OpenMP Advisor

Alok Mishra 计算机科学
石溪大学 纽约州石溪，美国
a
l
mishra@cs.stonybrook.edu

Abid M. Malik 计算机科学计划 布鲁克海文国家实验室
纽约州厄普顿，美国
amalik@bnl.gov

Meifeng Lin 计算机科学计划 布鲁克海文国家实验室
纽约州厄普顿，美国
mlin@bnl.gov

Barbara Chapman 计算机科学
石溪大学 纽约州石溪，美国
b
ar
bara.chapman@stonybrook.edu

arXiv:2301.03636v1 [cs.DC] 9 Jan 2023

摘要

随着高性能计算行业中异构架构日益多样化，将遗留应用程序移植到不同架构上运行已成为一项艰巨挑战。本文介绍了 OpenMP Advisor，这是首个利用机器学习和 OpenMP 将代码卸载到 GPU 的同类编译器工具。尽管该工具目前仅支持 GPU，但可以扩展为支持其他支持 OpenMP 的设备。该工具有两种模式：训练模式和预测模式。训练模式必须在目标硬件上执行。它以基准测试代码为输入，生成并执行所有可能在目标设备上运行的代码变体，然后收集所有已执行代码的数据，以训练供预测模式使用的基于机器学习的成本模型。然而，在预测模式下，该工具无需与目标设备进行任何交互。它接受 C 代码作为输入，并返回可用于将代码卸载到指定设备的最佳代码变体。该工具通过基于机器学习的成本模型预测内核的运行时间，从而确定最适合卸载的内核。该工具的主要目标是保持 OpenMP 的可移植性。借助我们的顾问工具，我们为七种不同架构上的多个应用程序生成了代码，并正确预测了每个应用程序在每种架构上排名前十的最佳变体。初步研究结果表明，该工具可帮助编译器开发人员和高性能计算应用研究人员将遗留高性能计算代码移植到即将到来的异构计算环境中。

关键词： OpenMP、图形处理器、机器学习、成本模型、Clang

1 引言

自所谓“摩尔定律的终结” [38] 以及从速度不断提升的单芯片转向多核处理器以来，已经过去了二十多年。然而，鉴于芯片上的晶体管数量不断增加，可以说摩尔定律尚未彻底失效；真正的问题是，晶体管的运行速度已无法持续提升。这是丹纳德缩放定律 [7], 所导致的结果。该效应与晶体管尺寸的缩小同时发生。丹纳德缩放定律的主要结果是，随着晶体管尺寸减小，功率密度仍保持不变。丹纳德发现，由于电压和电流随特征尺寸成比例缩放，而功率随面积成比例缩放，每瓦性能大约每两年翻一番。因此，问题并不在于无法蚀刻出更小的晶体管，而在于无法降低晶体管正常工作所需的电压和电流。

高性能计算行业的开发者开始超越摩尔定律着眼未来，而硬件开发者则通过配置越来越多的计算核心来提升芯片性能。架构硬件的快速变化要求更新编程语言并修改应用程序代码，以充分利用这些核心，例如在源代码中插入 POSIX 线程或 OpenMP [6] 结构。高性能计算社区很快就接纳了这种多核处理器技术。

过去十年间，许多高性能计算平台都在多核处理器上搭载了通用图形处理器（GPGPU），以利用其以较低功耗处理大量数据并行任务的能力。通用图形处理器最初用于图形运算，如今已广泛应用于高性能计算集群和图形处理器通用计算领域。尽管也有一些显著的例外，最新的 TOP500 榜单 [37] 清楚地表明，异构高性能计算平台正成为发展趋势，尤其是采用 GPU 的平台。相当一部分系统都是异构系统，其中 AMD 或 NVIDIA GPU 通常能够提供较高的每瓦性能。

*2023*年*1*月，*arXiv*2023年。美国计算机协会 ISB
N 978-x-xxxx-xxxx-x/YY/MM...$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

```
#pragma omp target
  for(int i=0; i<N_i; i++) {
    for(int j=0; j<N_j; j++) {
      for(int k=0; k<N_k; k++) {
        for(int l=0; l<N_l; l++) {
          /* ... 计算 ... */
        }
      }
    }
  }
```

代码 1。Wilson Dslash 算子的循环

1.1 挑战

许多程序员正在调整代码，以充分利用 GPU。不幸的是，在最大限度降低开销的同时优化 GPU 的计算能力，是一项繁琐的工作，可能需要重新设计大段代码和数据结构。为具有高度异构性且包含大量设备的系统开发代码，将面临更大的挑战。因此，必须开发能够减轻应用科学家这类开发负担的工具。

尽管有多种编程模型可供选择，但要优化由数万到数十万行代码组成的大型应用程序，仍然颇具挑战。即使采用 OpenMP 这类基于指令的编程模型，为每个内核添加指令也是一项重复且复杂的工作。OpenMP 提供了多种将内核卸载到 GPU 的选项。然而，应用科学家仍必须弄清所有复杂的 GPU 配置。为了展示将内核移植到新兴百亿亿次级硬件上的复杂性，我们使用了一个来自格点量子色动力学（LQCD） [2] 应用的内核；LQCD 是一个便于计算机处理的量子色动力学数值框架。LQCD 的关键计算内核之一是 Wilson Dslash 算子 [19],，它本质上是一个有限差分算子，用于描述夸克与胶子的相互作用。

四维时空中的 Wilson Dslash 算子 D由公式 1 定义。

$$D^{ij}_{\alpha\beta}(x,y) = \sum_{\mu=1}^{4}[((1-\gamma_\mu))_{\alpha\beta}U^{ij}_{\mu}(x)\delta_{x+\hat{\mu},y} + (1+\gamma_\mu)_{\alpha\beta}U^{\dagger ij}_{\mu}(x+\hat{\mu})\delta_{x-\hat{\mu},y})] \quad (1)$$

这里， x和 y是格点的坐标， α、 β是自旋指标， i、 j是颜色指标。 $U_\mu(x)$ 是胶子场变量，是一个 $SU(3)$ 矩阵。γ_μ 是固定的 4×4 狄拉克矩阵。复费米子场表示为一维数组，其大小为 $L_X \times L_Y \times L_Z \times L_T \times SPINS \times COLORS \times 2$，对应未预条件化的狄拉克算子，其中$L_X$、$L_Y$、 L_Z和 L_T分别是 x、 y、 z和 t 方向上的格点数。自旋和颜色自由度通常分别为 4 和 3，以变量 $SPINS$和 $COLORS$表示。

将方程 1 表示为 C++ 代码时，其中包含四个嵌套 for 循环，分别遍历 L_T、 L_Z、 L_Y、和 L_X，如代码 1 所示。当 L_T、 L_Z、 L_Y、和 L_X的值均设为 16 时，代码的 COMPUTE 部分包含超过 500 万个变量定义、12 亿个变量引用、超过 1.5 亿次加减运算、1.63 亿次乘法运算等。此外，此函数在整个 LQCD 应用中会被调用多次。

对于应用科学家而言，理解物理学、将其转化为计算机程序、分析可卸载的内核，然后思考如何将其并行化以便在高性能计算集群上高效执行，是一项艰巨的任务。要充分发挥图形处理器的性能，应用科学家需要深入了解底层架构、算法和接口编程模型。或者，他们也可以尝试各种图形处理器转换，直到找到最有效的一种。然而，这些策略都不太高效。

1.2 我们的贡献

本文介绍了**OpenMP Advisor**，这是首创的编译器工具，可为应用科学家提供各种 OpenMP 代码卸载策略的建议。该工具通过执行以下任务，成功应对有效转换 OpenMP 代码所面临的挑战：

