量

代码，但仅占用可忽略的运行时间，因此我们不希望将所有设置代码都移植到 GPU 上。这带来了一个问题，因为已移植的例程假定数据位于 GPU 上。现在我们使用统一托管内存，可以移除这些例程的仅 CPU 版本，因为在短暂设置阶段产生的 GPU-CPU 分页开销不应显著影响代码的整体性能。

F. 代码 *6 [D2XAd]*：所有循环均使用 *DC2X*，进行部分代码修改，并使用 *OpenACC* 进行数据管理

正如我们将在下一节看到的那样，使用统一内存会对 MAS 代码的性能产生非常不利的影响。因此，我们在代码列表中再加入一个版本。在这个版本（D2XAd）中，我们的目标与代码 5（D2XU）类似：尽可能减少 OpenACC 指令的数量（通过使用实验性功能、修改代码等方式）；但在这里，我们还要求性能达到代码 1 (A) 和代码 2 (AD) 的水平。为此，我们以代码 5（D2XU）为基础，重新加入所有 OpenACC 手动数据移动指令，使代码无需使用统一内存即可运行（这也要求我们重新加入上一节提到的所有重复的、仅供 CPU 使用的例程）。随后，我们修改代码，使用包装例程在图形处理器上创建并初始化数组，从而减少所需的数据移动指令数量。最终得到的代码包含 277 条 OpenACC 指令，比最初的代码 1 (A) 少 5 倍以上，比代码 2 (AD) 少近 50%，同时仍保持相近的性能（见下一节）。

V. 性能测试

A. 测试案例

为了测试我们各项实现的性能，我们使用了参考文献 [26] 中的一个生产级准稳态日冕背景模拟，该模拟采用完整的热力学磁流体动力学物理模型。该问题设置为 3,600 万个网格单元，并运行其 48 小时物理模拟时间中的前 24 分钟。所得解的可视化结果如图 1 所示。所选分辨率代表了一个中等规模的案例，也可以装入单个 NVIDIA A100（40GB）图形处理器的内存中。对于所有测试运行，我们都将所得解与原始代码的结果进行了验证，误差控制在求解器容差以内。

B. 硬件和软件环境

我们在 NCSA 的 Delta[13] 超级计算机上使用一台配备 8 个图形处理器的节点测试各个代码版本。该节点为双路插槽配置，搭载两颗 AMD EPYC Milan 7763 CPU，共配备八块通过 NVlink 互连的 NVIDIA A100（40GB）图形处理器，每块图形处理器的理论峰值内存带宽为 1,555 GB/s。我们使用 NVIDIA HPC SDK 的 nvfortran 编译器 22.11 版本和 OpenMPI 库 3.1.3 版本进行编译（我们没有使用搭配 UCX 的 OpenMPI v4，因为其性能和兼容性不及 OpenMPI v3）。在基准 CPU 测试中，我们使用圣迭戈超级计算机中心 Expanse[14]超级计算机上的双路插槽 AMD EPYC 7742 CPU 节点，每个节点的理论最大内存带宽为 381.4 GiB/s （ 409.5 GB/s ）。我们使用 GCC 的 gfortran 编译器 10.2.0 版本和 OpenMPIMPI 库 4.0.4 版本。

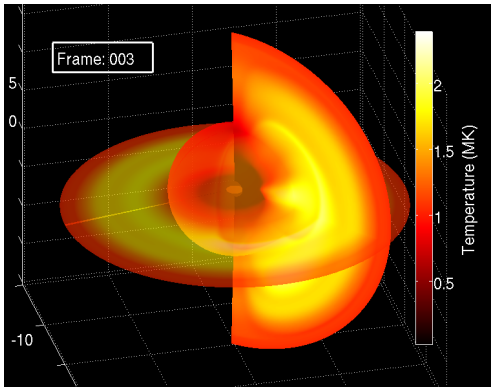


图 1。测试案例的 MAS 解的可视化结果。显示并绘制了最后一个时间步的温度切片，所用网格经过拉伸。

C. 性能结果

这里我们测试第 IV 节所述六个代码版本中每个版本的性能。首先，我们在 Expanse 的双路插槽 AMD EPYC CPU 节点上运行代码 1 (A) 和代码 2 (AD)。除了提供基准性能外，这还检验了使用并发循环不会对可移植性和性能产生负面影响（至少对于符合规范的版本而言）。结果如表 III 所示，表明该代码的并发循环版本在 CPU 上的运行性能与原始代码相当。

