arXiv:2512.22215v1 [cs.DC] 22 Dec 2025

SPUMA：一种用于 OpenFOAM® GPU 移植的最小侵入式方法

Simone Bnà[a, *]，Giuseppe Giaquinto[a, b]，Ettore Fadiga[a]，Tommaso Zanelli[a]，Francesco Bottau[a]

[a]*Cineca超级计算中心，Via Magnanelli 6/3，40033* 卡萨莱基奥迪雷诺，意大利[b]那不勒斯费德里科二世大学工业工程系，*80125* 那不勒斯，意大利

摘要

混合型集群上的高性能计算（HPC）为计算流体力学（CFD）带来了重大机遇，尤其是在现代加速器得到有效利用时。然而，尽管 GPU 已得到广泛应用，可编程性仍然是一项挑战，特别是在开源环境中。本文介绍 SPUMA[1], ，一种面向英伟达和 AMD GPU 的 OPENFOAM®[2] 完整 GPU 移植方案。该实现策略基于可移植编程模型，并采用内存池管理器，充分利用现代 GPU 的统一内存功能。本文还介绍了在欧洲的两个前百亿亿次级集群 LUMI 和 Leonardo 上开展的多项数值测试；这两个集群分别搭载 AMD MI250X 和 NVIDIA A100 GPU。在性能分析部分，我们展示了内存使用情况分析和内核运行时间的相关结果、内存池的影响，以及通过模拟著名的 DrivAer工业测试案例得到的能耗数据。GPU利用率对强可扩展性结果影响显著；当每个 GPU 的负载接近 800 万网格单元时，LUMI 和 Leonardo 上的效率均达到 65%。

*通讯作者。电子邮箱： simone.bna@cineca.it[1]SPUMA 是 Cineca 发布的一款基于 OpenFOAM技术的计算流体力学软件。该产品未经 OpenCFD Limited 批准或认可。OpenCFD Limited 是通过 www.openfoam.com 提供 OpenFOAM 软件的生产商和分销商，也是 OPENFOAM® 和 OpenCFD® 商标的所有者。[2]OPENFOAM® 是 OpenCFD Limited 的注册商标。OpenCFD Limited 是通过 www.openfoam.com 提供 OpenFOAM 软件的生产商和分销商。

在每个 GPU 处理 1000 万网格单元的负载下，20 个 GPU 上使用 OpenFOAM 原生多重网格求解器得到的弱扩展性结果，从 Leonardo 上的 75% 到 LUMI 上的 85% 不等。值得注意的是，切换至 NVIDIA AmgX 线性代数求解器后，效率不低于 90%。测试还表明，在 GPU 超额配置足够充分（每个 GPU 处理超过 1000 万个网格单元）的情况下，Leonardo 上的一块 A100 GPU 相当于 200 至 300 个 Intel Sapphire Rapids 内核。最后，与在 CPU 上执行的类似模拟相比，能耗最多可降低 82%。

---

1. 引言

过去几十年来，计算流体动力学（CFD）日益广泛地应用于各类工程问题，成为工业界和研究领域的重要工具。如今，商用和开源计算流体动力学软件在汽车工业、航空航天、涡轮机械、风能和空气污染物扩散等行业中都发挥着重要作用。

工业应用通常涉及复杂几何结构和湍流，还需要准确解析多个时间和空间尺度。因此，这些模拟往往需要消耗大量计算资源，使计算流体动力学成为高性能计算（HPC）领域的重要组成部分，也为参与百亿亿次计算发展的科学界带来了严峻挑战。

OpenFOAM [1]是一个开源的有限体积（FV）计算流体动力学框架，拥有庞大的国际用户群体，涵盖学术机构和私营企业。OpenFOAM 于20世纪90年代设计，其软件架构从根本上面向基于中央处理器的硬件。这使其无法充分发挥当前混合型超级计算机的强大计算能力。

在高性能计算领域，20世纪90年代，大量通用中央处理器内核构成的大规模并行架构取代了早期的向量机，成为主流。图形处理器（GPU）最初作为专用硬件开发，用于加速图形相关任务。在过去二十年里，由于处理海量数据的效率很高，GPU 在加速科学计算应用方面发挥着日益重要的作用。通用 GPU 计算推动了当前机器学习和人工智能应用的蓬勃发展

成为可能。尤其是后者，目前已成为推动 GPU 技术进一步发展的主要动力。因此，Top 500 榜单上的大多数超级计算机都配备了 GPU [2], ，云计算服务提供商也在逐步增加 GPU 加速计算服务。

过去十年间，许多研究人员致力于将 OpenFOAM 代码移植到图形处理器上。然而，大多数已发表的研究采用了混合方法，即仅将纳维-斯托克斯方程求解过程中的线性代数部分移植到图形处理器上 [3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16]。只有少数尝试 [17, 18, 19, 20]着眼于对整个算子分裂压强隐式法（PISO）和压力关联方程半隐式法（SIMPLE）求解器进行全面的图形处理器加速 [21], ；这两种求解器是 OpenFOAM 中用于解耦压力和速度的主要算法。

除 Bnà 等人近期发表的研究外 [3], ，官方 OpenFOAM 代码库中尚未纳入其他图形处理器开发工作，因此出现了大量基于旧版 OpenFOAM 的过时项目，如今可视为已被淘汰。一个例外是 Tandon 等人颇具前景的研究[20], ，该研究展现了主要图形处理器制造商重新投入资源、致力于全面移植 OpenFOAM 的意愿。

混合方法易于实现，但从求解成本的角度来看却很难证明其合理性：OpenFOAM 采用分离式方法求解纳维-斯托克斯方程，这意味着每个与偏微分方程相关的线性方程组都必须先复制、转换，最后再由 GPU 求解。矩阵组装不包含在卸载过程中，耗时可能高达整个模拟时间的 50%。在这种情况下，根据阿姆达尔定律，加速比不会超过 2。另一方面，阿姆达尔定律并未限制整个求解器的加速幅度。然而，像 OpenFOAM 这样的生产级代码由数百万行代码组成，其复杂性使软件集成成为一项艰巨的任务。

文献中少数几种针对整个 OpenFOAM 工作流所提出的方法并不完全令人满意：Miroslaw 等人 [17, 18] 使用 CUDA 将 OpenFOAM 完全移植到了 GPU 上，而 RapidCFD [19] 依赖于 Thrust 库——这是一个基于标准模板库的 CUDA/C++框架。唯一能够移植到多种架构上的方法基于 OpenMP 目标卸载 [20]。然而，该方法是为 AMD MI300A 等加速处理器（APU）设计的，这类处理器利用由 CPU 和 GPU 共享的硬件统一内存空间。

如 [20], 所示，独立式 GPU 上的性能会因页面迁移开销而严重下降：超过 65 % 的时间都花在更新 GPU 内存以及在主机与设备之间复制数据上。独立式 GPU 仍然存在于市场中，也存在于 TOP500 榜单上的超级计算机中（例如 Leonardo、Mare Nostrum 5 和 LUMI），而高性能计算未来的硬件发展趋势并不容易预测。

在这项工作中，我们介绍了 SPUMA（统一内存加速器上的模拟处理），这是一个面向现代异构硬件的可移植版 OpenFOAM。受 Neko 编程模型的启发 [22],，SPUMA 支持多种现代加速器后端，不依赖某一种专有技术，这与以往研究的做法不同。本文所述的移植策略无需对框架进行大规模重构，因为只对核心类进行了少量修改，从而减少了可维护性问题；该策略还可逐步应用于整个软件。

本文分为 5 个章节：第 2 节简要介绍 OpenFOAM，并说明其当前架构的关键方面；第 3 节介绍 SPUMA 中实现的编程模型，以及内部实现细节、设计方面和可移植性；第 4 节对软件进行验证，并展示 SPUMA 在欧洲的前百亿亿次级超级计算机（Leonardo 和 LUMI）上运行 DrivAer 汽车基准测试所取得的性能。最后，第 5 节总结全文，并展望未来的工作方向。

2. OpenFOAM 框架

OpenFOAM 是一个 C++ 数值框架，可用于开发和定制求解器，以模拟涉及多种可能物理现象的流体流动。

除了少数例外，OpenFOAM 中实现的所有求解器都是分离式的，这意味着每个耦合问题（即 Navier-Stokes 方程）都会转化为一系列互不耦合的问题（例如 PISO、SIMPLE、PIMPLE）。每个问题都采用基于非结构网格的共置有限体积法（FVM）进行离散。$C++$ 的面向对象编程特性被用于构建一种语法模型，以模拟方程，并对标量、向量和张量进行代数运算。

空间和时间离散采用低阶格式（通常为二阶格式），并且原生提供了大量预条件器和迭代求解器，用于求解线性系统。

最后，OpenFOAM 的设计基于消息传递并行（MPP），以实现区域分解算法。MPP 在框架的底层集成，提供了无需编写任何并行专用代码即可使用的高级应用程序接口。

2.1. 设计方面

OpenFOAM 设计于20世纪90年代，当时通用加速器尚不存在，可用内存通常也很少。OpenFOAM 在偏微分方程的求解过程中采用惰性策略；如果不考虑网格和解场，每个量都会按需分配到堆内存中，并在不再需要时立即销毁。缓存由用户显式执行，且仅适用于有限种对象类型。例如，在主时间循环中，每个标量方程的线性方程组和线性求解器都会在每个时间步创建并销毁。虽然不允许将这些对象整体缓存，但用户可以在指定的情况下缓存多重网格求解器的 GAMG结构。内存请求和释放过程由操作系统在中央处理器上（在 GPU 上则由驱动程序）管理，而非由开发者管理。正如第 4 节所示，这种方法在 GPU 上效率低下，但 OpenFOAM 并未提供更复杂的内存分配/释放策略。

另一点重要说明涉及 OpenFOAM 有限体积格式和线性代数求解器对特定数据结构的依赖，该数据结构由 $fvMatrix$ 和 $lduMatrix$ 类表示，在[3]中称为$LDUorderedCOO$。这些类依赖于隐式自由度（DOF）映射，这意味着网格数据结构反映了矩阵的稀疏模式。这种方法减少了算法的内存占用，但同时也降低了灵活性，只能求解标量问题。因此，若不对整个代码进行彻底修改，就无法在不显式转换的情况下采用其他矩阵数据结构。尽管这不是本文的主要关注点，但众所周知，近年来关于矩阵格式的一些进展，着重于最大限度地提升其在 GPU 加速器及现代硬件上的性能。

由于所有求解器、预条件器和平滑器都是原生实现，并依赖于专为 CPU 设计的特定数据结构，因此我们能够利用 OpenFOAM 框架提供的运行时选择功能，加入针对特定硬件优化的新线性求解器（例如代数多重网格）。更多详情见下一小节。

如前所述，OpenFOAM 采用分布式并行方法：在设置阶段将计算网格分解为若干子域，并在模拟过程中将每个子域分配给不同的并行进程。进程间的所有通信均由 OpenFOAM 的 PStream 库管理，该库封装了对用户可用的 MPI 实现所进行的 API 调用，并将其纳入框架的高级语法中。这一抽象层隐藏了与并行化相关的所有细节，并使 OpenFOAM 中的进程间通信必须以同步方式进行，从而优先考虑可读性和可维护性，而非最高性能；稀疏矩阵-向量乘法除外。让异步通信与计算任务重叠，是提升 OpenFOAM 在传统基于 CPU 的架构和加速器上的可扩展性的一条有前景的途径，但这需要对代码进行大量修改，与本工作的范围不符。

