arXiv:2303.08873v1 [cs.PL] 15 Mar 2023

面向异构系统可移植性能的机器学习驱动的自适应 OpenMP

Giorgis Georgakoudi、Konstantinos Parasyris、Chunhua Liao、DavidBeckingsale、Todd Gamblin 和 Bronis deSupinski

劳伦斯利弗莫尔国家实验室，美国{georgakoudis1, parasyris1, liao6, beckingsale1, gamblin2, desupinski1}@llnl.gov

2023年3月17日

摘要

异构性已成为高性能计算系统架构设计的主流选择。然而，异构性给实现执行的性能可移植性带来了重大挑战。将程序适配到新的异构平台既费时又费力，还要求开发人员手动探索庞大的执行参数空间。为应对这些挑战，本文提出了对 OpenMP 的新扩展，以实现自主的机器学习驱动适配。

我们的解决方案包括一组新颖的语言构造、编译器转换和运行时支持。我们提出一种生产者-消费者模式，可灵活定义多个不同的 OpenMP 代码区域变体，以实现适配。这些区域会在运行时进行透明分析，从而自主训练优化机器学习模型，动态选择速度最快的变体。通过利用机器学习技术以及 OpenMP 编译的代码生成能力，我们的方法大幅减少了用户在异构架构上编写自适应应用程序所需的工作。我们基于 Clang/LLVM 提供了完整的参考实现，并评估了自适应 CPU-GPU 执行的三个用例。针对 HPC 代理应用程序和基准测试开展的实验表明，所提出的自适应 OpenMP 扩展能够针对多种适配方式，在若干不同的 CPU 和 GPU 异构平台上自动选择性能最佳的代码变体。

1 引言

丹纳德缩放定律指出，通过晶体管微缩，处理器时钟频率会持续提升；而摩尔定律则认为，微芯片中的 CMOS 晶体管数量每两年翻一番。这两者的变化共同推动技术趋势转向并行架构。21 世纪初，并行计算机系统架构侧重于多核 CPU 架构。后来，GPGPU 范式的引入使技术趋势转向由多核 CPU 和 GPU 共同构成的异构系统。

这种异构性揭示了软件性能可移植性所面临的挑战。软件性能可移植性旨在通过单一应用程序实现，在不同底层硬件架构上实现相当的性能。OmpSs [9], OpenMP、Kokkos [10], 和 RAJA [15], 等编程模型提供抽象层，以隐藏在所有这些异构并行架构上开发应用程序所需的供应商特定接口，并提供统一的接口来表达并行性。尽管这些编程模型提供了一个简洁便捷的层，用于实现可移植代码，但同一应用程序在不同架构和系统上执行时，性能可能会有所不同。因此，这些编程模型能够高效地表达可移植代码，但应用程序在不同异构系统上执行时的性能可移植性并无保证。例如，高性能计算程序员发现，单一版本的源代码及其关联的静态执行参数定义（例如 GPU线程块中的线程数）通常不足以在不同异构系统上实现高应用程序性能。

（例如 GPU线程块中的线程数）等执行参数的静态定义通常不足以在不同异构系统上实现高应用程序性能。

手动为每个独立系统确定最优配置选项并设置其取值，代价高昂且难以扩展。因此，如今应用程序开发人员通过使用 Python 等脚本语言，手动组合一套自定义步骤流程，来自动执行配置搜索和调优过程。虽然这种方法可行，但程序员必须挑选合适的组件，以支持工作流程中的所有必要阶段，包括代码生成器、性能分析器和自适应运行时。总而言之，应用程序开发人员将算法移植到性能可移植编程模型上，使算法能够在任何受支持的架构上执行；随后，他们再通过自定义脚本针对某种特定架构对算法进行专门优化。整个过程自相矛盾。

我们认为，未来的性能可移植编程模型应提供编程语言构造，使开发人员能够根据指定的配置选项调优代码。软件栈（编译器、运行时库和操作系统）应自动、透明地调整执行方式并选择最优配置，同时尽可能减少对开发人员提供信息的要求。我们设想利用机器学习技术来指导参数选择。鉴于应用与平台之间存在复杂的相互作用，与试图理解软件和硬件内部机制的白盒方法相比，采用机器学习驱动的运行时自适应的黑盒方法，更适合用于指导代码变体的最优选择和配置。

本文提出一种新范式——机器学习驱动的自适应 OpenMP，旨在探索设计和实现新型编程模型特性，以应对适配挑战。我们认为，未来的编程模型（包括 OpenMP）应允许程序员表达与利用机器学习技术实现自动运行时自适应相关的丰富语义。由此形成的编程系统将借助机器学习技术，大幅减少用户在异构架构上编程所需的工作，从而实现性能可移植性，并提升高性能计算系统的计算效率。我们的工作贡献如下：

- 一组新的 OpenMP 指令和子句，用于表达自动进行模型驱动的运行时自适应所需的基本语义，适配由用户选择的 OpenMP 区域。我们的新颖设计提出了一种生产者-消费者模式来实现适配，该模式既能与现有的 OpenMP 适配机制组合使用，也能让用户直接在代码中指定适配；
- 结合编译器转换与运行时支持，实现自动多变体代码生成、透明的运行时性能分析和模型构建，从而自主地对自适应 OpenMP 区域进行模型引导的适配；
- 一个完整的参考实现[1], , 构建于 Clang/LLVM 之上，并连接到机器学习库的自适应运行时系统，可免费获取。
- 我们使用多种高性能计算基准测试和代理应用程序，对所提出的方法进行了详细评估。这些应用程序涵盖三个不同的用例，并在多种由 CPU 和 GPU 计算能力组成的异构系统上运行。

实验表明，所提出的自适应 OpenMP 能够利用运行时自动构建的决策树模型，在具有不同适配可能性的多个异构平台上，自动选择性能最佳的代码变体和配置。值得一提的是，我们的工作首次提出了一种完整、明确的自适应 OpenMP 解决方案，采用机器学习驱动的适配方式，具备稳健的编译时和运行时支持，并广泛适用于多种用例。

2 背景

现有的 OpenMP 适配 简而言之，OpenMP 5.0 [21] 引入了变体指令，例如元指令 和声明变体，通过在编译时适配 OpenMP 编译指示和用户代码来支持性能可移植性。OpenMP 上下文由活动 OpenMP 构造、设备、实现或用户定义条件的特征组成，可用于指定适配方式：将这些特征与指定属性进行匹配，从而选择适用于编译和执行的变体。OpenMP 5.0 仅支持匹配特征上的编译时条件；OpenMP 5.1 [22] 则将其扩展为支持运行时用户定义条件。

本研究尤其关注 OpenMP 元指令，因为我们通过自适应 OpenMP 扩展增强了它的功能。具体而言，OpenMP 元指令支持选择不同的 指令变体

表 1：Apollo API

C API	描述
void * __apollo 区域创建 (char *id，int 特征数，int 策略数，char *模型类型参数，int 最小训练数据量)	构建一个由 id 唯一标识的可调优区域，其中包含 num 个特征、num 个特征、num 个策略变体，并使用该模型类型中指定的类型，在达到最低数量后自动训练调优模型。返回指向区域处理器的不透明指针，用于 API 调用。
void __apollo 区域开始 (void * 区域)	开始对区域进行性能分析执行。
void __apollo_区域 end (void * 区域)	结束区域的性能分析执行。
void __apollo 区域集特征 (void *区域，float 值)	向区域的执行上下文添加一个特征值。
int __apollo_region_get policy (void *region)	根据特征查询 Apollo，以获取执行区域的策略变体。外层执行上下文的特征值，向 Apollo 查询要执行该区域的策略变体。返回一个介于 [0,策略数量-1] 之间的整数值。
void __apollo 区域 train (void * 区域)	在 Apollo 中使用迄今收集的任意性能分析数据，为某个区域训练模型 far.

```
1void 向量加法(double*A,double*B,double*C,size_tN)2{
3 #pragma omp metadirective \
4  when(user={condition(N<1024)} : parallel for) \
5  when(user={condition(N>=1024)} : \
6  target map(to: A[0:N], B[0:N]) map(from: C[0:N]) \
7  teams distribute parallel for)8 for(size_ti = 0;i <N; ++
i) 9 C[i] = A[i] +B[i]; 10}
```

清单 1：使用现有 OpenMP 适配的向量加法。

根据 OpenMP 上下文特征中的条件进行选择。它接受 `when` 子句，通过上下文选择器规范指定条件，以及当该条件求值为真时适用的指令变体。

例如，清单 1 展示了一段代码摘录，其中使用元指令，根据向量大小选择在 CPU 或 GPU 上执行实现简单向量加法的代码区域。第 3–7 行中的元指令根据向量大小N指定不同的指令变体：当N 小于固定阈值 1024 个单元时，内核在 CPU 上以工作共享循环的方式执行；否则，它会通过卸载指令在 GPU 上执行。编译元指令会为这两种指令变体生成代码，而运行时实际执行哪一种则取决于N的值。