表 III 在双路插槽 AMD EPYC 7742 CPU 节点上运行测试问题的墙钟时间（分钟）。

节点数	代码 1 (A)	代码 2 (AD)
1	725.54	725.53
8	79.58	79.64

图 2 展示了代码的六个版本在 Delta GPU 节点上使用 1 至 8 块 NVIDIA A100（40GB）图形处理器运行时的计时结果。可以看到，代码 1 (A)、代码 2 (AD) 和代码 6 (D2XAd) 起初呈现“超线性”扩展，随后扩展性降至理想水平以下。不过，由于初期的超线性扩展，这些代码在使用 8 块图形处理器时的扩展性都优于或接近理想水平。我们还看到，使用并发循环的代码 2 (AD) 和代码 6 (D2XAd) 都

[13]https://delta.ncsa.illinois.edu

[14]https://www.sdsc.edu/services/hpc/expanse

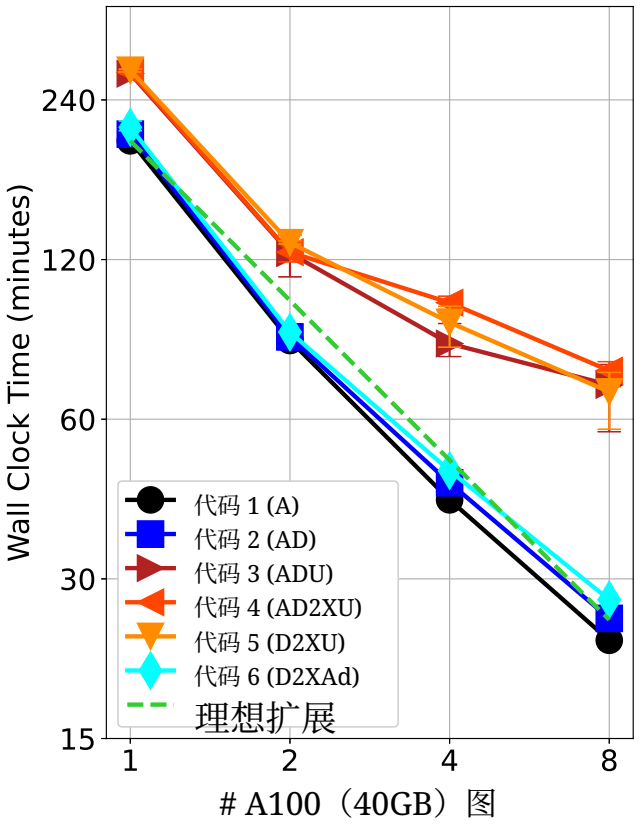


图 2. 在一个 8 块 A100（40GB）GPU 节点上运行测试问题的墙钟时间。每项结果显示三次运行的平均值，误差线表示运行时间的最小值和最大值。

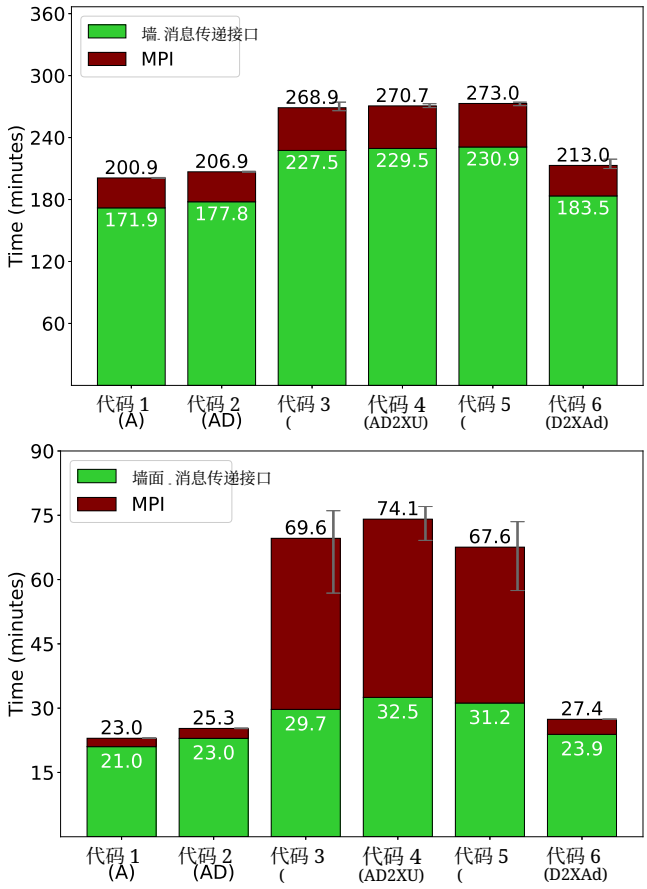


图 3. 在 1 块（上）和 8 块（下）A100（40GB）图形处理器上运行测试问题的时间。每项结果显示三次运行的平均值，误差线表示墙钟时间的最小值和最大值。消息传递接口时间（包括所有消息传递接口调用、缓冲区初始化/加载/卸载，以及负载不均衡导致的消息传递接口等待）以栗色显示，其余墙钟时间以绿色显示。