在混合型集群上，通常都能使用 GPU 感知型 MPI 实现。只要提供指向 GPU 内存的指针，这些实现便可直接在 GPU 之间进行通信，无需修改 API 调用。这样可以避免向主机内存传输数据所产生的高昂开销。我们的移植工作采用托管内存这一混合方法，在 GPU 内存和主机内存中镜像分配内容。更多详情见第 3.3 节。遗憾的是，直接进行 GPU 间通信需要纯 GPU 内存指针；虽然可以将托管内存指针传递给 GPU 感知型 MPI 实现而无需修改 API 调用，但这会触发 GPU 与主机内存之间的隐式传输。这些隐式传输会对代码在多 GPU 上的可扩展性产生负面影响。一种可能的解决方案是为 MPI 通信分配仅驻留在 GPU 内存中的缓冲区；不过，我们尚未实现这一方案。本文展示的所有结果都受到这些隐式内存传输的影响。

3. 编程模型

OpenFOAM 利用 C++ 的面向对象特性，定义了一种用于描述场的运算和操作的高级语法。例如，使用运算符重载来定义场之间的代数运算，同时构建矩阵来表示微分算子的离散形式。这种语法，即领域专用语言（DSL），使得实现自定义求解器、边界条件及其他功能变得相对容易。OpenFOAM 成功并长期发展的主要原因之一，是其 DSL 所带来的灵活性。另一个原因是使用了 C++ 本身——一种标准化且支持范围极广的语言。此外，OpenFOAM 所需的第三方库数量较少，这在历史上确保了它与众多平台的兼容性：最重要的前提是拥有可正常工作的 C++ 编译器。

对于图形处理器，目前还没有一种像 C++ 这样的、与厂商无关的标准化语言。过去十年间，出现了多种 GPU 加速编程方案：专有语言，如 CUDA [25], ；开放编程语言，如 OpenCL [26] 和 HIP [27], ；标准库，如 C++ 的 stdpar ；基于编译器指令的应用程序接口，如 OpenMP [28] 和 OpenACC [29], ；外部库，如 Kokkos [30], ；牛津并行 DSL：OP2 [31] 和 OPS [32], ；OCCA [33] 和 Thrust [34]。这些方案各有优缺点。它们对不同 GPU 厂商（即英伟达、AMD 和英特尔）以及计算科学中最常用的编程语言（即 C、C++ 和 Fortran）的支持程度各不相同，没有一种方案支持上述所有可能的组合。例如，CUDA 能确保在英伟达平台上实现高性能，但不受其他厂商支持。Kokkos 这类库充当软件与硬件之间的抽象层，可支持多个平台，但不原生支持 Fortran。由于这类抽象层需要特定的数据类型，它们要么迫使开发者采用同一种编程语言（Kokkos 使用 C++），要么要求维护混合语言源代码，后者会大幅增加工作量。此外，将代码的核心部分绑定到第三方库也会带来额外风险，因为库维护者可能会决定停止支持对代码至关重要的功能。

在 GPU 开发方面，不存在完美的解决方案。Nek5000 的开发者为其 GPU 移植所采用的折中方案 [22]是将代码的高级接口与低级数值后端分离开来；前者可以维护

为标准化语言（在他们的案例中是 Fortran 2008），而后者则可以使用针对特定硬件量身定制的多种实现，并为每种硬件选择最高效的可用工具。这种方法只有在尽可能减少代码中硬件特定部分的规模时，才可以被视为具有可持续性。如果在 OpenFOAM 中采用类似的方法，首先可以观察到，其高级语法基本无需改动，因为显式循环很少，且大多包含在 场 和几何场类的重载运算符中。然而，要针对多个不同平台，为这些低级循环分别实现可在图形处理器上运行的并行版本，是一项艰巨的任务，将大幅增加维护工作量。第 3.1 节介绍了一种可移植编程抽象，可减少将 OpenFOAM 完整移植到图形处理器所需的硬件特定内核数量。

3.1. 可移植编程抽象

OpenFOAM 基于一种粗粒度的分布式内存并行方法：计算域被划分为一系列子域，每个子域分配给整个求解器的一个独立实例。各个求解器实例（即进程）在求解过程中的特定阶段通过消息传递范式进行通信，其余时间则独立运行。这种方法适用于基于 CPU 的分布式系统，因为每个进程都可以在不同的 CPU 核心上顺序运行。GPU 本质上是并行设备，但采用的共享内存架构更适合细粒度并行。OpenFOAM 所基于的消息传递框架仍可用于将计算负载分配到多个 GPU 上，但此时需要实现两个层次的并行。为了利用每个 GPU 本身的并行特性，原本会在单个 CPU 核心上顺序运行的算法应改为并行执行。这是一项复杂的工作，因为任何结果取决于操作顺序的循环都无法轻易并行化，通常需要采用不同的算法。

将最初为顺序执行开发的算法并行化时，还会出现另一个问题：竞争条件。不同的并行线程可能会尝试同时访问同一内存位置，从而产生未定义行为。这在 OpenFOAM 中尤为突出，因为许多算法会遍历单元面，并访问与各相邻单元对应的内存位置。两个线程可能会同时尝试访问同一单元，导致竞争条件。这

可以在可能出现竞争条件的地方使用 原子操作来避免。原子操作是线程安全的基本操作实现，它会在访问内存位置之前检查该位置是否正在使用，但会带来一些额外开销。除了原子操作，还可以采用不同的算法和数据结构，从源头上避免竞争条件。在 OpenFOAM 中，这可能意味着改为对每个单元的面使用嵌套循环，而不是对所有面进行简单循环，正如 [35], 中所探讨的那样；也可能意味着为矩阵采用不同的数据结构（例如压缩稀疏行）。这种方法虽然可以避免原子操作带来的开销，却需要进行深入重构；而本研究旨在探究，在不显著改变 OpenFOAM 整体结构的前提下，将其移植到 GPU 所能带来的性能提升。

为探究能否在不进行大幅重构的情况下将 OpenFOAM 有效移植到图形处理器上，Cineca 开始着手对 OpenFOAM 的部分核心功能进行概念验证式的重新实现，并将其命名为 zepto- FOAM [36]。在 zeptoFOAM 中，每个可并行化循环都被替换为专门定制的可在 GPU 上执行的内核，同时保留 OpenFOAM 的整体数据结构。这表明，即使不进行大幅重构，也确实能够显著提升性能；这在很大程度上得益于近年来硬件供应商在优化 GPU 上的原子操作方面取得的进展。这项工作还表明，为每个循环分别编写不同的内核（并针对每种目标硬件重复这一过程）既耗时又容易出错，而且十分繁琐。将同样的方法应用于整个 OpenFOAM 并不切实际。

SPUMA 采用的更高效方法，是在观察到 OpenFOAM 中存在各种各样的可并行化算法后开发出来的；这些算法大体上可以分为三类：

1. *for* 循环：在 N 个不同的数据部分上重复执行代码块 N 次（例如，对两个数组的每个单元分别执行任意代数运算）。2. 归约求和：一种对一个或多个数组进行运算并生成单一结果的算法，该结果是各单元贡献值之和（例如，标量积会返回两个数组所有分量乘积之和）。3. 归约比较：一种对一个数组进行运算并通过对其每个单元应用比较运算来生成单一结果的算法（例如，返回数组最小值或最大值的最小值/最大值函数）。

值）。

可以定义一个名为执行器的抽象类，其中包含这三类可并行化算法的接口，并将需要并行执行 N 次的操作留作尚未定义的 lambda 函数（见清单 1）。执行器本身是一个模板类，它使用奇异递归模板模式（CRTP） [37]，提供静态接口，并接受一个派生类作为模板参数。执行器的派生类包含硬件优化内核，用于实现上述三类可并行化算法。这样，硬件特定代码就被限制在这些派生类的三个内核中，其余代码则可以通过父执行器 类提供的硬件无关接口调用相应的并行算法。随后，用户可以在编译时选择所需的后端。

```
1 namespace Foam 2 { 3 template <typename T>4 class 执行器 5 { 6 public :
```

```
27 {28   (static_cast  <T *  >(this))-> _升力系数a c k e n dR e d uc t i o nC o m pa r e (lambda, op, result, size);29 };30 }3132 ...3334
clas
laml
resu
_ba
```

清单 1：将 CRT 模式应用于执行器 类。

设置好执行器基础设施后，只需找出可并行化的代码片段，并将其封装在 lambda 函数中，再将其传递给与其类别相对应的并行执行器（例如 $parallelFor$），即可逐步完成 OpenFOAM 到加速器的移植。清单 2 展示了这一概念在mag 函数中的应用。

```
 1 template  <class Type  >2 void mag 3 ( 4 场  <typename typeOfMag  <
Ty
typ
f1.
```

```
checkFields(result,f1,"f1 =mag(f2)");自动 rp =result.begin
()
f1
si
```

图 2：将 *mag* 函数移植到 GPU。

3.2. 内存池

当内存容量有限时，在 NUMA 和 GPU 架构上进行高效的内存管理至关重要。内存池是一种算法，它依靠预先分配的内存块来高效管理动态内存分配。内存池通过重复利用已分配的内存来避免内存碎片并提升性能 [38, 20]。

正如我们在第 2 节中所述，OpenFOAM 会大量使用临时数组，这些数组需要在短时间内进行分配与释放：内存池是减少这些操作相关开销的有效工具。第 4.3 节将进一步考察内存池的影响，并比较不同的分配策略。

我们定义了一种半自动策略，用于确定哪些数组分配到内存池中：每个 场 对象，或从场派生的类实例，都会自动分配到内存池中；而对于 *List* 对象，则必须手动指定。该策略可避免将不参与 GPU 加速计算的小对象塞入内存池。由于不同硬件供应商的内存分配机制各不相同，我们开发了一个小型接口，向多个硬件专用后端提供少量应用程序接口。在此接口之上，我们实现了三种算法，用于协调多个内存块的分配与释放：虚拟内存池、固定大小内存池和动态大小内存池。

用于调试和性能分析的 虚拟内存池 以标准方式使用供应商 API 分配资源。固定大小内存池是一种稍微复杂一些的算法：通过供应商 API 分配一个大小恒定且足够容纳模拟过程中所使用全部场的单一内存块。每当程序请求新的内存空间时，内存池管理器都会检查是否有足够的内存可用。如果有，则不执行分配，而是将指向内存池中某个位置的指针

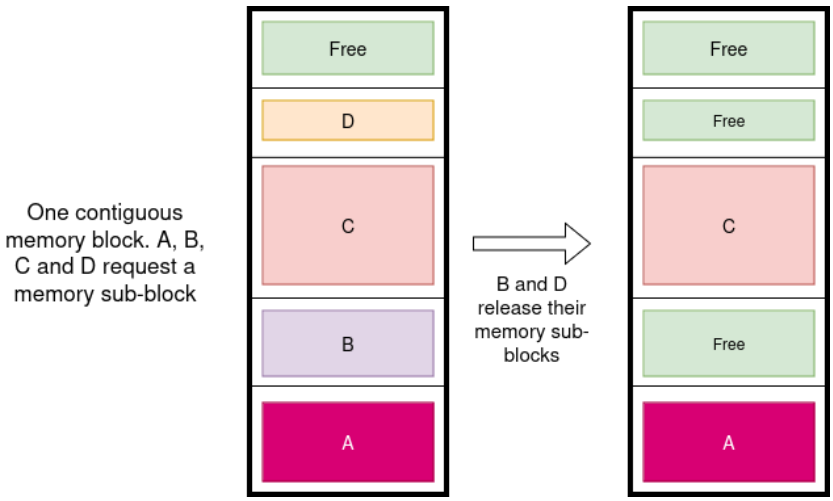


图1：固定大小内存池：一个固定大小的连续内存块被划分为许多子块；新内存空间的分配与释放仅涉及指针处理。

存储在列表中并返回。同时，可用空间的大小也会相应减少。如果没有足够的内存，则会发生致命错误并停止模拟。当某个内存空间不再需要时，内存池管理器不会执行任何释放操作，而只是从列表中删除该指针，并增加可用空间。该机制如图1所示。

