arXiv:2305.01886v1 [cs.DC] 3 May 2023

GPGPU 应用程序的性能和功耗预测

论文

作为部分满足学位要求而提交的
论文。

哲学博士

by

ALAVANI GARGI KABIRDAS

在.....的指导下

SANTONU SARKAR 教授

以及共同指导勘误 зегь

NEENA GOVEAS 教授



BITS Pilani
Pilani|Dubai|Goa|Hyderabad

比拉理工学院（皮拉尼）2023

摘要

Graphics 处理单元（GPU）已成为高性能计算不可或缺的一部分，有助于实现百亿亿次级性能。GPU 应用程序开发者的主要目标是对代码进行全面调优，以充分利用 GPU 上的各种资源，从而获得最佳性能。在高性能计算基础设施上提升应用程序性能的同时，开发人员还应确保应用程序的能耗尽可能低，尤其要考虑到数据中心和高性能计算服务器的巨大功耗。为此，开发者必须借助相应工具，在启动应用程序之前了解其能耗。我们坚信，了解并预测 GPU 性能和功耗，将有助于开发者针对给定架构，设计出以性能为导向且能效感知的应用程序。

本论文介绍了两种模型，它们通过对 CUDA 代码进行静态和动态分析，帮助开发者分析 CUDA 内核的能量耗散。第一种模型用于预测 CUDA 内核的执行时间。该模型对 PTX 代码进行静态分析，以提取指令特征、控制流和数据依赖。我们提出了两种满足性能约束和硬件约束的调度算法。第一种方法通过考量指令级并行（ILP）和线程级并行（TLP）来预测性能约束；GPU 利用这两种并行性来提升性能。第二种方法利用资源预留约束，在流式多处理器（SM）上调度各线程中的指令。我们通过动态分析建立一组内存访问惩罚模型，并结合调度信息，利用这些模型估算代码的执行时间。

第二个模型是基于静态分析、利用机器学习技术构建的功耗预测模型。构建该模型所用的特征通过对 PTX 代码进行静态分析得到。这些特征旨在揭示图形处理器功耗与程序特征之间的关系，从而帮助开发人员构建节能、可持续的应用程序。用于验证这两个模型的数据集包含来自不同基准测试套件的内核，涵盖不同的规模、类型（例如计算密集型、内存密集型）和复杂度（例如控制分歧、内存访问模式）。开发人员可以借助这两个模型重构代码，构建能效感知型 GPU 应用程序。我们还介绍了一款工具，并通过实际验证证明，这两个模型可作为图形处理器设计辅助工具，具有良好的实用性且易于使用。

目录

Page

表格列表 **ix**

图目录 **xi**

1 引言 **1**

1.1 研究动机.3 1.2 背景——图形处理器架构与编程模型.6 1.2.1 CUDA编程模型.9 1.2.2 CUDA编程模型到图形处理器架构的映射 . 10 1.2.3 GPU能耗建模.11 1.3 问题陈述.12 1.3.1 目标1.12 1.3.2 目标2.14 1.3.3 目标3.15 1.4 研究贡献.16 1.5 本研究的出版物.17 1.6 论文大纲.19

2 文献综述 21

2.1 设计辅助.21 2.2 GPU应用能耗预测.23 2.3 GPU性能建模.25 2.3.1 算法方法.25 2.3.2 统计方法.26 2.3.3 用于性能评估的模拟器.28 2.3.4 解析模型.28 2.3.5 延迟隐藏.33

2.3.6 微基准测试.34 2.4 GPU功耗建模.34

2.4.1 统计方法.36 2.4.2 动态电压频率调节 (DVFS).38 2.4.3 功耗预测模拟器.38

2.5 现有工作的局限性.39 2.5.1 性能预测.39 2.5.2 功耗预测.41 2.6 D

讨论.4 2

3 微基准测试 45

3.1 PTX指令集架构.46 3.1.1 生成PTX文件.47 3.2 计算指令.47 3.2.1 计算指令延迟结果.50 3.3 内存指令.52 3.3.1 指针追踪.53 3.3.2 全局内存指针追踪.53 3.3.3 全局内存延迟模型.54 3.3.4 共享内存访问延迟.56 3.4 内核启动开销.57 3.5 讨论.57

4 GPU性能建模 61

4.1 抽象 GPU 模型.62 4.2 静态分析.63 4.3 使用 ILP 和 TLP 建模 GPU 性能.65 4.3.1 提出的模型.66 4.3.2 延迟计算算法.67 4.3.3 GPU 调度算法.73 4.3.4 实验详情与数据收集.76 4.3.5 结果.77 4.3.6 根据并行计算侏儒对结果进行分类.77 4.3.7 所提算法的局限性.80 4.4 使用资源分配对 GPU 性能进行建模.81 4.4.1 分析循环和分支发散.83

4.4.2 调度模型.84 4.4.3 其他假设.86 4.4.4 调度算法.87 4.4.5 动态分析.91 4.4.6 结果.97 4.4.7 实验设置.97 4.4.8 数据收集.98 4.4.9 Kepler 架构的结果.100 4.4.10 不同 GPU 架构的结果.101 4.4.11 分析.105

4.5 讨论.109

5 功耗 预测 11 **1**

5.1 为什么采用基于机器学习的功耗预测方法? . . . . . 111 5.2 初步调查.112 5.2.1 考察的特征.113 5.2.2 特征工程.113 5.2.3 模型构建与结果.115 5.2.4 结果与分析.115 5.3 提出的模型.118 5.4 特征识别与获取.119 5.4.1 每线程寄存器数、每线程块共享内存.120 5.4.2 占用率.120 5.4.3 指令发射周期.121 5.4.4 内存访问开销.122 5.4.5 程序特征提取算法.124 5.4.6 数据集创建.129 5.4.7 数据集中噪声出现的可能性.130 5.4.8 其他假设.131 5.5 特征工程.133 5.5.1 派生属性：.133 5.5.2 非相关属性.133 5.5.3 显著影响属性.137 5.6 模型评估指标.138 5.7 模型选择与评估.139

5.7.1 人工神经网络实验与分析.140 5.7.2 基于支持向量回归的模型.143 5.7.3 基于随机森林的回归器.144 5.7.4 极端随机树回归器.146 5.7.5 梯度提升回归器.146 5.7.6 XGBoost回归器.147 5.7.7 CatBoost.148 5.7.8 模型构建与训练：实现细节.148 5.8 最终模型分析.151

5.8.1 从架构视角分析重要特征 . . . . 156 5.8.2 异常值检测.157 5.8.3 使用并行计算侏儒分析预测模型 161

5.9 讨论.162

6 工具设计 **165**

6.1 工具架构与实现.166 6.1.1 程序分析器.167 6.1.2 执行时间预测器.168 6.1.3 功耗预测器.168 6.1.4 报告生成器.168

6.2 插件使用详情.169 6.2.1 前置条件.169 6.2.2 插件设置.169 6.2.3 用户输入.170 6.2.4 结果生成.171 6.2.5 案例研究：矩阵乘法.171 6.3 功耗与执行时间预测结果比较.174 6.4 讨论.175

7 结论与未来方向 177

7.1 总结.177 7.2 未来工作方向.178

作者简介 **181**

参考文献 **183**

附录 A **199**

基于动态分析的模型.199 图形处理器事件.199 结果.202

附录 B **205**

微基准测试.205 内核启动开销代码片段.205 指针追逐基准测试代码片段.205 全局内存吞吐量模型的有效性.206 与 Volkov 等人的吞吐量模型比较.206 主要结论.208

表格列表

表格页

3.1 Tesla K20 计算指令摘要.51 3.2 测得的 PTX 指令延迟 $\mathscr{L}$.51 3.3 Tesla K20 GPU 全局内存访问指令的延迟计算 55 3.4 全局内存访问指令延迟模型的评估.55 4.1 模型属性所用符号.64 4.2 GPU调度算法结果.78 4.3 Dwarf 及其平均绝对误差.80 4.4 不同统计函数下动态分析参数的建模结果.93 4.5 全局和共享内存吞吐量指数模型.93 4.6 所考察 GPU 的硬件特征值.98 4.7 研究中的基准测试.99 4.8 实际执行时间与预测执行时间（μ秒）.101 4.9 不同 GPU 架构下各 Dwarf 的平均绝对百分比误差.104 4.10 吞吐量比较：我们的方法与 Volkov 的 [20] 方法 . . . . 105 4.11 与现有方法的比较.109 5.1 特征选择所考虑的特征.113 5.2 初步案例研究所考虑的特征.115 5.3 不同 GPU 架构下的结果.116 5.4 特征选择所考虑的特征。突出显示的行表示数据驱动方法（第5.5节）最终选出的特征.127 5.5 所收集功耗值的标准差.133 5.6 人工神经网络模型实验结果.144 5.7 机器学习模型及其超参数.148

5.8 不同 GPU 架构下机器学习算法的验证分数 . 151 5.9 运行 5 次后，三种架构下的 XGBoost 结果.153 5.10 XGBoost 与 CatBoost 评估.154 5.11 剔除异常值后机器学习技术的验证分数 . . . . . 161 5.12 并行计算侏儒的平均绝对误差.162 6.1 不同架构下矩阵乘法的结果. 172 6.2 所选基准测试的性能与功耗结果比较 174 1 机器学习算法的验证分数.203

图目录

图表 **Page**

1.1 Kepler GK110 架构.8 1.2 Tesla K20 SM 架构.8 1.3 GPU 内存设计.8 1.4 论文主题流程及映射至研究目标的出版物.18 2.1 跨越鸿沟.24 2.2 综述大纲.25 3.1 具有指令级并行的融合乘加 =1 的吞吐量.48 3.2 全局指令延迟的分段模型.58 3.3 内核启动开销.58 4.1 CUDA 代码的静态分析.65 4.2 提出的执行时间预测模型.68 4.3 执行时间预测模型.79 4.4 提出的性能预测模型.83 4.5 启动参数对内核性能的影响.85 4.6 算法6示意图.90 4.7 全局内存吞吐量计算模型.93 4.8 Kepler 架构两款 GPU 的误差观测结果.100 4.10 不同 GPU 架构下计算流体力学求解器的实际执行时间与预测执行时间103 4.11 Konstantidis 模型与所提出模型在 Tesla K20 GPU 上对 Rodinia 内核的预测误差对比.105 5.1 皮尔逊相关系数.114 5.2 Tesla K20 的相对特征重要性.117 5.3 功耗预测建模方法概述.119 5.4 程序特征提取算法示例.126

5.5 皮尔逊相关系数热图.136 5.6 肯德尔相关系数热力图.137 5.7 最终特征集的皮尔逊相关系数热图.138 5.8 使用 (a) 随机森林 (b) SHAP 绘制的特征重要性图.139 5.10 特征与功耗之间的关系.142 5.11 自适应学习率人工神经网络的损失函数图.143 5.12 常用基准测试的实测功耗与预测功耗 . . . . . 150 5.13 常用基准测试的功耗预测误差.152 5.14 最精确的基于树的方法的损失函数图.153 5.15 关键程序特征对功耗的依赖关系图 . . 155 5.17 XGBoost 所有特征的部分依赖图 (PDP).159 5.18 使用 HDBSCAN 得到的离群点得分密度.160

6.1 能耗估算工具概述.166 6.2 能耗估算工具的功能架构.167 6.3 能耗估算工具.170 6.4 能耗估算结果.171 6.5 插件中的 LIME 输出生成.172 6.6 使用 LIME 进行功耗预测分析.173 6.7 功耗与执行时间预测结果比较.175 1 能耗预测的特征重要性.203 2 功耗预测的特征重要性.203 3 吞吐量模型与 Volkov 模型对比.206 4 使用屋顶线模型进行比较.207

缩略语表

HPC	高性能计算
GPU	图形处理器
通用图形处理器	通用图形处理器。由 GPU 和通用图形处理器, 本文中所指的是 NVIDIA GPU
$CUDA$	统一计算设备架构
SM	流式多处理器
PTX	并行线程执行
SP	单精度核心
ILP	指令级并行
TLP	线程级并行
WS	线程束调度器
SFU	特殊函数单元
DFU	双精度单元
LSU	加载存储单元
SM	流式多处理器
RF	随机森林
SVR	支持向量回归
ANN	人工神经网络
XGBoost	极端梯度提升
CatBoost	类别提升
极端随机树	极端随机树

CHAPTER 1

引言

Scientific

20 世纪 90 年代初的计算需要数据并行计算，因此早期的高性能计算（HPC）机器 CM-5 应运而生，它由一个协处理器数组组成。CM-5 开创了将向量和矩阵计算卸载到协处理器数组上的理念。当时，高性能计算编程社区规模较小，成员高度专业化，擅长手工编写程序以充分利用硬件基础设施，并借助编译器提供的一些支持，例如对部分代码进行自动向量化。高性能计算硬件和软件

这类基础设施从未向普通编程群体开放。过去二十年间，随着 Cell 处理器、通用图形处理器或 Xeon Phi 加速器的问世，这一局面发生了变化。大量标量处理器与加速器相结合，在异构计算的框架下实现了百亿亿次浮点运算级性能 [1]。此外，这些处理器和加速器如今已作为商品化硬件供应。与此同时，标量 CPU 已发展为多核和众核处理器。因此，高性能计算

[1] https://www.top500.org/lists/

（HPC）如今已不再局限于科学界，编程群体也普遍能够负担得起。因此，计算范式已从串行转向并行 [1]。此外，这些处理器和加速器如今也以图形处理器（GPU）的形式作为商品化硬件供应。

图形处理器是大多数现代超级计算机、高性能计算集群、各种规模的服务器以及商品化硬件不可或缺的一部分。全球排名前五的超级计算机中，有三台利用图形处理器实现百亿亿次浮点运算性能。尽管图形处理器传统上是为渲染图像而设计的，因为多核机器中的每个核心都可以渲染图像中的一个像素。在过去二十年里，图形处理器的这一能力被用于执行计算密集型任务，这些任务需要高度的数据处理并行性，例如深度学习 [2, 3]。许多科学和工业应用程序都通过利用图形处理器的计算能力来实现，其中，应用程序中那些可轻松并行、且处理大型数据集的部分由图形处理器执行 [4]。由于每个数据元素都需要独立执行同一组操作，因此每个图形处理器线程都能够完成这项任务。例如，领先的汽车制造商特斯拉正在使用一台配备 NVIDIA GPU核心的超级计算机，训练用于特斯拉 Autopilot 自动驾驶系统的深度神经网络[5]。

尽管 GPU 因其可编程性和高吞吐量而广受欢迎，但其能耗是一个至关重要的问题 [6–8]。目前，全球最快的超级计算机 Fugaku 每天消耗 717576 千瓦时能量，几乎是其前代超级计算机 Summit（242304 千瓦时/天）的三倍 [2]。如果每台新的高性能需求机器都使能耗增加三倍，这将成为迈向可持续绿色计算道路上的最大障碍。此外，能耗的增加带来了温度管理问题，导致需要采用冷却系统，而冷却系统也会造成贡献

[2]https://www.top500.org/lists/

这会导致能耗增加以及维护成本上升。因此，理解和预测 GPU 能耗已成为当务之急。随着 GPU 每种新架构的演进，这项任务变得更加复杂且更具挑战性。在本论文中，我们致力于估算 NVIDIA GPU 应用程序在开发过程中可能消耗的能量。此类支持可以成为构建能效感知型 GPU 应用程序的有用工具，并以更广阔的视角推动可持续计算。我们首先讨论本项工作的研究动机，然后深入介绍其背景。

1.1 研究动机

能耗是高性能计算系统迈向百亿亿次浮点运算性能这一圣杯过程中面临的主要挑战之一 [6, 7]。过去，CPU 能效的提升是摩尔定律带来的副产品；摩尔定律准确预测了 CPU 上的晶体管数量每两年翻一番。然而，CPU 能效的提升速度一直在放缓。由于 CPU 的扩展已触及这一功耗墙，人们正将重点转向多核和并行架构，以提高能效。选择 GPU 的一个重要因素是，与 CPU 相比，GPU 在高吞吐量和高延迟应用程序中能够提供更高的每瓦性能比 [7, 8], 。然而，如果未能高效利用 GPU，那么 GPU 的能耗将令人严重担忧；其能耗几乎是 CPU 能耗的三倍 [9]。

为了避免能源债务 [10, 11], ——即因代码编写不佳而造成的能源浪费——深入了解应用程序在代码层面的能耗至关重要。绿色软件基金会发布了软件碳强度（SCI），这是一项用于评估软件系统碳排放的初步规范 [12]。SCI 旨在帮助用户和开发者根据碳排放情况做出明智的选择

，以决定在软件开发中使用哪些工具、方法、架构和服务。GSF 指出，减少软件碳排放有三种方式：（a）减少硬件使用，（b）降低能耗，以及（c）提高碳意识。考虑到能耗对碳排放的影响，我们必须为 GPU开发者配备相关工具，帮助他们构建节能应用程序，从而实现可持续计算的目标 [13]。在应用程序运行之前准确预测其能耗，是一项重要的研究课题。鉴于并行编程正迅速应用于各类应用领域 [14]，，如果现在不研究这一问题，开发者就只能运行程序来计算能耗，而这种做法并非总是可行、环保且经济。因此，在本论文中，我们尝试为这一关键的能耗预测问题提出解决方案。

应用程序的能耗可以通过静态 [15, 16] 或动态分析 [17, 18]来预测。纯静态分析预测是通过分析应用程序代码而不执行应用程序来实现的，而基于动态分析的方法则需要在目标硬件上运行应用程序进行评估。基于动态分析的模型可能更准确，因为收集到的数据能够最准确地反映应用程序的实际执行情况。然而，这违背了能耗预测的初衷，因为多次运行应用程序会导致能量浪费。此外，使用专用硬件的动态分析模型无法做到架构无关，因为该硬件可能与较新的架构不兼容 [6]。Chen 和 Shi[19] 主张使用基于软件的预测工具；尽管这些工具的预测准确率较低，但它们具有更好的灵活性，尤其是在无法获得所需专用硬件时。

考虑到硬件的架构复杂性，以及供应商不会透露关键信息，纯粹基于静态分析的模型是一个

这是一个具有挑战性的待解决问题。架构复杂性涉及诸多因素，例如图形处理器复杂的内部内存层次结构以及数千个处理器，这些因素给能耗建模带来了诸多障碍。此外，对于 GPU 应用程序，可以对 PTX（并行线程执行）代码进行静态分析。PTX 是 CUDA（统一计算设备架构）内核的一种中间表示形式。由于硬件级优化，PTX 代码中的所有指令可能并不都会在硬件上执行。

基于静态分析的模型的优势在于，它可用于设计能耗感知型 GPU应用程序。在开发应用程序时，它可以协助应用程序开发者使用适当的程序特征，从而使应用程序部署在 GPU 上时，图形处理器消耗更少的能量。如果没有任何应用程序开发支持，这种微调在实现过程中是无法实现的。因此，迫切需要一种足够准确的基于静态分析的能耗预测模型，该模型无需运行应用程序即可预测能耗，并且能够根据不同的 GPU 硬件特征进行建模。

针对图形处理器的能耗建模解决方案已采用基于硬件参数 [20, 21],、模拟器 [22, 23],以及指令级源代码分析 [24]的统计技术提出。我们观察到，与性能分析支持工具的开发相比，GPU应用程序开发投入了大量更多的精力。现有的性能分析器 [25, 26], 和模拟器 [22, 27]仅展示应用程序的执行性能数据。开发人员需要具备相应的专业知识，才能分析这些信息、理解应用程序的执行流程，并推断其性能瓶颈。这些任务无疑都非常耗时，而且高度依赖专业技能。

这些观测值促使我们构建一个能耗预测模型，用于估算 NVIDIA CUDA 内核应用的能耗，而无需执行该应用。我们提出了一种由两个子模型组成的图形处理器能耗模型，将二者结合后，

即可进行能耗估算。第一个模型用于预测应用程序的执行时间（性能）。第二个模型通过分析代码来估算功耗。这些模型可集成到设计辅助工具中，帮助 GPU程序员开发能耗感知型和节能型应用程序。所提出的方法应当与架构无关，并可通过调整一些参数扩展到更新的架构。

任何图形处理器应用程序开发者都可以轻松使用所提出的模型，并发现它们在多个领域中的实用价值。例如，可以在基于云的图形处理器租赁服务 [28]中使用性能预测模型。在这种场景下，开发者希望通过合理的近似预测所需的运行时间，以便确定需要租用该硬件多长时间，从而规划预算。

尽管我们计划设计一个用于能耗的静态分析模型，但能量耗散与架构组件之间的关系更加复杂。因此，如果不理解架构构造，就无法开发能耗建模解决方案。因此，在下一节中，我们将介绍 GPU架构的相关背景知识，这些知识对于理解我们提出的 CUDA 内核性能和功耗预测解决方案至关重要。如前所述，我们提出的是一种基于静态分析的解决方案，因此需要了解 GPU 所采用的编程模型。因此，我们还将在下一节中讨论 GPU 编程模型所需的背景知识。

1.2 背景——GPU架构与编程模型

图形处理器作为 CPU 的协处理器，可加速通用计算应用程序中的任务。CPU 会将部分计算密集且耗时的

代码卸载到图形处理器上，利用其强大的并行处理能力提升性能。要构建能够捕捉其执行细节的经验模型，深入了解图形处理器的架构至关重要。

本文提出的模型与架构无关，可适用于多种 GPU 架构。本文广泛用于实验和验证的图形处理器是 NVIDIA Tesla K20，下面将对其进行介绍，以便了解 GPU 的架构细节。
NVIDIA Tesla K20 GPU（图 1.1）基于 Kepler GK110 架构，计算能力为 3.5，由 13 个计算组件构成，这些组件称为流式多处理器（SM）。每个 SM 均包含完全流水化的 192 个单精度 CUDA 核心、64 个双精度单元（DPU）和 32 个特殊函数单元（SFU）[3]。每个 SM 将线程分成每组 32 个并行线程进行调度，这样的线程组称为线程束。每个 SM 配备四个线程束调度器和八个指令分派单元（图 1.2），因此可以同时发出并执行四个线程束。Kepler 的四线程束调度器会选择四个线程束，每个线程束每个周期可分派两条独立指令。

GPU架构提供多种内存选项，例如全局、共享、常量和纹理，开发者可以根据应用程序需求加以利用。常量内存和纹理内存是图形处理器上可用的只读内存空间。由于我们专注于通用GPU计算（通用图形处理器），因此在模型中考虑全局内存和共享内存，因为常量内存和纹理内存更多用于图形应用程序。我们在图1.3中展示了CUDA线程的内存设计。CUDA核心可以访问从寄存器文件中分配的私有寄存器集。共享内存分配给线程块，而全局内存空间位于网格内。因此，线程块内的线程可以使用共享内存共享数据，但要访问跨线程块的数据，必须从全局内存中访问这些数据。L1缓存与共享内存共存，并

[3]https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/tesla-product-literature/NVIDIA-Kepler-GK110-GK210-Architecture-Whitepaper.pdf



图 1.1：Kepler GK110 架构

全局内存访问始终缓存在二级缓存中。



图 1.2：Tesla K20 SM 架构

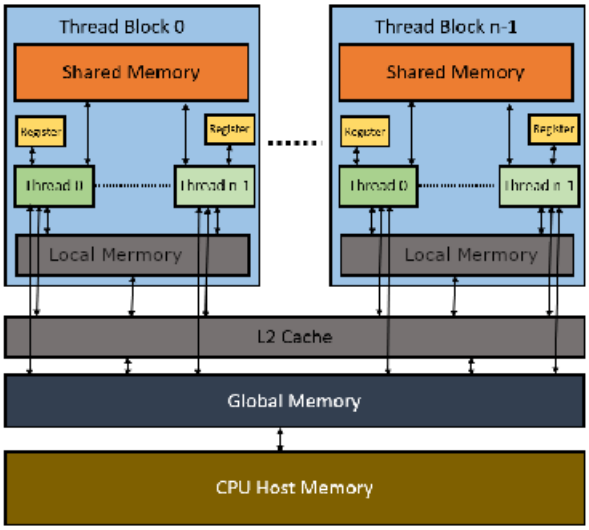


图 1.3：GPU 内存设计

1.2.1 CUDA编程模型

NVIDIA GPU 可通过 CUDA（统一计算设备架构）进行编程，以用于 GPGPU 应用程序。CUDA 是对标准 ANSI C 的扩展，其中预期由图形处理器中数千个线程执行的代码部分以内核函数的形式表示。尽管 CUDA 是 ANSI C 的扩展，但它提供了与 GPU 架构组件相关的底层构造，例如线程级操作，以及编程时对不同类型内存的选择。借助这些特征，CUDA 可直接扩展 GPU 架构。以往研究发现，CUDA 构造与能耗之间存在很强的相关性，例如全局内存指令 [29] ,块数 [30, 31] 等。因此，本节仅讨论 CUDA 编程模型中与硬件相关、且将用于构建所提出模型的特征。

在图形处理器上执行的内核函数使用 __全局__ 关键字声明。所有将在图形处理器上执行的指令都包含在此函数中。内核通过线程组层次结构启动：网格、线程块和线程。一个网格由一组执行内核函数的线程块组成，而每个线程块由数百个线程组成。

CUDA 通过定义设备的逻辑架构来提供设备抽象，即使图形处理器发生变化，该架构也保持不变 [32]。CUDA 还提供了分配资源的灵活性。例如，程序员可以使用线程块，通过共享内存共享数据，并利用内置原语同步操作。CUDA编程模型假设采用单指令多线程（SIMT）计算，指令以锁步方式执行。线程通过函数 同步_线程()进行同步，该函数为线程提供屏障。此外，程序员还可以选择在一组线程之间共享寄存器。

CUDA 通过提供 全局、局部 和 共享等关键字，实现对内存使用的抽象。此外，还有两种编程结构，分别是“常量”和“纹理”，对应两种特定的内存类型。“常量”内存对所有线程可见，用于存储在内核执行期间不会改变的数据。

接下来需要重点研究 CUDA 内核如何在图形处理器上执行。这与我们的工作相关，因为我们在本论文中开发了一种调度算法，试图模仿图形处理器的指令调度。

1.2.2 CUDA编程模型到GPU架构的映射

主机程序会在运行时为每次内核调用指定参数。必需的参数表示网格和线程块维度。用户还可以选择指定每个线程块可用的共享内存量。同一线程块内的线程会根据其坐标分配唯一标识符。同样，构成网格的线程块也会根据其坐标分配标识符。将线程 ID 与其所属线程块的 ID 组合起来，即可在网格中唯一标识一个线程。线程块内的线程可以通过共享内存共享数据，并可使用特殊的 CUDA API 调用执行屏障同步。如图 1.3 所示，CUDA 线程可以访问不同内存空间中的数据。

CUDA 执行模型通过将内核的线程块分配给流式多处理器来实现吞吐量 。启动内核时，网格中的线程块会根据硬件限制并行分配给流式多处理器。寄存器从按线程束划分的寄存器文件中分配，共享内存则为每个线程块分配。由于这些资源有限，因此流式多处理器上可同时执行的线程块和线程束数量受到硬件限制。

GPU 执行中一个非常重要的参数是线程束，它由 32 个线程组成，用于在 GPU架构上调度线程。流式多处理器以线程束为粒度调度指令。GPU 上的线程束调度单元负责调度线程束执行，而分派单元则为线程束中的所有线程发出相同的指令。由于控制流，线程可能会走上不同的执行路径。在这种情况下，所有不同的路径都会依次执行，非活跃线程则会禁用相应路径。