略慢于代码 1 (A)。可能的原因包括内核融合、异步内核丢失，以及 OpenACC 内核与并发循环内核采用了不同的编译器卸载参数。代码 6 (D2XAd) 也比代码 2 (AD) 稍慢，这可能是因为包装例程中增加了数组初始化内核，而原始代码在相应位置并未将数组初始化为零。代码 3 (ADU)、代码 4 (AD2XU) 和代码 5 (D2XU) 的性能和扩展性都大幅下降。图 3 展示了使用 1 块和 8 块图形处理器时的结果，并突出显示了消息传递接口开销时间。使用统一内存的代码，其消息传递接口时间显著增加，非消息传递接口时间也有所增加（但增幅小得多）。所有性能较差的代码计时结果相近，并且都使用统一内存。由于代码 3 (ADU) 等同于在启用统一内存的情况下运行代码 2 (AD)，这些结果表明，性能下降是由统一内存导致的，而非并发循环。我们通过在启用统一内存的情况下运行代码 1 (A) 和代码 2 (AD)，并获得与代码 3 (ADU) 相近的计时结果，证实了这一点。为探究统一内存导致性能大幅下降的原因，我们使用 NVIDIA NSIGHT Systems 性能分析器，对代码 1 (A) 在采用手动内存管理和统一内存两种情况下运行 8 块图形处理器的任务进行了分析。结果如图 4 所示。我们

可以看到，手动管理内存支持在消息传递接口的晕区交换中进行 GPU 对等数据传输，而统一内存会执行多次 CPU-GPU 传输，导致速度变慢。统一内存运行的开销也更大，这体现在内核启动之间的间隔更长。总体而言，手动内存管理运行几乎能在统一内存运行完成一次迭代所需的时间内完成三次完整迭代。英伟达的开发人员已经意识到这一问题，并正在设法解决[15]。

VI. 总结与展望

在我们测试的六个代码版本中，我们重点介绍三个。首先，代码 1 (A，即我们最初的 OpenACC 代码) 性能最佳。这得益于手动内存管理、OpenACC 支持异步内核启动以及内核融合。我们重点介绍的第二个代码