作为第三种选择，动态大小内存池的更复杂管理策略包含在*Umpire* 库中。动态大小内存池不需要评估模拟所需的最大内存；相反，仅需进行估算即可。在本工作中，关于*Umpire* 的唯一原创贡献，是开发了 SPUMA 与这一第三方库之间的接口。有关实现方面的更多细节，请参阅 [38] 。

3.3. 统一内存

统一内存（或托管内存）是一种厂商技术，允许应用程序分配数据，并由运行在中央处理器或图形处理器上的代码访问或修改。当应用程序访问分配在托管内存中的数据时，驱动程序和/或硬件会负责将缺失的内存页迁移到访问设备的内存中。

如果将统一内存与池策略结合使用，分配和数据迁移对开发者而言便完全透明，从而显著减少移植过程所需的工作量。第 3.1 节所述方法之所以高效，是因为开发者可以逐步将代码的不同部分卸载到图形处理器，而无需操心显式内存管理。由于数据可以自动从 CPU 移至 GPU，在将新算法移植到加速器后，开发者只需专注于验证结果。

当模拟过程中由 GPU 处理的数据不再来回传输时，GPU 移植便告完成。在这一过程中，最后只需将少量数据从 GPU 复制到 CPU，例如显示在屏幕上的标量参数，或用于判断是否继续执行模拟工作流的参数。大量数据传输仅在模拟开始时仍然存在，这是因为需要将网格量和解场卸载到 GPU；这些数据由 CPU 从磁盘读取，此外还涉及其他输入/输出操作。

3.4. 外部求解器：*amgx4foam*

开发专用于 GPU 的线性代数功能（例如预条件器、平滑器以及代数多重网格等多重网格算法），可以视为本文所述 GPU 移植策略的进一步推进。然而，求解计算流体力学问题离不开这些算法，而缺乏高效实现则会使人难以评估使用 GPU 加速纳维-斯托克斯方程数值解的实际收益。

文中提出的一种解决方案是将 SPUMA 与第三方库链接，这些库可在 GPU 上实现先进的线性求解器。具体而言，英伟达维护着一个经过厂商优化的库，可用于分布式代数多重网格算法（以及其他线性预条件器和求解器） [39]。该库名为 AmgX，借助 OpenFOAM 框架提供的运行时选择功能，即可轻松将其接入 SPUMA，正如第 2 节所述。

这项工作的一个副产品是对$amgx4foam$的全新实现。它是一个外部库，可扩展 SPUMA 中原生移植到图形处理器上的线性求解器和预条件器的数量。该库是对先前开发成果的重构，相关内容见 [3, 5, 6]。与这些研究中描述的混合方法不同，在本例中，矩阵

已位于图形处理器内存中，无需从 CPU 向加速器卸载数据。因此，现在只需将 OpenFOAM 原生格式转换为 CSR 格式。该转换采用了与 [5, 6], 中提出的相同策略，即根据基数排序算法对表示矩阵稀疏模式的 OpenFOAM 数组进行处理，并据此计算映射。相同的方法也可应用于其他外部线性代数库，例如 HYPRE [40], 、PETSc [41], 、Ginkgo[42], 、rocALUTION [43], 、Paralution [44], 、BootCMatchG（[45]）、Chronos [46],和 SParSH-AMG [47]——这些库均实现了 GPU 加速的代数多重网格线性求解器，例如 HYPRE 的 BoomerAMG 和 PETSc 的 GAMG——从而扩展 SPUMA 中可用的适配图形处理器的线性求解器列表。

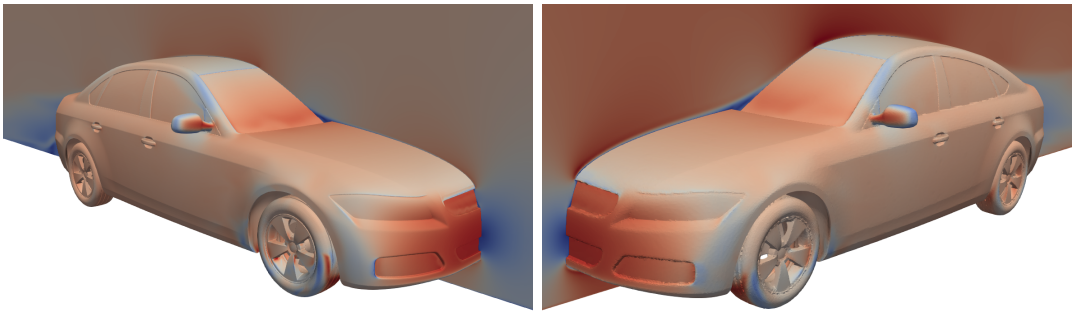
4. 性能评估

4.1. 实验设置

本节所述的数值实验使用 SPUMA 和 OpenFOAM-v2412，在 Leonardo 和 LUMI 上进行。这两台欧洲预百亿亿次级超级计算机分别配备 NVIDIA A100 和 AMD MI250X GPU 卡。两台集群的硬件特性列于表1和表2。表2显示，AMD MI250X GPU 卡包含两个图形计算芯片（GCD），每个芯片均作为独立的图形处理器运行。本文其余部分提及的 AMD GPU 均指单个 GCD。软件栈列于表3。

验证、性能分析和性能测试均使用 DrivAer 工业测试案例进行。这是一项广受欢迎的汽车基准测试，详见 [48]。车辆采用两种配置进行模拟：快背式几何形状和开闭式冷却版本（见图2）。表 4 列出了本文使用的、与模拟中不同网格和配置相关的缩写。两个测试案例的完整设置分别见参考文献 [49] 和 [50], ，对应快背式版本和开闭式冷却版本。在这两种情况下，原始问题均通过采用静止车轮和静止地面进行了简化。

在整篇论文中，仅使用 CPU 的运行均在 CPU 集群分区（即 Leonardo-DCGP 或 LUMI-C）上执行，而 GPU 运行则在 GPU 集群分区（即 Leonardo-Booster 或 LUMI-G）上执行。



(a) (b)

图2：三厢式（a）和快背式（b）配置下的 DrivAer模型。

分区/规格	Booster	DCGP
模型	Atos BullSequana X2135“Da Vinci” 单节点图形处理器刀片服务器	Atos BullSequana X2140 三节点CPU刀片服务器
节点	3456 个单路插槽、32个核心 英特尔 Ice Lake CPU	1536 个双路插槽、56 核 英特尔 Sapphire Rapids CPU
处理器	1 × 英特尔至强白金 8358 ， 2.60 吉赫兹，热设计功耗 250W	2 × 英特尔至强白金 8480+ ， 2.00 吉赫兹，热设计功耗 350W
加速器	每个节点配备 4 个 NVIDIA A100 DaVinci GPU 64GB HBM2e（1640 GB/s）NVLink 3.0（200GB/s）	-
核心	32个核心/节点	112个核心/节点
RAM	512（8×64）GB DDR4 3200 MHz	512（16 x 32）GB DDR5 4800 MHz
峰值性能	306.31 拍浮点运算/秒	9.43 拍浮点运算/秒
内部网络	每节点 2 个双端口 HDR100	每节点单端口 HDR100

表 1：Leonardo 硬件特性。

分区/规格	LUMI-G	LUMI-C
节点	2978 个单路插槽，64 核心	2048 个双路插槽，64 核心
处理器	1 × AMD EPYC 7A53 CPU（特伦托）， 3.5 吉赫兹 4 × AMD MI250X GPU	2 × AMD EPYC 7763 中央处理器（米兰）， 基础频率 2.45 吉赫兹，Boost 频率 3.5 吉赫兹
加速器	第二代 AMD CDNA 架构 每个图形处理器配备 2 x 64GB HBM2e（3200 GB/s）	-
核心	64 核/节点	128 核/节点
RAM	512 (8x64) GB DDR4	256 (8 x 32) GB（或更高）
峰值性能	531.51 拍浮点运算/秒	7.63 拍浮点运算/秒
内部网络	1 × 200 GB/s HPE Cray Slingshot-11 互连	

表 2：LUMI硬件特性。

集群	软件包	版本
Leonardo（英特尔 +英伟达）	CUDA	12.1
	NVCC 编译器	23.11
	OpenMPI	4.1.6
	AmgX	2.5
LUMI（AMD）	HIP 编译器（ROCm）	6.0.3
	AMD clang 编译器	17.0.0
	Cray-MPICH	8.1.29

表3：软件栈。

Name	配置	网格单元数	面数
DrivAer22M	快背式	22.555.718	68.559.955
occDrivAer65M	开放-封闭冷却切口	65.334.765	203.271.868
occDrivAer120M	开放-封闭冷却切口	120.899.906	374.098.306
occDrivAer235M	开放-封闭冷却切口	235.741.179	726.263.835

表 4：DrivAer 配置：DrivAer22M 取自 [49]，而 occDrivAer 配置取自 [50]。

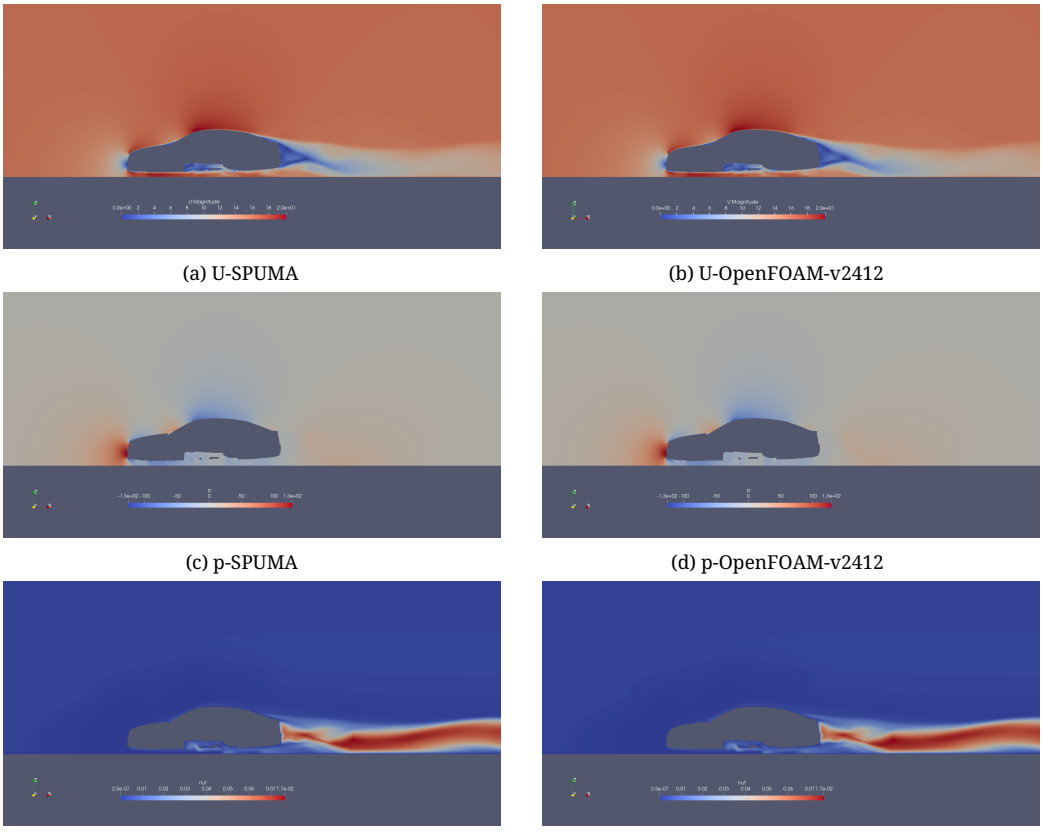
4.2. 验证结果

SPUMA 的一项重要要求是能够复现其中央处理器参考版本 OpenFOAM-v2412 的结果。采用雷诺平均纳维–斯托克斯 DrivAer22 M 测试算例验证结果，所用设置见附录 A。使用 SPUMA 和 OpenFOAM-v2412 运行模拟，共进行 100 次 SIMPLE 迭代，初始流场为 OpenFOAM-v2412 在时间 $=2000$ 得到的流场。