基于上述图形处理器架构和CUDA编程模型的背景信息，下面将讨论本文提出的图形处理器能耗建模方法。

1.2.3 图形处理器能耗建模

对机器进行能耗建模，需要测量其执行时间方面的性能以及运行时消耗的功耗。因此，可以基于两个分别计算功耗水平和执行时间的子模型来估算内核的能耗。我们可以结合性能模型和功率占用模型得到的结果，估算图形处理器的能耗。

$$E = P \times t_{kernel}$$

其中，E 是内核的能量估计值，单位为焦耳；P 是内核执行期间的平均功耗估计值，单位为瓦特；t_{kernel} 是内核的执行时间，单位为秒。

为了计算能耗，必须注意到，计算模型中可能存在若干误差来源，而要理解这些误差，需要对 NVIDIA 的基础设施有非常深入的了解。NVIDIA 的 GPU架构属于专有信息，除了通过在 GPU 上进行若干实验并进行严格的微基准测试 [33] 外，很难获取。由于硬件实现细节

并未公开，因此计算内核的执行时间和功耗是一项雄心勃勃且颇具挑战性的任务。

正如 Burtscher 等人关于 K20 图形处理器功耗测量的论文所述，内核执行期间图形处理器的实际功耗水平基本保持恒定。这是因为图形处理器中流式多处理器的组件处于活动状态。如果组件处于空闲状态，功耗水平就会大幅降低。需要准确估算图形处理器的功耗。为此，必须分析各种可能的方法，找出效果最佳的方法。可能的方法包括基于回归的功耗预测方法，以及根据图形处理器过去多次使用时的平均功耗水平进行统计估算的方法。

关于执行时间预测模型，Amaris 等人比较了解析模型和机器学习模型的结果，并得出结论：解析模型比其他方法更精确。在这项工作中，我们通过对程序执行过程进行数学抽象，构建了解析执行时间预测模型。该计算以函数的形式执行，并将目标架构、输入程序和启动参数纳入考量。有了这一背景，接下来我们将进一步详细讨论本研究的问题陈述。

1.3 问题陈述

基于我们利用性能和功率估算来建立基于静态分析的能耗预测模型的计划，我们提出以下目标，并在本论文中予以实现。

1.3.1 目标1

我们的目标是设计一个解析模型，用于预测 CUDA 内核的执行时间。为此，我们计划开展以下工作

1. 通过微基准测试确定图形处理器硬件的固有特性指令执行受一些关键参数影响，例如延迟、吞吐量等。供应商 [32]提供的有关这些参数（例如理论峰值性能）的信息非常有限。由于硬件的复杂性，这些细节可能无法准确反映实时指标，因此我们必须通过对硬件进行微基准测试来重新计算这些参数。

2. 创建准确的图形处理器执行模型

由于 GPU 在主流计算中的普及程度不断提高，分析和理解 GPU 应用程序的性能变得越来越重要。我们发现，相比于性能分析支持工具，GPU 应用程序开发投入了大量精力。现有的性能分析器 [25, 26] 和模拟器[22, 27] 只能展示应用程序的执行概要数据。开发人员需要具备相关知识，才能分析这些信息、理解应用程序流程，并推断性能瓶颈。这些工作无疑非常耗时，而且依赖专业知识。

CUDA程序的执行时间除了取决于数据依赖外，还取决于多种硬件特征，例如内存和缓存的使用情况、分支预测、流水线停顿、指令延迟等。要预测特定GPU架构的执行时间，需要深入了解其架构细节，而NVIDIA并未公开这些信息。在缺乏这些知识的情况下，在没有供应商提供任何架构细节的前提下，对GPU的各种架构特征进行建模，例如指令调度、估算每条指令的延迟、估算内存吞吐量以及对启动开销进行建模，都是相当困难的。

我们需要一个能够执行以下操作的性能预测模型工具

- 无需在 NVIDIA 图形处理器上运行 CUDA 内核，即可估算其执行时间

- 解决方案可以包括对代码进行大量静态分析；如果需要动态分析，则应只进行一次数据采集。

- 该方法可以针对程序设计问题进行调整，以帮助开发者深入了解这些问题。

1.3.2 目标2

我们的目标是构建一个预测模型，用于对 CUDA 内核的 GPU功耗进行建模。为此，我们计划开展以下工作

1. 选择可通过静态分析生成的特征计算设备的功耗与其硬件特征密切相关。因此，已有大量研究探讨硬件特征与功耗之间的关系 [36–39]。

然而，我们的目标是开发一种基于代码特征和最少运行时信息的功耗预测模型，使该模型无需运行 CUDA 程序即可用于功耗预测。据我们所知，目前采用这种方法开展的研究并不多。如何选择既能影响功耗，又能在不运行应用程序的情况下收集到的特征，是一项具有挑战性的任务。

2. 构建基于静态分析特征的数据集 据我们所知，目前尚无现成的数据集能够同时收集程序分析数据和功耗数据。因此，我们必须自行构建一个满足这一需求的数据集。为创建该数据集，我们需要以不同的启动配置（网格大小、块大小）运行 CUDA 基准测试，并测量每项

基准测试的功耗。我们还需要一种方法，无需使用任何外部硬件即可收集基准测试的功耗数据。这样，应用程序开发者便能轻松使用所提出的模型，而无需任何专用设备。因此，数据收集是一项繁琐且费力的任务，需要采用多种方法并利用多种资源，才能涵盖通过静态分析提取的各类特征。

3. 选择适合预测功耗的模型 我们没有发现任何用于理解 GPU 应用程序中功耗与程序特征之间关系的算法方法。因此，我们必须使用基于机器学习的模型来构建准确的功耗预测模型。我们需要系统地分析并从现有机器学习算法中筛选出最适合功耗预测的模型。

1.3.3 目标3

我们预计，在构建这些模型时获得的见解将有助于创建一个适用于 CUDA 内核能耗的架构无关模型，并且该模型可以针对多种架构进行调优。为此，我们计划开展以下活动

1. 研究不同架构，以便模型能够针对这些架构进行调优，从而适应更新的架构

在过去二十年里，NVIDIA 推出了十一种架构，不断提升 GPU 的性能效率，有时则以增加复杂度和功耗为代价 [4][6, 40]。因此，我们需要考虑模型对多种架构的适应能力，因为 GPU 硬件的演进速度远快于运行在其上的应用程序的生命周期。所提出的解决方案应以这样的方式设计，使其能够针对更新的架构进行调优：https://www.nvidia.com/en-in/about-nvidia/corporate-timeline/

[4]

将其架构细节作为输入。因此，我们需要研究不同的架构，并找出可调优的特征，以便模型能够适应更新的架构。

2. 在多种架构上验证模型

一旦我们找到用于将模型调优至更新架构的参数，就需要验证性能预测模型和功耗预测模型是否都适用于不同的架构。为此，我们必须识别较新的架构，以及属于这些架构的主流 GPU。由于我们希望采用解析模型，因此将性能模型适配到较新架构将比适配功耗预测模型更加容易。性能预测的数据收集仅涉及获取基准测试和微基准测试的实际执行时间。而功耗预测模型的数据收集则需要在不同的启动配置下执行数千次基准测试，因此这将是一项更加繁重的任务。

1.4 研究贡献

为了回答已确定的研究问题，我们提出了构建能量感知型GPU应用程序的解决方案。我们在本研究中的研究贡献如下：

- 我们提出了一组新颖的微基准测试，用于收集图形处理器的不同架构特征。

- 我们设计了一种轻量级方法，将 CUDA 应用程序的静态分析与图形处理器的架构特征相结合；这些特征是针对给定硬件仅执行一次动态分析后得出的。

• 我们基于一组 CUDA 基准测试应用程序创建了一个数据集，并定义了一套特征工程方法，用于识别影响功耗的一组特征。该数据集和特征提取代码已在 [5] 处公开。

• 我们开发了一种基于机器学习的功耗预测模型，利用程序分析和硬件属性来预测 NVIDIA 通用图形处理器应用程序的功耗。

• 我们分析功耗预测模型的结果，以了解程序特征对 NVIDIA GPU 功耗的影响。

• 我们研究了执行时间和功耗预测模型对不同架构的适应性。我们发现，我们的方法适用于四种架构：Kepler、Maxwell、帕斯卡和 Volta。

• 我们还介绍了一种设计辅助工具，可帮助异构应用程序开发人员构建能量感知型应用程序。

1.5 本研究的出版物

在图1.4中，我们将出版物与本论文涵盖的主题进行了对应。以下是我们研究成果所发表的出版物列表。

1. Santonu Sarkar 和 Gargi Alavani。2018年。为异构系统编写软件有多容易？ SIGSOFT 软件工程札记 42, 4（2017年10月），1–7页。

2. Gargi Alavani、Kajal Varma 和 Santonu Sarkar，《使用静态分析预测CUDA内核执行时间》，2018年 IEEE 国际并行与分布式处理及应用、普适计算与通信会议，

[5] https://github.com/santonus/energymodel

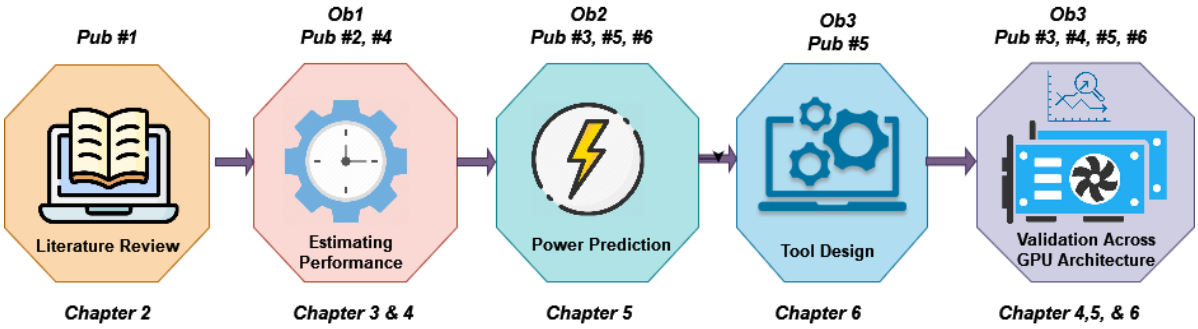


图1.4：论文主题流程及与研究目标对应的出版物

大数据与云计算、社会计算与网络、可持续计算与通信，2018年，第948—955页。

3. Gargi Alavani、Jineet Desai 和 Santonu Sarkar，《一种估算 CUDA 内核功耗的方法》，2020 IEEE 并行与分布式处理及其应用、大数据与云计算、可持续计算与通信、社会计算与网络国际会议，2020年，第984-991页。

4. Gargi Alavani、Santonu Sarkar，《使用静态和动态分析对图形处理器应用程序进行性能建模》，Concurrency and Computation: Practice and Experience，2021年，第34卷。

5. Gargi Alavani、Jineet Desai 和 Santonu Sarkar，《GPPT：CUDA 应用程序功耗预测工具》，2021年第36届 IEEE/ACM 国际自动化软件工程会议研讨会（ASEW），2021年，第247-250页。

6. Gargi Alavani、Jineet Desai、Snehanshu Saha 和 Santonu Sarkar，《基于程序分析和机器学习的 CUDA 内核功耗预测方法》，已提交发表

1.6 论文大纲

本学位论文其余部分的组织结构如下。**第2章** 综述最新技术水平，并将我们对该领域既有工作的研究贡献置于背景之中。**第3章**介绍微基准测试方法及其结果。**第4章** 阐述执行时间预测模型。**第5章** 提出一种基于机器学习技术的功耗预测模型。**第6章** 介绍本研究开发的设计辅助工具。**第7章** 总结本学位论文，并提出未来研究方向。

CHAPTER 2

文献综述

With 随着对性能导向型问题的需求不断增长，程序员通常会在图形处理器上执行应用程序中易于并行化的部分，以实现显著的加速。这些应用程序正变得日益复杂，运行时间也越来越长。传统程序员并不熟悉异构系统编程，而他们要成长为并行编程专家还需要相当长的时间。随着多核系统、加速器和混合系统的架构日趋复杂，亟需开发能够帮助应用程序开发者高效利用其计算能力的工具。

2.1 设计辅助

与串行编程相比，并行编程十分艰巨，因此需要不同类型的设计辅助工具。正如伯克利的技术报告所指出的， [14],除非编写可扩展并行程序能像编写

顺序计算机程序一样简单，否则开发人员将难以转向这一新范式。伯克利和伊利诺伊州并行处理小组的技术报告 [41]强调，除非提供此类支持工具和方法，否则异构计算将无法被主流接受。Wienke 等人 [42]的研究报告了一项实证研究，旨在确定影响程序员生产力的各种因素。尽管研究结果尚属初步，但表明，对并行计算平台的先验知识对程序员生产力的影响最大。不过值得注意的是，环境和工具对生产力的影响分别排名第二和第三。

迄今为止，并行计算软件领域的主要关注方向如下。

a. 编译器基础设施：这主要由 Intel、AMD 和 NVIDIA 等硬件制造商开创，他们大多致力于构建复杂的编译器。

b. 编程语言支持：这同样由 Intel、AMD、NVIDIA 等公司开创，他们构建编程语言扩展（如 CUDA、OpenCL）和库，以便编写并行程序。

c. 算法开发：许多先进的并行算法由受过专业训练的开发人员开发，用于充分发挥多核架构的计算能力，例如 CUDA 工具包 [43]。

尽管这些领域至关重要，我们认为并行计算软件还有其他重要的研究领域，具体如下。

面向异构计算环境的软件开发存在一些固有挑战：

（i）如何通过隐藏底层硬件的复杂性来简化软件开发？ [44–46]

(ii) 针对常见设计问题，有哪些现有解决方案？ [47, 48]

(iii) 可以使用哪些算法骨架来实现高效并行？[49–51](iv) 可以使用哪些可复用组件来加快软件开发？ [52, 53](v) 如何通过自动调优，使软件在硬件上获得最高性能？(vi) 有哪些技术可用于验证语义并指出软件缺陷？ [54–56](vii) 哪些工具可以帮助开发人员构建能效感知型可持续计算应用？ [12]

考虑到这些关键问题，开发人员发现，从顺序编程方式转向并应对并行程序的复杂性颇具挑战（图 2.1 中以一道鸿沟表示）。针对这些问题分别提出设计辅助解决方案，可以让并行开发人员的工作更加高效、轻松。在本论文中，我们聚焦于其中一个研究问题，即：“哪些工具可以帮助开发人员构建能量感知型可持续计算应用？”随着并行编程架构不断发展，亟需构建设计辅助工具，帮助开发人员打造节能工具，使这些架构成为可持续的计算替代方案。在本论文中，我们尝试提出设计辅助工具，以帮助开发人员构建能量感知型可持续计算 GPU 应用程序。因此，以下各节将综述现有的 GPU 能耗预测解决方案。

2.2 GPU 应用程序能耗预测

虽然超级计算机和并行计算机通常能大幅提升我们的计算能力，但它们会消耗大量能量并散发大量热量。因此，冷却系统构成了运营成本和电力成本中的重要部分（占比足够高，以至于...

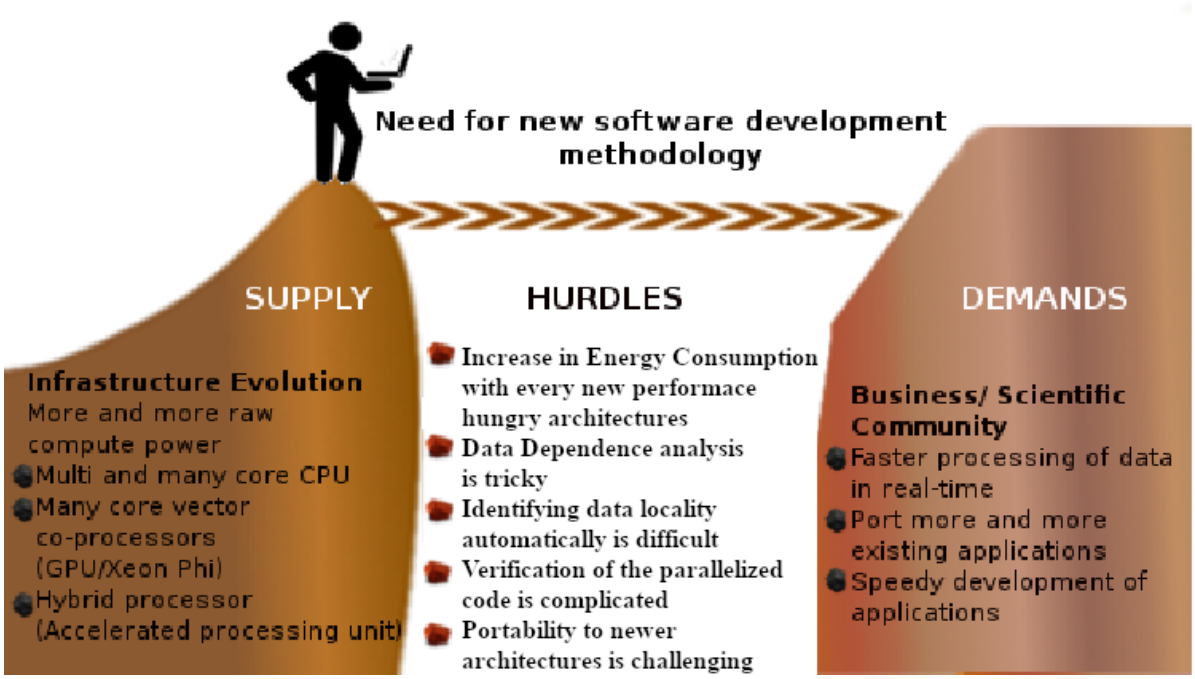


图 2.1：跨越鸿沟

为一座小城市供电） [6]。图形处理器开发人员的主要目标是全面优化代码，以获得最佳性能，并高效利用图形处理器上的各种资源。然而，在高性能计算基础设施上运行的应用程序应发挥最佳性能，同时还需要以尽可能低的能量消耗来运行。

开发者在启动应用程序之前，必须配备相应工具，以了解 GPU 的能耗。已有大量研究尝试解决 GPU程序能耗预测模型的各个方面 [7]。如前所述，我们将能耗问题划分为性能估算和功耗估算。因此，在本文中，我们综述了与 GPU 应用程序性能和功耗建模相关的既有研究。首先，我们详细讨论现有的性能预测文献。然后，我们回顾有关 GPU 功耗预测的现有文献。图 2.2 展示了这份文献综述的概览。

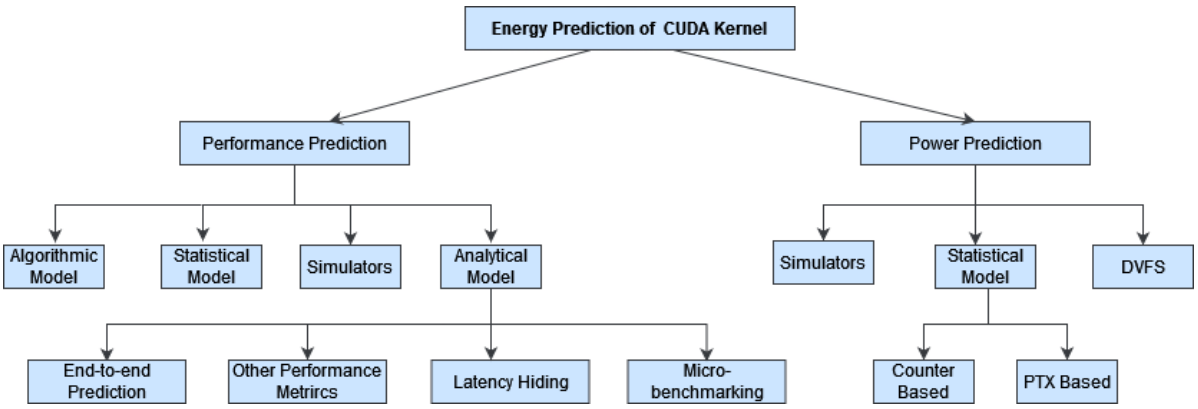


图 2.2：综述概述

2.3 GPU性能建模

性能建模是一个范围广泛且研究充分的主题，涉及延迟、吞吐量、并发性、内存访问效率等各种指标，涵盖不同粒度层级，从单条指令级别一直到整个应用程序[13]。我们的工作侧重于构建完整的 GPU 内核执行时间预测模型，而非在指令级别预测延迟和吞吐量。现有的性能预测研究可分为算法[57, 58], 解析模型[29, 35, 59–61], 统计方法[35, 62–64], 和仿真 [65]。此外，已有一些针对特定应用程序开发的性能模型，这些模型针对 GPU 应用程序的计算需求、内存使用和访问模式进行了调优 [66–68]。我们综述了这些性能预测技术，涵盖广泛的方法和现有解决方案。

2.3.1 算法方法

并行算法行为建模的早期工作始于 PRAM 模型[69],, 随后出现了网络模型[70], 、BSP 模型[71],、Log-P 模型 [72],和 QRQW 模型[73, 74]。这些模型有助于程序员理解多处理器系统中的并行性。

Amaris 等人 [75] 设计了一种简洁直观、基于 Bulk Synchronous Parallel（BSP）的执行时间预测方案。BSP 是一种并行计算模型，可对程序进行算法分析，考量图形处理器的计算量和内存访问次数，并结合通过性能分析收集的缓存使用补充信息。对于这两个基准测试，模型的结果为实际性能的 0.8 至 1.2 倍。Amaris 等人 [35], 在另一项研究中比较了解析模型和机器学习模型的结果，并得出结论：解析模型比其他方法更精确。不过，他们仅在两个基准测试上开展了研究，预测误差接近 20%。

由于算法方法并非特定于架构，因此无法使用这些模型进行针对特定架构的性能分析，也无法预测给定程序在图形处理器等复杂架构上的执行时间。下面介绍用于性能预测的统计方法/机器学习方法。

2.3.2 统计方法

Amaris 等人的机器学习模型 [35] 采用线性回归、支持向量机和随机森林来预测应用程序的执行时间。Eiger[40]是一种用于构建性能模型的自动统计方法。他们使用了从应用程序和硬件中提取的 47 个特征，这些特征通过在目标机器上运行应用程序收集。该方法使用 12 个基准测试进行了验证。Madougou 等人 [13] 将 Eiger 应用于图形处理器，发现其准确率在不同类型的基准测试中并不一致。

Shuaiwen 等人 [20] 的工作利用 CUPTI 库中的性能计数器构建了一个人工神经网络模型，用于预测执行时间，而不是使用 PTX 代码信息。作者声称，对于 12 个基准测试，平均绝对预测误差可控制在 6.7% 以内。他们的方法与我们方法的一个主要区别是，他们的

人工神经网络模型需要性能计数器值，而这些值只能通过运行 CUDA 内核获得。在我们看来，这违背了无需运行应用程序即可预测其执行时间的初衷。此外，他们的模型不适用于非对称工作负载，而我们的模型能够处理这些情况，正如前文所述。

Stargazer [76] 是一种自动化的 GPU 性能探索框架，通过随机采样整个 GPU 设计空间中的参数值来开展探索。它基于逐步回归建模，有助于了解对应用程序性能影响最大的架构参数。Stargazer 使用模拟数据（GPGPU-Sim[22]），且仅考虑硬件特征。这种方法非常耗时，也缺乏从应用程序角度进行的性能分析。

Zhang 等人 [63] 提出了一种基于随机森林（RF）的 ATI GPU 模型。他们利用随机森林的特征重要性，来了解对目标 GPU 执行性能（吞吐量）影响最大的变量。该方法在 22 个内核上进行了验证，用于构建模型的运行时数据通过 23 个性能计数器收集。基于计数器的方法无法扩展到不支持计数器的 GPU，而且现有计数器也无法涵盖所有性能因素。

ScaleClass [77] 可估算 OpenCL 内核在一系列硬件配置下的 GPU 性能和功耗。该方法使用神经网络分类器，根据内核的性能计数器来预测集群的扩展行为。然而，与早期基于 LR 和随机森林的统计方法相比，ScaleClass 的准确率较低，这可能是因为在准确率方面，基于回归的方法通常优于神经网络 [78]。现有统计方法利用性能计数器、运行时性能分析数据或仿真数据进行性能预测。接下来，我们将讨论模拟器如何促进性能预测。

2.3.3 用于性能评估的模拟器

模拟器对于研究基于 GPU 的工作负载特征非常有帮助。由于模拟器主要用于模拟机器的硬件行为，因此不会直接预测执行时间。不过，它们通过利用 GPU 运行时硬件行为的跟踪记录来构建预测模型，在性能预测中发挥着重要作用。例如，GPGPU-Sim [22] 被用于构建性能预测工具 [76]。因此，我们将简要讨论模拟器对性能预测的研究贡献。

GPGPU-Sim [22], GPU-Tejas [27],PPT-GPU [79]是适用于 NVIDIA 图形处理器的热门模拟器。尽管 PPT-GPU [79] 仿真框架声称能够快速、准确地预测应用程序的性能，但并未提供任何结果来支持这一说法。模拟器报告了执行仿真所需的执行时间。例如，GPGPU-Sim 工具的平均执行时间为 14100.29 秒，而 GPU-Tejas 报告的平均执行时间为 32.80秒 [27]。

2.3.4 解析模型

解析模型是对程序执行过程的一种数学抽象。其结果通常以函数的形式表示，该函数考虑了目标架构、输入程序和输入数据。解析模型既用于端到端预测执行时间，也用于通过吞吐量等指标表示图形处理器的性能。我们首先讨论现有的 CUDA 内核执行时间端到端预测方法，随后介绍用于性能预测的其他评估指标。

2.3.4.1 端到端执行时间预测

与我们的研究密切相关的一项重要工作是 Kothapalli 等人 [57] 的研究，其中作者提出了一种 GPU 专用预测模型，该模型受到队列读写 PRAM 模型 [73, 74]，的启发，用于粗略估算 CUDA 内核的执行时间。在这项研究中，作者提出了一种基于两种方法的调度模型：一种是在调度时考虑所有指令延迟，另一种则仅考虑内存延迟或计算指令延迟中的最大值。然而，结果表明，这种方法有时会严重高估，有时又会大幅低估。作者在案例研究中考虑了三个基准测试：矩阵乘法（误差为 31.25%）、列表排序（误差为 12.50%）和直方图。

Hong 和 Kim [80] 提出了一种早期的图形处理器执行时间预测模型。该方法会计算每个流式多处理器上可能容纳的最大线程束数，并统计等待一次内存操作期间可执行的计算指令（例如加法、乘法）的数量。该模型的一个显著缺点是没有考虑计算指令延迟，而这对任何性能预测都至关重要。无论占用率如何，他们都假设计算指令的发射延迟固定为四个周期。因此，如果基准测试属于计算密集型且内存指令很少，执行时间预测就会偏低。

Sim 等人[81]通过纳入算术延迟改进了 Hong 和 Kim 的方法 [80]。作者首先分别计算算术指令和内存指令所需的时间。随后，通过取两者中的最大值或使用他们提出的方程来计算总时间。除了指令级并行和线程级并行外，他们还考虑了内存级并行。然而，对于内存延迟较高的内核，该模型仍会产生较大的高估误差。