1. 检测可能可卸载的内核；2. 识别内核中的数据访问和修改；3. 推荐多种可将该内核卸载到图形处理器的 OpenMP 变体；4. 通过自适应成本模型评估每种内核变体的收益；5. 插入相关的 OpenMP 指令以执行卸载。

尽管该工具目前仅适用于 GPU，但可以扩展以支持其他支持 OpenMP 的设备。本文其余部分首先在第 2 节讨论与我们的工作相关且早于我们工作的最新进展。接着，我们在第 3 节定义 OpenMP Advisor，并在第 4 节介绍其实现。第 5 节介绍本文开展的实验及结果分析。最后，我们在第 6 节总结工作并讨论未来计划。

2 相关工作

许多研究探讨了在必须显式管理数据移动时，如何更好地管理 GPU 内存。Jablin 等人 [13]为 CUDA 程序提供了一套全自动系统，用于管理和优化 CPU 与图形处理器之间的通信。Gelado 等人 [9] 提出了一种异构计算编程模型，以简化和优化 GPU 数据管理。OMPSan [1]对已插入 OpenMP 代码中的显式数据传输进行静态分析。然而，这些研究并未揭示数据必须如何传输，也未说明如何利用 GPU 上的数据可复用性来管理内核之间的数据。

Mishra 等人 [26] 提出了一种技术，可静态识别每个内核使用的数据，并自动识别不同内核之间的数据复用机会。在我们的工作中，我们使用这项技术来管理 CPU 与图形处理器之间的数据。

近期，Roy 等人 [31] 提出了 BLISS，这是一种新颖的方法，无需插桩、领域专属知识或对应用程序的先验了解，即可对并行应用程序进行自动调优。借助贝叶斯优化，这款自动调优器会创建一组轻量级模型，其输出质量可与复杂、庞大的机器学习（ML）模型相媲美。BOHB[8], HiPerBOt [23], GPTune [20] 和 ytopt [40] 等其他自动调优器也经常在此场景中使用贝叶斯优化。然而，由于评估函数成本高昂，调优大规模应用程序仍然是一项挑战。此外，在异构硬件（如 GPU）和软件配置上优化高性能计算程序的需求，也阻碍了这些自动调优技术在百亿亿次级时代的广泛应用。高性能计算应用正变得极其异构、复杂，分析成本也日益高昂。

我们需要一种像我们这样的工具，帮助应用科学家将代码卸载到 GPU。Mendonça 等人 [22]和 Poesia 等人 [30]开展的其他自动 GPU 卸载相关研究，也可以通过将我们的数据优化技术和成本模型纳入其框架而受益，从而进一步降低在科学计算中使用 GPU 的难度。然而，开发成本模型耗时费力，而当今几乎所有编译器（例如 LLVM/Clang [21], ）都采用简单的“一刀切”成本函数，在异构架构差异极大的情况下无法达到最佳效果。COMPOFF [24]借助神经网络模型，提供了一种新的可移植成本模型，可静态估算 OpenMP 在不同架构上的卸载成本。我们将 COMPOFF 中定义的技术加以扩展，开发出一种可移植的静态成本模型，供我们的 OpenMP 顾问工具使用。

3 OpenMP Advisor

我们设计并开发了 OpenMP Advisor，这是一种编译器工具，可转换 OpenMP 代码，使其能够高效地卸载到图形处理器。该工具可识别 OpenMP 内核，提出若干种可将该内核卸载到图形处理器的 OpenMP 变体，并使用基于静态神经网络的编译时成本模型，分别预测运行这些内核的成本。OpenMP Advisor 的首个版本可帮助应用科学家编写面向图形处理器等加速器的代码。不过，该顾问工具可扩展为支持所有支持 OpenMP 的设备。

该工具有两种模式：训练模式和预测模式。训练模式必须在目标硬件上执行。它将所有基准测试代码作为输入，并生成所有可能的代码变体，以便在目标设备上运行。然后，它从所有生成的代码中收集数据，用于训练基于机器学习的成本模型，以供预测模式使用。在预测模式下，该工具无需与目标设备进行任何交互。它接受 C/C++ 代码作为输入，并返回可用于将代码卸载到指定设备的最佳代码变体。该工具可以按照第 4.2 节中的定义，使用基于机器学习的成本模型预测内核运行时间，从而确定最适合卸载的内核。

3.1 属性

以下内容 g 是关键 y 属性，适用于 O p enMP Advisor。

1. **静态** – 使用 GPU 卸载分析应用程序时，访问 GPU 是一项重大挑战。在开发过程中，许多应用科学家可能无法轻松访问 GPU。因此，在预测模式下，顾问工具必须在编译时完成所有分析，且不需要运行时性能分析，这一点至关重要。即使在训练模式下，有时也很难在代码生成期间预测某些值。为帮助顾问工具应对此类情况，我们提供了一个 JSON 文件供用户输入这些值。

2. **简约** – 顾问工具不会对内核主体进行任何更改。它只会识别应用程序中的 OpenMP target 区域，并添加各种 OpenMP 指令和子句，以生成多个内核变体。

3. **正确性** – 顾问工具会验证生成代码的正确性，以确保其符合 OpenMP 编程模型。目前，顾问工具不会修改内核主体，也不保证其内容的正确性。因此，尽管它能够预测 GPU 卸载的最佳方案，但仍会生成排名前 N 的代码变体（N 由用户指定），并由应用科学家选择要使用的代码变体。

4. **可移植性** – 顾问工具可移植到多种编译器和高性能计算集群。在我们的工作中，我们使用了四种不同的编译器：LLVM/clang（nvptx）、LLVM/clang (rocm)、GNU/gcc 和 NVIDIA HPC/nvc。该顾问工具也适用于多种配备不同 GPU 的高性能计算集群，例如使用 NVIDIA GPU 的 Summit [28], Ookami [3], Wombat [29] 和 Sea-wulf [39] 集群，以及使用 AMD GPU 的 Corona [17]。我们还在配备 Intel GPU 和 icpx 编译器的 Intel Devclouds [12] 上进行了有限的测试。

5. **适应性** – 顾问工具具有足够的适应能力，可以接受新的应用程序。尝试对新的应用程序进行预测时，如果实时数据收集困难或不切实际，应用科学家可以创建一个与目标应用程序相似的代理应用程序，并使用从该代理应用程序收集的数据训练模型。

3.2 设计

在开始实现之前，我们需要先解决一些设计挑战。首先

个问题是，在开发我们的框架时应采用哪种工具包或编译器。这个问题很容易回答，因为我们使用了当时最受欢迎的库——LLVM 编译器项目。LLVM 是一个用于构建、优化和生成中间代码和/或机器二进制代码的库。与大多数现代编译器一样，LLVM 将编译过程分为三个层次：

- **前端** 将高级语言（如 C/C++、Fortran）转换为 LLVM 中间表示。
- **中端** 对 LLVM 中间表示执行多个优化遍。
- **后端** 将优化后的 LLVM 中间表示转换为相应平台的汇编语言。

我们选择了 Clang 编译器（14.0.0 版），因为我们的目标应用程序是用 C/C++ 编写的。Clang 既是一个工具开发平台，也是 LLVM 项目中面向 C 语言家族的前端。由于采用了基于库的架构，它更易于在其他项目中复用和集成新功能，也更加灵活。