	C_d		C_l	
	Leonardo	LUMI	Leonardo	LUMI
OpenFOAM-v2412 (GAMG)	0.28498	0.28498	0.01816	0.01817
SPUMA (GAMG)	0.28498	0.28498	0.01816	0.01816
SPUMA (AMGX)	0.28498	-	0.01816	-
OpenFOAM-v2412（预条件共轭梯度法）	0.28498	0.28498	0.01817	0.01817
SPUMA（预条件共轭梯度法）	0.28498	0.28498	0.01817	0.01817

表5：DrivAer22M。使用不同压力方程求解器得到的平均阻力系数和升力系数。平均过程在区间 $[T_i = 2000, T_f = 2100]$ 内执行。

表5展示了采用不同压力求解器得到的平均 C_d 和 C_l 的比较结果，表明 SPUMA 与 OpenFOAM-v2412 具有良好的一致性。流场的可视化比较也显示出良好的一致性。图3展示了汽车对称面上的速度、压力和涡黏度场，这些场是在经过 $T = 2100$个时间步后得到的。



(a) U-SPUMA

(b) U-OpenFOAM-v2412

(c) p-SPUMA

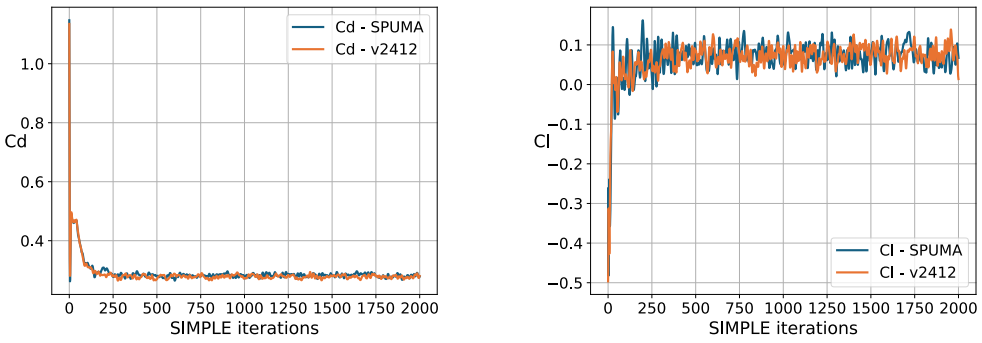
(d) p-OpenFOAM-v2412

(e) 涡黏度-SPUMA

(f) 涡黏度-OpenFOAM-v2412

图3：DrivAer22M。SPUMA 与采用预条件共轭梯度法作为压力求解器的 OpenFOAM-v2412 在 T=2100 时的速度场（上）、压力场（中）和涡黏度（下）对比。

采用 OpenFOAM-v2412（800 个核心）和 SPUMA（8 个 GPU）对 RANS occDri-vAer65M 测试算例进行了另一组模拟，并行运行该算例，最多迭代 2000 次。使用最后 1500 次伪时间迭代窗口评估平均力系数。阻力



(a) (b)

图4：occDrivAer65M。阻力系数（a）和升力系数（b）对比

	C_d		C_l	
	Leonardo	LUMI	Leonardo	LUMI
OpenFOAM-v2412（GAMG）	0.27847	0.28017	0.07793	0.07822
SPUMA（GAMG）	0.28031	0.28167	0.07812	0.07881

表6：occDrivAer65M。使用不同硬件计算的平均阻力系数和升力系数。平均过程在区间 $[T_i = 500, T_f = 2000]$ 内进行

和升力系数随时间的变化绘于图4，平均值则列于表6。阻力系数和升力系数的相对差异均小于 1%。这一微小差异是由于不存在稳态解，且解对误差传播具有很强的依赖性。在数百次迭代过程中，平均力系数的时间历程逐渐发散，呈现出混沌特征，见图4。

由于浮点代数在机器精度范围内不满足结合律和分配律，运算顺序可能会对结果产生细微影响。这些微小扰动在汽车尾流和底部区域会被放大，进而影响力系数。算例设置的任何变化，例如采用不同的计算域划分或编译器优化标志，都会在一定程度上产生不同结果。在图形处理器上，并行操作的执行顺序具有随机性，进一步降低了可复现性。

4.3. 性能分析

使用 CUDA 后端对应用程序的底层例程和内存利用情况进行性能分析，分别在串行和并行（2 个 GPU）模式下开展。在表 7 中，我们报告了 DrivAer22M 测试算例进行五次 SIMPLE 迭代（从第 2 次迭代到第 6 次迭代，不包括第一次迭代）期间，CUDA 内核和内存操作相对于 CUDA 上下文总耗时的时间占比。约 83% 的时间用于内核执行（即 归约 和 并行_*for*），其余时间用于内存操作（即 内存设置 和主机到设备内存复制）。此外，还列出了内存复制 操作和并行_*for* 内核的时间占比。对于后者，包含所有计算密集型内核——主要是稀疏矩阵-向量乘法（SpMVM）和梯度计算——但不包括单项占比低于 0.1% 的内核。在内存复制 操作中，开销最大的是设备间传输，约占总时间的 53%。

操作	内存池类型	时间 [%]	总时间 [毫秒]	次数	平均值 [μ秒]	中位数 [μ秒]	总大小	平均值	中位数
内存设置	dummy	28.0%	3757	6526	575.652	1.729	66.34 吉比字节	10.41 兆字节	232 字节
统一主机到设备	dummy	0.0	3.044	1207	2.521	2.272	12.13 兆字节	10.29 千字节	4.00 千字节
统一设备到主机	dummy	0.0	2.508	1205	2.081	1.728	12.13 兆字节	10,31 千字节	4,00 千字节
H2D	dummy	0.0	85.081	4244	20.047	4.640	49.95 千字节	12 B	8 B
D2H	dummy	0.0	10.907	3156	3.455	3.264	24.66 千字节	8 B	8 B
D2D	dummy	71.0	9455	3602	2625	185.807	230.75 吉比字节	65.60 兆字节	2.72 兆字节
内存设置	固定大小	11.0	56.411	6526	8.644	1.696	66.34 吉比字节	10.41 兆字节	232 字节
统一主机到设备	固定大小	0.0	2.764	1101	2.510	2.335	10.84 兆字节	10.08 千字节	4.00 千字节
统一设备到主机	固定大小	0.0	2.513	1300	1.932	1.727	10.84 兆字节	8.54 千字节	4.00 千字节
H2D	固定大小	8.0	40.846	4244	9.624	4.032	49.95 千字节	12 B	8 B
D2H	固定大小	2.0	10.603	3156	3.359	3.264	24.66 千字节	8 B	8 B
D2D	固定大小	77.0	387.542	3602	107.590	8.704	230.75 吉比字节	65.60 兆字节	2.72 兆字节

表 8：使用 SPUMA 和 1 个图形处理器运行 DrivAer22M 测试算例时，$simpleFoam$ 五次迭代（$T_f - T_i$、$T_f = 6$、$T_i = 1$）的内存传输统计。测试了 $dummy$ 和 固定大小 内存池。

Task	时间 %	项目	时间 %
		稀疏矩阵-向量乘法 (矩阵-向量乘法 + 转置矩阵-向量乘法)	21.79
		高斯梯度::梯度场	15.06
并行 for 循环	79.27	单元限制梯度	13.94
		近似对角不完全 LU 预条件器	9.07
		乘法	5.16
		表面插值格式::点积插值	4.05
		-=	3.26
		表面积分	3.22
		/=	2.84
		预条件双共轭梯度法::求解	2.13
		+=	2.10
		Richardson平滑器::平滑	1.88
		减去	1.23
		lduMatrix::sumA	1.06
		lduMatrix::negSumDiag	1.04
		lduMatrix::H + lduMatrix::H1	0.83
		有限体积边界::patchInternalField	0.82
		add	0.80
		场::negate	0.66
		GAMG求解器::缩放	0.64
		除	0.55
		GAMG聚合::限制场	0.55
		lduMatrix::非对角项绝对值之和	0.48
		GAMG求解器::接口内部场	0.46
		*=	0.42
		分量	0.41
		场替换	0.40
		T	0.40
		dev2	0.39
		线性迎风格式	0.38
		ldu接口场添加到内部场	0.32
		GAMG求解器延拓场	0.32
		GAMG求解器聚合矩阵	0.30
		mag	0.27
		max	0.25
		symm	0.24
		devTwo对称	0.23
		min	0.23
		dot	0.22
		GAMG 求解器::V循环	0.2
		pos0	0.18
		点乘	0.17
		外部	0.14
		有限体积矩阵::从列表设置值	0.14
		sqr	0.14
		sqrt	0.11
		表面法向梯度格式::表面法向梯度	0.10
		下对角上矩阵::面H	0.10
		其他（各占 < 0.1%）	0.27
归约	4.18		
自定义内存设置	9.88		
		D2D	52.70
		H2D	18.62
内存操作	6.67	D2H	15.44
		内存设置	11
		统一主机到设备	1.24
		统一设备到主机	1

表 7：CUDA 内核计算和内存操作相对于 CUDA 上下文总时间的耗时百分比分布。本次运行在 Leonardo 上进行，使用 DrivAer22M 测试算例，迭代五次（$T_f - T_i$, $T_f = 6$, $T_i = 1$），使用 2 个 GPU，以 GAMG 作为压力求解器（最粗层求解器 =DIC+PCG），并使用固定大小 内存池。

API	内存池类型	时间 [%]	总时间 [毫秒]	次数	平均值 [μ秒]	中位数 [μ秒]	最小值 [μs]	最大值 [μs]
cudaDeviceSynchronize	dummy	72.8%	54542	53259	1024	8.218	1.584	326289
CUDA 内存复制	dummy	12.7%	9536	6758	1411	14.144	9.888	31060
cudaFree	dummy	8.8%	6574	27192	241.761	9.962	5.464	13300
cudaMemset	dummy	5.1%	3812	6526	584.154	10.041	8.006	31982
cudaMallocManaged	dummy	0.3%	233.885	27192	8.601	6.529	1.499	367.717
CUDA 内核启动	dummy	0.2%	164.042	53259	3.080	2.756	2.266	35.659
cudaMemcpyAsync	dummy	0.0%	28.521	4244	6.720	5.876	4.391	24.201
cudaDeviceSynchronize	固定大小	92.0%	8403	53259	157.784	7.788	1.566	112744
CUDA 内存复制	固定大小	5.0%	460.657	6758	68.164	13.144	9.437	859.092
CUDA 内核启动	固定大小	1.6%	144.224	53259	2.708	2.550	2.278	241.593
cudaMemset	固定大小	1.2%	106.062	6526	16.252	9.304	8.094	383.221
cudaMemcpyAsync	固定大小	0.3%	24.014	4244	5.658	5.203	4.179	41.653