Huang 等人 [82] 提出了一种用于 GPU 性能建模的区间分析技术，称为 GPUMech。GPUMech 对多线程和资源

分配进行了建模，依据的是内存使用情况。该方法给出了两种调度方式的结果：轮询策略和先贪心后最老策略。他们使用 GPUOcelot [83]执行 GPGPU 内核，并收集每个线程束的指令跟踪记录。对于轮询调度策略，GPUMech 的观测误差为 13.2%；对于先贪心后最老策略，误差为 14.0%。

Baghsorkhi等人 [84] 首次提出了一种基于编译器技术的 GPU 内核执行时间预测模型，该模型同时考虑计算延迟和内存访问延迟。他们使用了一种名为“工作流图”的带注释程序依赖图，它是控制流图的扩展。后来，Volkov等人 [85] 通过实验表明，该模型在占用率达到100%时效果最佳，而在其他情况下均会失效。实际上，许多非平凡应用程序无法达到100%的占用率。我们还观察到，将指令吞吐量与每个流式多处理器上的线程束数量作图时，指令吞吐量呈线性或次线性增长，而他们的模型并未体现这一点 [84]。

Resios 等人 [59] 建立了一个参数化模型，用于计算计算指令和内存指令的延迟。然而，他们使用指令级并行（ILP）和线程级并行（TLP）等参数，导致计算出的延迟值过低，不切实际。他们的线程调度模型过于简化，因为该模型将执行时间乘以线程数。作者报告称，转置基准测试的精度为1.64，二维卷积滤波器基准测试的精度为1.89。

Williams等人 [86] 提出了适用于多核系统的屋顶线模型概念。该模型将处理器性能与片外内存流量联系起来，并使用峰值浮点性能、峰值内存带宽和运算强度来计算执行时间。该模型对若干与图形处理器性能相关的研究产生了影响。Konstantinidis等人 [87] 改进了屋顶线模型[86] ，仅使用多次执行CUDA内核代码所采集的运行时数据来构建他们的执行时间预测模型。为了提高模型的准确率，

作者提出将利用率因子（E_{util}）作为一个选项纳入模型。不过，E_{util} 也通过每个内核的实际执行时间计算得出。

Luo 等人提出了一种能够考虑指令级并行性和线程级并行性的 GPU 执行时间预测解析模型 [60]。该模型包含两个部分：内存子模型和计算子模型。内存子模型通过考虑活跃线程数量和 GPU 内存带宽来估算内存指令的开销。相应地，计算子模型通过考虑活跃线程数量和应用程序的算术强度来估算计算指令的开销。他们利用 Ocelot[83] 分析 PTX 代码，并获得每条 PTX 指令的时间开销。结果表明，对于本研究考察的六个基准测试，该模型的平均准确率可达 90%。

2.3.4.2 其他性能评估指标

现在，我们评估几篇重要的出版物，它们专注于预测端到端执行时间以外的性能指标。Li 等人 [88] 开发了一种名为*Transit*的交叉屋顶线模型，可预测计算吞吐量和内存吞吐量，并分别在 Fermi 和 Kepler 架构上进行了验证，预测准确率分别为 90% 和 83%。然而，与我们的方法不同，他们的模型无法预测 Gaussian、lavaMD 和 ParticleFilter 等复杂内核。

Volkov 等人 [85] 提出了一个用于预测算术吞吐量和内存吞吐量的解析模型。此外，该模型尚未在实际应用程序中进行测试，并且仅针对一个基准测试进行了测试。他们使用的指针追逐基准测试混合了算术指令和内存指令，其中所有算术指令都相同，所有内存指令也都相同。因此，指令会均匀交错排列，并且前后依赖。我们注意到 Volkov 的文章[85] 中还存在另一处不一致：文中提出的内存吞吐量模型并非指数形式，尽管

作者承认内存吞吐量应呈指数增长。通过广泛的实验研究，我们观察到实际的内存吞吐量确实呈现指数增长。

Huang 等人 [82] 提出了一种基于区间分析技术的 GPUMech 工具，用于模拟图形处理器中因内存使用而产生的多线程和资源分配。他们使用 GPUOcelot 框架，通过运行时分析收集每个线程束的指令跟踪。GPUMech 为经过分析的内存硬件组件提供每指令周期数（CPI）堆栈，可帮助开发者直观地了解内核的性能瓶颈。GPUMech 仿真模型[82] 显示，轮询调度策略的观测误差为 13.2%，先贪心后最老策略的观测误差为 14.0%。GPUMech 利用线程束执行的运行时跟踪记录以及指令延迟。

Lemeire 等人[89] 提出了一组基于占用率屋顶线方法的 OpenCL 微基准测试，并声称这些测试足以预测 GPU 的性能。为了计算每个计算单元的每指令周期数，他们将指令特征计算为其发射延迟和完成延迟。他们将每个计算单元建模为用于计算的流水线和用于内存访问的流水线。他们还测量了内核中独立指令的影响以及线程分歧的影响。

分支发散对应用程序的性能有着重要影响。如果 GPU应用程序 处理的数据导致具有某种模式的周期性分支分歧，那么如果能够通过动态分析 [90] 发现这种模式，有时就可以重构代码以消除这种分歧。然而，纯粹基于静态分析的方法极难预测潜在分支发散现象的任何特征。

我们已将现有方法的结果汇总于表4.11。在本论文中，我们提出了一种用于性能预测的解析模型。为了设计一个解析

性能模型，至关重要的是要了解模型如何对延迟隐藏进行建模。下面我们将讨论一些用于实现这一目标的方法。

2.3.5 延迟隐藏

对于任何 GPU 性能模型而言，延迟隐藏都是一个需要仔细建模的重要方面。GPU 中的延迟隐藏通过在多个线程束中运行独立指令来实现。我们将所提出算法的延迟隐藏效果与其他方法进行了比较。Hong 等人 [80] 假设算术吞吐量为每条指令 4 个周期，且不考虑 GPU 占用率。许多常用方法 [85, 89, 91] 通过实证表明，在占用率为 100% 时，GPU 几乎可以完全隐藏算术指令的延迟。在这种情况下，Hong 等人 [80] 的模型会严重高估性能；而当占用率很低时，则会导致低估（算术指令需要超过 4 个周期）。Volkov [85] 的模型通过一种称为算术强度的概念，结合算术指令和全局内存指令的延迟来实现延迟隐藏。然而，他们的模型是基于一个基准测试推导得出的，该测试中的算术指令和内存指令完全相同，且不包含独立指令。相比之下，我们的方法考虑了独立指令和资源分配。此外，我们还在来自不同基准测试套件、具有不同规模、特性（例如计算密集型、内存密集型）和复杂度（例如控制分歧、内存访问模式）的 44 个基准测试上评估了我们的方法。

Baghsorkhi等人提出的在基于图的模型中使用数据依赖和指令延迟的思想 [84] 与我们的模型类似。然而，在实现延迟隐藏时，他们通过解析方程从内存指令的延迟中减去未阻塞的算术指令来计算延迟。

如前所述，由于关键的硬件细节并未公开，微基准测试对于设计较为准确的性能预测模型至关重要。

因此，我们讨论一些现有的、广受欢迎的 GPU 微基准测试工作。

2.3.6 微基准测试

微基准测试的使用对于理解 GPU 指令执行的本质至关重要[33, 92–94] ，因为供应商不愿意分享这些关键细节。本文讨论了大量使用微基准测试来分析 GPU 性能 [89],、理解硬件特性 [33]以及将其应用于评估能源利用 [95]的代表性研究。

Lemeire 等人 [89]提出了 OpenCL 微基准测试，用于收集图形处理器的性能特征。作者研究了内核中独立指令和线程分歧的影响。

Wong 等人 [33] 测量了 Fermi 架构图形处理器内核执行的不同类型算术和逻辑运算的延迟和吞吐量。他们研究了分支发散和屏障同步的功能，还测量了内存缓存层次结构的结构和性能。

Andersch 等人 [96]使用 GPGPU-Sim 图形处理器时序模拟器分析了图形处理器微体系结构组件的静态和动态延迟行为。他们得出的结论是，GPU 在延迟隐藏方面并不像人们认为的那么有效，并建议除了吞吐量之外，延迟也应成为图形处理器设计的重要考量因素。

2.4 图形处理器功耗建模

过去几年里，了解高性能计算机器功耗的重要性显著提升。图形处理器功耗测量和估算大致可分为直接方法（包括使用外部和内部硬件监测功耗）以及间接技术，例如建模和仿真 [6]。使用外部硬件传感器对大规模分布式系统进行性能分析成本高昂，而且

内部硬件传感器的准确率和可靠性仍在验证中。因此，研究人员也在广泛研究间接功耗建模与测量。间接功耗建模包括基于硬件性能计数器、模拟器和 PTX 的功耗模型。我们的工作旨在为 GPU 的间接功耗建模做出贡献。我们使用机器学习技术构建了一个基于 PTX 的功耗模型。我们讨论了一些提出并评估间接功耗估算方法的研究工作。

Chen 和 Shi [19] 讨论了通过功耗分析来优化现有算法的必要性，并阐述了计算系统功耗建模中现有的各种硬件和软件机制。他们认为，尽管基于软件的工具准确率较低，但由于无需专用硬件，使用起来更加灵活。计算设备的功耗与其硬件特征密切相关。因此，大量研究致力于探究机器硬件特征与其功耗之间的关系 [36–39]。Mittal 和 Vetter [7] 综述了分析图形处理器能效的方法。他们总结认为，需要在芯片设计层面、体系结构层面、编程层面等采用多种方法，以最大限度地提升 GPU 能效。Bridges 等人 [6] 的综述也提倡通过应用层分析来预测功耗，尤其是采用基于 PTX 的功耗模型；他们认为，这类模型能够得出有价值且信息丰富的预测结果。然而，目前关于代码特征与功耗之间关系的研究并不多。据我们所知，目前仍没有任何算法方法能够揭示功耗与程序特征之间的关系。

2.4.1 统计方法

统计方法采用机器学习算法来预测图形处理器功耗。这些方法所使用的数据来自通过传感器和计数器采集的硬件事件，或基于应用程序的特征。Ma 等人[97]开展了最早一批关于功耗预测统计模型的研究。他们使用支持向量回归机（SVR）和支持线性回归机（SLR），估算图形处理器在较短时间窗口内的功耗。训练数据通过创建能够对不同 GPU 子单元施加压力的基准测试生成。SVR 的表现优于 SLR，这表明在功耗预测中，复杂模型比线性模型更有效。

2.4.1.1 基于计数器的模型

基于计数器的建模技术通过将应用程序的功耗与硬件事件相关联来预测其功耗。这些事件可通过性能计数器获取。性能计数器根据硬件操作生成（例如，已发射指令数、内存访问次数、缓存操作）。这些操作事件使用户能够了解底层硬件活动。利用能够解释硬件事件的计数器数据构建的模型，与功耗呈现很强的相关性。因此，这些模型可以生成准确的功耗模型。

Nagasaka等人 [98]提出了一种利用性能计数器估算 CUDA 内核功耗的统计方法。该方法将图形处理器性能计数器作为自变量，将功耗作为因变量，并训练一个线性回归模型。Ma等人 [97] 和Nagasaka等人 [98] 均强调，获取全局内存访问计数对于准确进行图形处理器功耗建模至关重要。Song等人 [20] 提出了首个使用神经网络构建的基于计数器的模型。

Jia等人 [99] 提出了 Starchart，这是一种使用再-

回归树来进行GPU性能与功耗调优。性能API（PAPI）库 [100] 经过扩展，除了性能计数器外，还可以测量能量和功耗。然而，在测量功耗时，它会产生一些异常。Song等人 [20]认为，目前尚无一种准确的模型能够兼顾简洁性、准确率以及对新兴GPU架构的支持，从而隔离能量与性能瓶颈并识别其根本原因。他们提出了一种神经网络模型，利用性能指标估算平均功耗。他们采用基于BP人工神经网络的预测模型，结合硬件性能计数器进行功耗预测，其预测准确率误差在2.1%以内。

2.4.1.2 基于PTX的模型

并行线程执行（PTX）是一种伪汇编代码，是 CUDA编译中的中间步骤。这种代码包含图形处理器将要执行的指令列表。若要先验地预测功耗，可以利用 PTX 指令分析 CUDA程序的功耗。这种方法使程序员能够在执行代码之前进行强有力的优化。

Hong 等人 [29]提出了一种针对 GPU架构的集成功耗与性能预测模型。他们通过数学方程对功耗与 PTX 指令之间的关系进行建模。他们设计了一组合成微基准测试，用于对 GPU 中不同的架构组件施加压力。随后，他们将这些组件映射到 PTX 指令，以了解指令对功耗的影响。他们采用这种方法来预测给定应用程序的最佳活动处理器数量。然而，他们的模型无法预测非对称应用程序和控制流密集型应用程序的性能。

Singh等人 [101] 将非线性函数应用于通过微基准测试收集的数据，以预测图形处理器的功耗。Zhao等人 [102] 提出了一种模型，通过将每条 PTX 指令计数作为输入来进行图形处理器的功耗预测。

2.4.2 动态电压与频率调节（DVFS）

动态电压频率调节（DVFS）是提高功耗效率的一种常用方法 [6, 36, 37]。Abe等人 [103] 探讨了动态电压频率调节技术在图形处理器加速系统中的应用领域；该技术在仅使用 CPU 的系统中能够轻松产生可预测的结果，但由于 GPU架构的复杂性，无法产生类似的结果。

我们无法将 DVFS 用于基于静态分析的功耗预测。这种方法不适合构建用于预测 CUDA 应用程序功耗的模型，原因如下：首先，GPU 频率与 CUDA 程序结构之间不存在既定的关联关系（无论是直接关联还是潜在关联）。此外，DVFS 在 GPU 中默认自动启用，不应将其禁用，因为这可能导致系统故障。另外，供应商不提供任何外部控制机制来操纵 DVFS，因此无法通过仅执行一次来构建能够捕获系统行为的动态模型。

2.4.3 功耗预测模拟器

尽管基于计数器的模型很受欢迎，也已证明行之有效，但执行应用程序却违背了其初衷。目前有许多基于仿真的功耗预测工具，无需执行应用程序即可帮助找到硬件特征与功耗之间的相关性 [23, 38, 104]。

Chen 等人 [105]提出了一种基于树的随机森林方法的图形处理器功耗模型，通过从 GPGPU-Sim 收集运行时数据，理解并估算基于性能变量的功耗。采用留一交叉验证（ LOOCV）时，随机森林的预测准确率为 7.77%，均方误差（MSE）为 302.6；回归树的预测准确率为 11.68%，均方误差为 637.8；线性回归的预测准确率为 11.70%，均方误差为 2548.0。然而，该模型无法对 BlackScholes 等计算密集型基准测试进行预测。该模型的结果表明，这种关系

是图形处理器功耗与其子进程之间的关系，且呈非线性并且复杂。因此，需要先进的机器学习算法来捕捉这种关系。

Lim 等人 [38] 将经验数据与 McPAT [106], （一种 CPU 功耗工具）提供的数据相结合，为 GPU 构建了一个功耗模型。该模型使用一组特定的基准测试进行训练，之后用于预测功耗。报告显示，该模型在 Merge 基准测试上的平均误差为 12.8%。尽管仿真能够提供有用的功耗预测，但对模拟器进行建模需要深入理解架构和相关过程，这相当复杂。

一种名为 GPUSimPow [39] 的模拟器能够估算给定 GPGPU架构上某个内核的功耗，无需在 GPU 上实际运行该内核，而是利用 GPU 内部组件的硬件模型进行估算。它将总体功耗估算拆分为各个组件的功耗。由于直接对硬件进行建模，作者声称，仿真可能是估算内核能耗最准确的方法。然而，这种方法也更加耗时，因为需要对内核从开始到结束的整个执行过程进行精确建模。在一组基准测试中，GPUSimPow 在 GT240 上的平均相对误差为 11.7%，在 GTX580 上为 10.8%。Wang[107]扩展了广受欢迎的 GPU模拟器 GPGPU-Sim [22] ，使其支持功耗建模。然而，他并未公布结果的准确率。

2.5 现有工作的局限性

2.5.1 性能预测

- 问题解决方法： 大多数现有的性能建模方法都源自运行时数据 [20, 80, 82, 84]。要利用这些方法来理解长时间运行的应用程序的性能，尚未

可行，但成本高且能量效率低。因此，构建一个无需执行即可预测应用程序性能的模型，是该领域的一项重要研究贡献。在 Konstantinidis等人 [87],的研究中，为提高模型的准确率，作者提出将利用率因子（E_{util}）纳入考量。然而，E_{util} 是根据每个内核的实际执行时间计算得出的，这违背了通过预测来节省能量的目的。

静态分析方法中的两项重要研究包括 Resios 等人的参数化模型，该模型为解析模型提供了良好的基础。然而，他们的图形处理器仿真模型较为薄弱，没有涉及线程调度。类似地，Kothapalli 等人的工作本应更直观，因为他们的方法有时会严重高估，有时又会低估。

• 进行微基准测试的必要性：微基准测试是探查图形处理器黑盒的关键，也是构建良好预测模型的第一步。尽管已有一些微基准测试套件，例如 Wong 等人的研究，但其结果是基于较早的 Tesla架构 NVIDIA GT200图形处理器。设计一种能够有效应用于新架构的微基准测试至关重要。我们还需要一套新的微基准测试，以捕捉内核启动开销、内存吞吐量、不同指令级并行度值下吞吐量的变化等关键特征。Baghsorkhi等人讨论了内核启动开销，以及它对执行时间极短的基准测试的重要影响。然而，他们并未提出计算该开销并将其纳入研究的方法。

• 验证： 尽管一些现有方法声称能够达到不错的准确率，但需要注意的是，其中大多数方法都是在仅包含两个到十个内核的极小数据集上进行验证的 [57, 59, 60, 75, 80]。构建性能预测

在相当数量的基准测试中得到验证的模型对于将其视为可靠的解决方案也至关重要。

- 特定架构模型： 据观察，现有的大多数方法仅针对一种架构进行了验证 [57, 59, 75, 80]。随着新架构在一两年内不断出现，构建特定架构模型在资源和时间方面似乎代价高昂。我们需要架构无关模型，只需调整少量参数即可在多种架构上运行。

2.5.2 功耗预测

- 问题解决方法： 计算设备的功耗与其硬件特征密切相关。因此，已有大量工作致力于研究硬件特征与功耗之间的关系 [20, 97, 98, 100, 105, 108]。这些运行时硬件数据可以通过功率计、硬件计数器或对 CUDA 内核的执行进行性能分析来收集。尽管使用功率计或基于硬件计数器的方法被认为更加可靠和准确，但有时需要多次运行应用程序才能收集这些数据。正如 Nagasaka等人 的工作所示 [98], , 某些硬件计数器可能不可用，从而影响预测模型。此外，某些架构允许对整个流式多处理器进行计数。在这种情况下，如果每个流式多处理器所利用的核心数量不均衡，预测结果可能会受到影响 [6]。

我们发现，基于静态输入特征的相关研究寥寥无几 [29, 102]。提出一种基于静态分析的功耗预测工具至关重要，它能够减少应用程序执行过程中的能耗，并实现准确的预测。这种方法有助于先验分析。Bridges 等人的一项综述 [6] 表明，基于 PTX 的功耗模型能够提供有价值且信息丰富的预测

当人们希望优化应用程序时，这种工具很有帮助。我们需要为开发人员提供一种工具，帮助程序员识别其 CUDA 内核中的能耗热点，并重构代码以提高能效。

- 特定架构模型：基于硬件计数器的功耗计算模型无法做到架构无关，因为不同架构所采用的硬件计数器数量和类型可能并不一致 [6]。仅基于硬件细节构建功耗预测模型，在较新的硬件上可能无法高效运行。在这种情况下，我们需要一种能够针对较新架构进行调优，且无需投入大量时间和精力的方法。

- 易用性：尽管直接方法是基础方法，但准确率与易用性之间仍需权衡。此外，在云计算时代，无法随时获得硬件来直接测量功耗。内部传感器易于使用，但供应商并未提供这些传感器的文档，例如，面向 GPU 的 NVML 库隐藏了关键的传感器信息。在这种情况下，能够在商品化硬件上运行、易于使用且基于程序分析的方法至关重要。

2.6 讨论

本章中，我们对该领域进行了综述，以准确了解 GPU 应用程序性能和功耗预测技术的最新技术水平。在此背景下，我们的文献综述涵盖了用于解决这两个问题的各种现有方法，包括基于运行时性能分析、硬件数据、模拟器和静态分析的方法。

用于性能预测的统计方法会利用性能计数器、运行时分析数据或仿真数据。使用运行时数据会违背预测的初衷，因为人们希望将这些结果用于先验分析。

现有基于静态分析的性能预测方法分为算法方法和解析方法。我们注意到，文献中已有的算法方法并非针对特定架构。因此，这些方法可用于对抽象并行机器建模，但无法用于针对特定架构进行性能分析，也无法预测特定设备（如图形处理器）的执行时间。现有的大多数基于静态分析的解析模型，都是利用某些指标（例如吞吐量）来预测性能，而非预测端到端执行时间。此外，许多模型只在一种架构上进行了测试，因此这些模型的可重用性尚不确定。

由于功耗与硬件行为的内在联系更为紧密，现有研究更侧重于使用性能计数器和模拟器，观察应用程序执行时的硬件行为。目前的文献中，关于基于静态分析的功耗预测方法的研究还不够充分。现有研究利用特定于硬件的运行时数据，无法扩展到更新的架构，并且在没有专用硬件的情况下，其可用性也受到限制。

我们希望通过提出一种较为准确、更加直观、基于静态分析且架构无关的性能和功耗预测模型，克服文献中发现的局限性。我们还认为，经过更多基准测试全面检验的研究会更可靠，也更容易被认可。

CHAPTER 3

微基准测试

As 正如第2章所总结的，微基准测试是探索图形处理器行为建模过程中至关重要的一步。本章讨论了本研究中针对各种原生指令集所设计的微基准测试。

设计微基准测试的目的是计算某些基本操作的属性，这些操作会影响更高层次的系统行为。微基准测试旨在测量一小段代码的性能，以便确定该代码作为应用程序一部分时的运行表现。这些微基准测试数据可用于 GPU 软件的优化、仿真和分析。

微基准测试数据已被证明是性能模型[59, 89, 91]中的关键输入。这是了解硬件架构并加深认识的一种方式。它有助于研究人员提供供应商未公开的处理器性能特性。我们可以利用微基准测试计算持续性能的性能上限。微基准测试也有助于发现架构中的性能错误。

每种架构都与其前代有所不同，因此指令执行特

性也可能因架构而异。指令执行受一些关键参数控制，例如延迟、吞吐量等。供应商提供的关于这些参数（例如理论峰值性能）的信息非常少。此外，这些数据可能无法准确反映实时指标，因此我们必须通过对硬件进行微基准测试来重新测算。

我们在文献 [59], [33] 和 [57] 中发现了关于 Fermi 架构微基准测试的相关研究。据我们所知，在开展微基准测试时，针对 Kepler 架构的研究还相对较少。此外，我们没有发现通过微基准测试计算实现峰值吞吐量所需的最少就绪线程束数量的研究。同样，我们也没有发现通过微基准测试量化内核启动开销的相关研究。

基于这些观点，本章将介绍以下内容：

- 为 GPU 架构开发的微基准测试，用于计算属于 PTX 指令集的计算指令的峰值性能值。
- 一种用于估算内核启动开销的模型，该开销会对内核的性能产生显著影响。
- 一种全局内存延迟模型，可根据访问步长预测指令延迟

我们首先了解 CUDA 内核的 PTX 表示中呈现的指令。

3.1 PTX指令集架构

PTX 是 Parallel Thread Execution（并行线程执行）的缩写，它是一种面向 GPU 应用程序的低级并行线程执行虚拟机和指令集架构（ISA）。PTX 是一种

稳定的 ISA，可跨越多个 GPU 世代。PTX 也是一种与机器无关的 ISA，适用于 C/C++ 和其他目标编译器。由于具备这些特征，PTX 被广泛用于手工编码和架构测试。

3.1.1 生成 PTX 文件

在构建时，向 nvcc 语句添加 –ptx 选项，即可将 CUDA 代码编译为 PTX 代码。该语句会生成一个 .ptx 文件。PTX 是一种中间代码表示形式，并非可执行文件。它与汇编代码非常相似，但并不是图形处理器实际执行的代码，因为它还需要经过几个中间步骤。编译后生成的 .ptx 文件包含对应 .cu 文件中定义的每个内核的代码。

基本的 PTX 代码由以下语句组成：注释、以“.”开头的指令、计算指令、全局内存访问语句、共享内存访问语句、同步语句、分支语句、标签以及返回语句。我们扫描 PTX 代码，并对上述每种语句进行分类。

我们分别分析计算指令和内存访问指令，因为对二者进行微基准测试的方法差异很大。我们首先讨论计算指令微基准测试的细节。

3.2 计算指令

计算指令是算术和逻辑指令，可以是整数运算（例如加法、乘法）、浮点运算（例如融合乘加）或特殊指令（例如平方根）。我们开发了微基准测试，用于计算计算指令的延迟、吞吐量和 PeakWarps。

为了计算延迟，我们启动一个线程，并使用两个相互依赖且属于同一指令类型的指令，以确保 GPU 在前一条指令完成之前无法发出下一条指令

```
__全局__ void 吞吐量_内核( int * dummy) {
  // 参数初始值设置
    int j = 块索引 . x*块维度 . x + 线程索引 . x + 块维度 . x*线程索引 . y ;
    for ( int i =0;i <内层循环迭代次数 ; i ++) {
  repeat256 (b+=a ; a+=b ; ) ;
    }
    dummy[ j ]= b+a; // 将结果存储到避免编译器优化
}
```

代码清单 3.1：吞吐量_内核



图3.1：指令级并行=1下融合乘加的吞吐量

在执行指令之前，记录开始时间 T_s 。然后，我们执行一组包含 256 条指令的指令，以确保总执行时间足够长，从而能够被记录。执行完成后，我们记录时间 T_e ，并测量执行时间 $T = T_e - T_s$。我们重复此过程 N 次，并累加每次运行的执行时间（$T_{tot} += T$）。最后，我们计算该指令的平均延迟，即 $\mathscr{L} = T_{tot}/(2 * 256 * N)$。

吞吐量和峰值线程束数值的估算过程相较于延迟计算要复杂得多，因为指令级并行（ILP）在其中发挥着重要作用。研究发现，在不存在任何独立指令的情况下，吞吐量会在达到某个线程级并行（TLP）值后趋于饱和。若要进一步提高吞吐量，就需要在 CUDA 内核中为每个调度器提供可用的独立指令。这可以通过在内核中添加独立指令来实现；每增加一条新的独立指令，ILP 值就会增加 1。下方展示的 ILP 为 1、ILP 为 2 和 ILP 为 3 时的指令集，阐释了这一方法。

指令级并行=1 指令：b+=a；a+=b；

指令级并行=2 条指令：b+=a；a+=b；c+=d；d+=c；

指令级并行=3 指令：b+=a; a+=b; c+=d; d+=c; e+=f; f+=e;