[15]Jeff Larkin，NVIDIA 公司个人通信

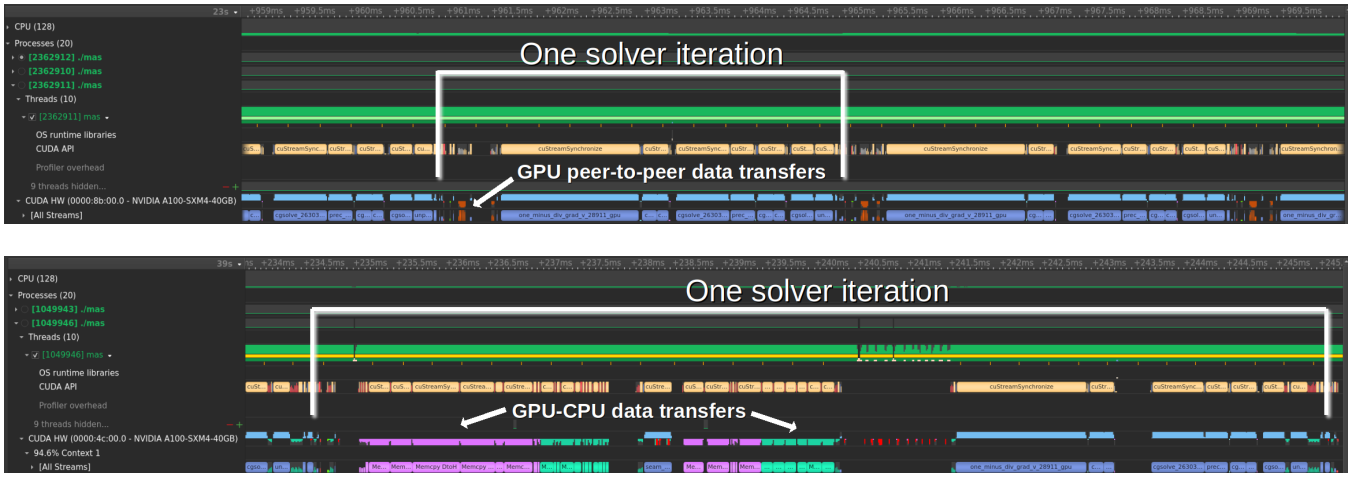


图 4. 使用 OpenACC 手动内存管理（上）和统一托管内存（下）时，MAS 中代码 1(A) 在 8 个 A100 图形处理器上运行粘度求解器迭代的 NVIDIA NSIGHT Systems 时间剖面。可以看到，手动管理内存时，MPI 光环交换过程中会进行图形处理器之间的点对点数据传输；而统一内存则会进行多次 CPU-GPU 传输，导致运行速度变慢。再加上额外开销，使用统一内存管理计算一次求解器迭代的速度比使用手动内存管理慢三倍。

是代码 5 (D2XU)。这段代码展现了标准语言的潜力：无需使用任何 OpenACC 指令，就能在多个 GPU 上运行。尽管统一内存导致了性能下降，但我们认为这只是暂时的挫折，随着编译器、消息传递接口库和系统集成的进一步改进，这一问题将得到解决。代码 5 (D2XU) 还要求我们手动内联一些函数、使用内联编译器标志，并采用另一种通过消息传递接口启动代码的方法。我们预计，未来的编译器版本也能免去其中一些要求。最后，我们重点介绍代码 2 (AD)。它符合当前的 Fortran 规范，性能几乎与代码 1 (A) 一样好，仍可通过所有主流 CPU 编译器进行编译，并将 OpenACC 指令的数量从 1458 大幅减少到 540。这表明，只要手动进行 GPU-CPU 数据移动，并发循环的性能就可以与 OpenACC 相媲美。加入代码 5 (D2XU) 和代码 6 (D2XAd) 的部分修改（以及其他修改），还能进一步减少 OpenACC 指令的数量；经过此类修改的版本已成为我们的新生产用 GPU 版本。

随着进一步开发和跨厂商支持，我们希望最终能拥有一个统一的代码库，无需任何指令即可在多个厂商的加速器硬件上运行。这对于维护高性能、可移植的加速代码将极具价值，同时也能让领域科学家使用标准 Fortran 开发代码。尽管指令比低级 API 更容易上手，但领域科学家仍需学习不熟悉的语法（以及如何正确使用它），从而影响工作效率。采用标准语言方案后，他们就可以专注于自己的实现，并继续使用自己几十年来熟悉的语言（只需做少量调整）。