表 9：使用 SPUMA 在 1 个图形处理器上运行 DrivAer22M 测试算例时，$simpleFoam$ 执行五次迭代（$T_f - T_i$、 $T_f = 6$、 $T_i = 1$）的 CUDA API 统计数据。测试了$dummy$ 和固定大小内存池。由于 SPUMA 中每次内核调用之后都会调用$cudaDeviceSynchronize$，因此该 API 统计的数据包括内核执行所花费的时间以及与同步相关的开销。

表 8 和表 9 从统计角度进一步分析了内存操作，使用一块图形处理器以及固定大小 和$dummy$ 内存池。分析表明，采用内存池对缩短求解时间有显著影响：由于碎片化程度降低，大幅减少了非连续内存访问，内存设置操作、设备间复制和内核执行（封装在$cudaDeviceSynchronize$中）的开销降低了一个数量级或更多。从主机到设备以及反向的内存复制开销仅略有降低。如表 9 所示，内存池还通过省去数万次对$cudaMallocManaged$ 和$cudaFree$的调用，降低了对象分配与释放的相关开销。

Case	无内存池	dummy	固定大小	Umpire
DrivAer22M-200 核心	154.43 秒	145.18 秒	144.64 秒	154.37 秒
DrivAer22M-400 核心	64.29 秒	62.40 秒	61.25 秒	63.97 秒
DrivAer22M-1 图形处理器	_	822.68 秒	141.40 秒	140.63 秒
DrivAer22M-2 图形处理器	_	431.81 秒	77.14 秒	77.55 秒

表10：从 T=10 开始，使用 3 种不同内存池（在 SPUMA 中实现）和不使用内存池（ OpenFOAM-v2412）时，执行 100 次迭代所用的时间。

表10的结果也凸显了内存池的优势。表中报告了采用三种不同内存池策略时，SPUMA 相较于 OpenFOAM-v2412 的加速比。加速效果并不局限于 GPU：使用固定大小 内存池或$Umpire$ 内存池时，GPU 上的性能提升尤为显著（约为 4 倍）； CPU模拟

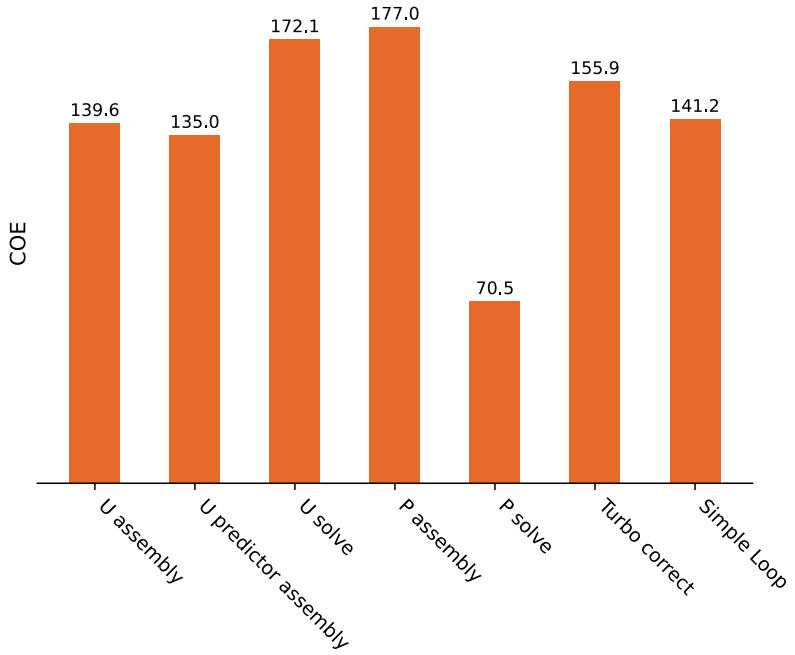


图 5：使用 SPUMA 在 2 个 GPU 上运行 $simpleFoam$，以及使用 OpenFOAM-v2412 在 400 个 CPU 核心上运行时的等效系数（COE）分解。压力方程使用 GAMG 求解器，最粗层级使用 PCG 求解器和 DIC 预条件器。该模拟使用 DrivAer22M 测试算例，在 Leonardo 高性能计算集群上执行。

也显示求解时间有所缩短。与作者开发的固定大小内存池相比，采用更复杂动态内存池的第三方库 Umpire 并未带来任何性能提升。

为比较 GPU 和 CPU 运行的性能，我们采用了等效系数（COE），其定义为 CPU 运行时间乘以核心数，再除以 GPU 运行时间乘以 GPU 数量，如方程 1 所示。

$$COE = \frac{T_{cpu} * N_{cores}}{T_{gpu} * N_{gpus}} \tag{1}$$

图 5 展示了从 $T = 1500$ 开始进行 100 次 SIMPLE 迭代后，SIMPLE 算法六个不同主要部分（例如动量方程的组装）所获得的 COE 值。该图并不均匀：

动量方程求解和压力线性系统组装过程中达到 180 的峰值，而动量方程右端项计算中的数值较低，约为 140；其特征是梯度计算，这是一个包含多个原子操作的内核。然而，最低值低于 100，出现在使用 GAMG 求解器进行压力求解时。

要有效地对 GAMG 求解器进行 GPU 并行化并非易事，这取决于粗化算法和平滑器类型。在本研究中，我们实现了两种完全在 GPU 上运行的并行平滑器，即 Richardson（或加权 Jacobi）和两阶段 Gauss-Seidel，具体如 [51], 所述；但粗化算法并未修改。表 11 报告了仅在粗层或在层次结构的所有层中更换平滑器的影响。采用所有层均使用 Richardson、最粗层使用对角预条件器的方案，可获得最快的求解速度。使用效果更好的 CPU 预条件器（如 DIC）会涉及数据传输，并不能缩短墙钟时间。关于这一主题的深入分析将留待未来的工作。

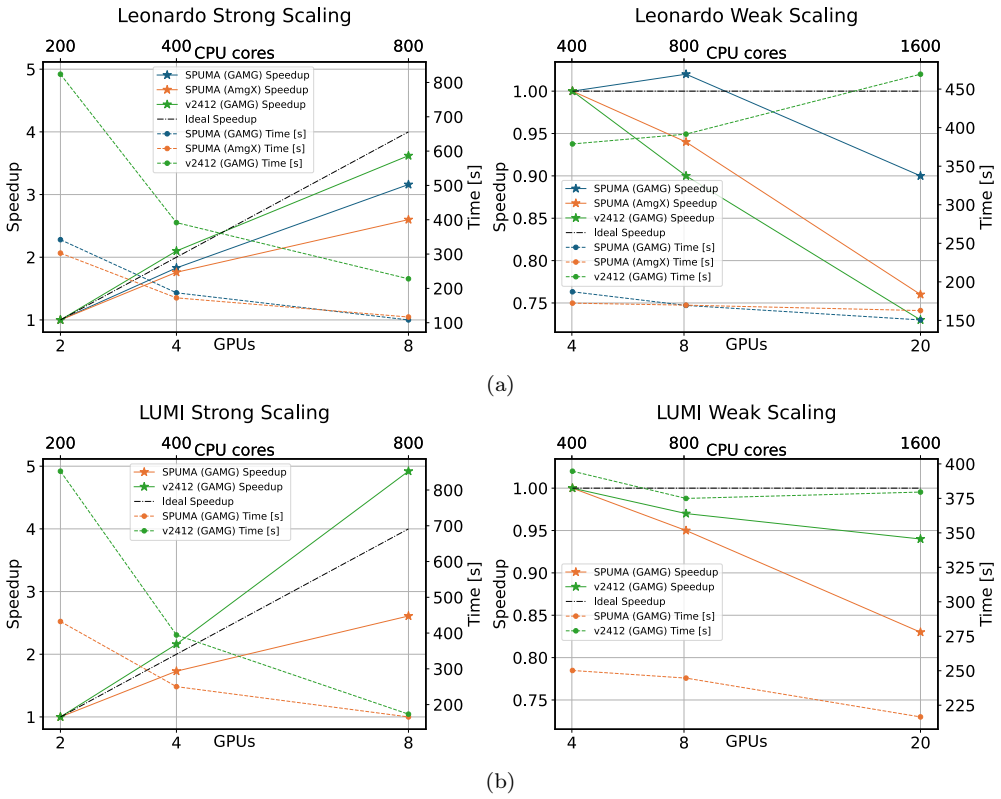
		粗层预条件器	
		DIC	对角
平滑器	Richardson（阻尼 Jacobi）	122.59 秒	115.76 秒
	两级高斯-赛德尔	132.02 秒	123.9 秒

表 11：使用不同平滑器和粗层预条件器的 GAMG 求解器求解压力方程时，DrivAer22M 测试算例进行 100 次 SIMPLE 迭代的墙钟时间。

4.4. 可扩展性测试

为评估并行性能，我们使用 OpenFOAM 的层次分解方法，在 occDrivAer 基准测试 [50] 上进行了弱可扩展性测试和强可扩展性测试。强可扩展性测试使用最小网格，即 65M 算例；弱可扩展性测试则针对表 4 中报告的全部三个网格进行。由于 Leonardo 和 LUMI 的 CPU 分区每节点最大核心数不同（见表 1 和表 2），所有模拟在两个集群上均以每节点 100 个核心运行。这一选择略低于两个集群的核心数上限，但仍与上限相近。对这些具有大容量缓存和较多核心的节点采用欠载运行是常见做法，因为这样有助于处理后台系统任务和系统维护任务，同时不会降低作业性能。

因此，使用 100 个核心是在集群间保持一致性、确保运行平稳，以及便于整除以获得规则的域分解（例如层次分解）之间的良好折中。



(a)

(b)

图 6：SPUMA 使用 AMGX（仅限 Leonardo）和 GAMG 求解器求解压力时的强扩展和弱扩展性能，与 Leonardo（a）和 LUMI（b）上的 OpenFOAM-v2412 进行比较。加速比以最小分解规模为基准。所报告的时间为使用 occDrivAer 测试算例完成 100 次 SIMPLE 迭代所需的时间。

图 6 展示了整个求解器的弱扩展和强扩展性能（包括加速比和计算时间），而表 12、13、14 和 15 则详细列出了求解器各部分所耗费的时间。图 7 展示了 occDrivAer65M 和 occDrivAer235M 算例的加速比，其基准案例是在 Leonardo 数据中心分区的 200 个核心（两个节点）上使用 OpenFOAM-v2412 运行的。

就强可扩展性而言，Leonardo 和 LUMI 使用 GAMG 求解器时的曲线相同。在 4 个 GPU 时，效率仍然良好，

但在 8 个 GPU 时开始下降，效率约为 65%。将压力求解器切换为 AMGX（仅在 Leonardo 上可用）后，由于其具备内置 GAMG 求解器所不具备的激进粗化功能，8 个 GPU 时的效率可提高至 80%。

