（2022 年），
1–6

Durham University

BGCE 学生论文奖

通过 C++ 标准算法在 ExaHyPE 中实现图形处理器卸载

Uzmar Gomez，[1] Gonzalo Brito Gadeschi，[2] 和 Tobias Weinzierl[1]

[1]杜伦大学计算机科学系，英国[2]英伟达，德国慕尼黑 通讯作者：
UzmarGomez，电子邮箱：uzmar.gomez@hotmail.com

arXiv:2302.09005v1 [cs.MS] 17 Feb 2023

摘要
ISO C++17 标准引入了并行算法，这是一种并行编程模型，旨在实现跨多种并行硬件的可移植性，包括多核CPU、GPU 和 FPGA。自 2019 年以来，NVIDIA HPC SDK 编译器套件便支持在多核CPU和GPU上使用这一编程模型。ExaHyPE 是用于研究复杂波动现象的双曲型偏微分方程求解器引擎。它支持多种数值方法，包括有限体积法和 ADER-DG，并采用自适应网格细化与基于空间填充曲线的动态负载均衡，以及基于任务的并行和 GPU 卸载。本研究将 ExaHyPE 中针对有限体积块的任务移植到 ISO C++ 并行算法编程模型，并将其性能和易用性与通过 OpenMP target指令实现卸载的 OpenMP 版本进行比较。结果表明，ISO C++ 对于像我们这种基于任务的自适应网格细化代码这样的非平凡应用而言，是一种可行的编程模型。该实现不依赖特定厂商的扩展，也不依赖非 C++ 扩展。不过，其速度比对应的 OpenMP 版本慢。

1. 引言

图形处理器（GPU）已成为驱动百亿亿次级系统的数据并行任务的主要加速器[1]。然而，将应用移植到 GPU 并设计 GPU 算法仍然颇具挑战。GPU 加速系统的主要高性能计算编程范式包括可在 GPU 上“自动”运行的 ISO 标准语言（C ++ 和 Fortran）、OpenMP 和 OpenACC 等基于注释的编程模型，以及 CUDA 或 SYCL 等 GPU 编程语言或扩展。这些不同方法各有优缺点。

NVIDIA HPC SDK 支持将 ISO C++ 并行算法 自动卸载到多核 CPU 和 GPU [2]。近期发表的研究展示了这些方法在迷你应用程序中的良好性能 [3]，而英伟达的文档列举了在大型应用程序中取得成功的案例，例如多物理场应用程序 m-AIA [4]和 STLBM 。

本研究旨在考察使用纯 ISO C++，以可移植的方式将大型应用并行化并面向多核 CPU 和多 GPU 系统的可行性。ExaHyPE 是一种用于复杂波动现象的双曲偏微分方程求解器引擎 [6] 。这是一个大型且复杂的应用程序，支持通过空间填充曲线实现动态负载均衡的自适应网格细化。在用于 CPU 的 MPI+X 并行性的基础上，它可以将各个单元的更新表示为任务：我们仅关注 ExaHyPE 有限体积求解器在分块结构 AMR 环境下针对小型笛卡尔网格块的计算内核 [7]——这些内核嵌入单元中，因此构成任务——并且已有相应的 OpenMP 卸载版和 SyCL 版本。更复杂的卸载策略，尤其是将执行逻辑移植到 GPU 上，不在本文讨论范围之内。

本工作的主要贡献是探索如何将各项任务映射到新的 ISO C++并行算法编程模型，并比较该编程模型与 OpenMP 在开发效率、易用性和性能方面的差异。我们还报告了在使用这种新颖的编程模型将 ISO C++ 卸载到 GPU 时遇到的技术挑战。

本文的组织结构如下。第2 节介绍问题域、数值格式以及 ExaHyPE 软件。第3 节讨论核心计算内核 C++ 的实现、图形处理器执行策略及实现挑战。第4 节展示初步结果，并据此得出结论。

2. ExaHyPE 中的块结构有限体积法

ExaHyPE 是一个用于编写双曲型一阶方程组仿真代码的引擎，其作用于函数 $\vec{q}(\vec{x}, t) = (q1, \ldots, q_s)$

$$\frac{\partial \vec{q}}{\partial t} + \sum_{j=1}^{d} \frac{\partial}{\partial x_j} \vec{f_j}(\vec{q}) = 0, \tag{1}$$