尽管 OpenMP 中现有的适配方案提供了有用的功能，但它并未解决这样一个难题：如何动态确定适配指定代码区域时所需的适当条件，例如示例中向量大小的具体阈值。手动确定这些条件既耗时又容易出错。开发者必须探索配置空间，包括硬件架构配置和可能的输入，收集性能分析数据，并将这些发现整合到某种模型中，以便在运行时优化变体选择。我们的 OpenMP 扩展通过自动化代码变体生成、数据收集和模型构建来弥补这些不足，用户只需提供一组输入特征和可能的变体。

先前提出的 OpenMP 适配扩展。 先前的工作 [18] 尝试仅为 OpenMP 元指令启用适配。该工作提出了一种声明适配 指令，将其绑定到后续的元指令，并要求用户提供一组特征，以定义一个机器学习模型，用于选择元指令的某个变体。然而，将适配仅绑定到元指令，会使其适用范围局限于通过元指令变体表达的适配可能性。此外，当元指令 的自适应模型影响绑定的元指令之外的其他代码区域时，还会引入前向和后向依赖，从而限制可能的适配。

```
1 void 向量加法(double*A , 双精度浮点数 *B , double *C, 大小_t N) {
2   静态 void*区域 =NULL;
3   枚举{ CPU  = 0, GPU  = 1 }; // Policy indices.
4
5   若(区域 == 空)
6     区域 = __apollo_区域_创建(/* id */ "向量加法",
7       /* num_features */ 1, /* num_policies */ 2,
8       /*model params */ "决策树,探索=轮询",
9       /* min_train_data */ 10);
10
11  __Apollo_区域_begin(region);
12  __Apollo_区域_set_特征(N);
13  int 策略 = __Apollo_区域_get_策略(region);
14  if(策略 ==CPU)
15    #pragma omp parallel for
16    对于(大小_ti  = 0;i <N; ++i)
17      C[i] =A[i] +B[i];
18  否则 /* policy == GPU */
19    #pragma omp target teams distribute parallel for \
20    map(to: A[0:N], B[0:N]) map(from: C[0:N])
21    对于(大小_ti  = 0;i <N; ++i)
22      C[i] =A[i] +B[i];
23
24  __apollo_区域_end(region);
25 }
```

图 2：使用 Apollo 进行向量加法适配

在我们的方法中，我们设计了一种新的适配指令，该指令是自包含的，并提出了用于适配的生产者-消费者模式。新的适配指令根据一组特征和一组变体生成自适应模型，无需与显式元指令关联，并支持将多个 OpenMP 元指令及其他代码区域作为模型输出的消费者。这种设计避免了先前工作的表达能力限制以及难以解决的依赖问题，从而带来了更大的灵活性。值得注意的是，后文介绍的用例中，需要协调多个代码区域进行适配的情形，在先前的工作中不可行或不切实际。

Apollo 调优 API 和运行时。 Apollo [5, 29] 是一款先进的、基于机器学习的调优库，使用户能够通过插桩代码来定义可调优区域、指定该区域的输入特征，以及一组抽象的变体（在 Apollo 术语中称为策略），以表达可能的调优选择。Apollo 配备了一个运行时库来实现其 API，内部会收集性能分析数据，并利用这些数据支持训练不同的机器学习模型。表 1 简要介绍了 API 调用及其功能，而清单 2 展示了如何使用 Apollo 进行适配，在向量加法内核上选择执行设备，而无需设定固定阈值。

在这种表述中，用户使用向量大小作为特征创建区域一次（第 5–9 行），将 CPU 或 GPU 执行列为可能的策略，并请求使用轮询方式探索策略来训练决策树模型，以收集训练数据。具体而言，Apollo 运行时会循环尝试可能的策略变体进行探索，并在收集到至少 10 个唯一的性能分析数据点后训练决策树模型——唯一性定义为收集具有不同特征和变体的性能分析数据。第 11–13 行对 Apollo API 的调用会在 Apollo 中开始区域执行，并查询运行时以获取执行策略。第 14–22 行根据执行策略的选择，实现 CPU 或 GPU 执行。最后，第 24 行表示 Apollo 中区域执行的结束。

Apollo 收集的性能分析数据用于测量每对开始/结束调用之间经过的执行时间，并存储在一个持久化数据库中，其中包含按区域划分的记录。Apollo 无论是在运行时还是在执行后，都会从这些记录中找出每个特征值对应的最快执行策略，以训练机器学习模型，从而在给定一个区域和一组可能未见过的特征时，预测最优策略选择。

Apollo 通过其 API 提供了有用的适配基础设施，但它完全不涉及 OpenMP。要通过 Apollo 对 OpenMP 程序进行适配，需要投入相当多的工作来为代码的不同部分插桩并定制实现。与直接使用 Apollo API 相比，我们的自适应 OpenMP 编程模型通过抽象 Apollo 中必不可少且必须显式进行的插桩，并借助编译器在与 OpenMP元指令结合时生成不同的代码变体，提升了可编程性。尽管如此，在我们的 OpenMP 适配 Clang/LLVM 实现中，代码生成阶段的目标设备是 Apollo 运

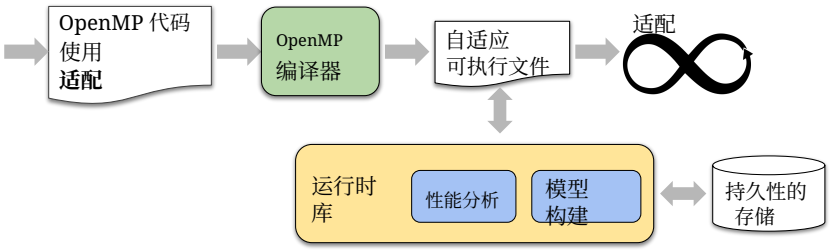

图 1：机器学习驱动的自适应 OpenMP

行时库 API，以利用其基础设施；同时需要指出的是，我们的自适应 OpenMP 也可以移植到符合所提语义的其他运行时实现。

3 自适应 OpenMP：设计与实现

图 1 概述了我们提出的方法，即扩展 OpenMP 以实现机器学习驱动的自适应。我们的扩展包括一种新的 OpenMP 构造 适配，用于定义自适应区域，以及用于在其他 OpenMP 构造中嵌入自适应功能的接口。编译器会生成实现自适应执行所需的所有代码，包括代码变体，以及用于性能分析、模型构建和模型引导的自适应的执行流水线；这些功能由运行时库提供支持。
生成的自适应可执行文件会经历一个自适应生命周期。运行该文件会收集带注释的自适应区域的性能分析数据，以评估不同执行变体。收集到足够的数据后，无论是由用户指示还是自主触发，可执行文件都会自动为每个自适应代码区域构建预测性机器学习模型，以选择各区域的优化变体。

用户可以根据应用程序和执行目标平台的特点定义自适应区域，提供一组特征来描述依赖于输入或数据的自适应可能性及可能的变体。此外，用户可以决定要训练哪种机器模型，以及要收集多少性能分析数据用于训练。自适应程序在执行常规计算的同时，会以透明的方式收集性能分析数据，用于训练自适应机器学习模型；这些数据会在多次运行中累积并存储于持久化存储中，以反映应用程序的性能。收集到足够的训练数据后，用户可指示自适应 OpenMP 为该区域训练自适应模型，以指导变体选择。对于迭代计算，通常可以在单次程序运行中收集足够的训练数据，从而在本次运行中训练并使用自适应模型；该模型也可供后续运行使用。否则，自适应 OpenMP 将利用多次运行中累积的数据训练模型。

3.1 OpenMP适配扩展

我们提出一种新的 OpenMP 构造 适配，用于表达在运行时自动进行机器学习驱动适配的语义。适配指令是一种独立指令，可采用 begin/end 指令对的形式，也可作为单条指令作用于代码的结构化代码块。它标示出用户代码中的一段区域，该区域构成执行过程中的自适应区域，将在运行时进行分析以收集数据。

语法和语义。适配 指令的语法如下：

```
#pragma omp begin 适配 [子句[[,]子句]...]换行 语句 #pragma
omp end 适配 [子句]换行or#pragma omp 适配 [子句[[,]子句]...]
换行 结构化代码块
```

用于指定自适应模型的相关子句如下：

- 模型_name(unique-id)(必需，唯一)
- 特征（特征列表）（可选，唯一）

- 变体（变体列表）（必填，唯一）
- 最小_训练_数据(int-expr)(可选，唯一)
- 模型_类型(kind-and-params)(可选，唯一)
- 模型_输出(identifier)(可选，唯一)

具体而言，t he 子句模型_名称 为自适应模型指定一个唯一的用户定义标识符，以便引用该模型 .
子句 features 接受一个基础语言标识符列表作为参数，这些标识符对应于作用域内的程序变量，并作为自适应模型的输入特征。区域本身是一个隐式特征，因为模型是按区域构建的。
子句 variants 接受一个变体列表作为参数。列表中的变体可以是用户定义的、类别标识符或数值表达式，也可以使用范围规格语法来定义可能数值变体的范围。范围规格的形式为 [BEGIN:END:STEP]，它会以BEGIN 数值为起点，以STEP 为增量，生成直到END （含该值）的变体范围。值BEGIN、END和STEP 必须是常量。例如，variants([10:20:2]) 等价于指定 variants(10, 12, 14, 16, 18, 20)。对自适应模型进行评估后，会选出一个变体。