作为示例，加法指令 对应的 CUDA 内核代码片段如清单 3.1 所示。

指令级并行度 1。此内核由清单 3.2 所示的主程序调用，用于计算此指令的吞吐量。一个重要观察是，对于不同的指令级并行度值，峰值吞吐量以及达到峰值吞吐量所需的活动线程束数量也各不相同。为刻画这一现象，我们使用不同的指令级并行度值运行此基准测试。对于每个指令级并行度，我们通过将第 6 行替换为相应指令级并行度的代码（如上所示），创建清单 3.1 中内核的不同版本，同时保持清单 3.1 和清单 3.2 中其余代码不变。不同指令级并行度值下计算指令（例如 加法指令）的吞吐量和峰值线程束计算过程总结如下：

1. 针对每种指令级并行度，设计多个吞吐量计算内核版本。

2. 对于每个 ILP，执行相应版本的内核。通过逐步增加启动的活动线程束数量来调用清单 3.2 中的吞吐量_ 函数代码，记录每个活动线程束 w 的每周期吞吐量 $tput$，从而生成数据集 $D(w, tput)$。

3. 对于每个指令级并行度（ILP），我们将峰值线程束数定义为

$$\mathscr{PW} = \min\{w \mid\mid D(w) - D(w') \mid > \epsilon, \forall w' \in W\}$$

这是根据 D 计算得出的，D 包含一组活动线程束 W；对于每个 $w \in W$，都有一个对应的每周期吞吐量 $tput = D(w)$。此指标通过修改标准 argmax 函数并引入一个较小的正阈值参数 ϵ 得出。这样便忽略了吞吐量值相对于最大值的细微变化（一个较小的正阈值 ϵ）。如图3.1所示，$\mathscr{PW}$（以红点标出）是红点在 x 轴上对应的线程束值。请注意，吞吐量值从这个标记点之后开始振荡。

```
1 in t 吞吐量_func ( dim3 Db, dim3 Dg) {
```

```
2    cudaEventRecord ( 开始 , 0) ; 吞吐量_内核 <<<Dg, Db>>>(d_dummy) ;
3
4
5
6
7  }
```

代码清单 3.2：PeakWarps 计算代码

为了运行基准测试程序进行测量，我们采取了以下步骤来避免量化误差。

- 编译器优化已禁用至零级，以避免即使在零级下也会执行不理想的优化。如清单 3.1 所示，图形处理器内核执行的结果存储在虚拟变量中。使用虚拟变量来避免编译器级优化，是设计微基准测试时的常见做法[110, 111]。

- 我们通过在一个网格中启动延迟内核，并且每个线程块仅包含一个线程，确保测量延迟值时只执行单个线程，从而避免并行性导致的延迟隐藏。

- 在计算吞吐量值时，我们确保流式多处理器的占用率始终高于 90%。为此，我们在每个流式多处理器上启动了每个线程块 256 个线程。

3.2.1 计算指令延迟结果

使用所提出的微基准测试获得的延迟值列于表 3.1 的第二列。如表 3.1 所示，对于每种类型的指令，改变指令级并行度值所得到的结果存在显著差异。在理想情况下，Tesla K20 的流式多处理器中有四个线程束调度器和两个指令分派器，

表 3.1：Tesla K20 计算指令摘要

指令	$\mathscr{L}$	**吞吐量 ($\mathscr{TP}$)**			**PeakWarps（$\mathscr{PW}$）**		
		指令级并行=1	指令级并行=2	指令级并行=3	指令级并行=1	指令级并行=2	指令级并行=3
addf,subf,乘法f	9	122	128	167	36	20	18
adds,subs,and	9	120	127	136	36	20	18
fma	10	119	95	143	36	20	16
mads	20	31	28	25	20	10	8
muls	9	28	32	32	8	8	8
divs	424	2.35	2.5	2.36	32	32	32
divf	894.5	1.066	1.0	1.02	32	32	32
sqrt	359	3.48	3.19	3.2	40	40	40
setp	22	50	50	50	36	28	28
cvt	10	31	31	31	12	12	12
mov	2	150	N/A	N/A	32	N/A	N/A

表3.2：测得的 PTX 指令延迟 $\mathscr{L}$

NVIDIA GPU ->	**Quadro K4200**	**Tesla K20**	**Tesla M60**	**GTX1050**	**Tesla V100**
架构 ->	Kepler	Kepler	Maxwell	帕斯卡	Volta
addf	10	9	15	15	15
adds	9	9	15	15	15
subf	10	9	15	15	15
subs	10	9	15	15	15
mulf	9	9	15	15	15
muls	9	9	86	86	15
and	9	9	15	15	15
fma	9	10	188	12	232
mads	18	20	100	15	30
divf	1252	894.5	1278	1398	977
divs	418	424	1026	503	815
cvt	33	10	195	195	218
sqrt	440	359	550	481	487
setp	22	22	30	30	30
mov	2	2	51	55	49
共享加载与存储	40	47	38	39	39

每个周期可以分派256条指令。未包含独立指令时观察到的吞吐量值（例如，对于ILP=1，每周期执行120条加法指令、减法指令）表明，由于同一线程束中缺少可用的独立指令，至少有几个分派器可能无法得到充分利用。我们还观察到，对于大多数指令，当ILP超过3后继续增加ILP，并不会显著影响吞吐量。这可能与以下事实有关：只要有三个以上的独立指令可用，调度器的两个分派单元就能得到充分利用。表3.2展示了多种GPU架构中计算指令的延迟。

3.3 内存指令

NVIDIA GPU 提供多种内存选项，例如全局内存、共享内存、常量内存和纹理内存。在当前工作中，我们仅关注全局内存指令和共享内存指令。我们开发了微基准测试，用于量化全局内存指令和共享内存指令的延迟与吞吐量。常量内存和纹理内存将作为未来的研究工作加以考虑。访问全局内存指令（全局加载和存储）的延迟取决于特定时刻正在访问的数据量，以计入资源限制所导致的额外等待时间。由于其延迟值较高，全局内存访问会对 CUDA 内核的执行时间产生显著影响。因此，准确估算全局内存访问的延迟值对于实现有效的执行时间预测至关重要[59, 80]。

在现有的各种方法中，基于指针追踪的微基准测试被认为是一种准确且流行的计算内存指令延迟的方法。GPU 的指针追逐算法分别在算法 1（主机（CPU））和算法2（设备）中给出。在主机算法中，数组先用步长值初始化，然后调用内核代码。在设备算法的内核代码中，记录指针追踪的内存访问指令的开始时间和结束时间。

记录该方法。在算法2的第4行，我们确保只有在前一条指令完成后，才执行下一条内存指令。因此，该方法可以记录执行单条内存指令所用的周期数。使用迭代器反复执行指令大量次，以确保延迟不会小到无法测量。我们将j 的值存储在一个虚拟变量中，以避免任何编译器级优化。

3.3.1 指针追踪

指针追逐微基准测试最初由 Saavedra 等人 [112] 为 CPU 提出，它会用下一次内存访问的索引初始化一组数组元素。连续两次内存访问之间的距离称为步长大小。内存访问延迟是指发出内存访问请求到数据在处理器寄存器中可用之间的时钟周期差。在指针追逐实验中，会依次遍历整个数组，以记录平均内存访问延迟。此方法也已应用于 GPU [33, 113]。我们修改了该方法 [113]，用于计算图形处理器内存指令的延迟。

3.3.2 全局内存指针追逐

GPU 的指针追逐算法分别在算法 1（主机（CPU）端）和算法2（设备端）中给出。在主机端算法中，数组使用步长值初始化，然后调用内核代码。在设备端算法的内核代码中，记录采用指针追踪方法进行内存访问的内存访问指令的开始时间和结束时间。在算法2的第 5 行中，我们确保只有在前一条指令执行完毕后，才执行下一条内存指令。因此，这种方法可以记录执行一条内存指令所用的周期数。通过使用迭代器多次执行该指令，确保延迟不会

短到无法测量。我们将j 的值存储在一个虚拟变量中，以避免任何编译器级优化。指针追逐的代码片段见附录

8.

算法 1 内存延迟主机（CPU）算法

```
1：初始化步长
2：对于k= 0到 N 执行
3:      h_arr[k]=(k+步长) % N;
4：将主机数组（h_数组）复制到设备（ d_arr）
5：memLatKernel«<Dg, Db»>(d_dummy,d_arr);                    ▷ 调用延迟内核
```

算法 2 内存延迟设备（图形处理器）算法

```
1: 过程 GMMEMLATKERNEL(d_dummy, d_array)
2:      start_time  =时钟();
3:      初始化 j=0;
4:      对于it=0次迭代执行
5:          j=d_arr[j] ;
6:      end_时间  = 时钟();
7:      d_虚拟  =j;
8:      延迟  = (end_time - start_time) / iteration          ▷ 平均内存延迟
```

3.3.3 全局内存延迟模型

我们观察到，在现有研究中，针对 Kepler 架构 [57, 93]，报告的全局内存指令延迟值超过 400 个周期。然而，我们在 Tesla K20 上进行的指针追踪微基准测试报告的平均延迟仅为 221 个周期！如果采用文献中报告的较高值（例如 [57] 中提到的 580 个周期），最终的预测模型可能会严重高估。我们进一步注意到，延迟值取决于启动参数，因此并非固定不变。这一观察促成了全局内存延迟回归模型的构建，该模型根据启动参数计算观测到的延迟。这将确保延迟值根据启动配置，在观测到的最低延迟与最高延迟之间变化。

表 3.3：Tesla K20 GPU 全局内存访问指令的延迟计算

步幅间隔	**全局加载/存储延迟**
nT_b*nB <4096	$\mathscr{L} = 0。02828 \times nT \_b \times nB + 220$
4096<nT _b*nB <24576	$\mathscr{L} = 0。004780 \times nT \_b \times nB + 251.7$
24576<$nT \_b * nB$ <991232	$\mathscr{L} = 0。0001679 \times nT \_b \times nB + 307.8$
991232<nT_b*nB <2203648)	$\mathscr{L} = -0。00002529 \times nT \_b \times nB + 501.8$

表 3.4：全局内存访问指令延迟模型的评估 s

MSE	均方根误差 R 平方 残差平方和			RSE P 值	
22.95	4.79	0.99	32801.69	4.80	2.20e-16

数据收集： 我们反复运行前面讨论的微基准测试，通过改变其启动参数（nB、nT_b）来记录全局加载/存储指令延迟。我们通过改变调度的线程数，确保测量的是每个线程执行一条指令所需的平均时间。在记录延迟时，我们禁用了内存指令的数据缓存。我们收集了大量数据点，以表示所观测到的广泛延迟值。每个数据点均通过对记录的时间取平均值得到。

1. 将记录的延迟值与访问步幅（$nT\_b \times nB$）绘制成图。如图3.2所示，该图呈非线性。存在三个断点，每个断点界定了两段线性关系。

2. 形式为 $\mathscr{L} = a \cdot nT\_b \cdot nB + b$ 的分段线性回归模型拟合效果最佳。表 3.3 给出了线性回归模型的四个方程。

所提出的延迟模型的评估指标为：均方误差：22.95，均方根误差：4.79，R-平方：0.99，残差平方和：32801.69，残差平方和：4.80。该模型的 R 平方值（0.9967）表明其对这些数据的拟合优度。

3.3.4 共享内存访问延迟

共享内存是一种由线程块内线程共享的非缓存内存。由于其位于芯片上的位置（每个流式多处理器都有专用的共享内存空间），其延迟远低于全局内存指令。为避免全局内存访问的长延迟，应用程序开发者可以在操作执行前将数据从全局内存移入共享内存，并在执行后将数据移出共享内存。对于共享内存，每个线程都会单独发出访问请求。我们使用算法3所示的指针追逐方法来测量共享内存访问延迟。该方法与算法2中所述的全局内存指针追逐方法类似。这里我们声明一个共享内存数组（`shdata[]`），并首先使用步长值对其进行初始化。在第8行中，使用指针追踪确保每个线程只执行一条指令。

共享内存被划分为大小相同、称为存储体的内存模块，这些模块可同时访问；而全局内存则按步长访问。全局内存报告的延迟至少比共享内存高 100 倍。与全局内存相比，共享内存报告的延迟值变化很小。因此，我们没有为共享内存采用任何回归模型。不同架构下记录的共享内存延迟结果如下

见表3.2。

算法3 共享内存内核算法

```
1: 过程 SMMEMLATKERNEL(d_dummy, d_数组)
2:     声明 shdata[] 为共享内存数组
3:     对于 i=0至 N 执行
4:         shdata[i] = d_数组[i];
5:     start_time  =时钟();
6:     初始化 j=0;
7:     对 it=0次迭代执行
8:         j=shdata[j] ;
9:     结束时间  = 时钟();
10:    d_虚拟变量  =j;
11:    延迟  = (结束时间 - 开始时间) / 迭代次数                ▷ 平均内存延迟
```

3.4 内核启动开销

CUDA 内核执行时间会受到在图形处理器上启动内核所需时间的影响，本文将其定义为内核启动开销。这是在执行内核指令之前和之后所耗费的时间。我们构建了一个经验模型来描述这一开销。我们使用不同配置（线程数、每个线程块的线程数）运行了一个空内核（不含指令）。

1. 我们通过更改启动参数（线程数、每个线程的线程块数），记录了这个空内核的执行时间。Resios 等人 [59] 声称，内核启动开销可以用常数建模，因为其值不会变化。然而，我们发现，启动的线程数增加时，空内核的执行时间也会增加。

2. 我们建立了三个统计模型来重现其行为，并使用 R 方拟合优度对它们进行了评估（见表4.4）。R 方值越高，说明模型对数据的描述越准确。

3. 线性回归模型（针对 Tesla K20 数据）的绘图如图3.3所示，该模型的 R 方值最高（0.9877）。Tesla K20 GPU 的内核启动开销（$l_$启动开销）模型为 $0.00002 \cdot nB \cdot nT\_b + 1.4489$。同样，我们也使用线性回归为所研究的每种 GPU 架构建立了内核启动开销模型。

内核启动开销的代码片段见附录 8。

3.5 讨论

我们为 NVIDIA GPU 设计了微基准测试，可用于 GPU 软件的优化、仿真和分析。微基准测试提供了

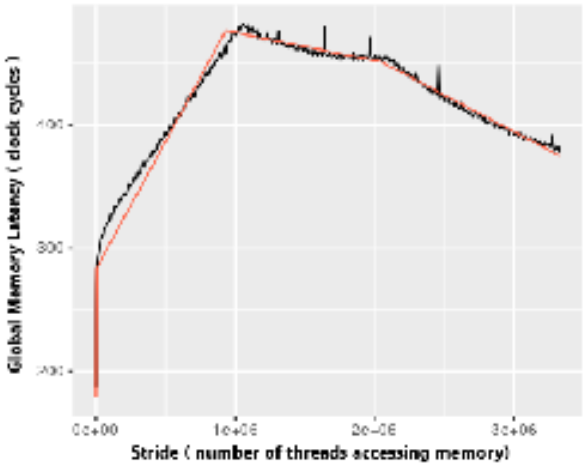


图3.2：全局指令延迟的分段模型

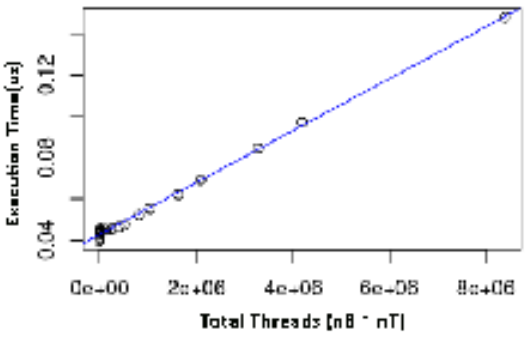


图3.3：内核启动开销

通过量化计算指令在延迟、吞吐量和PeakWarps指标方面的性能，我们对计算指令的执行有了更深入的理解。尽管已有针对GPU微基准测试的研究，但我们在呈现计算指令和内存指令的特性方面做出了创新性研究贡献。这项工作的创新之处之一，是报告了不同ILP值下吞吐量和PeakWarps的变化。吞吐量和PeakWarps值的这一变化被用于第4章公式4.3中的性能模型，以计算计算指令延迟。我们还提出了一种计算全局内存延迟的新模型，该模型结合了微基准测试和分段线性回归；据我们所知，其他研究尚未开展这项工作。该模型利用指针追逐方法进行微基准测试。我们基于这些结果拟合了一个分段线性回归模型，以确保全局内存延迟既不会被高估，也不会被低估。我们还计算了内核启动开销，它同样会影响GPU应用程序的性能。

我们通过与其他已有的微基准测试研究 [114, 115] 进行对比，验证了观察到的结果。计算指令和共享内存指令的延迟值与其他微基准测试研究的结果相同。吞吐量值与计算能力3.5的理论值相当接近，正如 [32] 中所述。

本章开发的微基准测试模型分别用于第 4 章和第 5 章的性能预测和功耗预测。内存指令执行会受到非合并访问和数据竞争等因素的影响。我们在延迟计算中未考虑这些因素。未来，我们希望对 GPU 内存层次结构中的各种内存指令进行微基准测试，重点研究影响内存指令性能的各种访问因素。

CHAPTER 4

GPU性能建模

Microbenchmarking

第3章讨论的方法为 GPU 内核执行时间预测模型奠定了基础。

我们利用微基准测试结果，通过分析 CUDA 内核的中间 PTX 代码，构建解析模型来预测 GPU 内核的执行时间，而无需执行该内核。

性能预测是指估算可执行软件的执行时间，或估算计算机的其他性能指标值，例如指令吞吐量、缓存未命中等。在本文中，我们使用“性能预测”一词表示对 GPU 内核执行时间的预测。

在缺乏 GPU 内部细节的情况下，我们利用从静态分析、一次动态分析以及供应商发布的资料中收集的特征，构建了一个抽象 GPU 模型，这些特征对于 GPU 性能建模至关重要。该抽象 GPU 模型充当实际 GPU 硬件，用于在计算某些性能参数（例如指令延迟）时提供特征参考。它还有助于构建架构无关模型，因为它有助于研究和呈现共同属性。

在多个 GPU 架构之间。现在，我们将介绍这一抽象 GPU 模型，并借助它提出性能预测模型。

4.1 抽象图形处理器模型

对于 t 为了估算内核的执行时间，我们将图形处理器表示为 a s

$$\mathscr{GPU} = \langle R, \mathscr{A}, T, \mathscr{PTX}, R_{map}, \mathscr{L}, \mathscr{TP}, \mathscr{BW}, \mathscr{P} \rangle$$

- R 表示每个流式多处理器的以下图形处理器资源类型：i) SP：单精度核心 ii) WS：线程束调度器 iii) SFU：特殊函数单元 iv) DPU：双精度单元 v) LSU：加载存储单元。

- $\mathscr{A}$ 表示 GPU 的以下属性集合：i) 访问大小：表示 GPU 每条内存指令访问的字节数 ii) nSM：流式多处理器的数量 iii) nTh_sm_M：每个流式多处理器的最大线程数 v) $L2\_sz$：L2 缓存大小 vi) ν_{gpu}：GPU 时钟频率，以及 vii) ν_{mem}：内存时钟频率 (viii) reg_b_M：每个线程块可用的寄存器总数 (ix) shm_b_M：每个线程块可用的共享内存总字节数 (x) nB_M：每个流式多处理器的最大线程块数 (xi) wSM_M：流式多处理器上的最大活动线程束数 (xii) Sz_w：线程束大小，表示每个线程束中的线程数 (xiii) Sz_{gl}：单次全局内存事务中的字节数

- T 是所有指令类型的集合，即 {计算（C）、全局内存（Gm）、

共享内存（Sm）、其他（M）}。每种指令类型都与一种资源相关联。

- $\mathscr{P}TX$ 表示可从 NVIDIA GPU 文档中获取的 PTX 指令集合。每条 PTX 指令都对应一种指令类型。

- R_{map}： $\mathscr{P}TX \rightarrow R$ 将一条指令唯一映射到一种资源类型。

- $\mathscr{L}$：PTX $\rightarrow \mathbb{N}$ 是延迟函数，它将 PTX 指令的延迟表示为正整数l。指令延迟表示一条指令完成执行所需的时钟周期数。

- $\mathscr{TP}$：表示计算指令、全局内存指令和共享内存指令的吞吐量。

- $\mathscr{BW}$：表示内存指令的全局加载和存储传输带宽

- $\mathscr{P}$：表示一组开销和惩罚，其中部分基于 $\mathscr{TP}$。我们考虑以下内容：i) $l_$开销：内核启动开销函数；ii) $gm_$惩罚：全局内存惩罚函数；iii) $sm_$惩罚：共享内存惩罚函数；以及 iv) $cm_$惩罚：缓存未命中惩罚函数

该模型针对特定的图形处理器架构构建一次。表4.1列出了模型属性所使用的符号。

4.2 静态分析

为了收集 CUDA 内核的各类执行特征，我们通过使用`nvcc` 编译器将代码编译为 PTX 格式 [109]，对 CUDA 内核进行静态分析。随后，我们为 PTX 表示中的每段内核代码构建控制数据流图。在这里，我们将每个基本块 [116] 建模为一个数据流图

表4.1：用于模型属性的符号

动态分析属性	
特征	**计算来源**
延迟 $\mathscr{L}$	微基准测试
全局内存访问延迟	线性回归模型
l 开销_	线性回归模型
gm_惩罚	指数模型
流式多处理器_惩罚系数	指数模型
cm_惩罚项	参数化方程
硬件属性	
设备属性 $\mathscr{A}$	设备查询
每种资源的实例数 $r \in R$	架构文档
输入	
nB：线程块数	用户输入
nT_b：每个线程块的线程数	用户输入
$nLoop$：循环迭代次数	用户输入
reg_t：每线程寄存器数	NVCC 编译器
shm_b：每线程块共享内存	NVCC 编译器

$G = \langle V, E \rangle$ ，其中每个顶点 $v \in V$ 表示一条 PTX 指令，有向边 $e: u \rightarrow v$ 表示 v 的计算依赖于 u 的数据。边 e 标注有延迟值 $\mathscr{L}(u)$。此外，对于每个 $v \in V$，可以使用 $Rmap(v)$ 函数获取对应的资源类型。

用于对整个内核建模的控制数据流图 [116],表示为 $G_{cfg} = \langle \mathbb{C}, \mathbb{E} \rangle$ ，其中每个节点 $B \in \mathbb{C}$ 都是一个与数据流图[117–119]相关联的基本块。由于控制数据流图中的边表示控制流，因此边上没有标注任何延迟信息。控制数据流图中的回边表示一个循环。回边会标注循环迭代次数 $nLoop$。由于 CUDA程序遵循结构化编程风格，因此可以将控制数据流图转换为一组区域 [120] ，其中每个区域都是一组基本块，而控制数据流图中的回边会变成循环区域 [120]的自循环。在代码生成期间，任何编译器中都很常见这种转换。因此，我们不再对此作进一步阐述。

图 4.1 展示了静态分析阶段的一个示例。

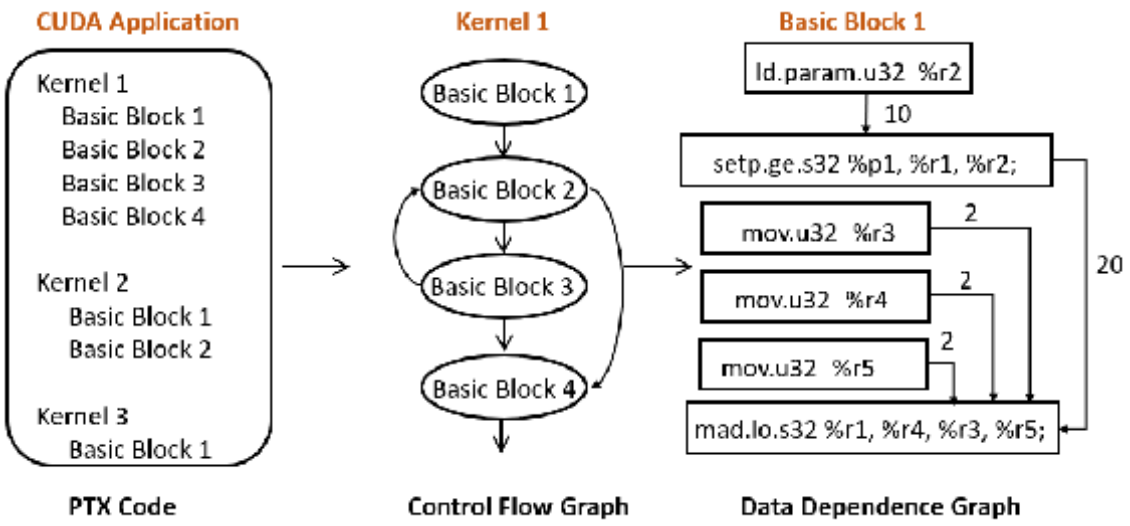


图 4.1：CUDA 代码的静态分析

我们提出了两种 GPU 性能建模方法。第一种方法使用参数化方程来捕捉指令级并行（ILP）和线程级并行（TLP）。第二种性能建模方法则受到资源分配的启发。在接下来的章节中，我们将详细介绍第一种方法，然后进一步阐述如何构建基于资源分配的模型来预测执行时间。

4.3 使用指令级并行和线程级并行对GPU性能进行建模

如文献综述（参见第2章）所述，并行性是影响图形处理器性能的重要因素。对于所有类型的指令，提高并行性（无论是增加线程数量，还是增加独立指令的数量）都会提升性能，但提升幅度受硬件约束限制。Resios 等人通过实验研究证明，性能会随并行性的增加而线性提升。他们通过传输开销、内存带宽和指令吞吐量验证了这一现象。

图形处理器中的并行性可通过两个方面实现：指令级并行性和线程级并行性。在指令级，通过在一个线程中安排独立指令来隐藏延迟，从而提升性能。在线程级，随着线程数量增加，独立指令也会增多，进而提升性能，直到达到硬件限制。过去，Resios 等人尝试采用并行性建模方法对图形处理器的行为进行建模。然而，他们是在 Fermi 架构上实现这一方法的，而且采用的调度算法非常简单。

这一观察促使我们探究指令级并行（ILP）和线程级并行（ILP）在决定 GPU 应用程序性能方面所起的作用。因此，我们计划使用参数化方程对 ILP 和 ILP 建模，以预测 GPU 应用程序的性能。这种方法的详情见第 4.3.1 节。

4.3.1 提出的模型

我们提出了一种解析模型，通过静态分析 PTX 代码来预测 CUDA 内核的执行时间，如图 4.2 所示，该模型包含两个主要阶段：

1。使用延迟计算算法（DCA）将各条指令的延迟相加，计算单线程的执行时间。

2。通过使用 GPU 调度算法（GSA）对流式多处理器（SM）上的线程调度进行建模，计算内核的整体执行时间。

本研究提出的执行时间模型被建模为这两个阶段的函数：

$$t_{kernel} = f(DCA, GSA) \tag{4.1}$$

在第一阶段，我们收集每种指令类型的微基准测试详情以及硬件特征。在分析 PTX 代码并构建

控制流图（CFG）后，我们利用这些详情，结合指令级并行、线程级并行、指令特征（延迟、吞吐量、PeakWarps）和指令特定特征（内存带宽、存储体冲突、内存访问合并），计算每条指令的延迟（d_i）。内核的总延迟（d_k）通过延迟计算算法计算，该算法利用d_i 和控制流图。

为了执行内核启动的线程，NVIDIA 调度器会将这些线程分成多个波次进行调度。供应商并未披露具体的调度方式。我们提出了一种简单的调度算法，作为本研究下一阶段的工作。