其中 $\vec{x} \in \mathbb{R}^d$ 以及方向项 f_j 构成通量函数。在 ExaHyPE 中，用户通过注入函数实现来采用这类通用偏微分方程表述，这些函数用于计算 $f_j(\vec{q})$。这些函数求值如何在数值格式中使用、在哪个核心上执行以及何时执行，对用户而言都是隐藏的。这就是 [8]所述的好莱坞原则。在本研究的实验中，我们对欧拉方程进行了基准测试。其守恒形式如下：

$$\frac{\partial}{\partial t} \begin{pmatrix} \rho \\ \vec{j} \\ E_t \end{pmatrix} + \nabla \cdot \begin{pmatrix} \vec{j} \\ \frac{1}{\rho} \vec{j} \otimes \vec{j} + pI \\ (E_t + p) \frac{1}{\rho} \end{pmatrix} = \vec{0}, \tag{2}$$

其中 $\vec{j} = \rho \vec{u}$ 是动量密度， ρ 是流体质量密度，p 是压力，E_t 是总能量密度。

ExaHyPE 的空间离散是固定的：代码采用由空间树 [8] 产生的块结构网格 [9], ： 一个立方体计算域沿每个坐标轴分割成三个块，然后在局部递归地重复这一过程。最终得到一个动态自适应网格

立方体网格块。在此空间离散的基础上，ExaHyPE 提供多种数值格式。这里我们采用块结构有限体积法：网格中的每个立方体都包含一个由 $p \times p$ 或 $p \times p \times p$ 个体单元组成的小型笛卡尔网格块。每个网格块都配有一个光晕层。

维护、存储并遍历块结构网格是 ExaHyPE 的职责。对于每个笛卡尔网格块，代码都会调用一个更新内核，使该网格块随时间向前推进。用户提供的通量函数会根据所关注的偏微分方程对该内核进行定制。之后，确保所有（halo）数据保持一致则是 ExaHyPE 的职责。

并行化与 *GPU* 卸载

ExaHyPE 通过 MPI 实现传统的非重叠域分解。随后，每个 MPI 进程内也会再次进行非重叠域分解。由此得到的子域会分配给各进程中的线程。

每个线程在每个时间步遍历其子域，并推进数值解。在遍历过程中，代码会优先处理 AMR 边界或子域边界上的网格块：遍历域时会立即更新这些网格块，因为它们的输出十分紧迫——这些输出既会用于域边界交换，也可能涉及开销较大的 AMR 例程。

其余网格块会暂存在线程本地缓冲区中。当该缓冲区超过给定阈值 N 时，我们会取出 N 个网格块，并将它们作为一个整体发送到加速器。遍历结束时，其余网格块都会作为常规任务派发，以充分利用主机核心。对网格块进行分类实现了域内任务调度的原则 [10], , 而缓冲则采用了 [7] 的概念。

通用有限体积内核

对于每个 $\mathcal{C}_i$ 中的每个 $p \times p \times p$ 网格块，我们对式（1）中的时间导数采用有限差分近似，并使用有限体积格式进行空间离散：

$$\frac{Q_i^{n+1} - Q_i^n}{\Delta t} + \frac{F_{i+1/2}^n - F_{i-1/2}^n}{\Delta x} = 0, \tag{3}$$

其中 $F_{ni-1/2}$ 表示穿过 $\mathcal{C}_i$ 的各个面上的通量。随后，我们对式（3）中的通量采用 Rusanov 近似：

$$F_{i-1/2}^n = \frac{1}{2}\left[f\left(Q_{i-1}^n\right) + f\left(Q_i^n\right) - a_{i-1/2}\left(Q_i^n - Q_{i-1}^n\right)\right] \tag{4}$$

其中

$$\begin{aligned} a_{i-1/2} &= \max\left(\left|f'(q)\right|\right) \text{ over all } q \text{ between } Q_{i-1}^n \text{ and } Q_i^n \\ &= \max\left(\left|\lambda_k\right|\right). \end{aligned} \tag{5}$$

λ_k 是偏微分方程雅可比矩阵的特征值。

借助这一通用数值格式，我们可以实现一个专门针对（2）的通用网格块更新内核，方法是将感兴趣的量的数量设为 $d + 2$，并注入两个例程：一个用于计算欧拉通量，另一个用于返回最大特征值。