子句 min_train_data 指定训练所需收集数据的最小数量。如果未指定，则默认设置由实现定义。在我们的实现中，最小数量等于可用变体的数量。

子句 模型_类型 以模型种类及其参数作为参数，用于指定构建自适应模型时采用的机器学习方法。若未指定，则模型类型和参数由实现定义。在我们的实现中，可选种类为 dtree 和 rfc，分别对应决策树分类器和随机森林分类器。此设计可扩展至其他机器学习分类器，例如神经网络或支持向量机（SVM）。种类 dtree 有一个可选的整数表达式作为唯一参数，用于指定最大树深度，例如 模型_类型(dtree, 4)。若没有参数，则最大树深度由实现定义。种类 rfc 有两个整数表达式作为参数，例如模型_类型(rfc, 10, 4)。第一个参数指定森林中的决策树数量，第二个参数指定这些树的最大树深度。两个参数必须同时定义，或者都不定义；若均未指定，则两个参数都由实现定义。在我们的实现中，默认模型类型是深度为 2 的决策树，先前的研究 [29, 18] 发现这种模型能够有效地实现自适应。

子句 模型_输出以基础语言标识符作为参数，用于存储运行时选定变体的数值。如果使用类别标识符指定变体，则基础语言标识符中的数值会按指定顺序，从 0 开始为这些类别编号。适配指令的结束部分只有一个子句 模型_名称，用于将此指令与其起始部分关联起来，从而支持自适应区域的嵌套。

适配的用法。此设计支持以两种方式使用适配指令中定义的模型：一种是在用户代码中直接实现执行变体，根据存储在 模型_输出所提供标识符中的值来选择变体；另一种是通过使用模型唯一标识符和元指令指令。我们为用户特征集扩展了元指令，加入适配特征选择器，其语法如下：

```
当（用户=适配（特征属性表达式））：\[指令变体])
```

与 OpenMP 中用于动态用户定义条件的条件特征选择器类似，适配选择器需要一个在运行时求值的布尔表达式，该表达式会对模型进行求值；模型由适配指令的 模型_名称子句中的标识符指定。表达式中的模型_名称标识符充当代理，代表在声明适配指令时，针对该执行所采用的特征值集合，为关联的自适应模型选定的具体变体。关系运算符运算符（==，!=，<，<=，>=，>）会在模型变体选择符合类别或数值变体标识符定义顺序所规定的隐式或显式排序时求值为 true。不同 when 子句中的布尔表达式应彼此互斥；若不互斥，则按照现有的 OpenMP 规范规则，优先采用求值为 true 的超集条件。

从概念上讲，适配 指令用于生成动态的变体选择，该选择取决于指定代码区域的输入特征。该选择的使用者包括 元指令，用于在运行时选择指令变体；还包括直接使用该选择的数值表达式的用户代码。这种设计为实现各种适配选择提供了充分的灵活性，其中包括根据模型输出选择不同代码版本。接下来我们将介绍三个用例。

```
1void 向量加法(double*A,double*B,double*C, 大小_t N){
2   #pragma omp begin adaptation \
3
4
5  #pragma omp metadirective \
6
7
8
9
10
11
12
13
14}
```

```
1void 向量加法(double*A , 双精度浮点数 *B , 双精度浮点型 *C , 大小_t N) {
2  静态 void *区域 = NULL;
3  枚举{ CPU = 0, GPU = 1 }; // Policy indices.
4
5  如果(区域 == 空)
6    区域 = __apollo_区域_创建(/* id */ "向量加法",
7    /* num_features */ 1,  /* num_policies */ 2,
8    /* model params */ "决策树. 探索=轮询".
9    /* min_train_data */ 10);
10
11 __Apollo_区域_begin(区域);
12 __Apollo_区域_set_特征(N);
13 int 策略 = __Apollo_区域_get_策略(区域);
14 if(策略 == CPU)
15   #pragma omp parallel for
16   对于(大小_ti = 0;i <N; ++i)
17     C[i] =A[i] +B[i];
18 else /* policy == GPU */
19   #pragma omp target teams distribute parallel for \
20   map(to: A[0:N], B[0:N]) map(from: C[0:N])
21   对于(大小_ti = 0;i <N; ++i)
22     C[i] =A[i] +B[i];
23 __Apollo_区域_end(region);
24}
```

列表 3：使用自适应 OpenMP（左）与 Apollo（右）进行向量加法适配，以选择最佳执行设备（CPU 或 GPU）。相同颜色表示功能对应的区域。

3.2 示例

列表 3、4 和 5 以向量加法内核的源代码为简单示例，对比了左侧使用自适应 OpenMP 的不同适配场景，以及右侧等效的 Apollo 实现。我们的设计支持所有这些适配方式，体现了其灵活性；我们也在一组高性能计算代理/迷你应用程序上评估了这些实际的适配选项。为强调两种方法实现适配所需的不同操作，我们在列表中用相同颜色标出两种实现中相对应的代码行：绿色，突出显示模型的实例化/声明；蓝色，突出显示对自适应模型决策的使用；灰色，突出显示适配后的算法；红色，突出显示模型的销毁。

3.2.1 最佳执行设备的选择

列表 3 展示了一个自适应执行示例，用于选择在 CPU 还是 GPU 上运行内核。在自适应 OpenMP 版本（列表 3 左侧）中，第 2–4 行包含`begin adaptation`指令，用于启动一个名为`by_len`的自适应区域，其唯一特征`N`用于模型构建，取值等于输入向量大小。用户定义的变体`cpu`和`gpu`在语义上分别对应 CPU 执行或 GPU 执行；由于未指定模型或最小训练次数，因此采用实现的默认值。第 13 行包含相应的`end adaptation`，用于标记该自适应区域的结束。第 5–9 行包含元指令，用于选择在主机 CPU 还是 GPU 设备上执行计算。

所提出的自适应 OpenMP 避免了 Apollo 接口冗长的问题。此外，它通过重复使用计算中 for 循环的单一实现，消除了难以维护的代码重复。模型决策与扩展的 元指令 相结合，非常适合生产者-消费者范式。模型生成一个选择，供元指令使用，以决定执行变体。

```
void 向量加法(double*A , 双精度浮点数 *B , 双精度浮点型 *C ,
    大小_t N){
  整数 GPU_线程;
  #pragma omp begin adaptation
    model_name(by_len) \
   features(N) \
   variants([64:1024:64]) \
   model(dtree) min_train_data(16 /* variants
    */ * 10) \
   model_output(GPU_THREADS)

  #pragma omp target \
    map(to: A[0:N], B[0:N]) map(from: C[0:N])
    \
    teams distribute" "parallel for \
    thread_limit(GPU_THREADS)

  用于(大小_t i  = 0; i  < N;  ++i)
   C[i] = A[i] +B[i];

  #pragma omp end adaptation
    model_name(by_len)
}
```

```
无效 向量加法(双精度*A , 双精度 *B , 双精度 *C ,
    大小_t N) {
  int GPU_线程;
  静态 void *区域 = NULL;
  整数 最小值_线程数=64, 最大值_线程数=1024,
    步骤=64:

  if(区域 == NULL)
    区域 = __Apollo_区域_create(/* id */
    "向量加法".
    /* num_features */ 1, /* num_policies */
    16,
    /*model params */
    "决策树. 探索=轮询".
    /* min_train_data */ 16 * 10);


  __apollo_区域_开始(区域);
  __apollo_区域_设置_特征(N);
  整数策略 =
    __apollo_区域_获取_策略(区域);
  GPU_线程 = 最小_线程 +策略*步长;
  #pragma omp target \
    map(to: A[0:N], B[0:N]) map(from: C[0:N])
    \
    teams distribute" parallel for \
    thread_limit(GPU_THREADS)

  循环(大小_ti  = 0;i <N; ++i)
   C[i] =A[i] + B[i];

  __apollo_区域_结束(区域);
}
```

列表 4：使用自适应 OpenMP（左）与 Apollo（右）进行向量加法适配，以选择每个团队的 GPU 线程数。颜色相同的部分对应功能相似的代码区域。

3.2.2 每个团队的最佳 GPU 线程数选择

代码清单 4 展示了一个动态选择每个团队 GPU 线程数的用例，并将所提出的自适应 OpenMP 方法与 Apollo 进行了对比。在自适应 OpenMP 中（代码清单 4 左侧），用户在第 3–7 行的适配指令中定义一系列数值变体，并提供一个程序变量 `GPU_THREADS`，用于存储模型评估的输出。该变量在第 8–10 行设置目标区域中可执行指令的线程数。用户请求使用决策树（默认树深度），并将最低训练数据量设为可能变体数量的 10 倍。模型将探索不同变体以收集数据，或在训练后选择能够优化执行性能的变体。

相比之下，Apollo 的实现（代码清单 4 右侧）除了需要设置检测详细程度外，还要求手动实现将 Apollo 策略索引转换回可能线程数选项的逻辑。自适应 OpenMP 借助编译器扩展生成代码，避免了这种复杂性，使用户能够简洁地定义和使用自适应模型。