由于 AmgX 是一种纯代数多重网格求解器，而且 SIMPLE 循环中的矩阵系数会随时间步而变化，因此无法像 GAMG 那样缓存多重网格层次结构中的粗网格，每次都需要重新计算。这解释了为何在使用 2 个 GPU 时，压力求解器的运行时间比 GAMG 慢（慢 2.5 倍）。不过，AmgX 更好的可扩展性使其在使用 8 个 GPU 时墙钟时间更短，这表明 GAMG 仍有尚未发挥的潜力。出现这种情况的原因是，采用默认设置的 GAMG 求解器会生成许多多重网格层级，其中许多层级的网格单元太少，无法有效利用 GPU，也无法实现通信与计算的重叠。相比之下，AmgX 的激进粗化生成的层级少得多，并会在网格单元数量过少时自动切换到中央处理器。提高 GAMG 求解器的可扩展性并确定最佳默认设置，将留待未来研究。

作为 GPU 模拟的参考，OpenFOAM-v2412 在中央处理器上的性能与 [52]中展示的结果一致：使用较少核心（即 400 个）时，OpenFOAM-v2412 在 LUMI 上略慢于 Leonardo，但其扩展性更好，且呈超线性增长，这证实了每个核心负载较低时，AMD CPU 的表现更佳。

	SPUMA GAMG			**SPUMA AmgX**			**OpenFOAM-v2412**		
	# GPU			**# GPU**			**# CPU**		
	4	**8**	**20**	**4**	**8**	**20**	**400**	**800**	**1600**
# 网格单元	65.3 M	121 M	236 M	65.3 M	121 M	236 M	65.3 M	121 M	236 M
# 单元格/进程	16.3 M	15.1 百万	11.8 百万	16.3 百万	15.1 百万	11.8 百万	163 千	151 千	147 千
T110-T10 [s]	**172.41**	**169.79**	**162.79**	**187.13**	**169.47**	**150.65**	**378.85**	**391.6**	**469.31**
SIMPLE 循环 [s]	1.748	1.735	1.696	1.894	1.721	1.530	3.844	3.994	4.792
U 组装 [s]	0.377	0.339	0.269	0.375	0.337	0.269	0.460	0.413	0.439
U 预测器 [s]	0.022	0.019	0.015	0.022	0.019	0.015	0.034	0.029	0.023
U 求解 [秒]	0.564	0.565	0.463	0.547	0.519	0.458	1.910	1.990	2.106
P 组装 [秒]	0.040	0.034	0.030	0.039	0.033	0.030	0.062	0.057	0.056
P 求解 [秒]	0.124	0.253	0.475	0.304	0.302	0.317	0.355	0.527	1.148
湍流 [秒]	0.491	0.413	0.349	0.482	0.404	0.350	0.816	0.777	0.810

表 12：在 Leonardo 高性能计算系统上使用 occDrivAer 测试算例，对比 OpenFOAM-v2412，详细分解 SPUMA 使用 AMGX 和 GAMG 求解器进行压力求解时的弱扩展性能。

	SPUMA GAMG			SPUMA AmgX			OpenFOAM-v2412		
	# GPU			# GPU			# CPU		
	2	4	8	2	4	8	200	400	800
# 网格单元	65.3M	65.3M	65.3M	65.3M	65.3M	65.3M	65.3M	65.3M	65.3M
# 单元格/进程	32.7M	16.3M	8.17M	32.7M	16.3M	8.17M	327k	163k	81.7k
T110-T10 [s]	**302.8**	**172.41**	**116.56**	**341.93**	**187.13**	**108.14**	**824.07**	**391.6**	**227.93**
SIMPLE 循环 [s]	3.068	1.748	1.195	3.458	1.894	1.093	8.347	3.994	2.326
U 组装 [s]	0.690	0.377	0.208	0.692	0.375	0.206	0.947	0.413	0.232
U 预测器 [s]	0.038	0.022	0.013	0.039	0.022	0.013	0.099	0.029	0.014
U 求解 [s]	1.095	0.564	0.338	1.102	0.547	0.316	4.180	1.990	1.115
P 组装 [s]	0.065	0.040	0.024	0.064	0.039	0.023	0.149	0.057	0.043
P 求解 [秒]	0.165	0.124	0.251	0.553	0.304	0.181	0.738	0.527	0.329
湍流 [秒]	0.806	0.491	0.284	0.802	0.482	0.280	1.786	0.777	0.465

表 13：在 Leonardo 高性能计算集群上使用 occDrivAer 测试算例时，SPUMA 使用 AMGX 和 GAMG 求解器求解压力的强扩展性能详细分项，与 OpenFOAM-v2412 的对比。

	SPUMA GPU 数量			OpenFOAM-v2412 # CPU		
	4	8	20	400	800	1600
# 网格单元	65.3M	121M	236M	65.3M	121M	236M
# 单元格/进程	16.3M	15.1M	11.8M	163k	151k	147k
T110-T10 [s]	**250.18**	**244.66**	**216.5**	**394.7**	**374.96**	**379.57**
SIMPLE 循环 [s]	2.535	2.494	2.238	3.982	3.820	3.869
U 组装 [s]	0.531	0.476	0.367	0.745	0.680	0.710
U 预测器 [s]	0.032	0.028	0.022	0.054	0.057	0.025
U 求解 [秒]	0.901	0.901	0.734	1.207	1.160	1.165
P 组装 [秒]	0.051	0.043	0.038	0.107	0.082	0.076
P 求解 [秒]	0.165	0.323	0.490	0.412	0.482	0.539
湍流 [秒]	0.693	0.581	0.471	1.156	1.067	1.062

表 14：在 LUMI 高性能计算系统上使用 occDrivAer 测试算例，对比采用 GAMG 求解器处理压力的 SPUMA 与 OpenFOAM-v2412 的弱扩展性能详细分项。

在弱扩展图中也能观察到类似趋势。采用 AmgX 的 SPUMA 在 Leonardo 上使用 4 至 20 个 GPU 时，效率不低于 90%；而使用 GAMG 的 SPUMA 和 OpenFOAM-v2412 的可扩展性则以相似幅度下降，在 20 个 GPU 时约为 75%。在 LUMI 上，这两种求解器的表现均更好：使用 20 个 GPU 时，OpenFOAM-v2412 的效率高达 95%，SPUMA 则约为 85%。

	SPUMA			OpenFOAM-v2412		
	# GPU			# CPU		
	2	4	8	200	400	800
# 网格单元	65.3M	65.3M	65.3M	65.3M	65.3M	65.3M
# 单元格/进程	32.7M	16.3M	8.17M	327k	163k	81.7k
T110-T10 [s]	**432.42**	**250.18**	**165.54**	**852.56**	**394.7**	**173.3**
SIMPLE 循环 [s]	4.376	2.535	1.691	8.605	3.982	1.758
U 组装 [s]	0.951	0.531	0.298	1.551	0.745	0.358
U 预测器 [s]	0.055	0.032	0.018	0.154	0.054	0.027
U 求解 [s]	1.680	0.901	0.549	2.594	1.207	0.508
P 组装 [s]	0.080	0.051	0.030	0.210	0.107	0.046
P 求解 [s]	0.228	0.165	0.303	0.937	0.412	0.185
湍流 [s]	1.122	0.693	0.396	2.511	1.156	0.494

表15：在 LUMI 高性能计算系统上使用 occDrivAer 测试算例时，SPUMA 使用 GAMG 求解器求解压力的强扩展性能详细分解，并与 OpenFOAM-v2412 进行比较。

4.5. 能耗

第 4 节所呈现的大多数结果都侧重于模拟求解时间相关的性能指标。然而，能耗和可持续性在高性能计算领域正变得越来越重要。因此，我们决定纳入一项初步研究，重点比较在 CPU 和 GPU 上运行 OpenFOAM模拟的能耗。

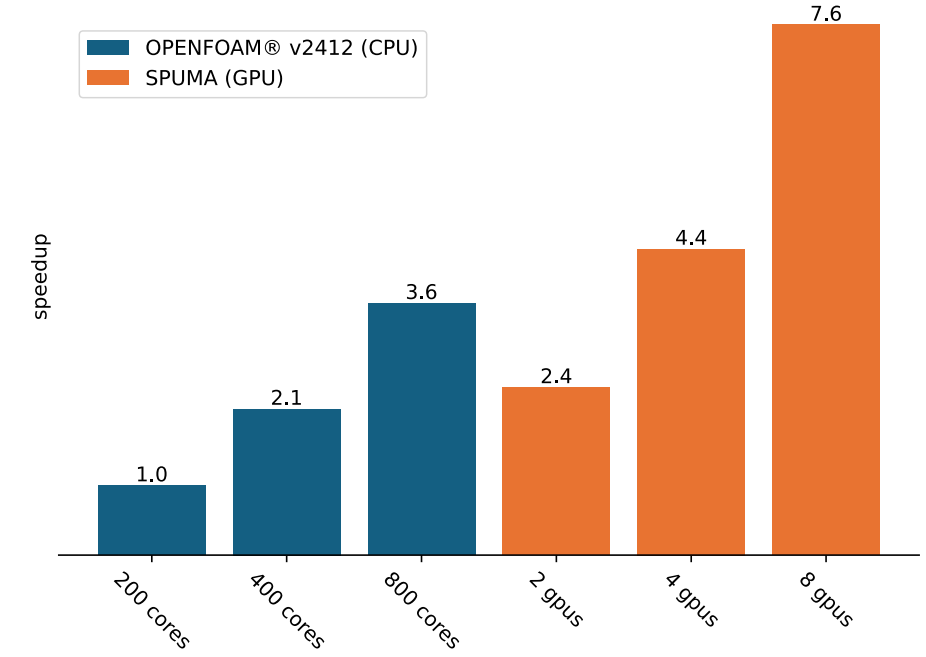
从前文介绍的算例中选取了三个：

- *occDrivAer65M，2 个 GPU，GAMG*：鉴于 GAMG模拟的强可扩展性仍远未达到最佳水平，该算例可视为本文介绍的效率最高的 GPU 算例。

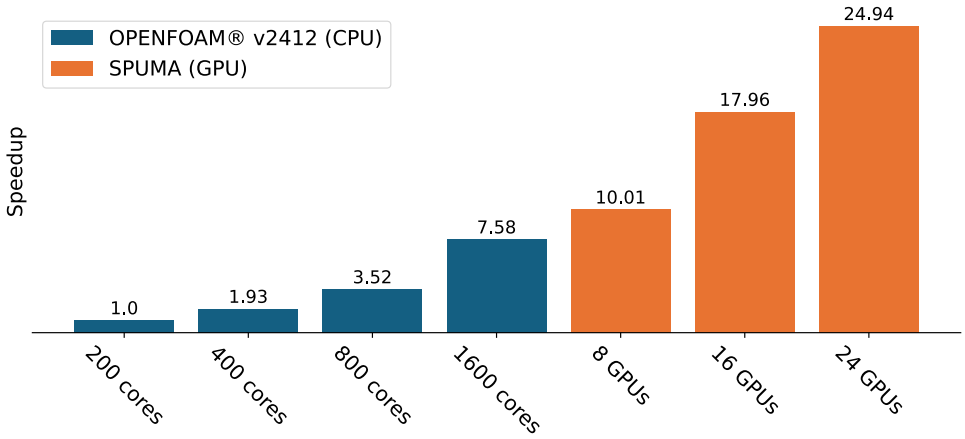
- *occDrivAer65M，4 个 GPU，GAMG*：从强可扩展性趋势来看，该算例的效率略低于前一个算例。不过，与 CPU 参考方案相比，该算例的求解时间更短。