接下来，我们必须决定应在哪个层级实现框架。尽管 LLVM 的优势在于 LLVM 中间表示，但我们的需求是为科学家生成并返回 C/C++代码。为了将 OpenMP 指令插入 C/C++ 文件，我们需要获取准确的源代码位置。虽然 LLVM 中间表示在源代码级调试信息中可以包含源代码信息，但无法保证源代码位置的准确性。另一方面，Clang 抽象语法树与编写的 C++ 代码和 C++ 标准都十分接近。Clang 为每个词元和抽象语法树节点提供一一对应的映射，并且能够出色地

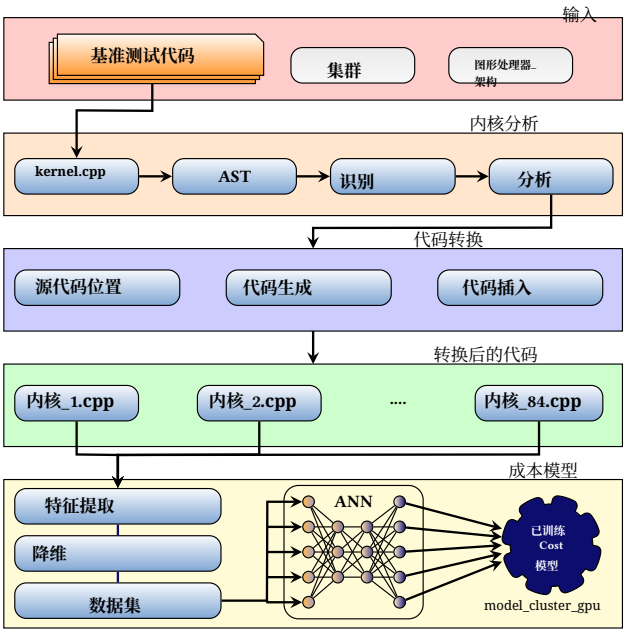


图 1。训练模式的高级流程

保留所有抽象语法树节点的源代码位置。抽象语法树是获取准确源代码信息的最佳位置。因此，我们在 Clang 中实现了顾问工具。顾问工具在训练模式下遵循图 1 所示的设计，在预测模式下遵循图 2 所示的设计。这两种模式都包含三个主要模块，将在以下小节中介绍：内核分析（第 4.1 节）、成本模型 （第 4.2 节）和代码转换 （第 4.3 节）。

4 实现

预处理后的抽象语法树比源代码级别的 C/C++ 代码更易于处理，而且我们始终可以通过 Clang 插件的 [5] 接口轻松参照原始代码。Clang 插件支持在编译期间执行额外的用户自定义操作。编译器会在运行时间从动态库中加载插件。FrontendAction 接口支持将用户专属操作作为编译过程的一部分执行，是 Clang 插件的典型入口点。为了对抽象语法树运行工具，Clang 提供了 AST 前端动作接口，用于处理操作的执行。我们定义并注册了 PluginOMPAdvisor，这是一个实现 CreateASTConsumer 方法的插件；该方法会返回一个 OMPAdvisorASTConsumer，用于消费（或读取）Clang 抽象语法树。借助 AST 消费者接口，我们可以对抽象语法树执行通用操作。该接口提供了 HandleTranslationUnit 函数，此函数仅在整个源文件解析完毕后调用。在本例中，

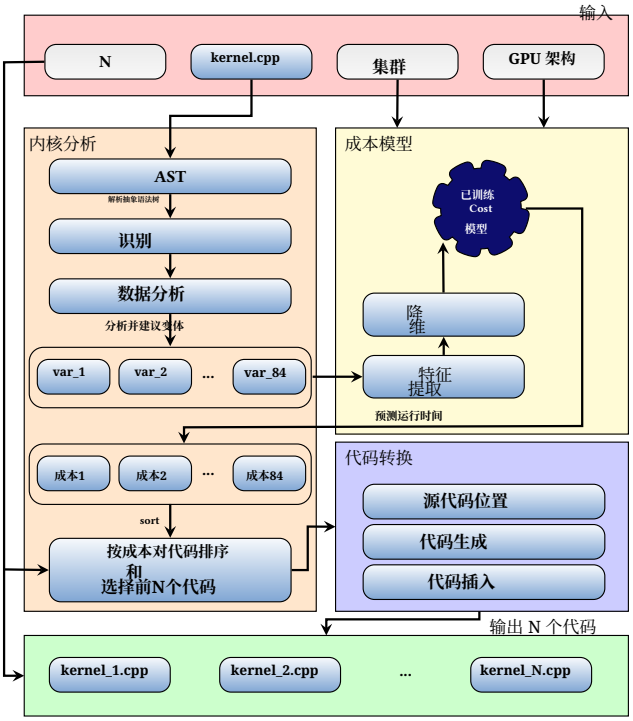


图 2. 预测模式的高级流程

翻译单元实际上代表整个源文件，而 ASTContext 对象则代表该源文件的抽象语法树。

Clang 还提供了 RecursiveASTVisitor 类，它会以前序或后序深度优先顺序遍历整个 Clang 抽象语法树，并访问每个节点。在我们的顾问工具中，我们定义了一个 OMPAdvisorVisitor，它是 RecursiveASTVisitor 的一个实例。通过简单地重写访问函数（例如 VisitStmt 或 VisitFunctionDecl），它可以访问任意类型的抽象语法树节点，例如 FunctionDecl 和 Stmt。我们没有直接调用任何 Visit* 函数，而是使用 TraverseDecl 函数，由它在后台调用相应的 Visit* 函数。为了继续遍历抽象语法树（以检查其他节点），我们在此类 Visit* 函数中返回 true；若要停止遍历并有效终止 Clang 编译过程，则返回 false。

4.1 内核分析

这是 Prediction 和训练模式下 HandleTranslationUnit 方法调用的第一个模块。顾名思义，该模块会分析一个 OpenMP 内核。总体而言，该模块以 C/C++ 源文件为输入，对其进行分析，并输出所有可能的图形处理器卸载变体。该模块负责三项任务：内核识别、数据分析 和变体生成。

4.1.1 内核识别。 用户输入用于帮助我们识别目标区域，因为本项目的范围并不包括自动并行化代码。应用科学家只需使用“omp target 指令”标记该区域。我们解析 Clang 抽象语法树，以查找 OMPTargetDirective 节点。该节点是 OMPExecutableDirective 类的子类，也是 Stmt 类的实例。我们重写 OMPAdvisorVisitor 类的 VisitStmt 方法，并检查每个访问到的 Stmt，判断其是否为 OMPTargetDirective 节点。识别出内核后，我们为其分配一个唯一 ID，并创建一个“Ker- nelInformation”类的实例。在该对象中，我们存储诸如唯一_ID、内核的起始和结束位置、调用内核的函数、内核是否在循环内调用以及嵌套 for 循环的数量等信息。之后，我们将利用所有这些信息来识别内核中访问的变量。

4.1.2 数据分析。 下一步是确定内核使用哪些数据。由于数据传入和传出图形处理器的成本都很高，因此在高性能计算中使用加速器时，必须谨慎进行数据管理以及主机与设备之间的映射。对于隐式数据传输，OpenMP 并未规定编译器应如何处理数据。有效管理数据的复杂性使应用科学家望而却步，降低了他们使用图形处理器的意愿。在应用程序中自动识别并利用图形处理器数据，可以减轻应用科学家的负担，并缩短整体执行时间。我们在 Mishra 等人关于数据传输与重用分析的工作 [26]基础上进行了扩展，该工作介绍了如何确定内核使用了哪些变量。我们使用 Clang 抽象语法树，为每个内核实现活变量分析，仅关注内核中使用的变量。这项工作的一个局限是，无法在编译期间确定数组的范围。在我们的实现中，我们扩展了分析功能，以确定代码中声明的数组范围。然而，这种分析并非总是可行，尤其是在处理全局变量或指针操作时。为了克服这一障碍，我们希望用户在数据映射子句中指定数组范围。如果用户未指定要映射到图形处理器的数组范围，我们就假定内核中使用的所有数组都在内核定义函数中声明或传入。我们继续执行活变量分析，以确定需要在 CPU 和图形处理器之间同步的变量，并计算它们的大小。我们的方法有一个局限：目前仅在内核执行前后在 CPU 与图形处理器之间映射数据。如何在内核执行期间管理数据传输，将留待未来研究。