3.2.3 GPU 与 CPU 执行协同调度

清单 5 展示了一个更复杂的用例：用户将计算划分到主机 CPU 和 GPU 设备上，以便同时利用二者。适配会动态优化设备之间的工作分配。使用自适应 OpenMP 时，如清单 5（左）所示，第 4–8 行的开始适配 指令指定了模型，其中的变体是数值表达式，表示要在 GPU 设备上执行的计算所占比例。用户指定程序变量 `GPU_FRAC`，用于存储模型选定的变体值。第 9–11 行和第 21 行的设备数据环境指令根据分配给 GPU 的工作比例进行参数化，以映射或取消映射数据。第 13–15 行包含 GPU 计算，其循环边界根据模型评估的变体进行调整；同样，第 17–19 行包含 CPU 计算。请注意，对于协同调度，目标区域标记为`nowait`，以便与主机上的并行区域并发执行；第 20 行的`taskwait` 则会同步执行，确保目标区域已完成，然后退出设备数据环境。

```
void 向量加法(double*A , 双精度*B , 双精度 *C ,
↪ 大小_t N){
  浮点型 GPU_分数;

  #pragma omp begin adaptation
↪  model_name(by_len) \
    features(N) \
    variants([0.0:1.0:0.1]) \
    model(dtree) min_training_data(11 * 10) \
    model_output(GPU_FRAC)")

  #pragma omp target enter data \
    map(to: A[0:GPU_FRAC*N], B[0:GPU_FRAC*N])
↪  \
    map(from: C[0:GPU_FRAC*N])
  // GPU execution
  #pragma omp target teams distribute parallel
↪  for nowait
  循环(大小_t i = 0; i < GPU_FRAC*N; ++i)
      C[i] =A[i] +B[i];
  // CPU execution
  #pragma omp parallel for
  循环(大小_t i = GPU_FRAC*N; i < N; ++i)
      C[i] =A[i] +B[i];
  #pragma omp taskwait
  #pragma target exit data map(from:
↪  C[0:GPU_FRAC*N])

  #pragma omp end adaptation
↪  model_name(by_len)
}
```

```
void 向量加法(double*A , 双精度*B , 双精度 *C ,
↪ 大小_t N) {
  浮点 GPU_FRAC;
  静态 void *区域 = NULL;
  浮点 MIN_FRAC=0.0f,MAX_FRAC=1.0f,
↪  步骤=0.1;

  如果(区域 == NULL)
    区域 = __apollo_区域_创建(/* id */
↪  "向量加法",
    /* num_features */ 1, /* num_policies */
↪  11,
    /*model params */
↪  "决策树.探索=轮询",
    /* min_train_data */ 11 * 10);

  __Apollo_区域_begin(region);
  __Apollo_区域_set_特征(N);

  int 策略 =
↪  __Apollo_区域_get_策略(region);
  GPU_FRAC = MIN_FRAC + 策略 * STEP;
  #pragma omp target enter data \
    map(to: A[0:GPU_FRAC*N], B[0:GPU_FRAC*N])
↪  \
    map(from: C[0:GPU_FRAC*N])
  // GPU execution
  #pragma omp target teams distribute parallel
↪  for nowait
  对于(大小_ti = 0;i <GPU_FRAC*N; ++i)
      C[i] =A[i] +B[i];
  // CPU execution
  #pragma omp parallel for
  对于(大小_ti =GPU_FRAC*N;i <N; ++i)
      C[i] =A[i] +B[i];
  #pragma omp taskwait
  #pragma target exit data map(from:
↪  C[0:GPU_FRAC*N])

  __apollo_区域_end(region);
}
```

列表 5：使用自适应 OpenMP（左）与 Apollo（右）进行 CPU-GPU协同调度时，向量加法适配如何划分工作量。颜色相同的部分对应功能相似的代码区域。

与 Apollo 实现（如清单 5 右侧所示）相比，自适应 OpenMP 表述更加简洁，同样避免了插桩和策略转换，同时凭借 OpenMP 更高层次的抽象支持类似的功能。

下一节将讨论用于降低自适应 OpenMP 的编译器扩展，并介绍实现中面向 Apollo 运行时 API 的目标设备。

3.3 编译器和运行时支持

我们扩展了成熟的生产级 Clang/LLVM 编译器框架，以实现所提出的 OpenMP 扩展。具体而言，我们修改了 Clang 前端，扩展解析和语义分析，使其能够处理新的适配指令、其子句，以及用于元指令的适配特征，并生成相应 LLVM 中间表示代码的抽象语法树节点。我们的代码生成扩展通过在 LLVM 中间表示中生成函数调用，将与适配相关的抽象语法树节点降级为对 Apollo 运行时 API 的 C 语言绑定的调用，其中包括自适应执行所需的全局状态（变量），以及用于动态选择要执行的首选变体的控制逻辑。

具体来说，开始适配 声明会生成一个全局变量，用于存储 Apollo 返回的区域处理器指针，该变量命名为 `__omp_适配_名称_<模型名称>` ，并初始化为 `null`。它还会生成全局变量，将分类变体参数转换为整数值，形式如下：

表 2：硬件和软件平台

	Power9 + V100	**英特尔 + P100**	**AMD +MI50**
CPU	IBM Power9	英特尔至强 E5-2695 v4 2.10 GHz	AMD 霄龙 7401 2.00 GHz
物理核心	10	18	24
处理器插槽	4	2	2
主存	256 吉字节	256 吉字节	256吉字节
GPU	NVIDIA Tesla V100-SXM2	NVIDIA Tesla P100	AMD Radeon Instinct MI50
设备内存	16 吉字节	16 吉字节	16GB

表 3：基准测试程序，包括简要描述、测试输入、用于建模和调优的特征

Name	**描述**	**特征**	**输入**	**CPU-GPU**	**GPU 线程**	**协同调度**
XSBench	蒙特卡洛中子输运 port	查找次数	$10000 \times [2^7, 2^8, \cdots, 2^{13}]$	✓		✓
RSBench	蒙特卡罗中子输运 port	查找次数	$10000 \times [2^8, 2^9, \cdots, 2^{13}]$			✓
LULESH	流体动力学代理应用程序方程。	单元数、节点数	$[50, \cdots, 120]$	✓	✓	
AMGMk（松弛）	代数多重网格基准测试-mark	矩阵行	$2^{12} \times [5^3, \cdots, 11^3]$	✓	✓	
Heat	热传导方程的显式有限差分热传导方程的方法	时间步	$512^2 \times [20^2, 21^2, \cdots, 32^2]$		✓	
FluidSim	使用二维流体模拟格子玻尔兹曼方法的二维流体模拟	网格大小	$512^2 \times [1^2, 2^2, \cdots 16^2]$		✓	

__omp_适配_<模型名称>_<变体>=<整数> 将最左侧的变体设为 0，其余变体依次递增。最后，它会生成一个名为 __omp_ 适配 _ 策略 _< 模型名称 > 的全局变量，用于存储 Apollo 返回的变体选择。假设指针类型和整数类型的大小均为 8 字节，则转换生成的状态所占内存为每个区域 $(N + 2) \times 8$ 字节，其中 N 为可能的变体数量。

此外，开始适配 时会发出对 __apollo_区域_create() 的调用，以创建一个区域，并将返回的指针存储到处理程序全局变量中，但仅当处理程序为 空时。随后，它会发出对 __apollo_区域_begin() 的调用，以启动该区域的上下文；还可能多次调用__apollo_区域_set_特征 ，以提供特征 子句中指定的特征值。最后，它会发出对__apollo_区域_get_policy() 的调用，以获取 Apollo 的变体选择，并将其存储在相关的全局变量以及模型_输出中定义的任何用户变量中。

结束适配声明会发出对 __apollo_区域_end() 的调用，以便 Apollo 收集内部性能分析数据，并在满足用户设定或实现默认值规定的最低训练数据要求后，触发优化型变体选择模型的训练。

4 实验设置

表 2 展示了评估中使用的三个异构系统的硬件和软件平台。遵循当前最佳实践，我们禁用同时多线程，并采用线程与内存绑定，以获得最佳性能并减少变异性。我们的实现基于 Clang/LLVM v14.0.0（提交 e08f3bf）扩展，加入了自适应 OpenMP 扩展，并以 Apollo 库（develop 分支，提交 3b5d38e）为目标。值得注意的是，Clang/LLVM 中的 OpenMP 卸载实现对 NVIDIA GPU（Power9+V100、英特尔+P100）已相当成熟，但对 AMD GPU（AMD+MI50）仍处于实验阶段。

表 3 列出了我们在评估中使用的程序。这些基准测试取自 HeCBench 基准测试套件 [16]，该套件收录了大量高性能计算基准测试、计算内核和代理应用/迷你应用，使用 CUDA、OpenMP 卸载、SYCL 和 HIP 编写。我们修改了 OpenMP 卸载实现，使其能够使用自适应 OpenMP 扩展，支持三种不同的适配方式：（1）选择执行设备，即 CPU 或 GPU；（2）为 GPU 执行时的每个自适应区域选择每个团队的线程数；（3）在 CPU 和 GPU 协同调度计算时选择工作划分。表中记录了哪些用例适用于哪些基准测试，因为这取决于具体应用。例如，由于不规则内存访问需要在设备间进行开销较高的内存更新，因此这类基准测试并不适合 CPU-GPU协同调度。