与此同时，对于我们的代码，我们预计未来数年仍需要一定程度的 OpenACC 支持。我们强烈鼓励供应商继续维护和／或增加此类支持（因为在图形处理器应用中，它经常被用来替代 OpenMP），直到标准语言能够实现全面的跨供应商、跨平台使用。

致谢

本研究得到美国国家科学基金会资助，项目编号为 AGS 2028154 和 ICER1854790；并获得美国国家航空航天局资助，项目编号为 80NSSC20K1582 和 80NSSC22K0893。本研究还通过 ACCESS 分配项目 TG-MCA03S014 使用了 NCSA 的 Delta 系统和圣迭戈超级计算机中心的 Expanse 系统。我们感谢 NVIDIA 公司的 Jeff Larkin 提供的宝贵见解，也感谢匿名审稿人的建议。

参考文献

[1] T. Deakin、J. Price、M. Martineau 和 S. McIntosh-Smith，《通过 BabelStream 评估并行编程模型可实现的内存带宽》，*Int.J. Comput.Sci.Eng.*，第 17 卷，第 247–262 页，2018年。[2] E. C. Davis、C. R. Olschanowsky 和 B. Van Straalen，《采用多面体数据流表示的结构化网格求解器》，载于并行计算语言与编译器国际研讨会。施普林格，2021年，第 127–146 页。[3] S. Chandrasekaran 和 G. Juckeland，《面向程序员的 *OpenACC*：概念与策略》。Addison-Wesley Professio nal，2017年。

[4] R. Van der Pas、E. Stotzer 和 C. Terboven，《使用 *OpenMP*：下一步——亲和性、加速器、任务处理和单指令多数据》。麻省理工学院出版社，2017年。[5]B. A. L. David Olsen、Graham Lopez。（2021年）使用 stdpar 通过图形处理器加速标准 C。 [在线]。可访问：https://developer.nvidia.com/blog/accelerating-standard-c-with-gpus-using-stdpar[6]J. Latt 和 C. Coreixas，《面向图形处理器编程的 C 并行算法：格子玻尔兹曼方法案例研究》，2021年。 [在线]。可访问：http://rgdoi.net/10.13140/RG.2.2.20168.24328[7]M. Stulajter、R. M. Caplan、J. Larkin 和 J. Linker。（2022年）使用 Fortran 标准并行编程实现 GPU加速。[在线]。可访问：https://developer.nvidia.com/blog/using-fortran-standard-parallel-programming-for-gpu-accelera

[8]I. Corporation.（2022）《英特尔Fortran编译器的新篇章》。[在线]。网址：
h
tt
ps://community.intel.com/t5/Blogs/Tech-Innovation/Tools/The-Next-Chapter-for-the-Intel-Fortran-Compiler/post/1439297[9]M. M. Stulajter、R. M. Caplan 和 J. A. Linker，“Fortran 的 ‘do concurrent’ 能否取代用于加速计算的指令？”载于使用指令进行加速器编程。施普林格国际出版公司，2022年，第3–21页。 [在线]。网址：https://doi.org/10.1007/978-3-030-97759-7 1[10] J. R. Hammond、T. Deakin、J. Cownie 和 S. McIntosh-Smith，“使用 BabelStream 对 CPU 和 GPU 上的 Fortran DO CONCURRENT 进行基准测试”，载于第*13*届 *IEEE* 高性能计算系统性能建模、基准测试与模拟国际研讨会论文集。美国计算机协会，2022年。[11] Z. Miki´c、J. A. Linker、D. D. Schnack、R. Lionello 和 A. Tarditi，“全球太阳日冕的磁流体动力学建模”，等离子体物理学，第6卷，第5期，第2217页，1999年。[12] J. A. Linker、Z. Miki´c、R. Lionello、P. Riley、T. Amari 和 D. Odstrcil，“磁通消失与日冕物质抛射”，等离子体物理学，第10卷，第5期，第1971–1978页，2003年5月。[13] R. Lionello、J. A. Linker、Z. Miki´c 和 P. Riley，“日冕磁力线对较差自转的响应及其纬向偏移：磁流体动力学模拟”，天体物理学杂志快报，第642卷，第1期，第L69页，2006年。[14]R. Lionello、J. A. Linker 和 Z. Miki´c，“首次全太阳月期间太阳的多光谱发射：磁流体动力学模拟”，《天体物理学杂志》，第690卷，第1期，第902–912页，2009年1月。[15] J. A. Linker、R. Lionello、Z. Miki´c、V. S. Titov 和 S. K. Antiochos，“由光球动力学驱动的开放磁通量演化”，《天体物理学杂志》，第731卷，第2期，第110页，2011年4月。[16] C. Downs、J. A. Linker、Z. Miki´c、P. Riley、C. J. Schrijver 和 P. Saint-Hilaire，“利用掠日彗星探测太阳磁场”，科学，第340卷，第6137期，第1196–1199页，2013年6月。