第二阶段以单个线程的总内核延迟为基础，并通过调度线程来模拟图形处理器的执行，以预测内核执行时间（ts_{kernel}）。该调度算法计算出的执行时间还会加上内核启动开销。我们还计算每个内存瓶颈的时间惩罚（b_penalty），并将其加到预测执行时间总值中。b_penalty 是 $sm_$惩罚 与$gm_$惩罚之和。

$$t_{kernel} = ts_{kernel} + l\_overhead + b\_penalty \tag{4.2}$$

这两个阶段的所有组成部分将在第 4.3.2 节和第 4.3.3 节中详细讨论。

4.3.2 延迟计算算法

在实际的图形处理器执行过程中，PTX 代码在图形处理器上执行之前，还必须经过若干额外的中间优化步骤。然而，我们无法访问优化后的代码。因此，我们剖析了这段 PTX 代码，以获取程序的属性和程序流程，并将其作为调度模型的输入。

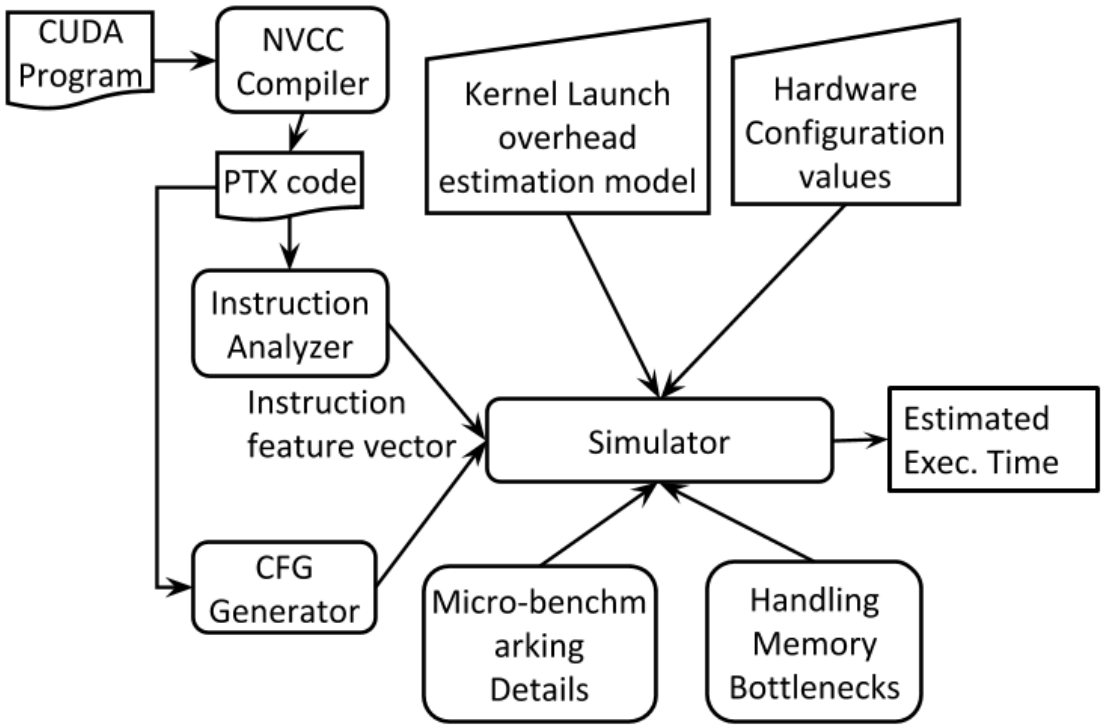


图 4.2：提出的执行时间预测模型

4.3.2.1 指令延迟计算算法

图形处理器调度模型需要计算完整执行一个单线程所需的 GPU 时钟周期数。内核程序的简化模型是一系列 PTX 指令，每条指令都以操作名称开头；如果指令使用操作数，后面还会列出寄存器操作数。我们从 PTX 代码中读取指令序列，然后逐条使用 Andreas 等人提出的参数化模型计算每条指令的延迟 [59]。以下各节将讨论本模型所采用的、源自这篇论文的参数方程。

虽然基础模型仍然相似，但要准确估算执行时间，就需要对其进行重大修改。具体而言，我们仅使用基础模型计算指令延迟，然后构建调度模拟来计算总执行时间。在调度算法中，为了计算周期数，我们

将指令依次分配给各个分派单元。但是，如果存在依赖指令，则会在下一个周期中考虑。为兼顾简洁与完整，我们在算法 4 中讨论了用于计算指令延迟的基础模型。

4.3.2.2 指令级并行性

在一个基本块中，指令级并行因子，即每周期有效执行的指令数，是遍历指令线程块后计算得出的，用于统计可以独立执行的指令数量。通过比较指令中使用的寄存器来检测它们之间的依赖关系。此处 指令级并行 $= n_{inst} n^{cycles}$，其中 n_{inst} 是基本块中已执行指令的总数，而 n^{cycles} 是独立执行周期数量的实际计数，用于计算我们依次分配给每个分派单元的指令所需的周期数。但是，如果存在依赖指令，则会将其计入下一个周期。 ____

4.3.2.3 线程级并行性

内核中的指令取自 PTX 分析阶段构建的基本块列表。SM 上活动线程束的最大数量wSM^M是硬件上限。由于它考虑了实际的内核启动参数，因此可将其视为初始线程级并行TLP。这样做是因为内核启动的线程块数量通常足以构成一个完整的波。为提高准确率，可以在每个阶段动态重新计算该值，但在大多数情况下，硬件限制可视为$TLP = wSM^M$。TLP是 SM 上活动线程束的数量。在估算所有指令的执行时间时，这是一个需要考虑的关键因素。

4.3.2.4 指令延迟计算

本研究使用的部分参数化方程受 Andreas 等人[59]的 GPU性能预测模型影响。他们已成功将这些方程应用于 Fermi 架构。我们正在通过改进这些方程，使其适用于 Kepler 架构。

按照他们的方法，如果线程束数量不足以达到峰值性能，那么所有指令都会并行执行，并且线程并行性会隐藏延迟。一旦指令流水线填满，后续线程束就需要等待核心空闲，因此会发生停顿。在这种情况下，需要额外的惩罚因子($\frac{Sz_w}{\mathcal{TP}}$)来衡量流水线停顿的影响。我们对 Andreas 等人方法的研究贡献是[59] 使用通过微基准测试计算得出的 PeakWarps 和吞吐量。Andreas 等人[59]使用每个 SM 的最大线程束数和吞吐量的理论值，而我们使用通过微基准测试测得的观测值计算指令延迟，从而提高了模型的准确率。

对于计算指令i，其延迟计算如下：

$$d_i^c = \begin{cases} \frac{\mathscr{L}}{ILP \times TLP} & if ILP \times TLP \leq peakWarps \\ \frac{\mathscr{L}}{ILP \times TLP \times peakWarps} + \frac{Sz_w}{\mathcal{TP}} & otherwise \end{cases} \tag{4.3}$$

同样，对于内存访问指令i，其延迟计算如下：

$$d_i^m = \begin{cases} \frac{\mathscr{L}}{ILP \times TLP} & if ILP \times TLP \leq peakWarps \\ \frac{\mathscr{L}}{ILP \times TLP} + p^m & otherwise \end{cases} \tag{4.4}$$

对于内存访问请求，当内存访问请求数量过多，图形处理器硬件受带宽和传输大小限制而无法一次性满足时，就会发生串行化

与计算指令的情况类似，会添加一个惩罚因子（p^m）来模拟串行化效应[59]。

4.3.2.5 内存访问延迟

内存访问指令的开销取决于访问是全局内存访问还是共享内存访问，因为这两类内存的架构和速度存在巨大差异。全局内存指令对总指令延迟的影响很大，因为该指令所需的时钟周期数在所有指令中最多。在全局内存访问的情况下，全局内存指令的开销为：

$$gm\_penalty = \frac{Sz_{gm} \times c}{BW_g} \tag{4.5}$$

对于每个基准测试，内存访问合并因子c 取决于访问步长，而这一因素无法通过分析 PTX 代码来确定。我们观察到，全局指令的内存延迟会随着访问步长的变化而变化。基于这一观察，我们针对全局加载/存储指令延迟与访问步长之间的关系建立了一个回归模型。该模型已在第3章中进行了详细说明。

在共享内存访问的情况下，惩罚计算为

$$sm\_penalty = \frac{Sz_w \times Sz_{sm}}{n_{sb} \times BW_s} + \frac{n_{bc} \times size}{BW_s}$$

其中，$Szsm$ 是与共享内存之间传输的字节数。共享内存被划分为有限数量的存储体n_{sb}。共享内存的设计使得对同一存储体的连续访问会被串行化，无法并行执行。通过分析 PTX 代码也很难对存储体冲突的数量（n_{bc}）进行建模，因此将留待未来工作研究。我们通过微基准测试观察到，共享内存访问指令的延迟为 47 个周期。

4.3.2.6 延迟计算算法

GPU核心上的指令以 SIMT 方式执行。如果线程束内存在分支发散，则两个分支中的指令都会依次执行；此时，不会进入某个分支的线程将处于空闲状态，而其余线程则执行相应指令。

为确定指令计数，我们假设每个线程束中的每个分支至少有一个线程会执行，即线程束执行期间不会有任何分支未被遍历。总指令计数通过累加顺序基本块中的指令数得到。如果检测到循环，则将循环路径的指令计数乘以迭代次数，然后继续计算。因此，内核的总延迟等于内核中所有基本块的延迟之和。我们将其视为单线程（或线程束）在图形处理器上执行所需的周期数。

单线程的执行时间t_{thread}为

$$t_{thread} = \frac{d_k}{\nu_{gpu}} \tag{4.6}$$

其中，内核(d_k)的总延迟为$d_k = \sum_{指令} d_i$。

算法4 延迟计算算法

过程 延迟计算($\mathcal{GPU}, \mathcal{L}, \mathcal{TP}$) $d_k = 0$
    对于每个*BasicBlock*在*Kernel*执行
        $d_B = 0$
        对于每个*Inst*在*BasicBlock*执行
            计算d_i
            $d_B = d_B + d_i$
        如果*BasicBlock*有循环**则**
            $d_B = d_B * n_i$
        $d_k = d_k + d_B$
返回d_k

4.3.3 GPU 调度算法

由于一个 GPU 核心一次只能执行一个线程，因此，GPU 上可同时启动的线程数量受核心总数限制。可能还有更多线程可供执行，但这些线程必须等待核心空闲后才能执行。由于这些线程需要等待，因此实际上它们是串行执行的。在我们的模型中，“波”是指在共享内存和寄存器需求等资源限制范围内，占满 GPU 上所有核心所需的线程数。所有线程都可以分组成这样的波。计算波的数量，就可以得到串行化线程组的数量，该数量由参数n_{serial}表示。

4.3.3.1 每线程执行时间

估算执行时间的第一步，是计算线程执行内核所需的 GPU 时钟周期数。这一估算基于这样一个前提：每条指令所需的时钟周期数各不相同，而这又取决于多种微架构硬件因素和程序特征。单个线程的执行时间（例如t_{thread}）为：

$$t_{thread} = \frac{\sum_{i \in insts} d_i}{\nu_{gpu}} \tag{4.7}$$

其中，d_i 的值取决于指令类型。

4.3.3.2 处理内存瓶颈

内存瓶颈对执行时间估算误差的贡献显著。这是因为内存瓶颈不仅取决于程序，也取决于硬件系统的状态。基于对调度算法的深入理解，可以通过理论方法分析程序的性能瓶颈。我们在该模型中引入了 Luo 等人提出的一种此类方法。 [60]

该方法计算一个名为 MPD（内存访问并行度）的参数，用于表示同一时刻可并行执行的最大线程束数。利用 MPD，我们可以计算每个瓶颈造成的时间开销，并将其加到预测执行时间总值中。全局内存带宽限制了内存传输速度。因此，如果过多线程束执行内存操作，受带宽限制，部分线程束就必须等待带宽可用。因此，这类执行会变为串行执行。为了计算可同时执行的线程束数上限，我们计算了每个线程束的内存带宽。然后，将 GPU带宽除以所有正在执行的线程束所需的带宽，得到 MPD。利用 MPD，我们可以计算瓶颈造成的时间开销，并在每次出现该瓶颈时将其加到总执行时间中。

4.3.3.3 调度算法中使用的参数

必须将内核启动参数，即线程块数（nB）和每个线程块的线程数（n_{tb}），提供给调度模型。启动的线程总数根据内核启动参数计算得出：$nT = n_{tb} * nB$；每个线程块的线程束数表示为$n_{wb} = n_{tb}Szw$。同一线程块中的所有线程都在同一个流式多处理器上执行。假设调度模型采用负载均衡方式，则图形处理器中的每个流式多处理器获得执行的线程块数量大致相等。因此，每个流式多处理器分配到的线程块数可通过将线程块均分到所有可用的流式多处理器上得出，即$nTh_sm = nTm_{tSM} * nSM$。利用每个流式多处理器的最大线程块数，可计算出一个波中每个流式多处理器可执行的最大线程块数：$nB^M = nTh\_sm/nSM$。

4.3.3.4 线程调度算法

CUDA 提供了一种抽象机制，用于描述线程的执行方式。在模拟图形处理器时，所使用的术语与程序启动参数的术语有所不同。本节将从基本单元的角度讨论仿真算法。

执行，即线程束。现在已获得延迟信息，必须在 GPU 上调度线程块，以便估算总体执行时间。仿真算法将 GPU 硬件规格以及 CUDA 占用率计算器报告的线程块数量限制作为输入，并计算 n_{serial} 的值，以确定总体执行时间。GPU 中的线程束调度器会从一个线程束中的每个线程调度一条指令，使其并行执行。这里通过假设一条指令 $Inst$ 在向所有线程束发出下一条指令之前，已在所有线程束上完整执行，来对这一过程进行建模。通过这种方式，所有指令都会在所有线程束中逐条执行。然而，这已足以计算总体执行时间。

我们假设所有基准测试中每周期执行一条指令。对基准测试进行性能分析并观察到每周期指令数（IPC）值介于 0.8 和 1.2 之间后，我们做出了这一假设。特定基准测试的指令发射周期数（n_{ic}）根据每周期发射指令的线程束数量和每个线程块中的线程束数量得出。随后，将n_{ic} 乘以t_{thread}，得到每个线程的执行时间（t_{thread}）。通过将线程块划分为若干组线程束，并分配给 GPU，直至每个 SM 达到每个 SM 的最大线程束数这一硬件限制（m_{wSM}）。这就是一轮分配。因此，一轮最多分配m_{wSM} *nSM个线程块。该过程持续进行，直到所有待调度的线程束都执行完毕。

仿真算法如算法 5 所示。在每个波次中，执行时间通过将调度的最大线程块数（$num_{blocksRound}$）乘以单个线程的延迟来计算。总执行时间是各个波次执行时间之和。所有可能的完整波次执行完毕后，再将剩余线程块的执行时间加到已计算出的执行时间中。由于线程块数量不足以占满所有流式多处理器，最后一个波次并不完整。最后，将内核启动开销和内存瓶颈惩罚加到执行时间上，得到总执行时间t_{kernel}。

第 21 行算法中线程调度所计算的时间ts_{kernel}。

算法 5 GPU调度算法

1: **过程** SCHEDULER($\mathcal{GPU}, d_k$)
2: $ts_{kernel} = 0$
3: $n_{ic} = \frac{kT_b}{n_{cSM}}$
4: $t_{thread} = \frac{d_k}{v_{gpu}}$
5: $t_{thread} = t_{thread} * n_{ic}$
6: **对** 每个 *ThreadBlock* **执行**
7: $num_{blocksRound} = 0$
8: 当前流式多处理器 $= 0$
9: 流式多处理器 计数器$[0..num_{SMs}] = 0$
10: **当**数量$_{blocksRound} <= nB^M$ **时执行**
11: 流式多处理器计数器[当前流式多处理器] = 流式多处理器计数器[当前流式多处理器] + 1
12: $num_{blocksRound} = num_{blocksRound} + 1$
13: **如果** $currentSM == num_{SMs} - 1$ **则**
14: 当前流式多处理器 $= 0$
15: **else**
16: 当前流式多处理器 = 当前流式多处理器 + 1
17: 秒$_{kernel}$ = 秒$_{kernel}$ + 最大(数量$_{blocksRound}$) * 吨$_{thread}$
18: $t_{kernel} = ts_{kernel} + l_$开销 $+ b_$惩罚 ty
19: **return** t_{kernel}

4.3.4 实验详情与数据收集

所提出的模型使用 Java 实现。除了将 PTX 代码作为主要输入外，模型还会读取配置文件，其中包含图形处理器的硬件特征，以及一些无法预测但应用程序开发者已知的、特定于内核的参数（启动参数、循环迭代次数）。流式多处理器上活动线程束数量与允许的最大线程束数量之比称为该流式多处理器的占用率。我们使用 NVIDIA 的 CUDA 占用率计算器 [121] 来获取块数和线程束数的最大值，并将其作为模型的输入，因为仿真阶段需要这些硬件限制。

本研究在 NVIDIA Tesla K20 GPU 上实现，该 GPU 由 13 个流式多处理器组成。此 Kepler 架构中的每个 SM 都配备 192 个单

精度 CUDA 核心、32 个加载/存储单元、32 个特殊函数单元（SFU）、64 个双精度单元和四个线程束调度器（每个线程束调度器有两个分派单元）。内核的执行时间在很大程度上取决于 GPU 底层的硬件特征。NVIDIA 的 CUDA 软件开发套件包含一个 *deviceQuery* 程序，运行后会报告正在使用的 GPU 的硬件特征列表。某些参数取自 GPU 的规格说明[122]。CUDA 软件开发套件还包含一个 *bandwidthTest* 程序，通过反复运行该程序来测得平均全局内存带宽。

4.3.5 结果

通过分析和调度 45 个基准测试内核，展示了所提出模型的结果。这些基准测试来自 CUDA 工具包[43] 和 Rodinia [123]。根据功能，基准测试可归类为常见的并行计算侏儒。将预测的内核执行时间与实际执行时间进行比较。每个内核的实际执行时间是通过在 Tesla K20 GPU 上使用 CUDA 事件运行内核得出的。所得平均绝对误差为 26.86%。这些基准测试的实际执行时间与预测执行时间对比图见图 4.3。该回归图的 R 平方值为 0.8245，残差平方和为 0.928。研究中的基准测试结果见表 4.2。

4.3.6 根据并行计算侏儒对结果进行分类

并行计算侏儒代表了计算与通信的模式 [1]。提出了 13 个 Dwarf，作为并行应用程序的代表，用于研究并行编程模型及架构。分析结果以了解哪些类型的内核模式能够获得较准确的预测结果，这将很有启发性。这有助于找出需要进一步研究的问题。

表 4.2：GPU调度算法的结果

基准测试	内核	执行时间（实际值）	执行时间（预测值）
二项式期权	BinomOpt	5541.7	4967
黑洞	BlackScholes	544.75	814.83
广度优先搜索	BFSv1 _65536	60.1	52.12
广度优先搜索	BFSv1 _4096	47.6	43.07
广度优先搜索	BFSv1 _1MW	119.9	226.6
CFD	CFD通量 _193K	613.2	355.999
CFD	计算流体力学时间步_导弹	119.8	81.44
CFD	计算流体力学时间步_193K	106.72	79.83
CFD	CFD初始化 _导弹	59.58	54.637
CFD	CFD初始化 _193K	58.66	53.89
CFD	计算流体力学时间步_97K	60.54	60.92
CFD	CFDStepF_193K	61.06	96.181
时钟	时钟	49.7	53.692
可分离卷积	可分离卷积 R_	1657.22	2422.2
卷积文本	ConvT _R	730.37	787.6
卷积文本	ConvT _C	621.41	787.675
快速沃尔什变换	FWT_M	799.42	775.623
高斯	GaussF2 _2048	1355.52	1134.2
高斯	GaussF2 _4096	5365.5	4651.41
高斯	GaussF1 _2048	334.144	299.3
高斯	GaussF1 _1000	45.024	43.074
K 均值	KmeansInvert	7715	3035.6
K 均值	KmeansPoint	2981.41	1247.89
LUD	LUDPeri	55.772	50.66
LUD	LUD对角线	52.5	47.97
矩阵乘法	矩阵乘法 200_	539.8	487.872
归并排序	归并MS_生成样本	3310.6	2830.641
NW	NW2	48.54	44.87
NW	NW1	43.81	44.83
粒子滤波器	并行滤波_Norm	368.67	330.88
saxpy	Saxpy	95.968	133.14
Scan	共享内存独占扫描	433.89	436.014
Shfl 扫描_	shfl_初始化图像	84.54	88.373
Sobol Qring	QR_inv	156.09	52.424
Sobol Qring	QR_QRNG	496.89	503.6
Srad	Srad1	81.952	66.29
立体	立体	2002.4	1256.9
转置	转置1_共享内存复制	42.465	42.651
转置	转置0_简单复制	101.12	116
转置	转置2_朴素	101.38	129.99
转置	转置7_对角线	86.75	149.46
转置	转置6_细粒度	78.3	141.12
转置	Tpose3_合并的	83.26	154.9
转置	Tpose4_优化	83.26	156.89
向量加法	VectAdd	2.64	2.4298

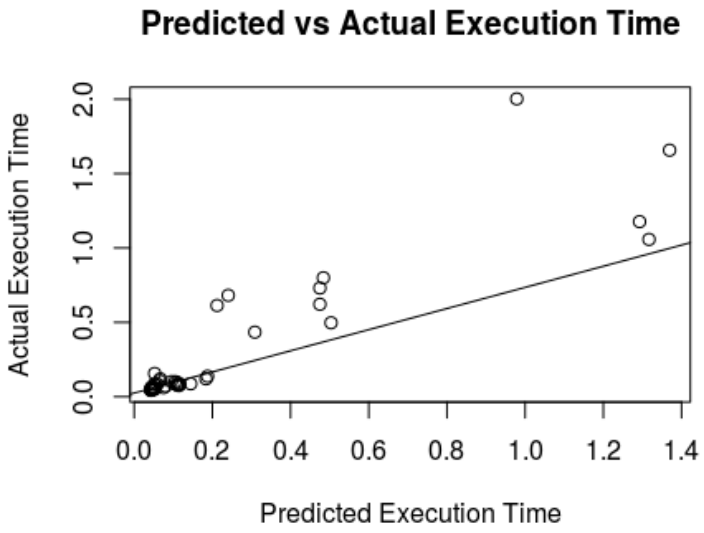


图 4.3：执行时间预测模型

为了了解这些并行小类如何帮助分析结果，我们将这些基准测试归类为并行小类[1][14]。表 4.3 展示了各小类的平均绝对误差。观察发现，对于由 Needleman-Wunsch算法（NW）基准测试内核表示的动态规划小类，其平均绝对误差为 4.93%；其次是稠密线性代数小类（例如矩阵乘法、高斯消元）。动态规划小类受内存延迟限制，而稠密线性代数受计算能力限制。由稀疏线性代数表示的内核同时受计算和内存限制，平均绝对误差为 29.41。

受内存延迟限制的谱方法 Dwarf（例如快速沃尔什变换）的平均绝对误差最高，为 39.46%。除蒙特卡洛之外，本研究考虑的所有其他 Dwarf 都是内存受限的；蒙特卡洛则取决于具体问题。基于这一分析，我们认为，与内存受限内核相比，所提出的模型对计算受限内核的效果更好。

表 4.3：Dwarf 及其平均绝对误差

Dwarf	内核数量	平均绝对误差
非结构化网格	9	29.07%
结构化网格	6	26.58%
谱方法	1	39.46%
稀疏线性代数	2	29.41%
蒙特卡洛	3	33.88%
图遍历	3	26.91%
动态规划	2	4.93%
稠密线性代数	19	26.47%

4.3.7 所提出算法的局限性

如前所述，使用静态分析以极高准确率预测 CUDA 内核的执行时间是一项难题。从这个角度来看，所提出模型的平均绝对误差为 26.86%，其表现尚属合理。在考察预测结果时，我们发现模型在许多情况下低估了执行时间，如图 4.3 所示。重新审视方法后，我们得出结论：在公式4.3和公式4.4中使用指令级并行（ILP）和线程级并行（TLP）等参数，会使该延迟值过低。

考虑具有指令级并行性的完全流水线架构时，指令执行所产生的延迟通常较小；然而，在某些情况下，它可能并不像使用 4.3 和 4.4 计算出的延迟那么低。尤其是在内核代码以少量线程执行某条指令的情况下，如果没有足够多的线程和指令同时执行，我们仍然必须等待该指令在已分配资源上完成执行（指令延迟）。如果根据该方程将此延迟值除以一个较大的数值，则计算出的延迟会低于实际延迟。因此，考虑资源分配方法有助于改善这种低估。在第一种方法中，除了最大并行执行线程束数量之外，模型在估算延迟时不考虑图形处理器硬件资源分配

。图形处理器具有单精度单元、加载存储单元、特殊函数单元等资源，分别用于处理不同类型的指令。根据与指令类型相关的硬件资源可用性详情来调度线程和指令，或许是设计更稳健、更高效模型的一种更直观方式。

在下一节介绍的第二种模型中，我们采用资源分配方法，从另一个角度对指令级并行（ILP）和线程级并行（TLP）模型进行建模。在这种方法中，我们根据指令并行执行的可能性，对指令到资源的映射进行建模，从而对指令级并行（ILP）进行建模。线程级并行（TLP）则通过对线程并行执行施加硬件资源限制来建模。我们沿用第一种方法中的 CUDA 代码静态分析，用于构建基本块和控制流图。然后，我们考虑硬件约束，对这些拓扑排序后的指令进行调度。下一节将详细介绍该模型。

4.4 使用资源分配对 GPU 性能进行建模

性能预测是指估算计算机的执行时间，或估算其他性能因素（例如缓存未命中）的执行情况。本研究使用“性能预测”这一术语来表示对 GPU 内核执行时间的预测。所提出的性能预测模型以 CUDA 程序作为输入，并估算其执行时间，如图 4.4 所示。

我们重复利用了第一种方法中的部分预处理步骤和数据。因此，图 4.4 和图 4.2 有显著相似之处，因为两种方法中通过静态分析 PTX 代码进行的预处理步骤完全相同。

两个模型都将内核启动参数作为用户输入，并使用微基准测试数据。然而，两个模型在内存行为建模和图形处理器调度算法方法方面存在显著差异。

整个过程由三个主要阶段组成：

1. 我们编译该程序以获得相应的 PTX 格式[109]。我们还会考虑与内核启动参数相关的输入，即网格大小（线程块数）和块大小（每个线程块的线程数）。然后对 PTX 代码进行静态分析。

2. 我们只需运行 NVIDIA 提供的设备查询应用程序一次，即可收集所考虑图形处理器的硬件详情，并据此开发一组模型。我们还会运行各种微基准测试程序，收集性能预测所需的其他若干参数。

3. 我们利用上述信息对 PTX 代码进行调度，以估算总延迟。使用我们提出的模型计算 CUDA 内核执行时间（t_kernel）可概括为以下公式：

$$d_{total} = d_{kernel} + l\_\text{开销} + gm\_\text{惩罚} + sm\_\text{惩罚} + cm\_\text{惩罚} \quad (4.8) \qquad t_{kernel} = \frac{d^{total}}{v_{gpu}} \quad (4.9)$$