表4：通过 Apollo 实现自适应执行的开销

	Power9 + V100	英特尔 + P100	AMD + MI50
插桩	646 纳秒	223 纳秒	200 纳秒
模型训练	280 微秒	96 微秒	60 微秒
模型推理	20 纳秒	7 ns	9 ns

我们以相对于基准配置的 加速比 作为性能指标。评估中，我们构建了最大深度为 2 的决策树模型；已有研究表明，这种模型 [29, 18]能够在模型构建和运行时评估之间取得良好的权衡。尽管如此，我们的工作也支持探索其他模型类型和超参数选择。每种适配方式都包含不同的比较器，以展示性能潜力和开销。对于每个程序、输入和比较器配置，我们均运行 5 次，并报告观测到的平均加速比。

4.1 CPU-GPU 执行。

基线是 GPU 执行。我们构建了一个自适应-*100* 模型，该模型使用所有运行的性能分析数据作为输入，包括用于训练的所有输入和变体，以评估拥有完整（100%）性能分析数据视图的模型的预测准确率和最终应用程序性能。此外，我们采用类似于K折交叉验证的实验方法，测试基于部分训练数据构建的模型。具体而言，我们将应用程序输入划分为K 个大小相等的组。使用其中比例为 f 的组进行模型训练，其余组用于测试，以评估自适应执行的性能。通过 K折构造，每个输入至少会在训练集和测试集中各出现一次。在实验中， $K = 4$，并测试不同的训练数据比例场景： $f = 25\%$、50%、75%。 K折构造重复 10 次，每次都打乱输入，以便在置换后的数据上训练不同的模型。我们在适用的测试输入上评估这些模型的性能，并排除用于训练的输入。我们将 f 比例场景称为自适应-*25*、自适应-*50* 和自适应-*75*。这些模型展示了适配的性能：模型基于用于收集性能分析数据的部分输入构建，并用于预测未见输入的优化变体。作为参考，我们还展示了 CPU 执行的性能，记为*Static,CPU*。

4.2 GPU 线程。

基线为使用每个团队 256 个线程进行 GPU 执行，这是原始 OpenMP 卸载实现中的默认配置。与 CPU-GPU 执行类似，我们展示了使用所有可能的数据训练的自适应-*100* 模型的结果。这些实验完全在 GPU 上执行计算，不跨越设备边界，因此计算数据环境完全位于 GPU 上。因此，我们还展示了在线 训练的加速比：让自适应 OpenMP 在每次运行中都训练一个模型，并使用最少量的训练数据，其数量等于不同线程配置（变体）的数量，同时利用程序中的迭代计算生成足够的训练数据。将在线 与自适应-*100* 进行比较，可以看出每次运行都构建专用模型与使用所有配置数据点构建通用模型之间的性能差异。*Static,Best* 展示了各区域中最佳的静态线程配置。

4.3 协同调度。

基线为仅 GPU 执行 。同样，我们构建了一个使用所有可能输入进行训练的自适应-*100* 模型，并纳入 e 基于模型的结果 在性能分析数据子集上训练（自适应-*25*、自适应-*50*和自适应-*75*）。

5 评估

5.1 自适应执行开销

表4展示了 Apollo 在每个不同硬件平台上进行自适应执行的开销。开销主要有三个来源：（1）在每次区域执行过程中收集性能分析数据所产生的插桩开销；（2）收集性能分析数据后，为每个区域训练模型所产生的一次性开销；（3）评估自适应模型所产生的开销，包括探索变体时的评估开销，或模型训练完成后为每次区域执行选择调优变体时的评估开销。对于模型训练和推理，开销测量假设采用一种最大深度为

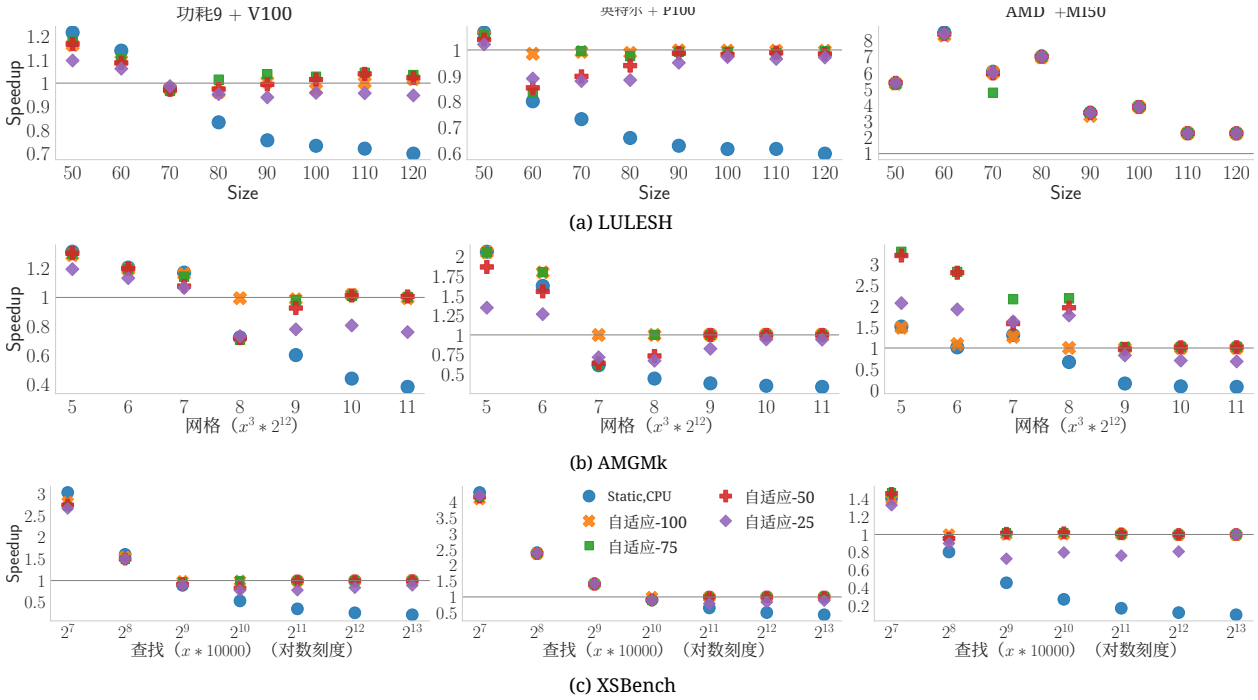

图 2：CPU-GPU 执行适配的结果。

2、基于 2 个不同特征和 10 个唯一性能分析数据元组构建的决策树模型。这是我们用例实验中遇到的最复杂建模方式，因此代表了开销上限。这些开销处于微秒和纳秒量级；鉴于现实应用中区域的执行时间通常显著更长，这些开销非常小。下文的应用程序性能测量不包括探索期间的数据收集，但在收集训练数据后，会计入自适应执行的插桩、模型训练和推理开销。

5.2 选择 CPU-GPU 执行。

图 2 展示了程序 *LULESH*、*AMGMk* 和 *XSBench* 的结果。纵观所有基准测试，首先观察 *Static,CPU* 结果可知，CPU 执行通常仅在较小输入规模下更快，因为这类输入无法充分利用 GPU 强大的并行性。CPU 执行何时会慢于 GPU 执行，具体取决于机器，因此需要进行动态适配。例如，对于 Power9+V100，输入域大小小于=60时，*LULESH* 执行更快；但在英特尔+P100 CPU 上，只有最小输入规模 50 时执行才更快。值得注意的是，在 AMD+MI50 上，CPU 执行始终更快；这是因为 AMD GPU 的 OpenMP 运行时实现尚不成熟，在迭代算法所需的主机与加速器设备之间的数据传输机制方面表现欠佳。尽管如此，自适应模型仍能识别这一情况，并预测 CPU 是执行速度最快的设备。*AMGMk* 内核和代理应用程序 *XSBench*也有类似的观察结果，不过对于较大的输入，AMD GPU 执行确实更快，避免了*LULESH*中出现的性能问题。动态选择 CPU 执行所能达到的加速比取决于具体输入和机器架构：*LULESH*在 Power9 上针对较小输入的加速比最高可达 1.25×，在英特尔上最高可达 1.1×；*AMGMk* 在 Power9 上的加速比最高可达 1.4×，在英特尔上最高可达 2×，在 AMD 上最高可达 4×；而*XSBench* 的加速比分别最高可达 3×、4×和 2×。

结果表明，自适应-*100* 始终能跟踪最快配置，并正确选择在 CPU 或 GPU 上执行，其加速比接近 *Static,CPU*。基于部分输入构建的自适应模型（自适应-*25*、自适应-*50*、自适应-*75*）的结果显示，平均而言，它们也能跟踪最快配置。对于较大的输入，它们从未错误地选择 CPU 执行；对于较小的输入，则通过选择 CPU 执行来获得加速比。随着训练折中纳入的数据增多，选择最快配置的准确率和可实现的加速比都会提高，因此自适应-*75* 的性能接近自适应-*100*。尽管训练数据有限，自适应-*25* 中最小的子集仍能实现加速比。图 3 展示了不同自适应模型和特征输入下，模型决策为在 GPU 上执行的比例，印证了上述发现。