在将变量存储到 KernelInformation 对象之前，我们会根据它们在内核执行之前、期间或之后的访问方式，将它们分为五组：

- 分配：这些变量首次在内核中赋值，因此无需向图形处理器传输数据。
- 传入：这些变量在之前已赋值，但在内核中仅被访问而未被修改。这些数据必须传输到设备，但无需传回主机。
- 传出：这些变量在内核中被更新，并在内核调用之后被访问。此类别无需向设备传输数据，但必须将数据从设备传输到主机。
- 双向传输：这些变量在之前已赋值，在内核定义期间被更新，并在之后被访问。此类数据必须在主机和设备之间双向传输。
- 私有：最后，这些变量仅在内核中定义和使用，无需在主机和设备之间传输。

标记为分配、传入和双向传输 的数据，会在内核之前通过“omp target enter data”指令进行映射；标记为传出 和双向传输 的数据，则会在内核之后通过“omp target exit data”指令进行映射。

4.1.3 变体生成。最后，我们生成多个不同的内核变体，以便将内核卸载到图形处理器上。首先，我们统计可折叠嵌套 for 循环的数量。在当前实现中，我们最多可以检查将 for 循环折叠至四层的情况。我们选择四层嵌套循环，是因为如代码 1 所示，Wilson Dslash 内核包含四个 for 循环。每个 for

```
#pragma omp 目标团队分发 collapse(1)
  for (int i  = 0;i <N_i;i++){  <= 循环 0
#pragma omp 并行 for 循环 collapse(3) schedule(静态)
    for (int j  = 0;j <N_j; j++) {  <= 循环 1
      for (int k  = 0;k  <N_k; k++) {  <= 循环 2
        for (int l  = 0;l  <N_l; l++) {  <= 循环 3
          ...
        }
      }
    }
  }
```

代码 2. 四层嵌套的“for”循环，在循环 0 中使用“teams distribute”，在循环 1 中使用“parallel for collapse(3)”。

循环都会获得一个从 0–3 开始的唯一循环编号。循环 0 始终预期会分布到图形处理器上的所有团队中。变体根据五个参数生成：

- 分发指令中使用的 collapse 子句的值
- parallel for 指令中使用的 collapse 子句的值
- parallel for 指令的位置
- 循环迭代的调度类型
- 主机与设备之间的数据传输

for 循环的总数以及并行 for 指令的位置决定了团队分发指令和并行 for 指令中可使用的 collapse 最大值。假设有四个 for 循环，如代码 2 所示。如果并行 for 指令位于循环 0，则“团队分发并行 for 循环”指令将合并，因此团队分发指令中没有 collapse 子句。如果并行 for 指令位于循环 x（其中 $1 \leq x \leq 3$），则团队分发指令中 collapse 的最大可能值为 x。而对于并行 for 指令，collapse 的最大可能值为（$NUM - x$），其中 NUM 是 for 循环的总数。

循环迭代的调度类型可以是静态、动态或引导式。通过对这些参数进行不同的排列组合，我们可以生成多种 GPU 卸载代码变体。生成所有变体后，我们使用静态成本模型来预测每个生成内核的运行时间。对于特定数量的 for 循环，可生成的变体数量如表 1 所示。

# for 循环	# 变体
1	6
2	18
3	42
4	84

表 1. 每种 for 循环数量预期生成的变体数量

4.2 成本模型

需要一个编译时成本模型，从内核分析模块生成的所有变体中选出最佳选项。大多数现代编译器都会采用解析模型，计算执行原始代码以及各种卸载代码变体的成本。为编译器构建这样的解析模型是一项艰巨的任务，需要编译器工程师付出大量努力。近年来，机器学习技术已成功应用于构建各种编译器优化问题的成本模型。我们扩展了 COMPOFF [24]，将其用作本工具的成本模型。COMPOFF 是一种基于机器学习的编译器成本模型，它使用人工神经网络模型静态估算 OpenMP 卸载的成本。其结果表明，该模型预测卸载成本的平均准确率超过 95%。COMPOFF 的主要局限在于，他们必须为每个变体分别训练一个模型。在我们的工作中，我们增加了训练数据，并对其进行扩展，使其能够为所有变体训练一个统一的成本模型。

一旦得知生成变体的预测结果，我们就会将其存储在 KernelInformation 类的实例中，以便内核转换模块使用。但实现基于机器学习的成本模型时，最大的挑战是缺乏可用的训练数据。为了解决这一问题，我们编写了额外的基准测试应用程序，并采用了 Rodinia 基准测试套件中的一些基准测试 [4]。我们的目标是纳入更广泛的基准测试类别，涵盖统计模拟、线性代数、数据挖掘等领域。为此，我们开发了表 2 中列出的应用程序。未来还会将来自其他不同领域的更多应用程序添加到此代码库中。

4.3 内核转换

在内核转换模块中，我们需要根据前面模块的分析和预测，实际转换原始源代码。在训练模式下，对于给定的内核，我们会生成所有可能的代码变体。然而，在预测模式下生成代码之前，我们必须先回答另一个关键问题：应该生成哪种代码？我们是否只应该为速度最快的内核生成代码？遗憾的是，指令就位后，顾问工具和 OpenMP 都不会验证内核的正确性。这也符合 OpenMP 的理念。因此，在我们的框架中，我们只能保证生成的 OpenMP 指令的正确性。

那么，我们该如何克服这个问题呢？我们可以像训练期间那样，为每种可能的变体生成代码，并让用户选择使用哪一种。但这样会让用户面对过多信息，不知所措。或者，我们可以要求用户提供要生成的代码数量上限。预测运行时间可以

应用	领域	描述
广度优先搜索(广度优先搜索) [34]†	图算法	广度优先搜索是一种图遍历算法，常用于各种学术和专业领域。
相关性系数（相关性） [32]	统计学	皮尔逊相关系数用于衡量两个数据集之间的线性相关性。
协方差（协方差）	概率论	样本协方差是根据一个或多个随机变量。
高斯消元（高斯） [33]†	线性代数	高斯消元法逐行计算结果，求解线性方程组中所有变量。系统。
K 近邻邻居（K近邻） [35]†	数据挖掘	K近邻技术利用邻近性对单个数据点进行分类或预测其所属类别。这是一种非参数的监督学习分类器。
拉普拉斯方程(拉普拉斯) [15]	数值分析	在物理学中，拉普拉斯方程是一个二阶偏微分方程。
矩阵-矩阵乘法 (矩阵乘法)	线性代数	最重要的矩阵运算无疑是矩阵乘法，它被广泛应用于各种领域，包括求解线性方程组、坐标系变换等。
矩阵-向量乘法 (矩阵向量乘法)	线性代数	一个简单的矩阵-向量乘法。
矩阵转置(mt)	线性代数	矩阵的转置就是将原矩阵翻转后的版本。
粒子滤波器（粒子） [36]†	医学成像	粒子滤波器是一种统计估计器，可根据含噪声的目标位置测量值以及对物体运动的了解，在贝叶斯框架。
代理应用（代理_应用程序）	-NA-	这是一个代理应用程序，其循环数量与我们的目标应用程序相同，执行的计算也与我们的目标应用程序 Wilson Dslash 算子类似。当难以收集真实应用程序的数据时，代理应用程序可以帮助我们收集更多数据。