算法 1 计算一个网格块的步骤。一次处理多个网格块的内核会增加一个额外的外层循环。

1 ^{new} = ^{old}: QQ：计算 $a_{i-1/2}Q^{old}$，其中 $a_{i-1/2}$ 由公式 5 给出。3：2
对 Q^{old}边界上的每个面，计算 $a = \max(\lambda_{max}Q^{old})$。4：通过 a $(Q^{old} - Q^{old-1})$更新体积。5：计算公式（4）给出的通量。6：计算 Q^{old} 与相邻体积之间通量的平均值。7：使用 $^{\Delta t}/_{\Delta x}$进行缩放，并利用其二维相邻面上的通量更新体积。8：计算有限体积的非保守乘积。9：更新 Q^{new}的值。10：计算网格块的全局最大特征值。

—

3. 一个 C++图形处理器计算内核

ExaHyPE 提供基于函数对象的通用计算内核实现，但我们也提供一个模板版本，可在一批中处理N 个网格块。它由一系列嵌套循环组成（算法 1）。我们遍历所有网格块。对于每个网格块，我们执行不同的步骤；每个步骤分别包含遍历单元或面的循环，以及偏微分方程的$s = 5$ 个未知量。

```
// 自动检测笛卡尔积类型using cart_prod_3d  =decltype(tl::views::
```

```
删除 [] 二维范围_单元格;
```

代码 1. 在堆上分配笛卡尔积（迭代空间）。这样，它就可以供图形处理器使用。

*SIMD*循环遍历和函数对象

```
std::for_each(std::execution::par_unseq,range2d_cell->
```

```
  内部::copySolution_LoopBody <求解器 >( QIn [网格块索引],
```

```
);
```

```
  [...]
  // 算法蓝图中的步骤（2 - -10）
}
);
```

代码 2. 将标准`std::for_each`算法卸载到图形处理器。

原始内核实现使用虚函数动态分派和单独编译的实现——即放在独立的 .cpp 文件中——以缩短编译时间并提高开发人员的工作效率。然而，英伟达对 ISO C++ 的实现不支持设备代码中的虚函数分派和分离编译。因此，我们在 ExaHyPE 中用定义在头文件中的模板函数和静态分派（代码 2）取代虚函数分派。对于（算法 5）中的最大值计算，我们用手动比较取代标准 C++ 函数调用。

虽然英伟达的实现支持通过非标准的类 CUDA`主机__ __设备__`注解进行分离编译，但这要求程序员在使用非标准扩展与将实现细节移至头文件之间做出选择，而后者会显著增加编译时间。手动实现 maximum 并非“真正的 C++”。

使用笛卡尔积的多维迭代空间

高效的 GPU 内核需要由折叠循环构成的大型迭代空间，以充分利用现代加速器所提供的全部并行性。因此，我们在代码++ 1 中构建了一种 C 并行算法，用于折叠多个`for`循环。它使用 C1 中的`std::views::iota`结构，并结合 C++23 的`std::views::cartesian_product`。该实现接收指向主机线程栈的指针。然而，根据 HPC SDK 文档，在本研究所用的硬件平台上，GPU 内核只能访问主机堆。因此，我们的代码 [11]，1 必须在堆上分配底层笛卡尔积。

由于本研究使用的 C++ 标准库中没有笛卡尔积的实现，我们必须加入一个初步实现，而该实现并未针对加速器进行调优。将其置于堆上纯粹是出于技术限制。

数据传输

ISO C++ 编程模型依赖统一内存，使加速器设备能够访问程序的全部内存。英伟达的实现不符合非一致性硬件平台上的规范，例如本研究所用的平台：设备仅允许访问主机堆内存。虽然 ExaHyPE 的数据主要存储在堆上，但指向这些数据的指针会方便地汇集在`hostPatchData`等结构体中，而这些结构体通常存储在主机的栈上。设备代码不允许访问这些值以加载指向堆数据的指针。因此，ExaHyPE 会在传递给并行算法的 Lambda 表达式中显式捕获这些指针。启动内核时，Lambda 捕获的内容会复制到 GPU 上；而这些指针所指向的堆数据，则会在 GPU 线程访问时按需迁移。数据会留在 GPU 上，直到 CPU 再次访问，此时数据会自动迁回。目前没有可移植的 C++ API 能够在主机和设备之间预取堆数据。

软件栈

我们的工作使用 NVIDIA HPC SDK 22.3，该 SDK 借助系统的 C++ 标准库（`libstdc++`），但采用基于 Thrust 库的自有 C++并行算法实现。许多高性能计算系统默认提供的 libstdc++版本不够新，因此我们必须通过编译器标志手动指定其他版本。我们报告了一些确实存在的`nvc++`错误，例如 [12]，这些问题将在未来版本中修复。