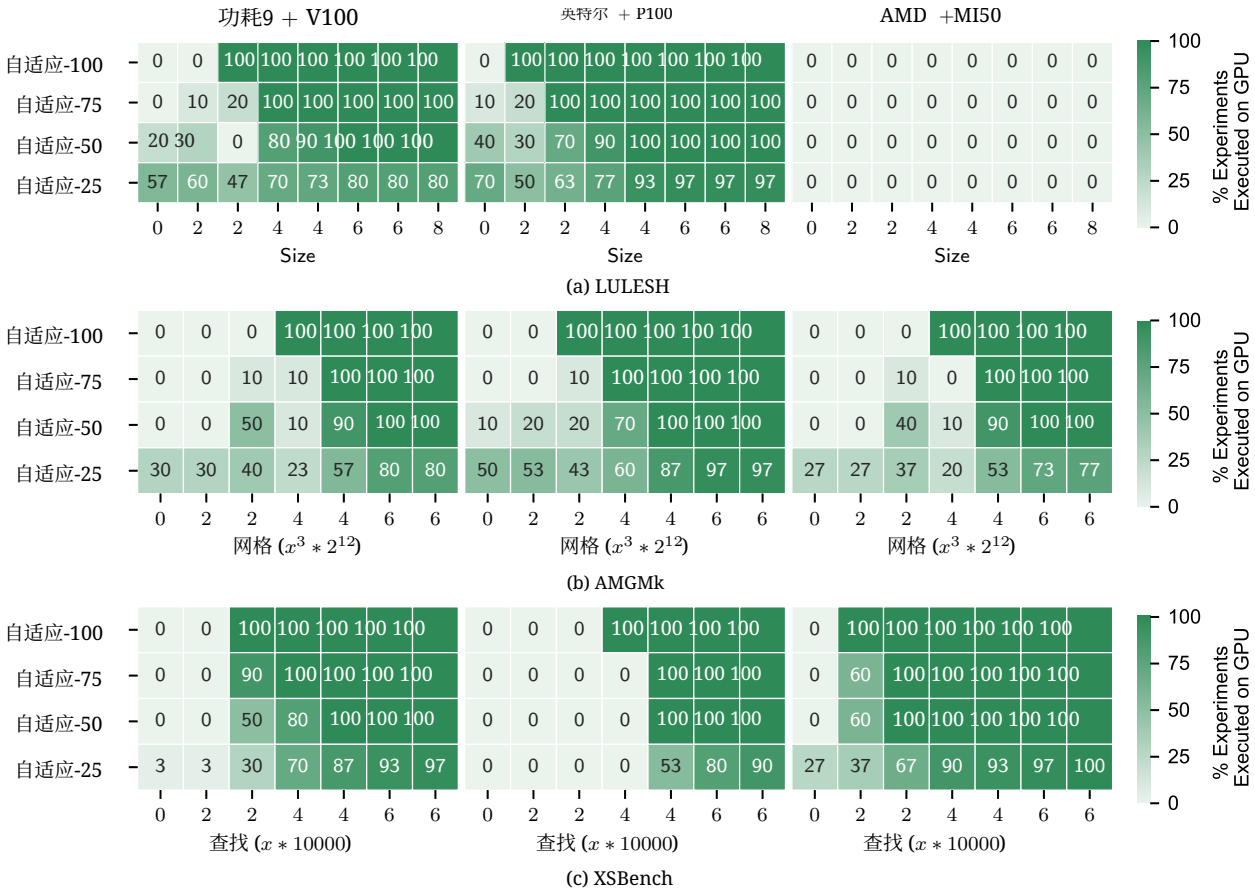

图 3：每 个模型决定在 GPU 上执行的实验百分比。X 轴显示特征值 es.

5.3 选择 GPU 执行的线程数。

图 4 展示了不同程序和系统的 GPU 线程配置适配结果。计算加速比时采用的基准，是使用 256 个 GPU 线程执行的结果；这是原始实现的默认值，按照启发式方法认为它总体上适用。结果验证了这一启发式方法，因为线程适配带来的加速比有限：许多情况下加速比微乎其微，观测到的最大加速比也从未超过 1.4×。不过，线程的最佳选择取决于程序、输入和机器。

进一步来说，如图 4a 所示，*FluidSim* 在英特尔+P100 和 AMD+MI50 上通过线程适配获得了加速比，最高分别达到 1.1× 和 1.15× 。*Heat* 在 Power9+V100 和 AMD+MI50 上均实现了加速比，最高为 1.2× 。*LULESH* 在所有机器上都实现了约 1.1×的加速比；但在 AMD+MI50 上，当输入大小为 70 时是个例外，加速比约为 1.4×。*AMGMk* 的线程适配仅在 Power9+V100 和英特尔+P100 上实现了加速比，最高分别达到 1.25×和 1.15× 。

有趣的是，*Static,Best*为所有区域选择相同的线程配置，因此不依赖输入，而是取决于平台和应用程序。其表现与默认的 256 个 GPU 线程配置相当或更好。对于较大的输入，自适应-*100* 和在线 中的自适应模型大多采用*Static,Best*的选择，获得相近的加速比，尽管它们可能会为每个区域选择不同的线程配置。在某些情况下，在线的性能超过自适应-*100*。前者利用应用程序在特定输入下单次运行的性能分析数据构建，后者则利用应用程序多次运行的性能分析数据构建。在这些情况下，我们发现，在线 针对特定输入特征构建的专用模型，相较于自适应-*100*构建的通用模型，能够生成更优的线程配置。