表 2。 基准测试应用程序。标有 † 的应用程序选自 Rodinia 基准测试套件 [4]。

作为注释放在每段代码中内核之前。这样，应用科学家就能更自主地决定接受还是拒绝生成的代码。待解决 OpenMP 代码正确性的验证问题后，我们便能够生成一份代码并提供给用户。在此之前，我们的顾问工具可以按照应用科学家的要求，生成排名最靠前的变体。

无论如何，我们都需要编写一个模块来修改现有源代码并生成新代码。Clang 提供了重写器接口，其主要功能是将高层请求路由到相关的底层重写缓冲区。重写器可以帮助我们管理代码重写任务。在重写器接口中，我们可以设置源管理器对象，用于将源文件加载到内存并进行缓存。该对象会为每条不同的 #include 链分配唯一的文件标识符，并拥有所有已加载文件的内存缓冲区对象。我们可以查询源管理器，以获取有关 SourceLocation 对象的信息，然后将其转换为拼写位置或展开位置。拼写位置表示与词元对应的字节，而展开位置表示代码在源文件中的位置。例如，在宏展开的情况下，拼写位置指明展开后的词元源自何处，而展开位置则指定它在何处展开。

重写器是我们插件内核转换模块的重要组成部分。这里采用的策略是在关键位置细致地修改原始代码，

```
0:// 预测运行时间：1.2 秒
1: #pragma omp target enter data map(...)
2:#pragma omp target teams distribute collapse 子句(2)
    for (int i = 0; i <N_i; i++) {
3:
      for (int j = 0; j <N_j; j++) {
4: #pragma omp parallel for schedule(动态)
        for (int k = 0; k <N_k; k++) {
5:
          for (int l = 0; l <N_l; l++) {
            ...
          }
        }
      }
    }
6: #pragma omp target exit data map 子句(...)
```

代码 3。 七处生成代码的位置。

以完成转换，而不是处理每一种可能的抽象语法树节点，再从抽象语法树生成代码。为此，重写器接口必不可少。它是一个高级缓冲区管理器，利用绳索数据结构有效地对源代码进行切片和处理。对于每个源管理器，重写器还会存储底层的重写缓冲区。结合 Clang 对所有抽象语法树节点源代码位置的出色保留能力，重写器能够在重写缓冲区中非常精确地删除和插入代码。更新完成后，我们可以将重写缓冲区写入文件，从而获得更新后的源代码。

在内核转换模块中，我们创建一个由七个字符串组成的向量。这七个字符串的位置如代码 3 所示。第一个字符串（位于 #0）始终是注释，用于保留文本 – “预测运行时间：## 秒”。正如文本所示，## 是此特定内核变体的预测运行时间。此字符串始终位于内核起始位置之前。接下来是 target enter data 构造（位于 #1）。此指令负责处理内核需要在图形处理器上创建哪些内存，以及执行前需要向图形处理器传输哪些数据。此字符串始终紧跟在注释字符串之后。下一个字符串（位于 #2）包含 OpenMP 指令，用于指定要卸载到目标设备的内核。为了充分发挥图形处理器的性能，我们应始终将内核分发到图形处理器上所有可用的团队。因此，此字符串始终包含指令 – “#pragma omp target teams distribute”。该变体决定此指令是否包含 collapse 或 map 等其他子句，以及这些子句的取值。此字符串始终紧邻内核起始位置之前，但位于 target enter data 字符串之后。其余字符串（#3、#4，以及该变体需要时的 #5）放置在对应嵌套 for 循环的起始位置之前。如果某个变体不需要这些字符串，则将其留空，并且不在相应位置插入代码。最后一个字符串（位于 #6）是 target exit data 构造，用于标识必须从图形处理器释放或传回 CPU 的数据。如果这些字符串非空，则每个字符串都始终以换行符结尾。将这七个字符串放置在正确位置后，代码会被写入一个新的 C++ 文件，并返回给应用科学家；他们可以根据注释中提供的内核运行时间选择最佳代码。他们可以选择接受或拒绝这些变换，然后选择要与其代码一起构建和链接的文件。

集群	GPU	编译器	版本
Summit [28]	NVIDIA Tesla V100	LLVM/clang（nvptx）	13.0.0
		GNU/gcc (nvptx-none)	9.1.0
Corona [17]	AMD Radeon Instinct MI50	LLVM/clang（rocm-5.3）	15.0.0
Ookami [3]	NVIDIA Tesla V100	LLVM/clang（nvptx）	14.0.0
Wombat [29]	NVIDIA Tesla A100	LLVM/clang（nvptx）	15.0.0
Seawulf [39]	NVIDIA Tesla K80	LLVM/clang（nvptx）	12.0.0
		NVIDIA/NVC	21.7
英特尔 DevCloud [12]	Intel Xeon E-2176 P630	Intel/icpx	2021.1.2
Exxact	NVIDIA GeForce RTX 2080	LLVM/clang（nvptx）	14.0.0

表3. 实验中使用的集群与编译器

5 实验与评估

我们使用了表3所示的多个集群与编译器，对我们的工具进行测试和评估。在本研究中，我们仅在集群的每个节点上使用一个 GPU。多个 GPU 的管理留待未来研究。第 4 节中介绍的三个模块需要进行不同的实验。

5.1 实验 1- 数据分析

首先，我们针对所有基准测试应用程序测试了顾问工具，以确定数据是否被正确识别和生成。为了开展这项实验，我们使用顾问工具生成代码，确保主机与设备之间使用了正确的数据。此外，我们手动修改了每个基准测试算法，将所有数据 map 子句映射到图形处理器并从图形处理器传出。我们在表3所示的每个集群上执行了所有代码，并收集了有关数据传输量和持续时间的数据。我们发现，顾问工具在所有情况下都改进了数据管理。图3展示了所有基准测试应用程序在转换前后 CPU 与图形处理器之间传输的数据量（以 GB 为单位）。应用转换后，我们可以清楚地看到，数据传输量确实大幅减少。数据传输量的减少会对所有应用程序的数据传输时间产生