- *occDrivAer65M，800 个核心，GAMG*：中央处理器端最快的算例。

运行所选测试算例时，对计算节点的能耗进行了监测。此项分析仅在 Leonardo 上进行，因为它



(a)



(b)

图 7：occDrivAer65M 测试算例（a）和 occDri-vAer235M 测试算例（b）的强扩展结果：相对于 200 核心中央处理器运行的加速比；SPUMA 使用 AMGX 作为压力求解器。

这使得作者所在机构的主要高性能计算系统更容易使用功耗监测工具。硬件功耗通过**perf**（一种 Linux 性能分析工具）测量中央处理器的能耗，并通过**nvidia-smi**（英伟达内置的系统管理接口，SMI）测量图形处理器的能耗。

关于所考察算例的求解时间，有一点值得注意：中央处理器算例比使用 2 个图形处理器的算例快约 25%。不过，在这项比较中，1 个图形处理器相当于约 301 个 CPU 核心。另一方面，使用 4 个图形处理器运行的算例比中央处理器基准算例快 24%，且 1 个图形处理器相当于 264 个 CPU 核心。这些结果列于表16。

	设备数量	能耗 [Wh]	热设计功耗 [W]	平均功耗 [W]	负载
OpenFOAM-v2412	16 个 CPU（800 个核心）	281.81	350	346.5	99%
SPUMA	2 个 GPU	43.21	440	258	59%
	1 个 CPU（32个核心）	10.67	250	127.3	51%
SPUMA	4个GPU	44.12	440	232	53%
	1个CPU（32个核心）	6.54	250	137.6	55%

表16：对 occDri-vAer65M 测试算例执行100次 SIMPLE 迭代时，CPU和GPU的能耗。TDP指热设计功耗，即芯片在典型高负载使用情况下能够产生的最大热量。

两个GPU算例显示，其综合能耗（同时考虑加速器和主机CPU节点）显著低于CPU算例。定量而言，运行该算例相比原始CPU执行可节省超过80%的能耗。显然，对于中小规模的模拟（例如本文展示的包含6500万个单元的模拟），CPU和GPU的求解时间可能相近；但切换到GPU对节能具有巨大影响。另一方面，考虑到求解时间和能耗，对于规模更大的分析，GPU执行应当是更好的选择。值得强调的是，在考虑能耗时，4个GPU的算例不仅速度更快，而且比2个GPU的算例效率更高。

SPUMA 目前尚未包含针对 GPU 的专门优化，但近期可能会加入。我们认为，这一因素进一步提升了本节所呈现的能耗分析结果的价值。另一方面，LUMI 的 EPYC Milan处理器在速度和扩展性方面均优于 Leonardo 的 Sapphire Rapids；要全面评估能耗与求解时间表现，需要更广泛的测试硬件

组合和深入研究，这超出了当前工作的范围。

5. 结论与未来工作

本文介绍的软件是计算流体力学领域的一项重要创新，使应用最广泛的开源模拟框架能够在现代异构硬件上运行，而这类硬件由中央处理器和 GPU 加速器组成。本文提出的主要特性可概括为以下三点：

- 透明的内存管理：引入内存池简化了中央处理器与相关加速器之间内存管理所涉及的复杂交互。通过开发由少量专用于内存相关操作的应用程序接口组成的接口，所有 OpenFOAM 开发者都能应对将代码新部分移植到加速器上的挑战，即使他们并非 GPU 编程专家。这是一项显著优势，因为公认的 OpenFOAM 优势之一，便是为其发展做出贡献的开源社区。

- 兼容的编程模型：可移植编程抽象的实现使 GPU 内核能够集中在少数专用类中。与内存管理类似，这种方法简化了移植过程。此外，它还大幅提升了代码的可维护性和升级潜力，因为引入或弃用后端只会影响少数几个类。

- 渐进式编程模型：如果内存池是通过统一内存技术分配的，那么移植过程不完整唯一造成的影响就是内存页迁移导致速度变慢，但模拟仍能提供正确结果。这样便可以逐步引入新功能，并使用性能分析工具找出速度变慢的原因，再替换导致迁移的代码段。

这三项特性使我们能够将 SPUMA 视为一个开发沙盒，让官方维护者和第三方开发者都能以此为起点开展图形处理器移植工作。

本文的发现并不局限于上述软件特性。通过在欧洲两台主要的超级计算机 Leonardo 和 LUMI 上模拟一个公认的工业测试案例 DrivAer，SPUMA 的潜力得以展现。DrivAer 测试的一些亮点包括：

- 在 Leonardo 上，一块 NVIDIA A100 的性能相当于 200 至 300 个 Intel Sapphire Rapids 内核，因此能效表现出色：使用 GPU 运行可节省 80% 以上的能耗。在 LUMI 上，一块 AMD MI250X 的性能相当于 100 至 150 个 AMD Epyc Milan 核心。随着人工智能市场的增长推动 GPU 技术迅速发展，我们有望在不久的将来看到更好的性能。

- 与 Leonardo 相比，OpenFOAM-v2412 和 SPUMA 在 LUMI 上均表现出更好的扩展性。不过，在两台超级计算机上，OpenFOAM 的 GAMG 实现运行于 GPU 时仍会带来一定的性能损失。NVIDIA AmgX 的效率更高，证实了这一点；在 Leonardo 上，其弱扩展性和强扩展性分析结果都显著更好。

这些结果是在未针对 GPU 硬件特性调整 OpenFOAM 的任何算法和数据结构的情况下取得的。未来的开发工作可着重改进多重网格求解器，以进一步提升性能，例如优化粗化算法、并行平滑器和 MPI 通信。其他重要工作还可包括采用循环融合技术、缓存数据和矩阵，以及总体上优先采用能够提高计算强度的操作。

在横向拓展方面——即扩展支持 GPU执行的 OpenFOAM 功能数量——我们计划纳入可压缩求解器和计算域接口，以及目前尚不支持的多相流和传热求解器。假设硬件相似且 GPU 内存超额订阅程度相当，这些求解器也有望取得与 DrivAer 测试算例相近的性能提升。此外，SPUMA 可视为一个对开发者友好的项目，这意味着广大的国际 OpenFOAM 社区将在进一步扩展 GPU 支持的功能方面发挥重要作用。

6. 资助与致谢

我们感谢 EuroHPC联合事业授予我们使用意大利 Cineca 的 Leonardo 和芬兰 CSC 的 LUMI 的权限，分配编号为 EHPC-DEV-2025D01-014。

本研究由 EuroCC 2 共同资助。EuroCC 2 获得了欧洲高性能计算联合事业（JU）根据资助协议第 101101903 号提供的资金，以及意大利企业与意大利制造部根据国家复苏与韧性计划（任务 4，组成部分 2——投资 2.2——研究与创新伙伴关系——欧洲地平线计划）提供的资金（ Progn.P/050001/03/X65）。JU 获得了欧盟数字欧洲计划以及德国、保加利亚、奥地利、克罗地亚、塞浦路斯、捷克、丹麦、爱沙尼亚、芬兰、希腊、匈牙利、爱尔兰、意大利、立陶宛、拉脱维亚、波兰、葡萄牙、罗马尼亚、斯洛文尼亚、西班牙、瑞典、法国、荷兰、比利时、卢森堡、斯洛伐克、挪威、土耳其、北马其顿共和国、冰岛、黑山、塞尔维亚、波斯尼亚和黑塞哥维那的支持。

本研究作为 exaFOAM项目的一部分开展，该项目获得了欧洲高性能计算联合事业（JU）根据资助协议第 956416 号提供的资金。JU 获得了欧盟地平线2020研究与创新计划以及法国、德国、西班牙、意大利、克罗地亚、希腊和葡萄牙的支持。

参考文献

[1]H. G. Weller、G. Tabor、H. Jasak、C. Fureby，采用面向对象技术的计算连续介质力学张量方法，Computer in Physics 12 (6) (1998) 620–631。arXiv:https://pubs.aip.org/aip/cip/article-pdf/12/6/620/7865493/620_1_online.pdf，doi:10.1063/1.168744。网址 https://doi.org/10.1063/1.168744

[2] https://top500.org/（2024年）。

[3] S. Bnà、I. Spisso、M. Olesen、G. Rossi，Petsc4foam：一个将 PETSc 接入 OpenFOAM 框架的库，PRACE白皮书（2020年）。

[4] S. Zampini、S. Bnà、M. Valentini、I. Spisso，使用 PETSc4FOAM 进行 GPU 加速的 OpenFOAM 模拟，载于：第八届 ESI-OpenFOAM 会议，2020年。

[5] M. Martineau、S. Posey、F. Spiga，面向 OpenFOAM 的 Amgx GPU 求解器开发，载于：第八届 ESI-OpenFOAM 会议，2020年。[6] M. Martineau、S. Bnà、S. Posey、F. Spiga，支持 GPU 求解器的 OpenFOAM，载于：第九届 ESI-OpenFOAM 会议，2021年。[7] G. Olenik、M. Koch、Z. Boutanios、H. Anzt，面向平台可移植线性代数后端的 OpenFOAM，Meccanica（2024年）。doi:10.1007/s11012-024-01806-1。[8] https://github.com/chegdan/cufflink-library，基于 CUDA C/C++ 与 OpenFOAM 连接数值方法的库。27-01-2016（2011年）。[9]J. Dyson，面向 OpenFOAM 的 GPU 加速线性方程组求解器及其在喷雾中的应用，博士论文，伦敦布鲁内尔大学机械、航空航天与土木工程系、工程、设计与物理科学学院，论文可见于 htt ps://bura.brunel.ac.uk/bitstream/2438/16005/1/FulltextThesis.pdf（2018年）。[10] A. Monakov，面向 OpenFOAM 的高效多 GPU CUDA 线性求解器，技术报告，俄罗斯科学院系统编程研究所（2013年3月20日）。[11] A. Monakov, https://github.com/amonakov/ispm-sparse-lib, spm-sparse-lib27-01-2016（2011年）。[12] A. Alonazi、D. E. Keyes、A. L. Lastovetsky、V. Rychkov，ParCFD 2014：面向混合异构 HPC 平台的基于 OpenFOAM 的 CFD 应用设计与优化，2018年。网址 https://api.semanticscholar.org/CorpusID:265096037[13] Z. Koza、A. Kosinor、M. Matyka、J. Pola、L. Miroslaw，使用 Speedit 技术进行 OpenFOAM 多 GPU 模拟，载于：第七届 OpenFOAM 研讨会，2012年。[14] Z. Jamshidi、F. Khunjush，新兴多核平台上 OpenFOAM 线性求解器的优化，载于：2011 年 IEEE 环太平洋通信、计算机与信号处理会议，2011年，第 824–829 页。doi:10.1109/PACRIM.2011.6033001。