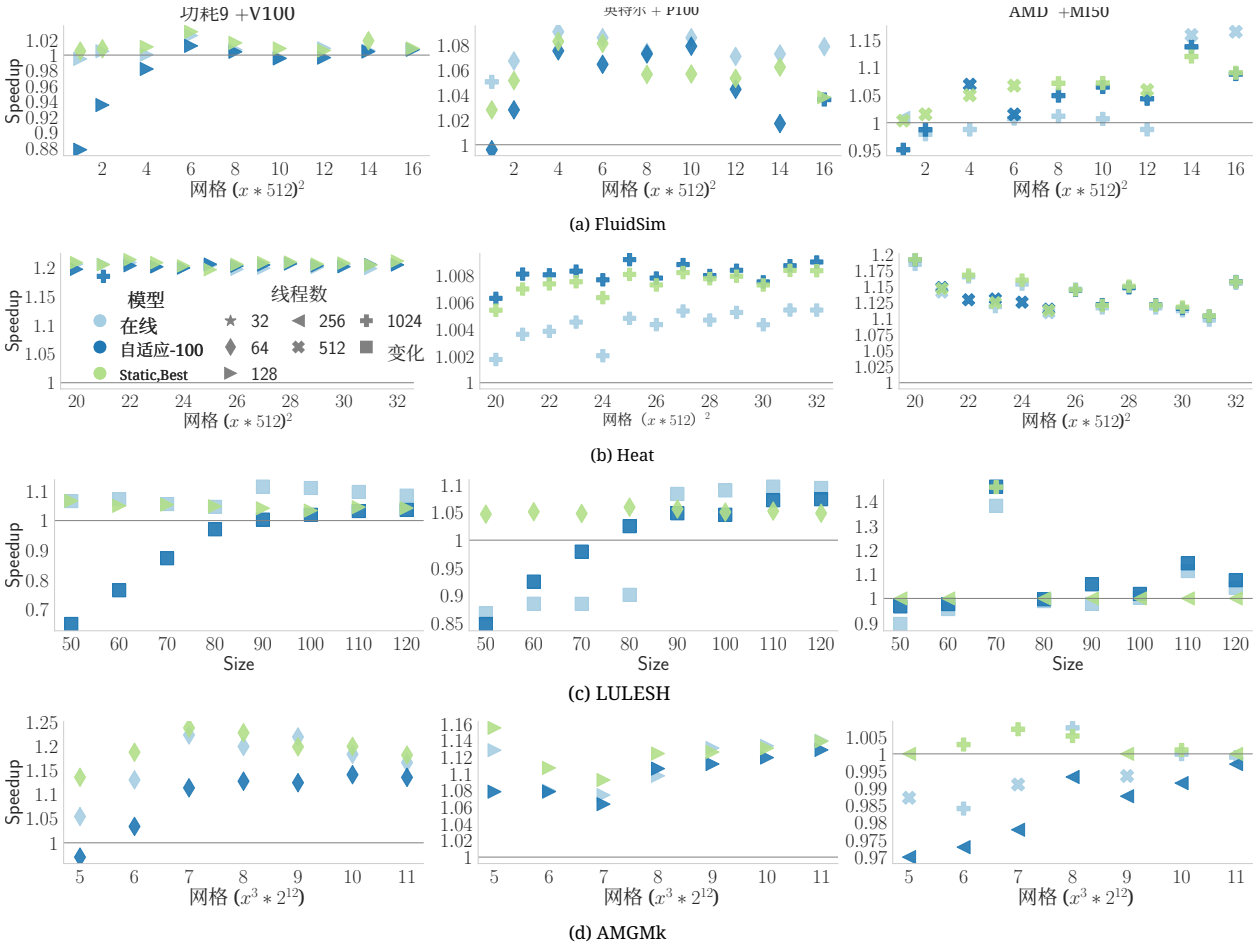

图 4：GPU线程适配的结果。

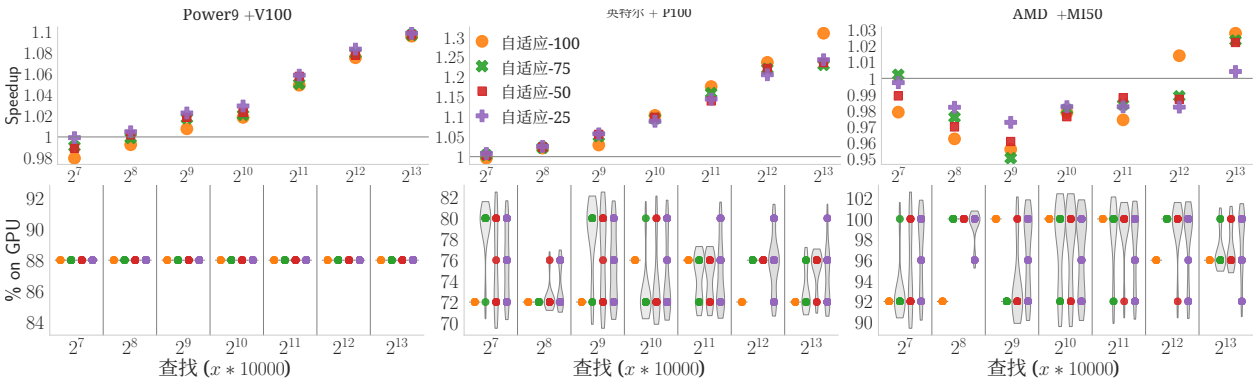

图 5：XSBench。 CPU-GPU协同调度的结果（上）。小提琴图（下）展示了模式 1 决策 ·

5.4 CPU和GPU上的协同调度执行。

图 5 和图 6 展示了*XSBench* 和*RSBench*在 CPU 与 GPU 之间划分计算时的加速比和模型决策分布，并以仅 GPU 执行为基准。由于 LLVM 的 AMD GPU 后端存在编译错误，缺少*RSBench* 在 AMD+MI50 系统上的结果。可能达到的加速比仍会因程序、输入和所部署的系统而异，这再次说明了适配的必要性。

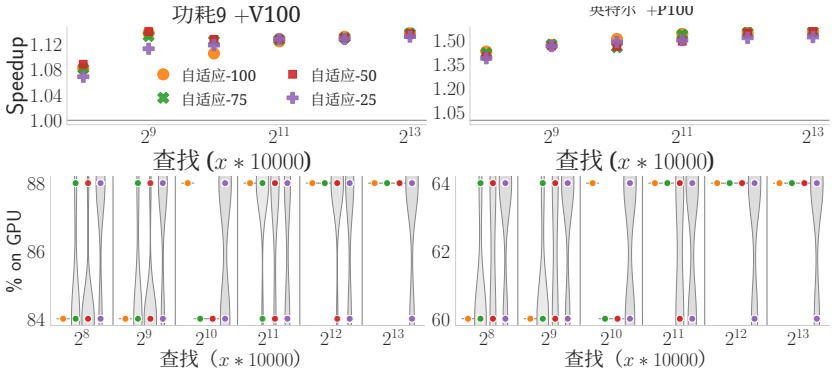

图 6：RSBench。CPU-GPU协同调度的结果（左）。小提琴图（右）展示了模型决策的分布。

具体而言，对于 *XSBench* （图 5），在 Power9+V100 上的加速比有限，只有较大的输入才能实现，在最大输入规模下观察到的最高加速比为 1.1× 。相比之下，在英特尔+P100 上运行的 *XSBench* 始终能从协同调度中获益：最小输入下的加速比为 1.02× ，随着输入规模增大而稳步提升，在最大输入规模下达到约 1.25× 。在 AMD+MI50 上进行协同调度，仅在较大输入下实现了小幅加速。对于 *RSBench* （图 6），协同调度在所有输入规模下均能实现加速；对于最大输入，Power9+V100 上的最高加速比约为 1.14×，英特尔+P100 上则为 1.5× 。自适应-*100* 和数据子集折叠（自适应-*25*、自适应-*50*、自适应-*75*）在两个程序中实现了相近的加速比。有趣的是，仅使用 25% 数据进行训练的 自适应-*25* 也实现了相当的性能。图 5、6 中的小提琴图展示了各模型决策的分布，从而更深入地了解不同模型的分区决策。使用不完整训练数据训练的自适应模型，其预测结果与自适应-*100*模型相近，因此实现的加速比也相当。

6 局限性与挑战

本节将讨论自适应执行的两个主要挑战。

6.1 选择特征

所提出的方法为开发者提供了一种易于使用的编程模型，使其能够适配应用程序的执行，并选择最优变体，以最大限度地缩短应用程序的执行时间。该方法要求开发者确定一组与自适应区域执行时间高度相关的特征变量。在本研究中，我们考察依赖于应用程序输入、能够控制区域计算复杂度的特征变量，例如需要执行的并行迭代次数。我们在评估中通过实验证明，作为一项指导原则，选择这类特征变量足以实现有益的适配。然而，适配所带来的收益和准确性取决于开发者运用领域知识，在我们的自适应方法中选择有用特征。不过，对于某些算法而言，很难或不切实际地确定与代码区域执行时间高度相关的特征变量。例如，共轭梯度（CG）算法的收敛所需迭代次数，很大程度上取决于待求解线性系统的条件数。实际上，在应用程序执行期间可能难以准确估算条件数；若将其排除在特征之外，则会降低适配的潜在效果。尽管如此，底层线性系统的维度，或许还有条件数的粗略估计等特征值，仍能为适配提供实用信息，从而提升性能。

6.2 模型训练的性能分析数据收集

我们的方法需要收集最低数量的性能分析数据点，以训练自适应模型。根据具体用例，数据收集阶段可能需要多次运行应用程序。为了有效训练自适应模型，用户应根据应用程序的典型用例确定代表性输入，并收集其性能数据，这与性能分析引导的优化技术类似。以图 2 所示的 *AMGMk* 中执行设备（CPU 或 GPU）的选择为例，自适应模型需要收集来自较小和较大输入维度的性能分析数据，才能训练出能够泛化到未知输入的有效模型。

应对这一挑战有两种选择。用户可以有意使用代表性输入执行一组训练运行，以覆盖整个领域，但这需要投入一些人工精力和时间来进行训练运行。或者，用户可以完全将数据收集工作交由自适应执行，通过指定一个可能较大的最低训练数据收集量；这意味着在生产环境中运行应用程序时，能够以可接受的探索开销，针对可能的指定变体捕获所需的适应效果。我们更倾向于后一种方案，因为它只需用户进行最少干预；我们认为，采用贝叶斯优化方法探索变体空间，有望在提供有效适配、实现高性能的同时降低探索开销。

7 相关工作

数十年来，研究人员一直在广泛研究利用机器学习技术指导编译器优化。Wang 等人 [27]给出了一份全面的综述。Ashouri 等人 [3] 综述了用于解决编译器优化选择和阶段排序问题的机器学习技术。Milepost GCC 项目 [12] 将 GCC 与机器学习相结合，以预测有利的优化，从而适应不同的架构。为了获得 CPU 与 GPU 计算之间的最佳划分比例，Luk 等人 [19]对执行变体进行性能分析，并建立线性回归模型。
机器学习也是帮助应用程序选择使用 CPU 还是 GPU 的一种常用方法。Grewe 等人 [13]使用决策树模型来判断在 GPU 上运行 OpenCL 内核是否有利。Hayashi 等人 [14] 使用离线监督式机器学习技术，通过即时编译器为各个 Java 内核从 CPU 和 GPU 中选择首选计算资源。为避免手动定义模型特征，DeepTune [8]直接将原始源代码作为输入，构建深度神经网络，以指导 OpenCL 程序的最佳映射。
已有大量研究致力于自动调优技术。典型示例包括 ATLAS [28], Active Harmony [26], FFTW [11], POET [30], CHILL [6]。OpenTuner [2]提供了一个通用框架，用于构建领域专用多目标程序自动调优器。它支持采用多种搜索技术的集成方案。CLTune [20]是一个通用的 OpenCL 内核自动调优器，支持用户定义可能参数值组合的搜索空间。Bliss [24] 提出采用概率式贝叶斯优化，为整个应用程序调优硬件参数（核心频率、超线程）和软件执行参数（OpenMP 线程、算法替代方案），并由用户进行筛选。ANGEL [7]采用分层方法，实现多个目标的在线调优，例如权衡执行时间与功耗。Bari 等人在 [4] 中介绍了 ARCS 框架，用于调优 OpenMP 程序，以优化功耗。