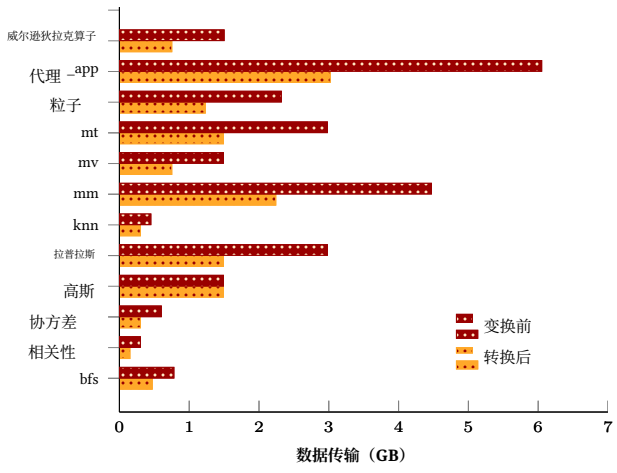


图3. 所有基准测试在数据分析前后的总数据传输

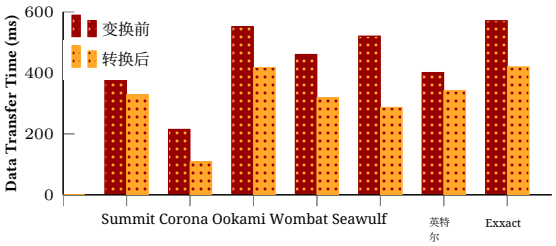


图 4. Wilson Dslash 算子在不同集群上转换前（1.5GB）和转换后（0.75GB）的数据传输时间（毫秒）

影响。我们在所有可用的集群上运行了这些应用程序，并收集了数据传输时间。图4展示了 Wilson Dslash 算子在不同集群上的数据传输时间。

5.2 实验 2——代码生成

在第一次实验中，已经测试了用于数据分析的代码生成。但该实验不会生成不同的代码变体。在第二次实验中，我们使用顾问工具，按照第 4.1.3 节中的讨论，针对每个基准测试应用程序生成所有可能的代码组合。表 4 显示了该实验的结果，与表 1 中的预期结果一致。我们使用表 3 中所示的编译器，为各种集群编译了所有这些代码。某些编译器（Seawulf 上的 NVIDIA/NVC 和 Wombat 上的 LLVM/Clang 15）不支持在图形处理器上进行动态调度或引导式调度，因此导致编译失败。除此之外，所有代码都成功编译，并在各自的集群上运行。我们收集了本次实验中每个内核的运行时间，供成本模型使用。

我们在一段时间前收集了 Intel 架构的数据，目前无法访问集群来开展新实验。因此，我们关于 Wilson Dslash 算子的可用数据非常有限，并且没有关于 Intel 架构上的代理_应用程序的数据。由于无法获得计算节点，我们无法为 Exxact 机器（配备 NVIDIA GeForce GPU）收集大量数据点。这两个集群各自只有约 2，000 个数据点。

Seawulf 配备了 NVIDIA K80 GPU，这是我们实验中使用的 GPU 里速度最慢的。因此，每个内核在 Seawulf 上的运行时间都比在其他集群上更长。另一方面，由于 Wombat 上的编译器不支持 GPU 上的动态调度和引导式调度，大多数内核变体都无法编译。由于这些原因，我们只能在 Seawulf 和 Wombat 上各收集约 3，000 个数据点。我们的所有内核都在 Summit、Corona 和 Ookami 上成功编译并运行，并且在这些架构上各收集了超过 10，000 个数据点。

应用	内核	# 循环	变体生成
bfs	kernel1	1	6
	kernel2	1	6
相关性	kernel1	1	6
协方差	kernel1	1	6
	kernel2	1	6
高斯	kernel1	2	18
拉普拉斯	kernel1	2	18
	kernel2	2	18
knn	kernel1	1	6
mm	kernel1	2	18
mv	kernel1	1	6
mt	kernel1	2	18
粒子	kernel1	1	6
	kernel2	1	6
	内核3	1	6
	内核4	1	6
	内核5	1	6
	内核6	1	6
	内核7	1	6
代理应用	kernel1	4	84
Wilson Dslash	kernel1	4	84

表4。为所有应用程序生成的变体数量

5.3 实验 3- 成本模型

为了构建成本模型，我们将 COMPOFF 模型从六种变体扩展到了全部 84 种变体。我们在测试模式下构建成本模型，然后用它在预测模式下预测运行时间。我们的成本模型采用一个包含六层的多层感知机模型：一个输入层、四个隐藏层和一个输出层。我们根据输入特征数量的倍数来设置神经元数量，而不是随机选择每个隐藏层的神经元数量，也不进行穷举网格搜索（即搜索第一层的神经元数量）。因此，在有 33 个输入特征的情况下，第一、第二、第三和第四个隐藏层分别有 66、132、66 和 33 个神经元。

线性层的权重采用 glorot 初始化方法设置，相关说明见 [10]。偏置设为 0，所有运行中训练数据的批量大小均设为 16。作为底层优化算法，我们评估了随机梯度下降（SGD）、Adam [16] 和 RMSprop [11]。由于 RMSprop 在两个高性能计算集群上的变换任务中表现最佳，因此我们选择 RMSprop 优化算法，初始学习率设为 0.01，每 30 个训练轮次缩小 0.1 倍，权重衰减设为 0.0001，共训练 150 个训练轮次。所有模型均使用均方根误差（RMSE）损失函数进行训练，其定义见公式 2，其中 $\bar{y}_i$ 和 y_i 分别表示预测运行时间和真实运行时间。

$$RMSE(\bar{y}, y) = \sqrt{\frac{1}{N}\sum_{i=0}^{N}(\bar{y}_i - y_i)^2} \quad (2)$$

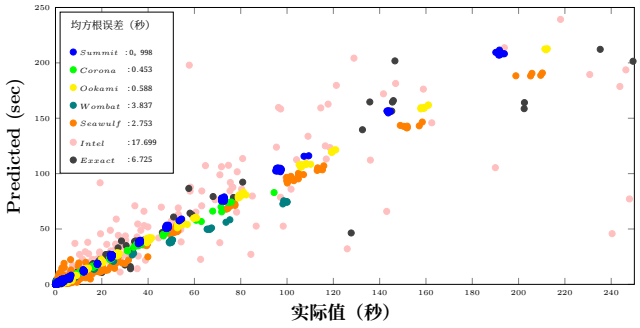


图5。 在不同集群上验证成本模型

我们将所有基准测试应用程序使用的数据集分为两部分：训练集（80%）和验证集（20%）。验证集不会用于模型的任何学习。对训练数据和验证数据采用的唯一增强方式是 Z分数标准化。模型使用训练集进行训练，之后将测试数据传入模型，以检验其性能。为确定预测误差的标准差，我们计算均方根误差（RMSE）。该值越低，模型越好。不过，RMSE 是否理想取决于计算 RMSE 所依据的数据范围。判断 RMSE 值是否“理想”的一种方法是使用以下公式对其进行归一化：

formula

$$NRMSE(\bar{y}, y) = \frac{RMSE(\bar{y}, y)}{(y_{max} - y_{min})} \tag{3}$$

这会得到介于 和 之间的一个值，d 值越接近 0，表示模型拟合效果越好。在本研究中，归一化均方根误差小于 0.1 的模型被视为成功。

从图5中实际数据与预测数据之间的强相关性可以看出，简单的多层感知机在我们的所有应用程序中都表现出色。原本预计，由于数据不足，英特尔和 Exxact 的模型会表现最差。然而，Exxact 的模型表现远好于英特尔上的模型（也呈现出收敛迹象），因为我们能够在 Exxact 系统上为代理应用收集一些数据。与使用更大数据集训练的其他模型相比，Wombat 和 Sea wulf 的表现中等。究竟需要多少数据才能训练机器学习模型，目前仍是一个开放问题。不过，从图5中我们观察到，如果模型拥有超过 10,000 个数据点，就能训练出一个更令人满意的模型。