4. 结果

图 1 展示了我们的计算内核在英伟达 2080TI 上以二维设置处理$p = 17$时，针对不同补丁集大小N的基准测试结果。所有计时均包含主机与设备之间的全部数据传输。对于N个网格块，内核既有逐补丁实现（外层笛卡尔循环遍历网格块），也有批处理版本（按顺序执行算法 1的各个步骤，并在每一步中循环遍历所有网格块、所有体单元和所有未知量）。

两种内核变体都利用了式（4）中的对称性：面通量会进入左、右相邻体单元的解中。我们按体单元计算f（以及特征值），并针对左、右相邻单元，分别冗余地计算式（4）中这些值的线性组合。中间的f和 λ 值可以采用结构体数组（AoS）、数组结构体（SoA）或数组的结构体数组（AoSoA）的形式存储。

批处理形式与逐补丁形式的区别源于笛卡尔积中循环顺序的置换（代码 1）。然而，我们的自适应网格细化代码采用局部自适应时间步进，因此每次内核调用都需要额外输出每个网格块的最大特征值。我们需要进行逐补丁归约。为避免数据竞争，这种归约通过顺序`std::for_each`循环完成，但也可以使用另一种标准算法或原子内存操作来并行化。

结果表明，对于 C++ 实现，批处理变体始终比逐补丁变体更快。将一个算法拆分为多个步骤，并为每个步骤使用较大的迭代空间，是有利的。随着卸载的网格块数量增加，有限体积更新所需的时间会减少，因此我们可以认为，GPU 内核能够很好地将两个步骤之间的数据保留在图形处理器上。计算过程中临时数据的数据布局对性能的影响有限。

对于 OpenMP，我们发现逐个处理网格块可能更具优势。我们假设在这种情况下，不同网格块最终会运行在图形处理器的不同流式多处理器上，并且彼此之间基本独立。总体而言，OpenMP 移植版本明显更快——对于较小的N值尤其如此——比 C++ 移植版本更快。NSight 分析表明，这得益于定制的 OpenMP 内存管理：数据卸载时，我们可以通过设备指针进行显式控制。相比之下，C++版本的实现中内置了按需内存传输。