鉴于 OpenMP 在高性能计算领域的普及，针对异构平台调优 OpenMP 程序正受到越来越多的关注。早期的一个例子是 Liao 等人 [17] 开发的源代码提取技术，用于对从大型应用中提取的 OpenMP 循环进行自动调优。Sreenivasan 等人 [25] 提出了一种轻量级的 OpenMP 语用指令自动调优器，用于优化调度策略、块大小和线程数等执行参数。Pennycook 等人 [23] 使用 Mantevo 套件中的 miniMD 基准测试，研究了使用 OpenMP 5.0 引入的*元指令* 和*声明变体* 的优势。他们证实，这些特征能够以更精简的源代码形式表达代码变体，从而实现具有竞争力的性能可移植性，不过这些特征仅支持编译时适应。
此前已有多项工作尝试通过编程接口集成基于机器学习的适配。先前的工作 [18] 尝试使用机器学习增强 OpenMP 中的适配。然而，这项工作仍处于初步阶段，因为所提出的指令仅支持与元指令关联的单一自适应代码模式。此外，该论文在评估中也仅使用了一个内核。Apollo [5, 29] 是一种先进的自动调优库，支持用户定义的可调优区域。用户可以为每个区域指定输入特征和一组代码变体（按 Apollo 的术语称为*策略*），以表达可供调优的选项。通过一系列协同的 API 调用，Apollo 运行时为程序员提供性能分析数据收集、模型构建和模型驱动适配等功能。然而，由于 Apollo 被设计为运行时库，使用它需要大量手动改写代码。

我们的工作与上述研究不同之处在于，我们将机器学习直接融入基于指令的编程模型，从而能够在程序内细粒度代码区域的层面实现更加灵活、透明且自动化的模型驱动运行时自适应。通过支持多个适配代码变体，我们的方法显著提升了面向异构架构的高性能计算编程的可移植性、性能和生产力。我们还通过采用广泛的基准测试，以及使用生产级编译器实现的原型，对该方法进行了更全面的评估。

8 结论

本文提出了一组新的 OpenMP 扩展，使程序员能够表达动态的、由机器学习驱动的自适应。借助生产者-消费者模式，这些扩展支持不同类型的自适应用例，并自动完成模型驱动适配的整个工作流程，包括代码变体生成、性能分析和模型构建。通过参考实现和一组代理应用基准测试，我们证明了该方法切实可行且具有实际价值。

未来，我们计划扩展 适配 指令，使其支持更多类型的代码变体。我们还将研究高级分析方法，以自动确定适合适配的代码区域、代码变体的选择、构建模型所需的特征，以及合适的机器学习模型。

致谢

本研究由劳伦斯利弗莫尔国家实验室在美国能源部的支持下完成，合同编号为 DE-AC52-07NA27344（LLNL-CONF-833682）；部分经费由 LLNL-LDRD 计划项目编号 21-ERD-018 以及美国能源部科学办公室先进科学计算计划（ASCR SC-21）提供。

参考文献

[1] 自适应 *OpenMP* 实现。将以开源形式提供。2023年。[2] Jason Ansel 等人。“OpenTuner：一种可扩展的程序自动调优框架”。载于：第*23*届并行体系结构与编译国际会议论文集。2014年，第303–316页。[3] Amir H Ashouri 等人。“基于机器学习的编译器自动调优综述”。载于：*ACM*计算调查（*CSUR*）51.5（2018），第1–42页。[4] M. A. S. Bari 等人。“ARCS：面向功耗受限 OpenMP 应用的自适应运行时配置选择”。载于：*2016*年 *IEEE* 国际集群计算会议（*CLUSTER*）。2016年，第461–470页。DOI：10.1109/CLUSTER.2016.39。[5] David Beckingsale 等人。“Apollo：用于快速动态调优输入相关代码的可复用模型”。载于：*2017*年 *IEEE* 国际并行与分布式处理研讨会（*IPDPS*）。IEEE。2017年，第307–316页。[6] Chun Chen、Jacqueline Chame 和 Mary Hall。*CHiLL*：用于组合高级循环变换的框架。技术报告。Citeseer，2008年。[7] Ray S Chen 和 Jeffrey K Hollingsworth。“Angel：一种面向多目标在线自动调优的分层方法”。载于：第*5*届运行时与超级计算机操作系统国际研讨会论文集。2015年，第1–8页。[8] Chris Cummins 等人。“优化启发式方法的端到端深度学习”。载于：*2017*年第*26*届并行体系结构与编译技术国际会议（*PACT*）。IEEE。2017年，第219–232页。[9] Alejandro Duran 等人。“OmpSs：一种面向异构多核体系结构的编程方案”。载于：《并行处理快报》21.02（2011），第173–193页。[10] H. Carter Edwards、Christian R. Trott 和 Daniel Sunderland。“Kokkos：通过多态内存访问模式实现众核性能可移植性”。载于：《并行与分布式计算杂志》74.12（2014）。高性能计算的领域专用语言与高级框架，第3202–3216页。ISSN：0743-7315。DOI：https://doi.org/10.1016/j.jpdc.2014.07.003。URL：http://www.sciencedirect.com/science/article/pii/S0743731514001257。[11] Matteo Frigo 和 Steven G Johnson。“FFTW3 的设计与实现”。载于：《*IEEE*会刊》93.2（2005），第216–231页。[12] Grigori Fursin 等人。“Milepost GCC：支持自调优编译器的机器学习”。载于：《国际并行编程杂志》39.3（2011），第296–327页。[13] Dominik Grewe、Zheng Wang 和 Michael FP O’Boyle。“面向异构系统的 OpenCL 数据并行程序可移植映射”。载于：*2013*年 *IEEE/ACM* 代码生成与优化国际研讨会（*CGO*）论文集。IEEE。2013年，第1–10页。[14] Akihiro Hayashi 等人。“用于运行时 CPU/GPU 选择的机器学习性能启发式方法”。载于：*Java*平台编程原理与实践会议论文集。2015年，第27–36页。[15] Richard D Hornung 和 Jeffrey A Keasler。“RAJA 可移植性层：概述与现状”。载于：（2014年）。

[16] Zheming Jin。*SYCL* 中的 *Rodinia* 基准测试。技术报告。橡树岭国家实验室（ORNL），美国田纳西州橡树岭，2021年。[17] Chunhua Liao 等人。“支持全程序经验优化的有效源到源提纲化”。载于：并行计算语言与编译器国际研讨会。Springer。2009年，第308–322页。[18] Chunhua Liao 等人。“扩展 OpenMP 以支持机器学习驱动的自适应”。载于：第八届基于指令的加速器编程研讨会（2021年）。[19] Chi-Keung Luk、Sunpyo Hong 和 HyesoonKim。“Qilin：通过自适应映射利用异构多处理器中的并行性”。载于：*2009*年第*42*届 *IEEE/ACM* 微体系结构国际年会（*MICRO*）。IEEE。2009年，第45–55页。[20] Cedric Nugteren 和 Valeriu Codreanu。“CLTune：一种通用的 OpenCL 内核自动调优器”。载于：*2015*年 *IEEE* 第*9*届嵌入式多核/众核片上系统国际研讨会。IEEE。2015年，第195–202页。[21] Open MP 架构审查委员会。*OpenMP* 应用程序编程接口 *5.0*。2018年。URL：https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf。[22] OpenMP 架构审查委员会。*OpenMP* 应用程序编程接口 *5.1*。2020年。URL：https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-1.pdf。[23] Simon J. Pennycook、Jason D. Sewall 和 Jeff R. Hammond。“评估拟议的 OpenMP 5.0 特性对性能、可移植性和生产力的影响”。载于：*2018*年 *IEEE/ACM* 高性能计算中的性能、可移植性与生产力国际研讨会（*P3 HPC*）。2018年，第37–46页。DOI：10.1109/P3HPC.2018.00007。[24] Rohan Basu Roy 等人。“Bliss：使用多样化轻量级学习模型集合自动调优复杂应用”。载于：第*42*届 *ACM SIGPLAN* 编程语言设计与实现国际会议论文集。PLDI 2021。线上会议，加拿大：计算机协会，2021年，第1280–1295页。ISBN：9781450383912。DOI：10.1145/3453483.3454109。URL：https://doi.org/10.1145/3453483.3454109。[25] Vinu Sreenivasan 等人。“支持 OpenMP 自动调优的框架”。载于：*OpenMP* 国际研讨会。Springer。2019年，第50–60页。[26] Cristian Tapus、I-Hsin Chung 和 Jeffrey K. Hollingsworth。“Active Harmony：迈向自动化性能调优”。载于：*2002*年 *ACM/IEEE* 超级计算会议论文集。SC ’02。马里兰州巴尔的摩：IEEE计算机学会出版社，2002年，第1–11页。ISBN：076951524X。[27] Zheng Wang 和 Michael O’Boyle。“ 编译器优化中的机器学习 ”。载于：《 *IEEE* 会刊》 106.11 （ 2018 ），第 1879–1901页。[28] R Clinton Whaley 和 Jack J Dongarra。“自动调优的线性代数软件”。载于：*SC’98*：*1998* 年 *ACM/IEEE* 超级计算会议论文集。IEEE。1998年，第38–38页。[29] Chad Wood 等人。“Artemis：使用机器学习自动调优并行执行参数”。载于：高性能计算国际会议。Springer。2021年，第453–472页。[30] Qing Yi 等人。“POET：用于经验调优的参数化优化”。载于：*2007*年 *IEEE* 国际并行与分布式处理研讨会。IEEE。2007年，第1–8页。