公式 4.8 汇总了指令执行造成的延迟以及惩罚，这些因素对内核的总执行时间有显著影响。我们通过使用第 4.4.2 节中讨论的算法 6-8 分析 PTX 代码，来计算内核的模拟延迟（d_{kernel}）。在此延迟的基础上，我们加上启动开销、全局内存惩罚、共享内存惩罚和缓存惩罚，这些因素对内核的总执行时间有显著影响。这些惩罚在所使用的资源方面彼此独立，因此可以累加。我们通过运行按第 4.4.5 节详细讨论的方法设计的微基准测试，来计算这些内核惩罚因子。

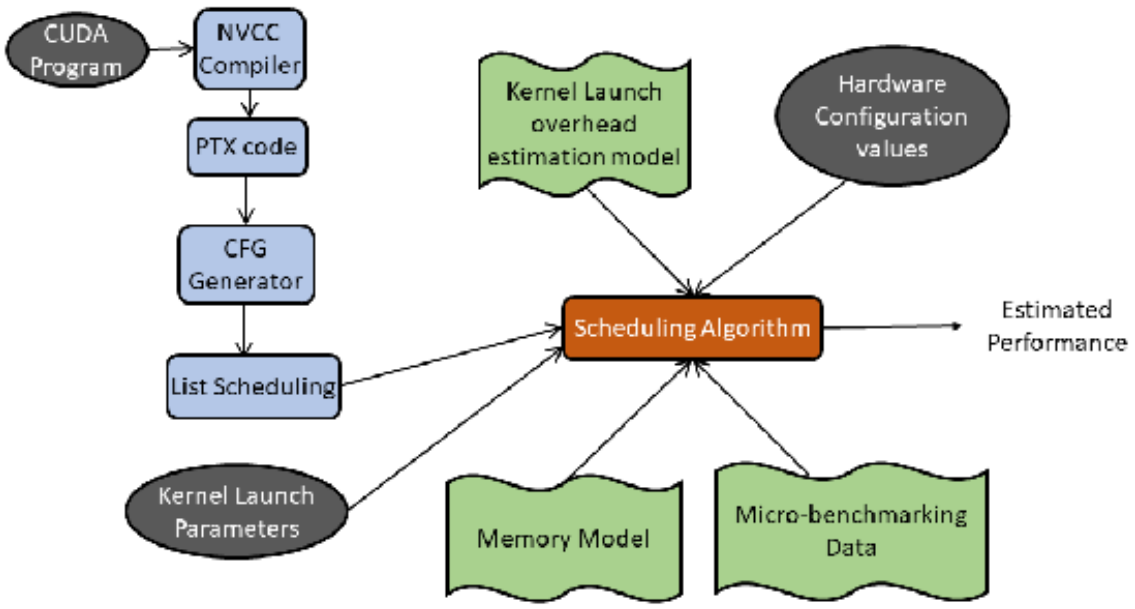


图 4.4：提出的性能预测模型

4.4.1 分析循环和分支发散

总体指令计数考虑了循环估计（基于迭代次数）和分支发散。如果识别出循环，则需要将循环路径的指令延迟乘以用户提供的输入（循环迭代次数），然后继续进行计算。由于大多数实际应用程序都是非仿射的 [124], ，且非仿射基准测试的循环迭代次数预测问题尚未得到解决，因此我们必须依赖用户提供循环迭代次数。

GPU 流式多处理器按照单指令多线程（SIMT）模型执行每条指令，在该模型中，一个线程束（由 32 个线程组成）中的线程执行相同的指令。当一条指令属于分支类型时，线程束中的所有线程可能不会采取相同的分支，从而导致分支发散 -[125]。与循环一样，很难预测分支发散。

所提出的方法假设每个线程束中的每个分支至少会被一个线程执行。换句话说，我们假设在线程束执行期间不会有任何分支未被遍历。这意味着，执行该分支的线程将执行该

指令，而其余线程将执行空操作（NO-OP）。因此，执行时间将根据控制数据流图中的最长路径计算。然而，如果在线程束执行期间从未选择最长分支，则最终的指令延迟可能不够准确。

4.4.2 调度模型

利用上一节讨论的静态分析数据，我们建立了一个调度模型，用于刻画图形处理器的行为特征。由于供应商不会公布流水线细节、分支预测和缓存行为等关键信息，因此我们对图形处理器的运行方式做出了一些假设，以实现图形处理器调度。首先，我们讨论资源分配，它在调度中起着至关重要的作用。

4.4.2.1 资源分配

如第4.1节所述，$\mathcal{GPU}$ 由一组资源R构成。算法6以SIMT方式对指令u 进行调度和资源分配，以计算延迟。在此过程中，该算法为u分配多个该资源类型的实例。

例如，假设一条加法指令使用的资源类型是“单精度加法器”。调度加法 指令时，算法会分配 GPU 中的所有单精度加法器（例如 32 个）。这样一来，如果可用资源数量足够多，多个线程束可能会同时处于活动状态。实际上，调度机制和线程束管理相当复杂，因为分派执行的指令可能混合来自多个线程束。若不了解 NVIDIA GPU 所采用的确切调度策略，就无法对此类复杂情况进行建模。我们在算法 6-8 中未对这种行为进行建模。

4.4.2.2 波的概念

准备执行的线程数可能超过 GPU 中允许的最大线程数（每个流式多处理器）。在这种情况下，所提出的调度器会分批次执行线程。我们在本文中将这种分批执行称为“波次”。在我们的模型中，一个执行波次表示 GPU 一次可以启动的线程数。随后，将观测到的单个波次执行时间乘以启动的波次数，即可得到总执行时间。

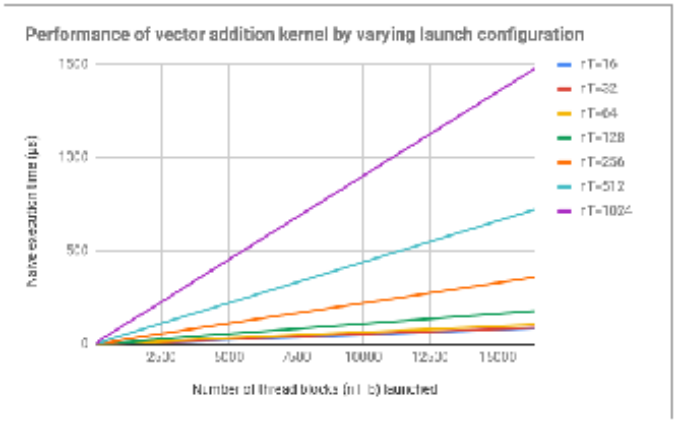


图 4.5：启动参数对内核性能的影响

为了计算波次数量，我们尝试在 Tesla K20 GPU 上运行向量加法内核，该 GPU 有 13 个流式多处理器；运行时使用不同的 nB （块数）和 nT_b （每个线程块的线程数）值作为启动参数。结果如图 4.5 所示。例如，我们观察到，当 $nB = 15$ 时，SM0 获取线程块 0 和线程块 13，SM1 获取线程块 1 和线程块 14，其余流式多处理器各获取一个线程块，而这与 nT_b 的值无关。直观来看，这意味着波次数量可以计算为 $波次 = \lceil nB/nSM \rceil$。Kothapalli 等人 [57] 也提出了这种方法。然而，仔细观察图 4.5 会发现，接下来将解释这种直观的公式并不成立。

假设 K20 图形处理器执行 13 个向量加法内核线程块（每个流式多处理器执行一个线程块）需要 t 个时间单位。那么，

如果这种直观的估算足够准确，那么尝试启动 26 个线程块时，图形处理器应该需要接近 2t 个时间单位。然而，图3中的 nT_b 曲线都没有呈现这种行为。例如，nB=5000 的实际执行时间并非 nB=2500执行时间的两倍。因此，我们不能得出这样的结论：一般而言，计算出一个波的执行时间后，将其乘以 $\lceil nB/nSM \rceil$，就能得到总执行时间。

在所提出的方法中，我们首先计算每个流式多处理器（SM）需要调度的线程总数（nTh_Schd），计算方式为 $nB * nT\_bnSM$。接下来，根据寄存器和共享内存需求，计算单个波次中每个流式多处理器（SM）可容纳的最大线程数（nTh_sm^M）。算法8使用 nTh_Schd 和 nTh_sm^M 计算波次数量。

在每个波次中，我们采用一种资源分配技术来建模图形处理器上线程的执行，该技术的灵感大致来自 [126]。

4.4.3 其他假设

其他假设如下：

- 图形处理器采用超标量架构；流水线填满后，每个周期至少有一条指令完成执行[20]。为对此进行建模，我们在算法6第6行的指令调度过程中，将指令延迟划分为流水线延迟（$\mathscr{L}_{\text{流水线}}$）和指令延迟（$\mathscr{L}$）。

- 波次的数量取决于执行该类型指令所需资源的可用情况。因此，我们在线程调度过程中计算波次数量。

- 无论每个线程块的线程数是多少，一个线程块都会被调度到一个流式多处理器上。当所有流式多处理器都各自分配到一个线程块后，下一轮分配将从第一个流式多处理器开始。实验验证了这一现象。为适应这种行为，我们在算法8第2行通过用线程块数（nB）除以流式多处理器数（nSM），计算 nTh_schd。

• 由于多个线程束会依次启动，因此无法按照图形处理器上实际的指令执行顺序进行模拟。线程束调度器可以选择n 个线程束，并根据架构资源限制[127]，每个周期分派m 条独立指令。因此，当调度了多个线程束时，不同线程束中的指令不会遵循特定顺序。然而，实现这种行为极具挑战性。在此，模型假设一个线程束的指令全部执行完毕后，下一个线程束才恢复执行，如算法6第15行所示；其中，根据所使用的资源为每条指令增加延迟。

4.4.4 调度算法

我们希望找到一条指令的最优执行顺序，因此实现了一种基于列表调度 [126, 128] 的算法。该调度遵循两个约束条件，即：

1. 优先约束： 调度器在调度操作时使用与基本块G 关联的数据流图，并确保具有边（u、v）的两个节点之间的最小延迟为节点 u 的延迟($\mathscr{L}(u)$)。在算法7中，调度器还通过考虑整个内核的控制流图G_{cfg}来遵循线程块之间的控制流。

2. 资源限制： 由于图形处理器的计算遵循 SIMT 模型，因此一条指令由多个线程执行，每个线程处理不同的数据元素。即使调度器能够调度大量线程来执行一条指令，执行最终仍受限于可用于该指令的资源总量。例如，如果有N 个线程准备执行加法运算，且有$R[SP]$ 个单精度核心可用（每个流式多处理器），则该加法运算将分为 $\left\lceil \frac{N}{R[SP]} \right\rceil$ 个批次执行。所提出的调度

算法在计算执行周期时会考虑这一资源限制。

算法6 线程块调度

1: **过程** SCHEDULEBLOCK($\mathscr{GPU}$, $G = \langle V, E \rangle$, nTW)
2: $V' \leftarrow$ 拓扑排序(G)
3: 调度器 $\leftarrow$ []; 保留表 $\leftarrow$ [][]; ▷ 调度器 负责跟踪每个节点的调度情况，而保留表 是资源预留表 , 初始化为 0
4: $dela$ 秒 $\leftarrow 0$;
5: **对于** $v \in V'$ **执行**
6: $t \leftarrow \max_{(u \to v)} \left\{ Sch[u] + \mathscr{L}(u) + \mathscr{L}_{流水线} \left(\left\lceil \frac{nTW}{R[R^{map}(u)]} \right\rceil - 1 \right) \right\}$
7: $j_d \leftarrow R^{map}(v) + \mathscr{L}_{pipeline} \left(\left\lceil \frac{nTW}{R[R^{map}(v)]} \right\rceil - 1 \right)$
8:
9: 令 $(t1 \geq t)$ 为满足以下条件的索引：$Rsv[t1, j] \cdots Rsv[t1 + (d-1), j]$均为 0。
10: $Sch[v] \leftarrow t1; i \leftarrow 0$;
11: **对于** $i < d$，**执行**
12: $Rsv[t1 + i, j] \leftarrow 1$; i++;
13: $dela y \leftarrow Sch(v) + d$;
14: **返回** 延迟

算法7 调度控制流图

1: 过程 SCHEDULECFG($\mathscr{GPU}$, Gcf g ← 〈ℂ,𝔼〉, nTW) 2: BB′ ←拓扑排序(Gcf g) 3: d ela y ← [ ]4: 对 G ∈BB′ 执行 5: d ←最大值$_{e=G_p \to G}$ {d ela y[G p]}6: d_G ← SCHEDULEBLOCK($\mathscr{GPU}$,G, nTW, n gM, nShM) 7: 如果 G 有回边，则 8: $d_G \leftarrow d_G *$ nLoo p 9: d ela y[G] ← d + d_G10: d_{cf}g ← d ela y[G]11: 返回 d_{cf}g ▷ 控制流图的延迟

4.4.4.1 CountingMemory 指令

除了指令调度之外，还需要统计内核控制流图中的全局和共享内存访问指令总数，分别记为 nGM 和 $nShM$,

算法8 调度内核

1: **过程** SCHEDULEKERNEL($\mathscr{GPU}$, $G_{cfg} = \langle \mathbb{C}, \mathbb{E} \rangle$, nB, nT_b, reg_t, $shmb$)
2: $nTh\_schd \leftarrow \lceil \frac{nB\ n流式多处理器}{} * nT\_b \rceil$;
3: 波次 $\leftarrow 0$;
4: $nTh\_流式多处理器 \leftarrow nTh\_schd > nTh\_流式多处理器^M ?\ nTh\_流式多处理器^M :\ nTh\_schd$;
5: 最大寄存器 $\leftarrow$ 每个线程组_的 × 个线程，_个流式多处理器 $>$ 每个线程组_个块 $^M ?$ 寄存器-块^{M}寄存器_线程×个线程_块×个线程_块： 线程组_个流式多处理器；
6: $maxSh \leftarrow (shmb \times nTh\_sm\ nT) > shm\_b^M ?\ shm\_b^M\ shmb \times nT\_b :\ nTh\_sm$；
7: $maxB \leftarrow (shmb \times nTh\_sm\ nT) > nB^M ?\ nB^M \times nT\_b :\ nTh\_sm$；
8: $nTh\_sm \leftarrow min(maxR,\ maxSh,\ maxB,\ nTh\_sm)$;
9: **当** nTh _调度 ≥ 0 **执行**
10: 第 n 个_波 $\leftarrow$ 第 n 个_流式多处理器;
11: $d_{kernel} \leftarrow d_{kernel} +$ SCHEDULECFG($\mathscr{GPU}$, G_{cfg}, 第 n 个_波);
12: 第 n 个_调度 $\leftarrow$ 第 n 个_调度 $-$ 第 n 个_流式多处理器^M;
13: 波次 $\leftarrow$ 波次 $+1$;
14: 填入 $nSHM$ 和 nGM 的值;
15: **返回** (d_{kernel}, 波次, nGM, $nSHM$); ▷ 内核延迟

这两个计数用于估算全局内存和共享内存访问开销，下一节将对此进行介绍。统计工作在调度基本块中的每条指令时执行（算法6的步骤6）。为简洁起见，此处不展示这一简单的统计机制。我们强调，算法8在步骤10中获取这两个计数。

4.4.4.2 图示

我们使用第 4.4.2.2 节中介绍的向量加法内核示例，说明如何进行调度。如第 4.4.3 节所述，图形处理器在执行过程中会运行多种指令，而我们无法对其建模。因此，所提出的算法假设向量加法 内核以 100% 的占用率运行，并在调度某条指令时确保该指令所需的所有资源均可用（如第 4.4.4 节所述）。

图 4.6 展示了相应 PTX 代码片段以及该 PTX 片段的数据依赖图，其中 SB 和 EB 是两个虚拟节点，分别表示开始。

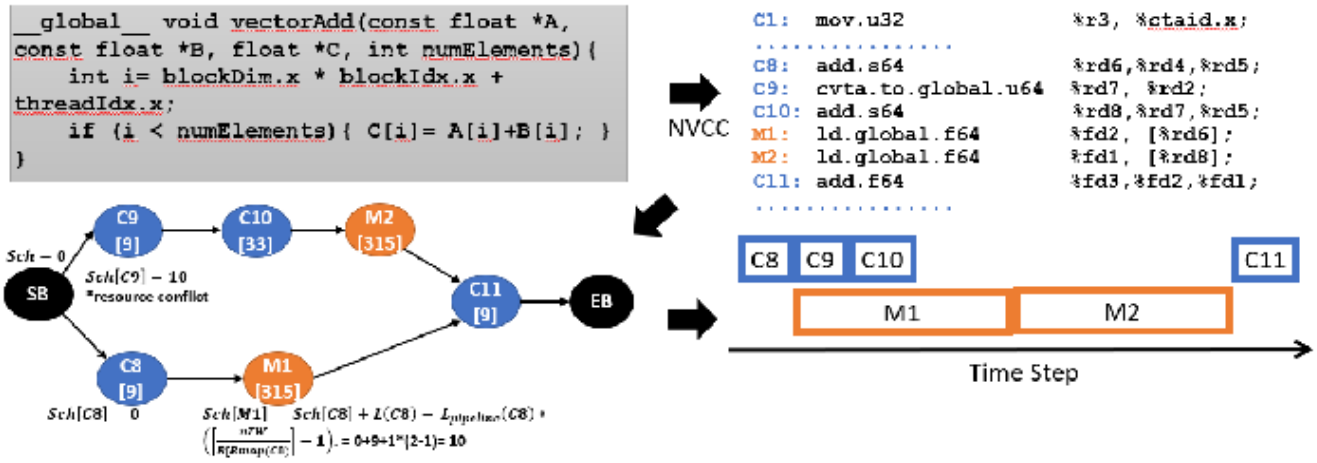


图 4.6：算法6示意图

分别为基本块的末尾。假设目标 GPU 为 Tesla K20，其 $\mathscr{L}$(ld) 的延迟为 315 周期，$\mathscr{L}$(加法)= 9 周期，$\mathscr{L}$(cvta)= 33 周期。稍后我们将说明这些延迟值是如何得出的。Tesla K20 拥有 192 个核心和 32 个加载存储单元，因此 $R[SP]=192$，且 $R[LSU]=32$。假设此内核的启动参数为 13 个线程块，每个线程块有 256 个线程，因此 $nTW=256$。

算法6按拓扑序对该图进行排序，以便逐一调度节点。其中一种拓扑序为 C8、C9、C10、M1、M2、C11。由于 SB 是一个虚拟节点，$Sch[C8]$,表示用于调度 C8 的时间单位（相对于线程块），因此其值为 0。步骤 11 和 12 将资源 SP 线程块 10 个周期。在下一次迭代中，对于 C9，算法6的步骤 6 和 7 将计算 $t=0$。然而，C8 已经为加法 保留了资源 10 个周期，即 $Rsv[0,SP]\cdots Rsv[9,SP]=1$。因此，C9 只能在第 10 个周期开始（$Sch[C9]=10$）。算法的步骤 8 和 9 会计算出这一结果。内存受限指令 M1 在 C8 之后开始，并与 C9 并发执行。虽然 M2 也可以与 M1 并行开始，但算法6会在 M1 之后调度 M2，因为假设加载存储单元（加载存储单元）已被 M1 完全占用（算法6的步骤 8 和 9）。由于 C11 对 M1 和 M2 存在数据依赖，因此会为其可能的调度计算 $\max\{\langle M1\rightarrow C11\rangle,\langle M2\rightarrow C11\rangle\}$。这反过来意味着，在算法6的步骤 6 中，$t\leftarrow\max\{332,655\}=655$。由于没有其他

使用资源的指令，用于加法、$Rsv[655,加法] = 0$ 和$Sch[C11] = 655$。

延迟隐藏 延迟隐藏涉及将指令的执行时间与另一个 [85] 的执行时间重叠。算法6在确保资源限制的同时实现延迟隐藏。在此示例中，C9 和 M1 相互独立，其资源 $R^{map}(C9) = SP$ 和 $R^{map}(M1)$ = 加载存储单元 不同；因此，它们会按图 4.6 所示同时进行调度。当两条指令同时开始时，C9 指令的延迟被 M1 的执行隐藏。实际上，C9 和 C10 的延迟都通过使用 M1 得到隐藏。

接下来，我们强调另一种由指令流水线化引起的执行延迟。在我们的模型中，一条指令 v 在 $nTW/R[R_{map}[v]]$ 个批次中执行。除第一个批次外，一个批次中 v 的延迟会被下一个批次中的延迟所隐藏。我们假设，当v 完全实现流水线化时，其延迟为 1。让我们考虑 M1 的调度，它在图 4.6 中的 C8 之后开始。尽管 $\mathscr{L}(C8) = 9$, $\lceil nTW/R[SP]\rceil = \lceil 256/192\rceil = 2$。通过指令流水线化，C8 将耗费 $9 + 1 * (2 - 1) = 10$ 个周期。由于$Sch[C8] = 0$，$Sch[M1] = 0 + 9 + 1 = 10$个周期。

4.4.5 动态分析

内核执行时间会受到在 GPU 上启动内核所需时间的影响，这通常称为内核启动开销。这是执行内核指令前后所消耗的时间。我们构建了一个经验模型来描述这一开销。

NVIDIA GPU 具有复杂的内存层次结构，包括全局内存、共享内存、纹理内存和常量内存。内存访问指令需要区别处理，因为它们造成的延迟不仅受指令并行性的影响，还受资源限制的影响。我们提出的内存访问模型由

三个部分组成：i) 访问延迟模型；ii) 访问惩罚模型；iii) 缓存未命中惩罚模型。

4.4.5.1 内存访问惩罚模型

由于其他因素，例如全局内存的访问模式以及共享内存的存储体冲突，内存指令的延迟往往更高且难以预测。在计算 CUDA 内核的总延迟时，我们需要将内存指令的这一特性考虑在内。仅凭一条全局内存指令和共享内存指令的计算延迟，不足以对访问模式或存储体冲突这一现象进行建模。为在计算延迟时纳入这一特性，我们采用了为内存访问指令引入相应惩罚因子的方法。将这些惩罚与调度器得出的延迟相加，即可得到总执行时间，如公式 4.8 所示。

图形处理器提供多种内存选项，例如全局内存、共享内存、常量内存和纹理内存。我们在模型中考虑全局内存和共享内存的开销，因为常量内存和纹理内存更多地应用于图形应用程序。我们专注于通用GPU计算（GPGPU）。我们在图1.3中展示了CUDA线程的内存设计。

我们为图形处理器的全局内存指令（公式 4.10）、共享内存指令（公式 4.11）和缓存行为（公式 4.12）建立了惩罚模型。以下各小节将详细介绍每个惩罚模型。

4.4.5.2 全局内存惩罚

对于全局内存而言，影响其性能的一个关键因素是我们无法在调度期间建模的内存访问合并效应。当 GPU 流式多处理器执行一条内存指令时，该流式多处理器会创建内存请求，并切换到另一个线程束，直到该线程束请求的所有内存值都可用为止[59, 129]。我们的模型

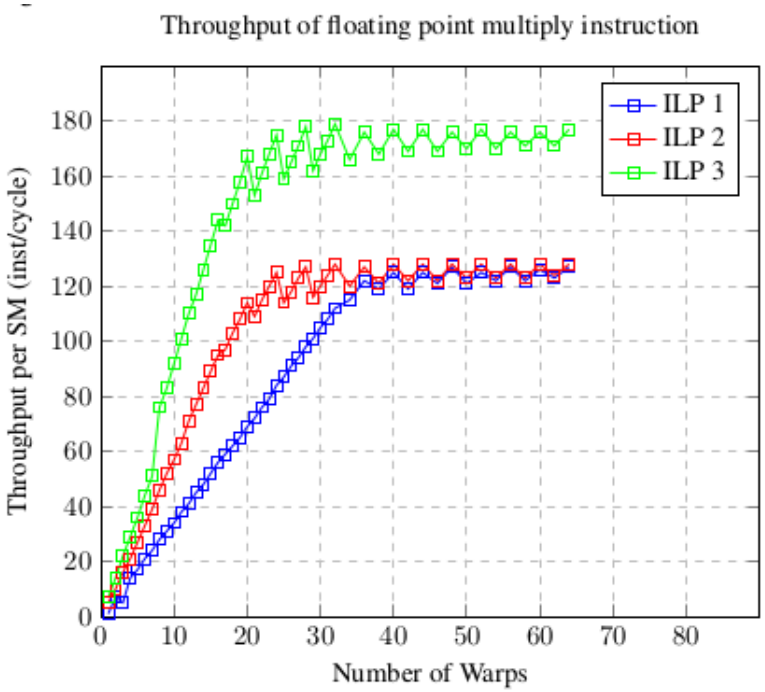


图 4.7：计算全局内存吞吐量的模型

表4.4：不同统计函数下动态分析参数的建模结果

参数	线性回归		对数增长 指数增长			
	R_平方	均方根误差	R_平方	均方根误差	R_平方	均方根误差
内核启动开销	0.9877	13.15	0.3240	97.5958	0.8398	47.95
共享内存吞吐量	0.2472	181202.3	0.8360	83413.19	0.9782	41347.23
全局内存吞吐量	0.1896	23075.85	0.8798	8885.28	0.9528	5568.59
	R_平方		RMSE			
缓存惩罚	0.9045		3797.42			

表 4.5：全局与共享内存吞吐量的指数模型

图形处理器架构 全局内存吞吐量 R_{global} 共享内存吞吐量 R_{shr}		
Tesla K20	76363.8*(1.04-exp(-0.00021342*ngM)	823761.8*(1 - exp(-0.000013830*纳秒M)
Quadro K4200	68145.9*(1.03-exp(-0.00027584*ngM)	70628.9019*(1.03-exp(-0.000301068*nsM)
Tesla M60	71453.6*(1.22-exp(-0.000033469*ngM)	1618990*(1-exp(-0.0000055295*nsM)
GTX 1050	47244.33*(1 - exp(-0.001223*ngM)	613743.6*(1.12-exp(-0.00000694*纳秒M)

通过加上一个作为惩罚值的额外延迟来纳入这一现象。该惩罚值取决于全局内存请求的数量 nGM，由算法8推导得出。全局内存惩罚的推导如下：

1. 我们通过运行包含全局加载/存储指令的应用程序，收集了内存吞吐量数据（即读取或

存储在全局内存中的数据速率）。内核使用不同的启动参数（线程块数和每个线程块的线程数）启动，以确保通过改变指令数来记录吞吐量。

2. 我们通过尝试拟合线性、对数和指数增长函数来构建吞吐量模型。表4.4列出了这些模型各自的 R 平方值。根据结果，我们得出结论：指数增长函数

$$\mathscr{TP}(gm) = a \times (b - e^{-c \times \mathscr{L}(gm)})$$

代表全局内存事务的模型具有最高的R平方值和最低的均方根误差值。该指数增长模型如图 4.7 所示，适用于 Tesla K20。我们已在附录 8 中讨论了如何测试全局内存吞吐量模型的有效性。

3. 全局内存惩罚首先通过计算使用加载存储单元（$^{nB \cdot nT\_b}_{R[\text{加载存储单元}]}$）进行的内存访问次数来确定。然后，我们将$accessGm_{sz}$ 除以其 $\mathscr{TP}$（gm），以测量每次访问所需的时间。