5.4 实验4——预测

最后，在最后一组实验中，我们使用顾问工具为 Wilson Dslash 算子预测排名前 10 的最佳变体。确定这 10 个变体后，我们使用代码转换模块生成对应的 10 个代码变体，并将其返回给用户。顾问工具以 Wilson Dslash 基础内核作为输入，并根据成本模型预测生成排名前 10 的最佳内核。如图 6 所示，我们绘制了在 Summit 超级计算机上运行某个内核时，全部 84 个生成变体的实际运行时间和预测运行时间（按实际运行时间排序）。可以清楚地看到，所有变体的实际运行时间与预测运行时间之间都存在很强的相关性。在几乎所有集群的内核中都能观察到同样的相关性。图 7 展示了 Wilson Dslash 算子在各集群上的均方根误差和归一化均方根误差。图中各集群名称下方标出了其运行时间范围（秒）。目前我们无法使用 Intel 集群进行新实验，而且 Intel 数据集中来自目标 Wilson Dslash 内核的数据很少，也不包含任何来自代理_应用程序的数据。因此，即使我们使用该模型进行预测，也无法验证结果。鉴于此，我们没有在 Intel 架构上开展这项实验，结果标记为 $-NA-$。不出所料，由于 Exxact 数据集中的数据较少，目标内核的均方根误差显著增大（11.153 秒）。其归一化均方根误差为 0.279，仍未达到我们 0.1 的预期。尽管如此，该模型仍显示出实际数据与预测数据之间存在一定相关性。相比之下，Wombat 和 Seawulf 的表现相当不错，尽管均方根误差分别为 4.273 秒和 3.375 秒，仍能预测排名前 10 的内核变体。不过，它们的归一化均方根误差分别为 0.033 和 0.066，完全符合我们的预期。据我们观察，为这些集群添加更多数据也能进一步降低其均方根误差。最后，如图 7 所示，Summit、Corona 和 Ookami 的均方根误差均低于一秒，并且都能准确预测排名前十的内核变体。

6 结论与未来工作

在本文中，我们介绍了 OpenMP Advisor，这是一款编译器工具，可为应用科学家提供各种 OpenMP 代码卸载策略的建议。尽管该工具目前仅支持 GPU，但可以扩展以支持其他支持 OpenMP 的设备。借助我们的顾问工具，我们为多个应用程序生成了适用于七种不同

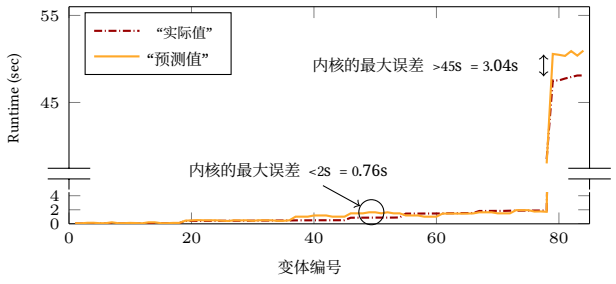


图 6. Summit 上所有变体的 Wilson Dslash 算子的实际运行时间和预测运行时间（单位：秒），按运行时间排序，其中 L_X, L_Y, L_Z和 L_T设置为 32,32,32 和 16。

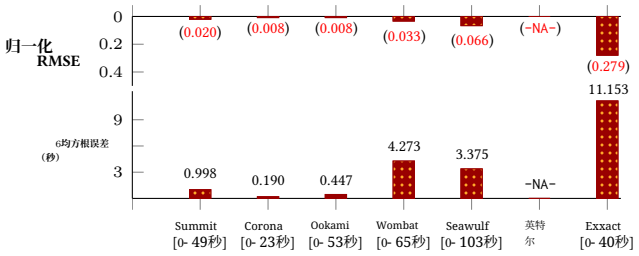


图 7。在不同集群上预测 Wilson Dslash 算子所有变体运行时间的均方根误差和归一化均方根误差，L_X，L_Y，L_Z，L_T分别设置为 32、32、32、16。每个集群的运行时间范围标注在其名称下方。

架构的代码，并且能够正确预测每个应用程序的十大最佳变体，其中包括在每种架构上运行的真实应用程序（Wilson Dslash 算子）。初步研究结果表明，该工具可帮助编译器开发人员和高性能计算应用科学家将遗留高性能计算代码移植到即将到来的异构计算环境中。下一步，我们将扩展该工具，使其涵盖以下任务：

- 内核执行期间主机与设备之间的数据同步。

- 通过任务将计算卸载到多个图形处理器 [14] 。
- 扩展基准测试，纳入更多内核，以覆盖更广泛的应用程序。

- 收集所有架构的更多数据，并将研究扩展至新的 GPU 和编译器。
- 我们的成本模型目前能够预测数据大小和并行度在编译时已知的内核的运行时间。我们可以扩展模型，使其能够预测各种数据大小下的最佳变体，然后使用 OpenMP 元指令 [27]指令为每个范围生成多个指令变体。

- 根据以往的研究 [18, 25], ，利用 CPU 和图形处理器之间的统一内存，可以提升 OpenMP GPU 卸载的性能；这类分析也应成为我们的顾问工具的一部分。

- 支持 OpenMP Advisor 分析所有 OpenMP 指令，而不仅仅是 target 卸载。
- 将顾问工具扩展到其他基于指令的模型，例如 OpenACC。

这款工具是首次尝试利用 OpenMP 和机器学习构建自动 GPU 卸载框架，因此仍有很大的改进空间。

致谢

本研究得到了百亿亿次级计算项目（17-SC-20-SC）的支持。该项目由美国能源部科学办公室与国家核安全管理局合作开展。本材料也基于美国国家科学基金会资助的工作（资助编号：CCF-2113996）。本研究使用了橡树岭国家实验室橡树岭领导力计算设施的资源，该设施由美国能源部科学办公室根据合同编号 DE-AC05-00OR22725 提供支持。作者感谢石溪大学研究计算与网络基础设施以及石溪大学高级计算科学研究所提供 SeaWulf 计算系统的使用权限。该系统的建设得益于美国国家科学基金会 140 万美元的资助（# 1531492）。

参考文献

[1] Prithayan Barua、Jun Shirako、Whitney Tsang、Jeeva Paudel、Wang Chen 和 Vivek Sarkar。2019年。OMPSan：对 OpenMP 数据映射构造进行静态验证。载于*OpenMP* 国际研讨会。Springer，新西兰奥克兰，3–18。[2] Brower、Richard、Christ、Norman、DeTar、Carleton、Edwards、Robert 和 Mackenzie、Paul。2018年。美国能源部百亿亿次级计算项目中的格点量子色动力学应用开发。*EPJ*网络会议175（ 2018 09010 10 1051 201817509010 ），。[3] https://doi.org/ 。/epjconf/ Andrew Burford、Alan Calder、David Carlson、Barbara Chapman、Fi- rat Coskun、Tony Curtis、Catherine Feldman、Robert Harrison、Yan Kang、Benjamin Michalowicz 等。2021年。Ookami：部署与初期体验。载于高级研究计算实践与经验。美国计算机协会，纽约，1–8。[4] Shuai Che、Michael Boyer、Jiayuan Meng、David Tarjan、Jeremy W Sheaffer、Sang-Ha Lee 和 Kevin Skadron。2009年。Rodinia：异构计算基准测试套件。载于*2009*年 *IEEE* 工作负载特征分析国际研讨会（*IISWC*）。IEEE，44–54。[5] Clang 插件 2022。Clang 插件。检索于2022年4月20日，网址：https://clang.llvm.org/docs/ClangPlugins.html [6] Leonardo Dagum 和 Ramesh Menon。1998年。OpenMP：共享内存编程的行业标准API。*IEEE*计算科学与工程5，1（1998），46–55。[7] Robert H Dennard、Fritz H Gaensslen、Hwa-Nien Yu、V Leo Rideout、Ernest Bassous 和 Andre R LeBlanc。1974年。具有极小物理尺寸的离子注入 MOSFET 设计。*IEEE* 固态电路期刊9，5（1974），256–268。[8] Stefan Falkner、Aaron Klein 和 Frank Hutter。2018年。BOHB：大规模稳健且高效的超参数优化。载于国际机器学习会议。机器学习研究论文集，1437–1446。[9] Isaac Gelado、John E Stone、Javier Cabezas、Sanjay Patel、Nacho Navarro 和 Wen-mei W Hwu。2010年。适用于异构并行系统的非对称分布式共享内存模型。载于第十五届编程语言与操作系统体系结构支持国际会议论文集。347–358。[10] Xavier Glorot 和 Yoshua Bengio。2010年。理解深度前馈神经网络训练的困难。载于第十三届人工智能与统计国际会议论文集。JMLR研讨会与会议论文集，249–256。[11] Geoffrey Hinton、Nitish Srivastava 和 Kevin Swersky。2012年。机器学习神经网络第 6a 讲：小批量梯度下降概述。引自14，8（2012），2。[12] 英特尔。2021年。英特尔开发者云。检索于2022年10月25日，网址：
h
t
t
p
s://
w
w
w.intel.com/content/www/us/en/developer/tools/devcloud/overview.html
[13] Thomas B Jablin、Prakash Prabhu、James A Jablin、Nick P Johnson、Stephen R Beard 和 David I August。2011年。CPU-GPU通信管理与优化自动化。载于第 *32* 届 *ACM SIGPLAN* 编程语言设计与实现会议论文集。142–151。