虽然我们的软件已准备好支持多个任务同时将工作卸载到图形处理器上，但 C++ 和 OpenMP 均不支持此功能。两者都出现了段错误。

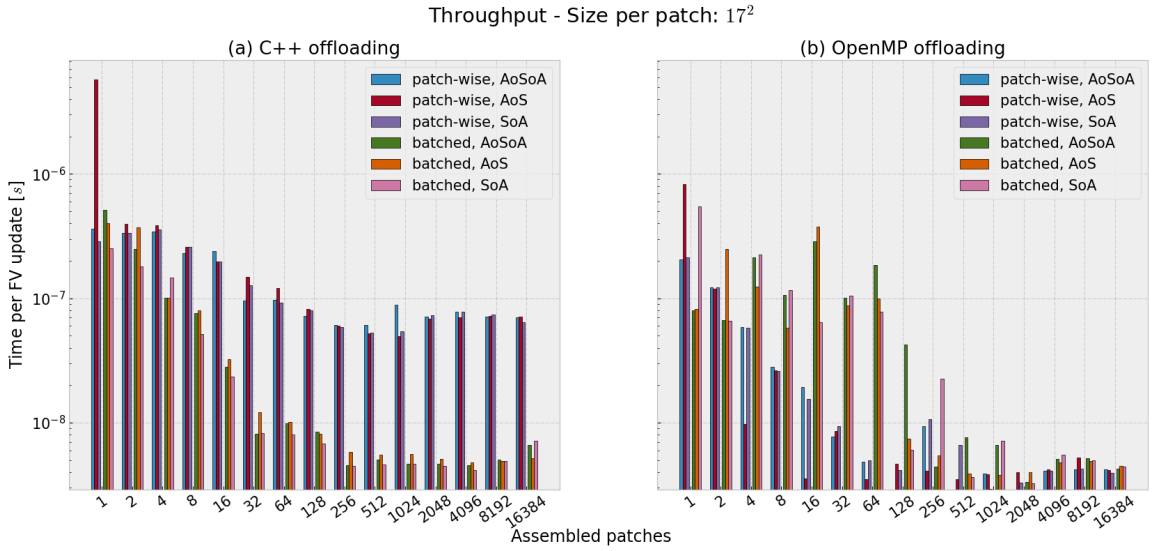


图 1. 不同实现变体以及每次内核调用处理的网格块数量 N 下，每次有限体积更新所需的时间。

5. 展望与结论

原则上，通过 C++ 标准算法中的 C++ 执行策略进行 GPU 卸载，对于像我们的基于任务的自适应网格细化代码这样的非平凡测试案例是可行的。最终实现没有依赖特定的（OpenMP）扩展，代码也很简洁；不过，这只是主观评价。就性能而言，C++ 移植版比对应的 OpenMP 版本慢。很难确定这种表现的原因。二维欧拉方程的算术强度很低。对于 ADER-DG ExaHyPE 求解器等计算更复杂的情况，性能差异或许会消失；这一现象有待未来研究。

现在评判 C++GPU 卸载的优缺点还为时尚早。除了显而易见的磨合问题外，我们的工作还凸显了开发者面临的一项特殊挑战：尽管 C++ 在语法和语义上可能正确，但其 GPU 数据传输仍可能失败（例如，数据并未“人为地”分配在堆上）；或者代码因按顺序执行而符合要求，却违背了程序员将计算卸载到加速器上的初衷。此外，C++社区需要一种可移植的替代方案，取代英伟达特有的、用于分离编译的 __主机__ __设备__ 注解。理想情况下，面向 GPU 的多遍编译或链接时编译将使这些注解不再必要。

作为这种自动化方式的替代方案，C++ 语言可以在现有执行策略的基础上，补充声明式 GPU 卸载策略。采用此类策略后，如果循环体不适用于 GPU，那么原本正确的循环体也会变得无效。目前尚不清楚如何在语言中妥善体现这一点，或者是否应完全由编译器在整个转换过程中处理。

参考文献

[1] Volodymyr V. Kindratenko、Jeremy J. Enos、Guochun Shi、Michael T. Showerman、Galen W. Arnold、John E. Stone、James C. Phillips 和 Wen-mei Hwu。面向高性能计算的 GPU 集群。发表于 *2009 年 IEEE 集群计算与研讨会国际会议*，第 1–8 页，2009 年。[2] David Olsen。使用 PGI C++17 并行算法在 NVIDIA GPU 上进行 C++。发表于 *GPU 技术大会*，2019 年。

[3] Wei-Chen Lin、Tom Deakin 和 Simon McIntosh-Smith。在异构高性能计算系统上评估 ISO C++ 并行算法。载于第 13 届 IEEE 高性能计算系统性能建模、基准测试与仿真国际研讨会，2022 年。[4] 欧洲百亿亿次计算卓越中心。百亿亿次级人工智能与仿真工程研究。https://www.hpccoe.eu/2021/ 06/04/m-aia/，2021 年。最后更新：2022 年 9 月 1 日。[5] Jonas Latt、Christophe Coreixas 和 Joël Beny。面向众核格子玻尔兹曼模拟的跨平台编程模型。PLOS ONE，16(4):e0250306，2021 年 4 月。[6] Anne Reinarz、Dominic E. Charrier、Michael Bader、Luke Bovard、Michael Dumbser、Kenneth Duru、Francesco Fambri、Alice-Agnes Gabriel、Jean-Matthieu Gallard、Sven Köppel、Lukas Krenz、Leonhard Rannabauer、Luciano Rezzolla、Philipp Samfass、Maurizio Tavelli 和 Tobias Weinzierl。ExaHyPE：用于并行动态自适应波问题模拟的引擎。Computer Physics Communications，254:107251，2020 年。[7] Han Zhang、Tobias Weinzierl、Holger Schulz 和 Baojiu Li。修正引力中的碰撞气体球形吸积 I：自相似解与一种新的宇宙学流体动力学代码。皇家天文学会月刊，515(2):2464–2482，2022 年 7 月。[8] T. Weinzierl。Peano 软件——基于并行自动机的动态自适应网格遍历。ACM 数学软件汇刊，45(2):14，2019 年。[9] A. Dubey、A. S. Almgren、J. B. Bell、M. Berzins、S. R. Brandt、G. Bryan、P. Colella、D. T. Graves、M. Lijewski、F. Löffler、B. O’Shea、E. Schnetter、B. van Straalen 和 K. Weide。块结构自适应网格细化软件包中高级框架综述。Journal of Parallel and Distributed Computing，74(12):3217–3227，2016 年。[10] Dominic Etienne Charrier、Benjamin Hazelwood 和 Tobias Weinzierl。动态自适应网格上间断伽辽金方法的域内任务调度。SIAM 科学计算杂志，42(3):C69–C96，2020 年。[11] NVIDIA HPC 编译器。C++ 并行算法。https://docs.nvidia.com/hpc-sdk/compilers/c++-parallel-algorithms/index.html，2022 年。[12] NVIDIA 错误报告。https://nvbugs/3706250，2022 年。