[15] T. Rathnayake、S. Jayasena、M. Narayana，使用 AMGX 在 GPU 上运行 OpenFOAM，载于：第25届高性能计算研讨会论文集，HPC '17，国际计算机仿真学会，美国加利福尼亚州圣迭戈，2017年。[16] J. Williams、C. Sarofeen、H. Shan、M. Conley，面向非结构网格计算流体力学计算的 GPU 加速迭代线性求解器，《Procedia计算机科学》80（2016）1291–1300，2016年国际计算科学会议（ICCS 2016），2016年6月6日至8日，美国加利福尼亚州圣迭戈。doi:https://doi.org/10.1016/j.procs.2016.05.504。网址 https://www.sciencedirect.com/science/article/pii/S1877050916309917[17] T. Tomczak、K. Zadarnowska、Z. Koza、M. Matyka、Łukasz Mirosław，图形处理器上的完整 PISO 和 SIMPLE 求解器（2012）。arXiv:1207.1571。[18]K. Zadarnowska、Z. Koza、M. Matyka、L. Miroslaw，图形处理器上迭代纳维–斯托克斯求解器的加速，《国际计算流体动力学杂志》27（2013）201–209。doi:10.1080/10618562.2013.804178.UmUrmfl7KgY。[19] D. Jasiński，针对大规模并行GPU架构改造 OpenFOAM，载于：第三届OpenFOAM用户会议，斯图加特—德国，2015年。[20] S. Tandon、L. Grinberg、G.-T. Bercea、C. Bertolli、M. Olesen、S. Bnà、N. Malaya，利用统一内存和 OpenMP® 将高性能计算应用移植到 AMD Instinct™ MI300A，载于：ISC高性能计算2024研究论文集（第39届国际会议），2024年，第1–9页。doi:10.23919/ISC.2024.10528925。[21] J. H. Ferziger、M. Perić，《计算流体动力学方法》，第3版，施普林格，柏林，2002年。doi:10.1007/978-3-642-56026-2。网址 http://dx.doi.org/10.1007/978-3-642-56026-2[22]N. Jansson、M. Karp、A. Podobas、S. Markidis、P. Schlatter，Neko：一种现代、可移植且可扩展的高保真计算流体动力学框架，《计算机与流体》275（2024）106243。doi:https://doi.org/10.1016/j.compfluid.2024.106243。

[23] M. Kreutzer、G. Hager、G. Wellein、H. Fehske、A. R. Bishop，一种统一的稀疏矩阵数据格式，可在配备宽 SIMD 单元的现代处理器上高效执行通用稀疏矩阵-向量乘法，《SIAM科学计算杂志》36（5）（2014）C401–C423。doi:10.1137/ 130930352。[24] G. Goumas、K. Kourtis、N. Anastopoulos、V. Karakasis、N. Koziris，现代架构上的稀疏矩阵-向量乘法性能评估，《超级计算杂志》50（2009）36–77。doi:10.1007/s11227-008-0251-8。[25] https://docs.nvidia.com/cuda/（2024）。

[26] https://www.khronos.org/opencl/（2024年）。

[27] https://rocm.docs.amd.com/projects/HIP/en/latest/（2024年）。

[28] https://www.openmp.org/（2024年）。

[29] https://www.openacc.org/（2024年）。

[30]H. Carter Edwards、C. R. Trott、D. Sunderland，Kokkos：通过多态内存访问模式实现众核性能可移植性，《并行与分布式计算杂志》74 (12) (2014) 3202–3216，《面向高性能计算的领域专用语言和高级框架》。doi:https://doi.org/10.1016/j.jpdc.2014.07.003。网址 https://www.sciencedirect.com/science/article/pii/S0743731514001257

[31] G. Mudalige, M. Giles, I. Reguly, C. Bertolli, P. Kelly，《OP2：用于在多核和众核架构上求解基于非结构网格的应用的主动库框架》，载于：2012年创新并行计算（InPar），2012年，第1–12页。doi:10.1109/InPar.2012.6339594。[32] I. Z. Reguly, G. R. Mudalige, M. B. Giles, D. Curran, S. McIntosh-Smith，《OPS：用于多块结构网格计算的领域专用抽象》，载于：第四届领域专用语言与高性能计算高级框架国际研讨会论文集，WOLFHPC '14，IEEE Press，2014年，第58–67页。

[33] D. S. Medina, A. St-Cyr, T. Warburton, 《OCCA：一种统一的多线程语言方法》（2014年）。arXiv:1403.0968。网址 https://arxiv.org/abs/1403.0968[34] https://nvidia.github.io/cccl/thrust/（2024年）。[35] G. Malenza, 《使用现代 C 并行化技术获得的 OpenFOAM 性能分析》，硕士论文，SISSA（2022年）。++S. Bnà, F. Bottau, E. Fadiga, G. Giaquinta, T. Zanelli，https: // gitlab.hpc.cineca.it/openfoam/yoctoFOAM-herd/zeptoFOAM（2024年）。[36]D. Abrahams, A. Gurtovoy, 《C++ 模板元编程：来自 Boost 及其他领域的概念、工具与技术》（C++ 深入系列），Addison-Wesley Professional，2004年。[37] D. A. Beckingsale, M. J. McFadden, J. P. S. Dahm, R. Pankajakshan, R. D. Hornung, 《Umpire：面向应用的复杂分层内存管理与协调》，《IBM研究与开发杂志》64（3/4）（2020年）00:1–00:10。doi:10.1147/JRD.2019.2954403。++ M. Naumov, M. Arsaev, P. Castonguay, J. Cohen, J. Demouth, J. Eaton, S. Layton, N. Markovskiy, I. Reguly, N. Sakharnykh, V. Sellappan, R. Strzodka, 《AMGX：用于图形处理器加速代数多重网格和预条件迭代方法的库》，《SIAM科学计算杂志》37（5）（2015年）S602–S626。doi:10.1137/140980260。网址 https://doi.org/10.1137/140980260++ R. D. Falgout, U. M. Yang, 《HYPRE：高性能预条件器库》，载于：P. M. A. Sloot, A. G. Hoekstra, C. J. K. Tan, J. J. Dongarra（编），《计算科学 — ICCS 2002》，Springer Berlin Heidelberg，柏林，海德堡，2002年，第632–641页。[38]R. T. Mills, M. F. Adams, S. Balay, J. Brown, A. Dener, M. Knepley, S. E. Kruger, H. Morgan, T. Munson, K. Rupp, B. F. Smith, S. Zampini, H. Zhang, J. Zhang, 《面向基于 GPU 的百亿亿次级系统，推动 PETSc 实现性能可移植性》，《并行计算》108（2021年）102831。doi:https://doi.org/10.1016/j.parco.2021.102831。

网址 https://www.sciencedirect.com/science/article/pii/S016781912100079X

[42] H. Anzt, T. Cojean, G. Flegar, F. Göbel, T. Grützmacher, P. Nayak, T. Ribizel, Y. M. Tsai, E. S. Quintana-Ortí，Ginkgo：面向高性能计算的现代线性算子代数框架，ACM数学软件汇刊 48 (1)（2022年2月）。doi:10.1145/3480935。网址 https://doi.org/10.1145/3480935

[43] https://github.com/ROCm/rocALUTION（2024年）。

[44] https://www.paralution.com/（2024年）。

[45] M. Bernaschi, P. D'Ambra, D. Pasquini，BootCMatchG：一种适用于GPU的自适应代数多重网格线性求解器，Software Impacts 6（2020年）100041。doi:https://doi.org/10.1016/j.simpa.2020.100041。网址 https://www.sciencedirect.com/science/article/pii/ S2665963820300324

[46] G. Isotton、M. Frigo、N. Spiezia、C. Janna，Chronos：一款面向高性能计算的通用经典AMG求解器，《SIAM《科学计算杂志》》43 (5) (2021年) C335–C357。arXiv:https://doi. org/10.1137/21M1398586, doi:10.1137/21M1398586。网址 https://doi.org/10.1137/21M1398586

[47] S. Ganesan、M. Shah，SParSH-AMG：一个用于混合中央处理器-图形处理器代数多重网格和预条件迭代方法的库（2020年6月）。arXiv: 2007.00056, doi:10.48550/arXiv.2007.00056。

[48] A. Heft、T. Indinger、N. Adams，《DrivAer模型的实验与数值研究》，载于《流体工程分部夏季会议论文集》，美国机械工程师学会，第44755卷，美国波多黎各里奥格兰德，2012年7月8日至12日，第41–51页。doi:10.1115/FEDSM2012-72272。[49] https://github.com/OpenFOAM/OpenFOAM-12/tree/master/tutorials/incompressibleFluid/drivaerFastback.[50] https://develop.openfoam.com/committees/hpc/-/tree/develop/incompressible/simpleFoam/occDrivAerStaticMesh。

[51] L. Berger-Vergiat、B. Kelley、S. Rajamanickam、J. Hu、K. Swirydowicz、P. Mullowney、S.Thomas、I. Yamazaki, 《GPU集群上克雷洛夫求解器的两阶段高斯–赛德尔预条件器和平滑器》（2021年）。arXiv:2104.01196。网址 https://arxiv.org/abs/2104.01196

[52] F. C. C. Galeazzo, M. Garcia-Gasulla, E. Boella, J. Pocurull, S. Lesnik, H. Rusche, S. Bnà, M. Cerminara, F. Brogi, F. Marchetti, D. Gregori, R. G. Weiß, A. Ruopp, 《不同高性能计算架构上的计算流体力学微基准测试性能比较》, Computers 13 (5) (2024年)。doi:10.3390/computers13050115。网址 https://www.mdpi.com/2073-431X/13/5/115

附录 A. 仿真设置

验证运行中使用的线性代数求解器见表 A.17。清单 3 列出了线性代数求解器的设置、SIMPLE

方程	求解器	预条件器/平滑器	容差	相对容差
p	GAMG	Richardson	1e-9	1e-3
p	PCG	对角	1e-9	1e-3
p	AmgX	雅可比	1e-9	1e-3
U, k, ω	预条件双共轭梯度法	代数 DILU	1e-8	1e-3

表 A.17：线性求解器设置——验证运行。

算法条目和松弛因子（OpenFOAM 中的 fvSolution 字典）。清单 4 包含离散格式（OpenFOAM 中的 fvSchemes 字典）。

```
求解器
{


   PCG-AMG-X雅可比
   {
      求解器                    AMGX;
      矩阵类型                  压缩稀疏行;
      数据位置 设备;
      mode  dDDI;
      AmgXconfig
      {
```

```
12 config_版本 2：13 determinism_标志 1：14 求解器 15 { 16
预
求
24
sol
SIZ
层
37
件
43
pr
49
CC
差
```

57 } 58 } 59 60 pcgDiag 61 { 62 求解器

PC
容差
次数
GA
09;
77
pG
pcg
nu
代数
94
97

```
102 残差控制 103 { 104 p 1e -02; 105 U 1e -03; 106 " ( k | omega | epsilon ) " 1e -03;
107 }
113 1
123 {
mega
```

清单 3：验证算例 - fvSolution。

```
1    时间导数离散格式2    {3        默认 稳态 :4    }56    梯度
```

离
1;1
lin

```
{
    默认 无 ;
    散度 ( phi ,U ) 有界 高斯线性迎风格式 梯度 ( U ) ;
    散度 ( phi ,U ) 有界 高斯线性迎风格式 梯度 ( U ) ;
}
```

清单 4：验证算例 - fvSchemes

可从 [50], 获取可扩展性测试所用的设置，但线性求解器（表 A.18）和对流项离散格式（使用 linearUpwind 而非 linearUpwindV）除外。

方程	求解器	预条件器/平滑器	容差	相对容差
p	GAMG	Richardson	1e-7	0.1
U, k, ω	预条件双共轭梯度法	代数 DILU	1e-6	0.1

表 A.18：线性求解器设置——可扩展性测试。

z