[17] Z. Miki´c、C. Downs、J. A. Linker、R. M. Caplan、D. H. Mackay、L. A. Upton、P. Riley、R. Lionello、T. T¨or¨ok、V. S. Titov 等，“预测2017年8月21日日全食期间的日冕”，《自然·天文学》，第1页，2018年。[18] P. Riley、J. A. Linker、R. Lionello 和 Z. Mikic，“最近一次太阳活动极小期的共转相互作用区：全球磁流体动力学建模的优势与局限”，《大气与太阳-地球物理学杂志》，第83卷，第1–10页，2012年7月。[19]X. Feng，《空间天气磁流体动力学模拟的现状》。新加坡：Springer Singapore，2020年，第1–123页。 [在线]。网址：https://doi.org/10.1007/ 978-981-13-9081-4 1[20] R. Lionello、C. Downs、J. A. Linker、T. T¨or¨ok、P. Riley 和 Z. Miki´c，“行星际日冕物质抛射的磁流体动力学模拟”，《天体物理学杂志》，第777卷，第76页，2013年11月。[21] T. T¨or¨ok、C. Downs、J. A. Linker、R. Lionello、V. S. Titov、Z. Miki´c、P. Riley、R. M. Caplan 和 J. Wijaya，“2000年7月14日‘巴士底日’爆发的太阳到地球磁流体动力学模拟”，《天体物理学杂志》，第856卷，第1期，第75页，2018年。[22] R. M. Caplan、J. A. Linker、Z. Miki´c、C. Downs、T. T¨or¨ok 和 V. S. Titov，“使用 OpenACC 对成熟的太阳磁流体动力学代码进行 GPU 加速”，《物理学杂志：会议系列》，第1225卷，第012012页，2019年5月。[23] C. Downs、J. A. Linker、V. S. Titov、R. M. Caplan、P. Riley 和 R. Lionello，“在全球尺度上关联日冕加热与日冕结构及复杂性”，第三届三年一度地球-太阳峰会（TESS），第54卷，第7期，2022年。[24] C. Downs、J. A. Linker、R. M. Caplan 和 V. S. Titov，“关联表面复杂性与日冕及太阳风结构”，准备中，2023年。[25] R. M. Caplan、Z. Miki´c、J. A. Linker 和 R. Lionello，“热力学磁流体动力学模型中的抛物型算子推进：显式超时间步进与采用 Krylov求解器的隐式格式之比较”，《物理学杂志：会议系列》，第837卷，第012016页，2017年5月。[26]K. K. Reeves、T. T¨or¨ok、Z. Miki´c、J. Linker 和 N. A. Murphy，“探索三维日冕物质抛射模拟中电流片区域的等离子体加热”，《天体物理学杂志》，第887卷，第1期，第103页，2019年12月。 [在线]。网址：https://doi.org/10.3847/ 1538-4357/ab4ce8