附录1. OpenMP 参考实现

我们的 OpenMP 原始版本在主机上的一个结构体中存放 N 个网格块（单元数量）的数据。这里的数据包括实际的有限体积数据，以及单元（网格块）的大小和位置等元数据。

OpenMP 变体会在内核调用之前，将这些数据显式映射到图形处理器上（代码3）。类似地，对各个网格块进行循环，可将每个网格块的有限体积数据传输过去（此处未显示）。

```
#pragma omp target teams distribute parallel for simd
      collapse (3) device( targetDevice )
用于 (整数  网格块索引 =0; 网格块索引 <单元数量;
      网格块索引 ++)
对于 (整数 y   = 0; y  < 每个网格块轴上的体单元数; y++)
对于 (整数 x   = 0; x  < 每个网格块轴上的体单元数; x++) {

     图形处理器单元数据 补丁数据;
     补丁数据. 单元数量                    = 单元数量;
     补丁数据 . 单元中心                   =(tarch ::la:: 向量 <
      维度, 双精度浮点数 >*)(rawPointerToCellCentres);
     补丁数据 . 单元尺寸 =(tarch ::la:: Vector <
      维度, 双精度浮点数 >*)(rawPointerToCellSizes);
     补丁数据 .QIn = mappedPointersToQIn ;
     补丁数据 .QOut                        = mappedPointersToQOut ;
     补丁数据 .t                           = t;
     补丁数据 .dt                          = dt;
     补丁数据 . 最大特征值                 = 最大特征值;

     [.....]

     copySolution_循环体 <求解器 >(
          补丁数据.QIn [网格块索引 ],
          QIn枚举器,
          补丁数据.单元中心 [网格块索引],
          补丁数据 . 单元尺寸 [网格块索引 ],
          网格块索引,
          二维体积索引 (x,y),
          补丁数据 .t[网格块索引 ],
          补丁数据 .dt [网格块索引],
          补丁数据 .QOut [网格块索引 ],
          QOut枚举器
     );

     [...]
}
```

代码 4。 OpenMP 代码的主循环体。等同于代码2。

我们的 OpenMP 参考实现通过纯嵌套 for 循环实现了嵌套的笛卡尔迭代空间（代码 4）。它们通过 pragma 合并。从这个意义上说，这种实现更接近 C，而非其 C++GPU 卸载对应实现。

在循环体中，代码使用设备指针手动重建主机端结构体，即在图形处理器上创建一个对应的结构体，并手动填入经过转换的图形处理器指针。另一种实现方式是为承载 N 个网格块数据的结构体添加 OpenMP 映射。

函数 `copySolutionAndAddSourceTerm_LoopBody` 以及算法 1中的其他计算步骤实现均为模板，因此定义在头文件中。因此，编译器可以直接将它们内联，并根据上下文推断出需要将它们转换为图形处理器代码。

附录 2. 测试环境

我们提供了一个 Docker 镜像，用于复现本文中获得的结果。该定制镜像基于最新的 HPC SDK 编译器，并包含一些 Python 库以及运行 ExaHyPE 及其底层 Peano AMR 框架所需的其他软件。如果要在图形处理器上运行该软件，请使用以下命令拉取镜像并启动 Bash 终端

```
docker run --gpus all -it ghcr.io/ uzmargomez/peano: hpcsdk22 .9 bash
```

如果无法访问图形处理器，则必须省略 `--gpus all` 标志。接下来，我们在容器的终端中运行以下命令

```
cd / git clone https :// gitlab .lrz.de/ hpcsoftware /Peano.git && cd Peano &&
```

如果图形处理器不可用，HPC SDK 编译器可以使用多核 CPU 并行化标准算法。在这种情况下，图像的应用基准测试仅测试 CPU 内核的性能。为此，我们将标志`-stdpar=gpu`更改为`-stdpar=multicore`。

```
const int numberOfCells = hostPatchData . numberOfCells ; double * rawPointerToCellCentres = hostPatchData . 单元中心 [0]. data (); double *
```

代码 3. OpenMP 设备上变量的分配与释放 .