[14] Vivek Kale、Wenbin Lu、Anthony Curtis、Abid M Malik、Barbara Chapman 和 Oscar Hernandez。2020年。通过 taskloop 和用户定义的调度支持多 GPU 目标。载于*OpenMP* 国际研讨会。Springer，295–309。[15] Kushal Kedia。2009年。使用 *OpenMP* 和 *MPI* 进行混合编程。技术报告。技术报告 18.337 J。麻省理工学院。[16] Diederik P. Kingma 和 Jimmy Ba。2015年。Adam：一种随机优化方法。载于第三届国际学习表征会议（*ICLR 2015*），美国加利福尼亚州圣迭戈，*2015*年*5*月*7*日至*9*日，会议论文集，Yoshua Bengio 和 Yann LeCun（编）。http://arxiv.org/abs/ 1412 6980 。[17] 劳伦斯利弗莫尔国家实验室。2019年。LLNL - Corona。检索于2022年10月25日，网址：https://hpc.llnl.gov/hardware/compute-platforms/corona [18] Lingda Li、Hal Finkel、Martin Kong 和 Barbara Chapman。2018年。在统一地址空间下管理 OpenMP GPU 数据环境。载于*OpenMP* 国际研讨会。Springer，69–81。[19] M Lin。2016年。Columbia Physics System 中 Domain Wall Dslash 内核的优化。（2016），269。[20] Yang Liu、Wissam M Sid-Lakhdar、Osni Marques、Xinran Zhu、Chang Meng、James W Demmel 和 Xiaoye S Li。2021年。GPTune：用于百亿亿次级应用自动调优的多任务学习。载于第 *26* 届 *ACM SIGPLAN* 并行编程原理与实践研讨会论文集。234–246。[21] LLVM。2021年。*LLVM* 成本模型。https://llvm.org/doxygen/CostModel 8 cpp.html[22] Gleison Mendonça、Breno Guimarães、Péricles Alves、Márcio Pereira、Guido Araújo 和 Fernando Magno Quintão Pereira。2017年。DawnCC：用于数据并行与卸载的自动注释。*ACM* 建筑与代码优化汇刊（*TACO*）14，2（2017），13。[23] Harshitha Menon、Abhinav Bhatele 和 Todd Gamblin。2020年。使用贝叶斯优化为高性能计算应用自动调优参数选择。载于*2020*年 *IEEE* 国际并行与分布式处理研讨会（*IPDPS*）。电气电子工程师学会，831–840。[24] Alok Mishra、Smeet Chheda、Carlos Soto、Abid M. Malik、Meifeng Lin 和 Barbara Chapman。2022年。COMPOFF：一种利用机器学习预测 OpenMP 卸载成本的编译器成本模型。载于*2022*年 *IEEE* 国际并行与分布式处理研讨会（*IPDPSW*），*2022* 年 *5* 月 *30* 日至*6*月*3*日。电气电子工程师学会。[25] Alok Mishra、Lingda Li、Martin Kong、Hal Finkel 和 Barbara Chapman。2017年。基准测试与评估用于 OpenMP GPU 卸载的统一内存。载于第四届高性能计算 *LLVM* 编译器基础设施研讨会论文集。1–10。[26] Alok Mishra、Abid M Malik 和 Barbara Chapman。2020年。使用 OpenMP 进行 GPU 卸载的数据传输与重用分析工具。载于*OpenMP* 国际研讨会。Springer，280–294。[27] Alok Mishra、Abid M Malik 和 Barbara Chapman。2020年。扩展 LLVM/Clang 框架以支持 OpenMP 元指令。载于*2020*年 *IEEE/ACM* 第六届高性能计算 *LLVM* 编译器基础设施研讨会（*LLVM-HPC*）与百亿亿次计算的分层并行研讨会（*HiPar*）。电气电子工程师学会，33–44。[28] 橡树岭国家实验室。2017年。橡树岭领导力计算设施 - Summit 超级计算机集群。检索于2022年10月25日，网址：https://www.olcf.ornl.gov/summit/ [29] 橡树岭国家实验室。2020年。橡树岭领导力计算设施 - Wombat 集群。检索于2022年10月25日，网址：h ttps:// www.olcf.ornl.gov/olcf-resources/compute-systems/wombat/ [30] Gabriel Poesia、Breno Guimarães、Fabricio Ferracioli 和 Fernando Magno Quintão Pereira。2017年。异构设备上的计算静态放置。*ACM* 编程语言论文集1，OOPSLA（2017），1–28。[31] Rohan Basu Roy、Tirthak Patel、Vijay Gadepally 和 Devesh Tiwari。2021年。BLISS：使用多种轻量级学习模型组成的模型池，为复杂应用程序进行自动调优。载于 第*42*届 *ACM SIGPLAN* 编程语言设计与实现国际会议论文集。1280–1295。[32] Patrick Schober、Christa Boer 和 Lothar A Schwarte。2018年。相关系数：恰当使用与解读。《麻醉与镇痛》126卷第5期（2018年），1763–1768。[33] Rodinia 基准测试套件。2018年。高斯消元法。https://www. cs.virginia.edu/rodinia/doku.php?id=gaussian_elimination [34] Rodinia 基准测试套件。2018年。图遍历 - 广度优先搜索。https://www.cs.virginia.edu/rodinia/doku.php?id=graph_traversal [35] Rodinia 基准测试套件。2018年。*K*近邻。https://www.cs. virginia.edu/rodinia/doku.php?id=k-nearest_neighbors [36] Rodinia 基准测试套件。2018年。粒子滤波器。https://www.cs.virginia. edu/rodinia/doku.php?id=particle_filter [37] Top500。2021年。Top500 超级计算机站点。https://www.top500.org/ lists/top500/2022/06/ [38] Ilkka Tuomi。2002年。摩尔定律的兴衰。*First Monday*（2002年）。[39] 石溪大学。2019年。SeaWulf：石溪大学的计算集群。检索日期：2022年10月25日，网址：https: //it.stonybrook.edu/help/kb/understanding-seawulf[40] Xingfu Wu、Michael Kruse、Prasanna Balaprakash、Hal Finkel、Paul Hovland、Valerie Taylor 和 Mary Hall。2022年。使用贝叶斯优化，通过 LLVM Clang/Polly 循环优化编译指示对 PolyBench 基准测试进行自动调优。《并发与计算：实践与经验》34卷第20期（2022年），e6683。