4. 将调度算法获得的全局内存指令数量（nGM）乘以单条全局内存指令的惩罚值，从而得到所有全局内存加载和存储指令的总惩罚值。

我们通过以下公式计算全局内存惩罚：

$$gm\_penalty = \frac{nB \cdot nT\_b}{R[LSU]} \times \frac{accessGm_{sz}}{\mathscr{TP}(gm)} \times nGM \tag{4.10}$$

4.4.5.3 共享内存惩罚

在访问共享内存时，CUDA 内核的执行也会受到存储体冲突的影响。我们通过为共享内存访问添加惩罚值来对此进行建模。惩罚值的计算方式是采用通过以下方式计算出的吞吐量。

受 bankconflicts[59] 影响的 observedthroughput。可通过将访问大小除以该观测吞吐量，计算这些共享内存请求（$nShM$）的估计延迟。

共享内存惩罚通过以下步骤计算：

1. 与全局内存吞吐量的行为类似，共享内存吞吐量在相等的时间间隔内按相同比例增长。为了根据共享内存加载/存储事务的数量对共享内存吞吐量（$\mathcal{TP}sm$）进行建模，我们分别测试了线性、对数和指数增长模型，其R平方值见表4.4。其中，指数增长模型

$$\mathcal{TP}(sm) = a \times (b - e^{-c \times \mathcal{L}(sm)})$$

产生最大的R平方值和最小的均方根误差（RMSE）值，从而确保良好的拟合效果。

2. 然后，通过计算每个流式多处理器（SM）调度共享内存指令的次数，计算一条共享内存指令的惩罚——$nB \cdot nT\_b$ $R[LSU] \cdot nSM$。接下来，我们计算每条共享内存加载或存储指令的调度耗时（$access\mathcal{TP}^{sz}(sm)$）。其中，$accessShm_{sz}$ 表示每条共享内存加载和存储指令访问的字节数。

3. 在最后一步中，我们使用调度算法中统计的共享内存指令数量（nShM），为所有已调度的共享加载和存储指令计算共享内存惩罚。

sm_惩罚的计算方法如下：

$$sm\_penalty = \frac{nB \cdot nT\_b}{R[LSU] \cdot nSM} \times \frac{accessShm_{sz}}{\mathcal{TP}(sm)} \times nShM \tag{4.11}$$

4.4.5.4 缓存未命中惩罚

准确建模缓存行为是一项无法实现的任务，因为它取决于多个运行时因素。NVIDIA GPU 二级缓存存储了部分全局内存，而全局内存是

指令延迟的主要来源。我们通过利用可用的架构和程序详细信息计算 L2 缓存未命中惩罚来近似这种行为，以捕捉图形处理器的行为。缓存惩罚的计算步骤如下：

1. 为了估算缓存已满且缓存未命中会导致惩罚（我们将其建模为 $cm\_penalty$）的次数，我们采用算法6、7和8中计算出的“波次”这一概念。

2. 符号 $L2\_sz$ 表示 L2 缓存的大小，该大小通过设备查询得出。

全局内存指令的总数 nGM （在调度期间静态计算）除以可用缓存内存，由此得到缓存未命中次数的近似值。

3. 随后，使用该值以及全局内存延迟（通过微基准测试计算，见表 3.3），计算内核执行期间可能发生的缓存未命中的缓存惩罚。

缓存未命中惩罚函数采用以下公式建模：

$$cm\_penalty = \frac{nT\_b \cdot nB \cdot nGM}{\frac{waves \cdot L2\_sz}{access_{sz}}} \times \mathscr{L}(gm) \tag{4.12}$$

我们通过将提出的公式与使用nvprof性能分析工具[1]对多个基准测试进行性能分析并记录 L2 缓存未命中率所得的数值进行比较，验证了该公式。根据该未命中率，我们计算出实际缓存惩罚以进行验证。我们将实际惩罚值与通过上述公式得出的缓存惩罚进行比较。表4.4列出了实际值与预测值之间的 R平方值和均方根误差，以佐证我们的结论。

[1]https://docs.nvidia.com/cuda/profiler-users-guide/index.html

4.4.6 结果

我们开发了一款基于 Java 的预测器工具，它可以分析 PTX 代码，以内核启动参数（块数、每个线程块的线程数、循环迭代次数）和动态分析模型的结果为输入，预测 CUDA 内核的执行时间。为了收集动态分析结果，我们通过微基准测试收集每种指令类型的延迟值。其他运行时特征（gm_penalty、sm_penalty）则通过基于`nvprof`记录的数据构建模型来收集。我们还在表4.8中记录了该工具执行预测所需的时间。平均而言，我们的预测工具预测 CUDA 内核执行时间需要 66.48ms。在使用搭载 8GB 内存的 Intel i3-5005U CPU、JVM 版本 15.0.1 进行分析时，观测到的最短预测时间为 1.031ms，最长为 206.62ms。

4.4.7 实验设置

本研究中使用的 CUDA 基准测试涉及不同的应用领域，例如向量运算（如向量加法、Saxpy）、矩阵运算（如转置、矩阵乘法）、数学问题（如 BFS、高斯）以及高级运算（如 BlackScholes）。我们使用了来自 CUDA 工具包 [43] 和 Rodinia 测试基准 [123] 的应用程序。

在构建模型的过程中，我们设计了实验，并使用基于 Kepler 架构的 Tesla K20 GPU 验证了结果。我们还使用同样基于 Kepler 架构的 Tesla K4200 GPU 验证了结果。为了探究所提出的模型在多大程度上与架构无关，我们在 Maxwell（Tesla M60）和帕斯卡（GTX 1050）架构上重复了实验。架构特征值见表 4.6。

表4.6：所研究 GPU 的硬件特征值

GPU 硬件特征	Quadro K4200	Tesla K20	Tesla M60	GTX1050
架构	Kepler	Kepler	Maxwell	帕斯卡
计算能力	3.0	3.5	5.2	6.1
流式多处理器数量	7	13	16	5
每个流式多处理器的核心数	192	192	128	128
GPU时钟频率（MHz）	706	784	1178	1493
内存时钟频率（MHz） 2700		2600	2505	3504
L2 缓存（字节）	524288	1310720	2097152	524288

4.4.8 数据收集

我们通过运行针对每种指令类型的基准测试应用程序，收集了微基准测试数据。这些应用程序在每种架构上仅运行一次。同样，我们使用`nvprof`对包含相应指令的应用程序进行性能分析，运行应用程序以收集不同指令数量下的全局内存吞吐量和共享内存吞吐量。硬件属性则通过运行 CUDA示例中提供的设备查询应用程序来收集[43]。

为了测试和验证我们提出的模型，我们在每种架构上运行了许多常用的基准测试，其中一些基准测试包含多个内核，如表4.7所述，并使用`nvprof`记录其执行时间。有些内核可以使用不同的启动参数启动，而有些内核则需要固定的启动参数。我们考虑所有允许的启动配置，为这些基准测试创建了108种配置。有些基准测试可以在不同的数据集上运行，而对于某些基准测试，其样本数量受基准测试设计者随附的数据集限制。平均而言，每种配置有四个样本，我们有400多个数据样本用于测试模型。结果见表4.8和表4.9。

表4.7：研究中的基准测试

基准测试	内核名称	来源	Dwarf类型
广度优先搜索	Kernel1	Rodinia 基准测试套件	图遍历
广度优先搜索	Kernel2	Rodinia 基准测试套件	图遍历
卷积 可分离卷积 convolutionRowsKernel		CUDA 工具包	结构化网格
卷积 可分离卷积 列卷积核 CUDA 工具包			结构化网格
卷积纹理	convolutionRowsKernel	CUDA 工具包	结构化网格
卷积纹理	列卷积核 CUDA 工具包		结构化网格
Srad	srad_cuda_1	Rodinia 基准测试套件	结构化网格
Srad	srad_cuda_2	Rodinia 基准测试套件	结构化网格
Hotspot	计算_临时值	Rodinia 基准测试套件	结构化网格
归并排序	mergeSortSharedKernel	CUDA 工具包	回溯与分支定界
归并排序	generateSampleRanks	CUDA 工具包	回溯与分支定界
归并排序	mergeRanksAndIndices	CUDA 工具包	回溯与分支定界
归并排序	合并基本区间	CUDA 工具包	回溯与分支定界
Needleman-Wunsch算法	针_cuda_共享_1	Rodinia 基准测试套件	动态规划
Scan	共享内存独占扫描	CUDA 工具包	稀疏线性代数
Sobol Qring	sobol图形处理器_内核	CUDA 工具包	蒙特卡洛
计算流体力学求解器	计算_步骤_因子	Rodinia 基准测试套件	非结构化网格
计算流体力学求解器	CUDA_计算_通量	Rodinia 基准测试套件	非结构化 Goverheadrid
计算流体力学求解器	初始化 _变量	Rodinia 基准测试套件	非结构化网格
计算流体力学求解器	cuda_时间_步	Rodinia 基准测试套件	非结构化网格
时钟	timedReduction	CUDA 工具包	非结构化网格
BlackScholes	BlackScholesBodyGPU	CUDA 工具包	稠密线性代数
LU 分解	lud_对角线	Rodinia 基准测试套件	稠密线性代数
LU 分解	lud_周长	Rodinia 基准测试套件	稠密线性代数
LU 分解	lud_内部	Rodinia 基准测试套件	稠密线性代数
K 均值	KmeansInvert	Rodinia 基准测试套件	稠密线性代数
K 均值	K 均值点	Rodinia 基准测试套件	稠密线性代数
矩阵乘法	矩阵乘法CUDA	CUDA 工具包	稠密线性代数
高斯	Fan1	Rodinia 基准测试套件	稠密线性代数
高斯	Fan2	Rodinia 基准测试套件	稠密线性代数
转置	朴素转置	CUDA 工具包	稠密线性代数
转置	粗粒度转置	CUDA 工具包	稠密线性代数
转置	对角线	CUDA 工具包	稠密线性代数
转置	无存储体冲突	CUDA 工具包	稠密线性代数
转置	细粒度	CUDA 工具包	稠密线性代数
转置	粗粒度	CUDA 工具包	稠密线性代数
转置	copy	CUDA 工具包	稠密线性代数
转置	复制共享内存	CUDA 工具包	稠密线性代数
向量加法	向量加法	CUDA 工具包	稠密线性代数
立体视差	立体视差	CUDA 工具包	稠密线性代数
Saxpy	Saxpy	CUDA 工具包	稠密线性代数
快速沃尔什变换	fwtBatch1	CUDA 工具包	稠密线性代数
快速沃尔什变换	fwtBatch2	CUDA 工具包	稠密线性代数

4.4.9 Kepler 架构的结果

首先，我们在基于 Kepler 架构的 Tesla K20 GPU 上测试了模型，该 GPU 包含 13 个流式多处理器。每个流式多处理器包含 192 个单精度 CUDA 核心、64 个双精度单元、32 个特殊函数单元、32 个加载存储单元和 4 个线程束调度器[127]。为了评估预测方法的准确率，我们采用了平均百分比误差 (MPE) 和平均绝对百分比误差 (MAPE)。

Tesla K20 GPU 的记录 MPE 和 MAPE 分别为 −8.88% 和 28.3%。接下来，我们在同样采用 Kepler 架构的 Quadro K4200 GPU 上测试了该模型。为测试该模型，我们针对 Quadro K4200 的硬件特征对其进行了调优。该设备的 MPE 和 MAPE 值分别为 −5.66% 和 29.4%。图 4.8 展示了两款 Kepler GPU 上若干常用基准测试的预测绝对百分比误差。

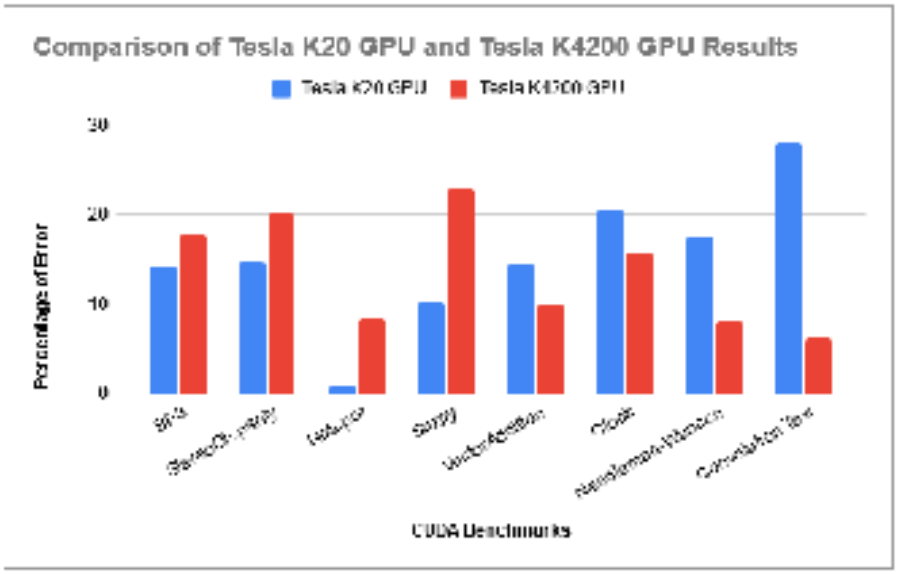


图 4.8：Kepler 架构的两款 GPU 的误差

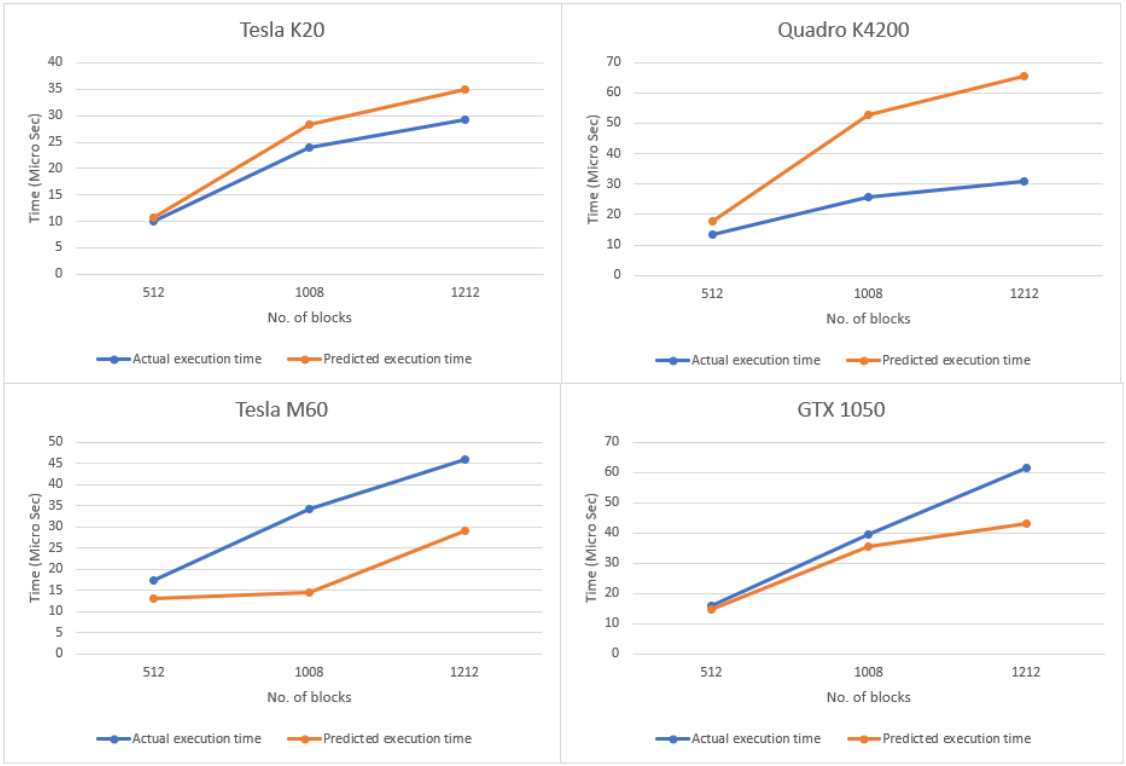
表 4.8：实际执行时间与预测执行时间（μ秒）

基准测试	内核	GPU架构							
		Tesla K20		Quadro K4200		Tesla M60		GTX 1050	
		实际的	预测值	实际值		预测值	实际值	预测值	
计算流体力学求解器	步长因子	55.39	71.28	55.84	80.95	41.09	40.54	61.10	60.89
计算流体力学求解器	Flux	820.19	1,519.40	1,050.90	1,638.00	338.30	1,519.40	631.60	1,130.14
计算流体力学求解器	InIt	29.15	34.86	30.84	65.61	45.76	29.07	61.50	35.52
计算流体力学求解器	时间步	47.17	49.03	47.07	103.40	47.49	25.59	62.80	69.93
时钟	timedReduction	5.10	4.05	8.02	6.75	4.15	4.05	6.17	5.48
可分离卷积	convolutionRowsKernel	1,665.50	2,801.80	1,751.90	1,937.43	1,765.77	2,801.80	2,890.90	2,959.94
可分离卷积	列	1,165.50	985.57	1,751.90	1,205.50	2,483.50	2,057.84	1,651.08	1,284.50
卷积纹理	convolutionRowsKernel	620.85	794.89	1,271.50	1,352.65	698.57	794.89	1,596.70	1,253.06
卷积纹理	列卷积核	681.04	794.89	1,286.80	1,352.65	691.53	794.89	1,594.40	1,253.06
广度优先搜索	Kernel1	14.98	17.14	20.87	17.14	14.98	10.37	16.98	20.73
广度优先搜索	Kernel2	15.14	17.46	11.87	19.23	11.01	17.14	12.31	20.73
BlackScholes	BlackScholesBodyGPU	547.12	819.18	631.51	819.18	660.25	819.18	813.50	970.27
快速沃尔什变换	fwtBatch1	459.66	390.25	486.91	520.62	571.66	414.85	684.03	720.84
快速沃尔什变换	fwtBatch2	879.38	770.58	1,367.70	1,280.54	569.74	410.80	1,389.20	1,148.25
高斯	Fan1	2.14	1.91	4.54	2.07	5.65	2.91	2.82	1.44
高斯	Fan2	326.63	147.93	533.50	251.47	374.26	147.93	549.41	189.12
Hotspot	calculate_temp	6.98	7.05	5.99	6.50	6.24	5.75	5.02	4.80
K 均值	KmeansInvert	7,566.60	6,565.74	12,934.00	11,194.97	7,483.20	5,065.74	9,354.88	9,190.17
K 均值	KmeansPoint	2,909.70	2,515.04	3,024.20	2,861.97	2,994.30	2,065.74	3,791.30	3,655.93
LU 分解	lud_对角线	16.29	14.04	20.06	11.17	16.13	15.30	20.26	16.44
LU 分解	lud_周长	15.95	20.24	12.48	18.55	5.45	20.24	2.75	10.09
LU 分解	lud 内部_	93.98	81.59	110.86	70.86	4.45	2.03	1.47	0.89
矩阵乘法	矩阵乘法CUDA	6,997.75	5,825.05	8,600.28	5,583.24	4,917.40	5,825.05	8,514.90	9,099.98
归并排序	mergeSortSharedKernel	2,184.26	3,102.00	5,683.00	5,014.00	1,946.60	2,645.00	4,459.00	5,728.00
归并排序	generateSampleRanks	183.33	150.60	257.88	234.86	146.78	118.28	122.30	92.50
归并排序	mergeRanksAndIndices	8.38	7.17	24.42	21.58	5.34	3.74	7.55	7.14
归并排序	合并基本区间	1,114.10	910.58	1,723.60	1,526.89	589.74	328.60	1,412.20	1,283.60
Needleman-Wunsch算法	针 _cuda_共享 _1	18.75	15.45	15.49	14.24	13.63	15.45	10.59	7.82
Needleman-Wunsch算法	针 _cuda_共享 _2	22.24	16.99	15.52	15.63	13.51	16.99	7.39	8.55
Saxpy	Saxpy	79.18	71.09	160.11	123.51	78.30	71.09	93.23	77.55
Scan	共享内存独占扫描	361.83	678.01	501.37	887.07	425.17	678.01	894.11	622.25
Sobol Qring	sobolGPU 内核_	110.66	115.24	104.80	120.76	119.33	115.24	150.80	151.54
Srad	srad_cuda_1	11.26	14.84	18.56	26.04	8.42	14.84	14.53	19.20
Srad	srad_cuda_2	13.46	11.56	20.66	19.56	8.26	6.56	7.74	6.86
立体视差	立体视差	379.60	322.96	286.10	343.86	363.46	322.96	295.36	302.09
转置	transposeNaive	100.83	182.47	429.98	261.22	146.52	182.47	374.05	150.47
转置	合并访问转置	78.05	234.03	160.10	319.02	166.94	573.03	124.56	213.61
转置	对角线	83.17	69.75	162.21	144.69	93.23	74.75	128.64	122.18
转置	无存储体冲突	77.35	64.32	158.81	131.41	72.72	54.32	107.07	113.57
转置	细粒度	73.41	63.23	132.86	138.76	121.20	83.23	92.81	156.19
转置	粗粒度	74.88	58.65	147.90	119.13	301.14	70.65	476.12	450.06
转置	copy	62.15	53.03	87.10	57.89	66.94	52.28	93.02	82.56
转置	复制共享内存	68.96	79.28	385.53	401.20	274.42	119.80	423.20	451.20
向量加法	向量加法	2.12	1.81	2.20	1.98	1.82	1.49	1.65	1.39
平均百分比误差		−8.88%		−5.66%		−10.64%		−3.94%	
平均绝对百分比误差		28.3%		29.4%		47.8%		28.5%	
预测器工具的执行时间		总计：2925毫秒		平均值：66.48毫秒		最小值：1.031 毫秒		最大值：206.62 毫秒	

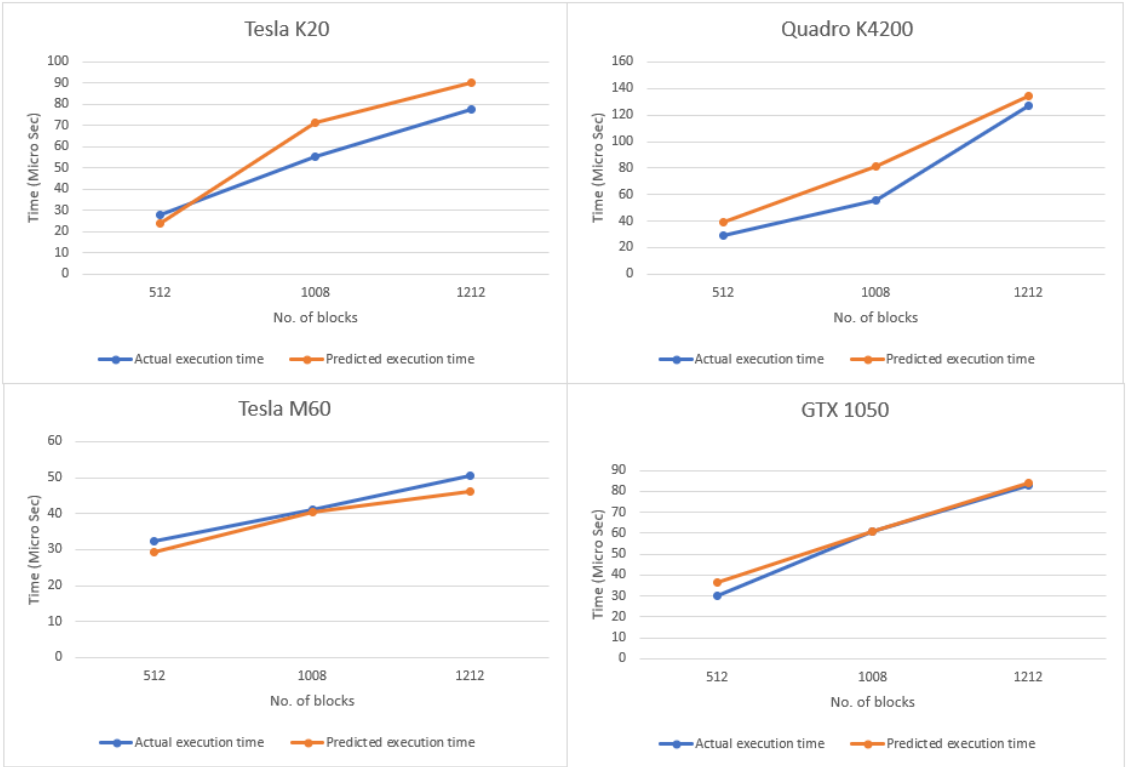
4.4.10 不同 GPU 架构上的结果

对于本研究考察的所有 GPU，我们首先使用设备查询应用程序和供应商文档收集架构细节，然后计算延迟值，最后构建前文所述的内存访问模型，从而针对每种架构调优性能预测模型。

我们已在所有 GPU 上运行表4.7中所述的基准测试，以了解我们的方法在不同架构上的泛化能力。我们对每个



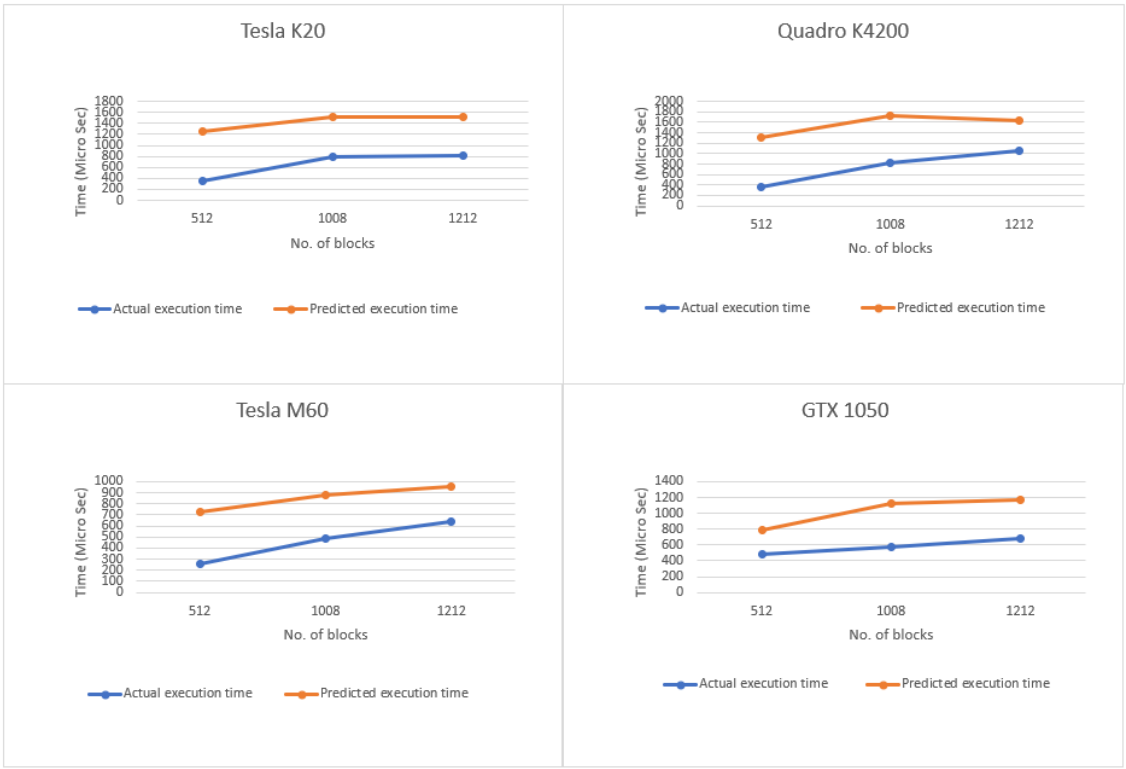
(a) initialize_variables Kernel



(b) compute_step_factor Kernel



(c) cuda_time_step Kernel



(d) cuda_compute_flux Kernel

图 4.10：不同图形处理器架构下计算流体力学求解器的实际执行时间与预测执行时间

表4.9：不同图形处理器架构下各 Dwarf 的平均绝对百分比误差

Dwarf 名称	**Tesla K20**	**Quadro K4200**	**Tesla M60**	**GTX1050**
动态规划	20.6	4.39	19.56	20.93
稠密线性代数	29.95	27.92	51.75	34.67
稀疏线性代数	87.38	76.93	59.47	30.41
回溯与分支定界	23.15	10.93	32.42	16.86
图遍历	14.87	39.64	43.23	45.24
蒙特卡洛	4.14	15.23	3.43	0.49
结构化网格	25.05	15.34	29.88	16.49
非结构化网格	31.61	69.82	87.09	28.81

通过改变网格和块大小，为多个启动配置运行内核。对于每种配置，我们尽可能改变数据集大小。对于某些基准测试，预测准确率较高（误差<10%）；而对于其他一些基准测试，则存在预测值过高或过低的情况。例如，在转置基准测试中，转置合并内存访问内核在所有架构中出现了最大程度的高估。经过进一步调查，我们发现，存储体冲突、内存访问合并和缓存是导致高估的主要原因。这两款 GPU 的 MPE 和 MAPE 值已列于表 4.8 中。

为了突出我们的估算模型的各个方面，我们选取了 Rodinia 测试套件中的计算流体力学（CFD）求解器基准测试，并将结果绘制在图 4.10 中。计算流体力学是一种相当复杂且广泛应用的算法，常见于工程和科研领域。该基准测试包含四个内核。我们在包含 97K 个数据点的问题规模下执行了这些内核。

图 4.10 展示了全部四种 GPU 架构的结果。图表显示，我们的模型能够以较高的准确率预测三个内核（初始化、compute_step_factor 和 time_step）的执行时间。我们针对三种不同的网格大小配置（506 个线程块、1008 个线程块和 1212 个线程块）运行了计算流体力学基准测试。结果表明，我们的预测模型具有可扩展性，因为它在这三种配置下都表现良好。

表4.10：吞吐量比较：我们的方法与 Volkov 的方法[20]

基准测试详情		Tesla K20（算术吞吐量值）			Tesla M60（算术吞吐量值）		
基准测试	内核	Volkov[85]模型	提议模型	实际	Volkov[85]模型	提议模型	实际
向量加法	向量加法	6.481	6.4107	5.4733	5.5907	7.7875	6.3755
矩阵乘法	矩阵乘法CUDA	39.0696	1.5933	1.3263	23.7688	0.4621	0.5474
Hotspot	计算_温度	5.2443	7.0464	7.1171	4.682	7.8197	7.2056
高斯	Fan1	1.2217	2.7528	2.4569	1.0399	1.5576	0.8022
Saxpy	Saxpy	6.6365	36.1331	32.4413	4.0374	21.6778	19.6817
归并排序	mergeRanksAndIndices	21.6338	19.4705	16.6591	15.6288	26.7592	18.7414
转置	合并访问转置	17.8371	15.1779	45.5105	10.8516	4.5358	15.5694
转置	对角线	16.4114	47.6405	39.9534	9.9842	32.2877	25.8877
计算流体力学求解器	Flux	38.7485	3.9998	7.4097	23.5734	2.9726	13.3509
时钟	timedReduction	17.2784	20.1448	15.9973	12.0472	14.3252	13.98
BlackScholes	BlackScholesBodyGPU	37.5466	33.7858	50.5861	22.8423	25.2992	31.389
Needleman-Wunsch算法	针 _cuda_共享 _1	0.1886	0.3139	0.2587	0.1621	0.2552	0.2893

然而，compute_通量内核的预测误差更高。我们对 com- pute_通量内核进行了性能分析，以了解预测误差的原因。我们发现内核执行受到了分支发散的影响，而这点我们在当前模型中尚未考虑。

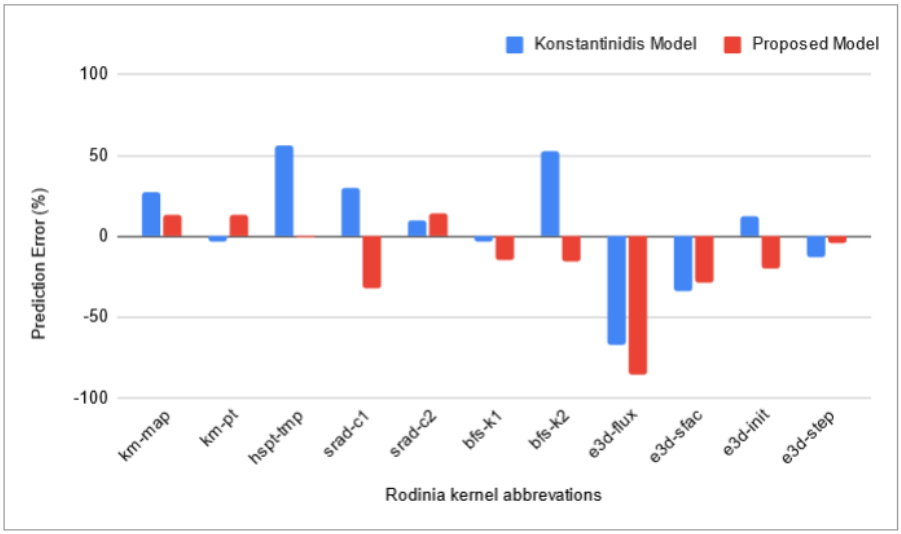


图 4.11：Konstantidis 模型与所提出模型在 TeslaK20 图形处理器上对 Rodinia 内核的预测误差

4.4.11 分析

我们在三种架构上测试了该模型：Kepler、Maxwell 和帕斯卡，详情见表4.6。各架构在流式多处理器数量、每个流式多处理器的核心数、时钟频率和缓存大小方面各不相同。本研究考虑的基准测试见

表4.7。这些基准测试被归类为并行计算侏儒，以研究特定通信和计算模式下并行算法的结果。我们在第4.4.9节中展示了属于 Kepler 架构的两款 GPU 的结果。图4.8中观察到的误差表明，对于给定架构，这些基准测试的精度尚可接受。Saxpy、时钟、向量加法和卷积文本基准测试的观测误差差异极小。这说明，尽管性能能力不同，我们的模型仍能对属于同一架构的两款 GPU 模型作出合理预测。我们从实验中得出的见解如下：

准确率： 在表4.8中，我们展示了跨GPU架构的基准测试结果。对不同架构结果的分析表明，我们的模型在所考虑的三种架构——Kepler、Maxwell和帕斯卡——上均能较好地运行。结果显示，Kepler 架构的预测准确率最高，其次是帕斯卡架构。在实验中，Maxwell 架构的预测准确率最低。

> 基于这些结果以及四种GPU架构下的误差值，我们相信，当该模型应用于未来的GPU架构时，也能够以相当的准确率进行预测。

最佳情况： 正如表 4.9 所示，我们观察到，该预测模型对于属于动态规划、结构化网格、回溯与分支定界以及稠密线性代数的基准测试效果最佳。向量加法、Saxpy 和矩阵乘法属于稠密线性代数 Dwarf，且受计算限制[14]。根据表 4.8 和表 4.9 中的结果，我们发现模型对于受计算限制的基准测试效果良好。我们还观察到，该预测模型对于执行时间较短的基准测试（例如向量加法和 Needleman-Wunsch算法）以及基准测试效果良好。

这些应用的执行时间较长，例如矩阵乘法和 Black-Scholes。因此，我们的模型的准确率并不严重依赖于执行时间。

对结构化网格和动态规划基准测试进行性能分析后，我们观察到，全局内存带宽是这些基准测试的主要瓶颈。看来，我们与全局内存访问相关的经验模型能够有效预测这些基准测试中内存受限指令的延迟值。

蒙特卡洛基准测试既非计算密集型，也非内存密集型。正如 [1] 中所讨论的，其执行特征高度依赖问题的性质。尽管我们的模型能够有效预测蒙特卡洛基准测试的表现，但我们不能声称这一预测方法普遍适用于此类基准测试。

最差情况：如表 4.9 所示，稀疏线性代数小类的结果最差，其次是非结构化网格。稀疏线性代数小类包含间接且不规则的内存访问模式 [130], ，因此预测其执行时间是一项复杂的工作。根据 Asanovic 等人 [1] 的报告（该报告提出了并行计算侏儒），非结构化网格可以宽泛地视为稀疏线性代数问题。因此，预测稀疏线性代数时面临的挑战同样适用于非结构化网格。

结果表明，在所有架构上，高斯基准测试的 *Fan1* 内核的性能都被低估了。Fan1 中有若干 `if`语句，可能导致分支发散。我们还观察到，由于数据请求，该内核有 16.34% 的停顿。目前，我们的预测模型还无法模拟这种停顿现象。在大多数情况下，转置基准测试的 粗粒度转置 内核的性能被高估了。Greg 等人在一项研究中 [131], 指出，该内核的性能瓶颈在于将数据写入全局内存中转置后的位置

。我们发现，该内核的内存访问模式并非合并访问，这一点尚未纳入我们的模型。

带宽预测：我们还观察到，对于该内核，粗粒度转置的有效带宽（GB/s）非常低。这似乎表明，我们的模型无法纳入有效带宽较低所造成的影响。向量加法、Saxpy 和矩阵乘法的带宽利用率都很高，因此预测模型对这些操作的预测效果良好。

与其他方法的比较Lemeire 等人 [89] 提出了一组基于微基准测试应用程序的方程，用于计算发射和完成延迟（记作 Δ），这与我们的方法不同。为了进行比较，我们将 Kepler 架构上算术指令的 Δ值（9 个周期）与我们的结果进行了比较。我们发现，我们的结果（见表 3.2）确实与 Lemeire 等人的结果一致。这两种方法[85][89] 都尚未使用真实应用程序进行充分测试。

Volkov 等人 [85] 提出了一种解析模型，用于预测算术吞吐量和内存吞吐量，而我们的模型预测端到端执行时间。我们提出的模型也展现了内存吞吐量的指数特性。为了进行比较，我们使用自己的预测模型生成算术吞吐量。我们将我们的方法和 Volkov 的 [85]] 模型应用于本研究中介绍的一组内核，并计算算术吞吐量值。表4.10对比了算术吞吐量与 Volkov[85] 模型的结果，表明使用我们的方法计算出的吞吐量接近实际吞吐量。我们选取了 Kepler 和 Maxwell 架构，因为 Volkov 的研究[85] 也涵盖了这两种架构。

与 Konstantinos 等人[87] 临时提出的屋顶线模型不同，我们的方法无需任何运行时数据即可预测 CUDA 内核的执行时间

表4.11：与现有方法的比较

参考文献	方法类型	分析类型	数据集大小	准确率
Resios 等人 [59]	参数化模型	静态	2	精度：转置（1.64）， 二维卷积滤波器（1.89）
Kothapalli 等人 [57]	解析模型	静态	3	矩阵乘法（31.25%）， 列表排序（12.50%）
Hong 等人 [29]	解析模型	静态	6	平均绝对百分比误差：13.3%
Elias 等人 [87]	使用运行时数据的屋顶线模型	运行时	30	平均绝对百分比误差：27.66%
Amaris 等人 [75]	基于直观 BSP 的模型	静态	2	在 0.8 至 1.2 倍范围内 实际执行时间
Luo 等人 [60]	解析模型	运行时	6	平均准确率为 90%
Amaris 等人 [75]	机器学习模型	运行时	2	在 0.5 到 1.5 倍之间 实际执行时间
我们的方法	解析模型	静态	45	平均绝对百分比误差：28.3%，平均百分比 误差：-8.88%

该内核的执行时间；我们认为，使用内核的运行时数据来预测其执行时间，在很大程度上违背了这一做法的初衷。图 4.11 展示了 Konstantinidis [87]模型和我们的模型共有的基准测试的预测误差（非绝对误差）。比较 Konstantinidis 的方法[87]与我们的方法时，他们报告的平均绝对误差为 27.66%，而我们的模型在他们同样使用的 Tesla K20 GPU 上报告的平均绝对百分比误差为 28.3%。因此，与 Konstantinidis 等人基于运行时数据得出的结果相比，我们基于静态数据的预测并未相差太远。表 4.11 对我们的结果与现有方法进行了比较研究。如表所示，考虑到用于测试的数据集规模以及基准测试特征的差异，我们的模型取得了不错的结果。

4.5 讨论

本章介绍、评估并分析了两种性能预测模型。第一个模型基于参数化方程，利用指令级并行和线程级并行对并行性建模，以计算执行时间。为改进那些低估执行时间的基准测试，我们提出了第二种基于硬件资源分配方法的方案。我们复用了初始预处理步骤、微基准测试结果以及第一个模型中的启动开销模型，

并重新构建了 GPU 调度算法。此外，我们还引入了一种更复杂的、基于一次性动态分析的内存模型。我们在大量基准测试中对所提出的两种模型进行了测试和验证。我们证明了所提出的模型很容易针对更新的架构进行调优。性能预测是一项难题，尤其是对于并行计算机和 GPU 而言，因此，所提出模型的准确率是性能预测研究中的一项新颖研究贡献。

CHAPTER 5

功耗预测

As 如目标2所述，计算设备的功耗与其硬件特征密切相关。因此，大量研究致力于探讨硬件特征与功耗之间的关系 [36][37][38][39]。然而，我们的目标是基于代码特征和尽可能少的运行时信息来开发功耗预测模型，使其无需运行 CUDA 程序即可预测功耗。据我们所知，采用这种方法开展的研究并不多。

5.1 为什么采用机器学习方法进行功耗预测?

大多数解析功耗预测模型会利用图形处理器的运行特征及其底层电气属性，来预测图形处理器消耗的功耗。对于在构建解析模型时已考虑其运行属性的系统，这类模型表现良好。此外，我们没有发现任何

用于理解 GPU 应用程序中功耗与程序特征之间关系的算法方法。此外，无法捕捉图形处理器在指令级上的功耗。为此，我们采用基于机器学习的方法，对应用程序、架构特征与功耗之间的复杂关系进行建模。

数据驱动的功耗预测模型是一种替代方法，可用于估算图形处理器中各种运行组件所影响的功耗，而这种影响具有高度非线性。由于数据驱动方法具有固有的普适性，只要数据点充足，数据驱动模型就有望捕捉到图形处理器的不同组件协同运行以执行代码时所产生的复杂性。当然，模型必须基于规模足够大且多样性充分的数据集进行充分训练。数据驱动方法的普适性在于，模型既不需要图形处理器内部组件的任何功能属性或物理特征，也不需要图形处理器的任何细微架构细节。在我们的方法中，我们有意通过选择具有不同特征的基准测试来增加这种多样性，这或许是论文中所展示的模型具有稳健性的原因。

5.2 初步调查

在最终确定功耗预测模型的特征之前，我们进行了一项初步研究，其中包含仅通过代码静态分析得出的特征以及硬件特征。我们采用数据驱动方法，首先考虑所有我们认为可能影响功耗的特征，然后系统地选择最终的特征子集。

表 5.1：特征选择所考虑的特征

符号	描述
块大小_	每个线程块的线程数
网格_大小	块数
计算_指令_内核	内核中的计算指令数
计算_指令_模拟	模拟计算指令数
glob_inst_内核	内核中的全局内存指令数量
glob_inst_sim	模拟全局内存指令数量
inst_issue_周期	指令发射周期
misc_inst_内核	内核中的杂项指令数量
misc_inst_sim	杂项指令数量
占用率	流式多处理器上的活动线程束与最大数量之比 流式多处理器支持的活动线程束数量
内核中的共享内存指令_ _	内核中的共享内存指令数
模拟共享内存指令_ _	模拟计算指令数
总 _线程数	启动的线程总数
总计_inst	模拟的指令总数
波次	在流式多处理器上执行的线程块波次

5.2.1 考察的特征

我们考察了通过程序分析提取的、可能导致功耗增加的特征。其中大多数特征的含义不言自明，关于派生特征（如 inst_issue_cycles）的详细信息，请参阅第 5.4 节。程序特征提取算法将在第 5.4.5 节中介绍。本初步研究中考察的此类特征共有 16 个，具体列于表 5.1 并加以描述。

5.2.2 特征工程

随后，我们利用皮尔逊相关系数观察各属性之间的关系，并剔除高度相关的特征。我们已在第5.5节中详细讨论了皮尔逊相关性分析。通过图5.1所示的皮尔逊相关系数热图，我们可以观察到以下几点：

- 波次、网格_大小和总线程数等特征与

特征 inst_issue_cycles 的相关系数分别为 0.85、0.92 和 0.85。因此，我们从模型中删除这些特征，只保留 inst_issue_cycles。

- comp_inst_sim、glob_inst_sim 和 total_inst 之间的相关性极高，相关系数为 1。因此，我们的模型只考虑 glob_inst_sim，并舍弃另外两个特征。

图 5.1：皮尔逊相关系数

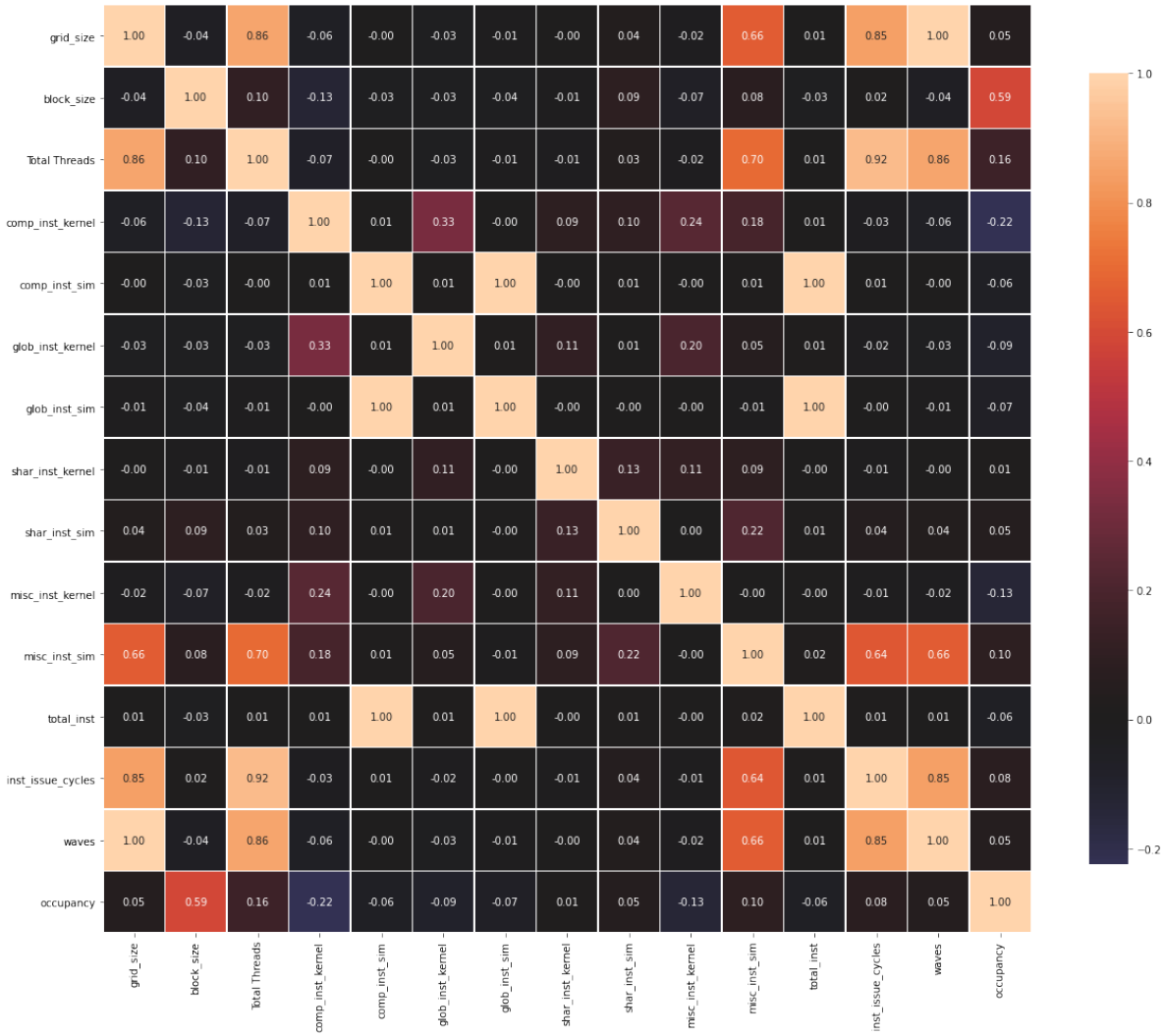


表5.2：初步案例研究考察的特征

符号	来源
块大小_	用户输入
comp_inst_内核	PFEA
glob_inst_内核	PFEA
glob_inst_Sim	PFEA
inst_issue_周期	PFEA
misc_inst_内核	PFEA
misc_inst_sim	PFEA
占用率	CUDA 占用率计算器
shar_inst_内核	PFEA
shar_inst_sim	PFEA

5.2.3 模型构建与结果

决策树 [132] 基于树形结构构建回归或分类模型，其中包含决策节点和叶节点。使用决策树的成本与用于训练模型的数据点数量呈对数关系。决策树可以为评估其他复杂树模型在功耗数据上的性能提供基本思路。我们使用 sklearn 库构建决策树模型，该模型采用经过修改的 CART （分类与回归树）算法。我们通过调整各种超参数来训练模型，例如拆分内部节点所需的最小样本数、树的最大深度，以及进行最佳拆分所需的最大特征数；此外，还采用剪枝来避免模型过拟合。

5.2.4 结果与分析

我们基于 5 折交叉验证分数的平均值，使用三项关键指标验证模型：R^2 分数、均方根误差（RMSE）和平均绝对误差（MAE）。表 5.3 汇总了所考虑的三台图形处理器机器在这些指标上的结果。

表 5.3：不同 GPU 架构的结果

架构 R^2评分 均方根误差			MAE
Tesla K20	0.8199	10.8832	5.7835
Tesla M60	0.8533	7.3676	3.3581
Tesla V100	0.8973	12.2025	5.4570

R^2 分数在所有三种架构上都相当不错，因此我们认为模型能够做出准确的预测。所有三种架构的 RMSE 值也表明模型具有稳健性和准确性。如图 5.12 所示，对于所有常见基准测试，我们预测的功耗与实测值相当接近。

通过这项初步研究，基于观察到的结果和模型分析，我们尝试回答以下研究问题（RQ）：

RQ1：能否构建一个不依赖架构的模型，以可容忍的误差和准确率预测 CUDA 程序的功耗？从实验结果可以看出，我们在三种不同的 GPU 架构上验证了模型：Kepler、Maxwell 和 Volta。三项验证指标，即 R^2 score、均方根误差和平均绝对误差的结果均表明，我们的模型不依赖架构，因为它在这三种架构上都表现良好。我们在各种 CUDA 内核基准测试上测试了该模型，这些基准测试在内存访问模式、启动配置、适用性和执行受限类型方面各不相同。由此可对 RQ1 作出肯定回答。

RQ2：本研究所考虑的 CUDA 程序中，哪些特征对图形处理器的功耗影响最大？

观察到的功耗预测模型结果表明，我们可以将 CUDA 内核消耗的功耗与程序特征关联起来。我们分析了本研究考察的哪些程序特征对功耗预测的贡献最大。如图5.2所示，基于针对 Tesla K20 使用决策树进行的特征重要性分析，我们发现，在三种架构中，指令 inst_issue_cycle 对功耗的影响最大。

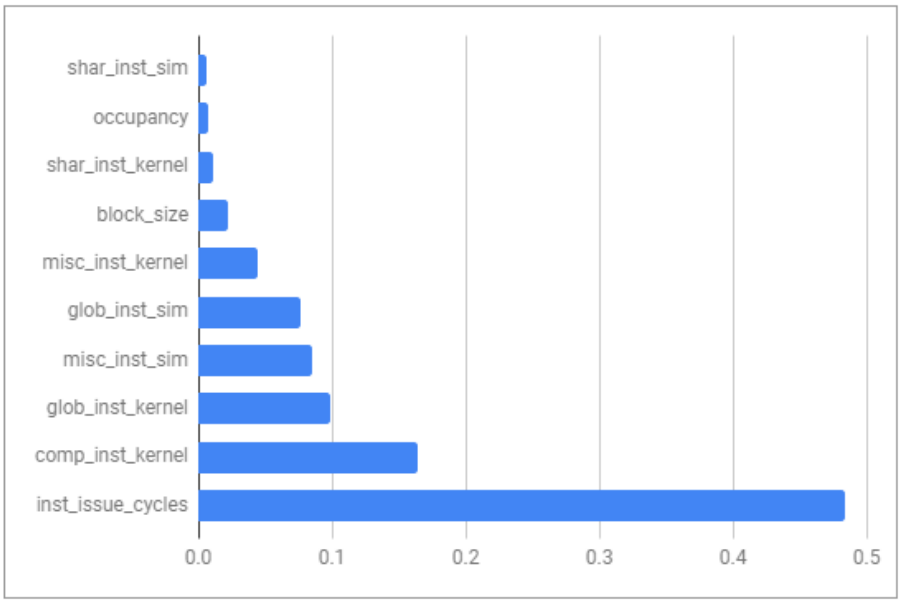


图 5.2：Tesla K20 的相对特征重要性

全局内存访问指令对功耗有显著影响。在三种架构中，指令计数（glob_inst_kernel）和模拟指令数量（glob_inst_sim）的排名都较高。这印证了 Song等人 [20]关于全局内存对功耗影响的观察。计算指令计数（comp_inst_kernel）也会影响功耗预测，其影响与杂项指令计数（misc_inst_kernel）以及模拟杂项指令的数量（misc_inst_sim）相近。某些计算指令（浮点乘加、浮点除法）的延迟远高于其他指令（加法、乘法） [133]。作为本研究的延伸，我们可以进一步探究各类指令的延迟对图形处理器功耗的影响。

如前所述，已启动线程块的数量（block_size）对预测结果的影响取决于特征。已启动线程块的数量可以表示 CUDA 内核执行期间激活的流式多处理器数量，因为每个新线程块都会映射到一个流式多处理器。然而，据我们观察，它对功耗的影响程度适中。

我们已在早期发表的论文 [134] 中介绍了这项研究。尽管观察到的研究结果尚可，但我们坚信，纳入利用架构特定细节收集的动态分析特征，能够构建出更可靠、准确的模型。此外，探索能够改善观察到的预测误差的其他机器学习技术，对于构建更准确的模型也很重要。我们还对使用运行时特征预测执行时间、功耗和能量进行了初步研究，相关内容见附录 8。

因此，为了进一步增强模型，我们提出了一项深入研究：从架构角度收集更多特征，通过一次性动态分析，借助更新的机器学习算法构建更稳健、准确的预测模型。

5.3 提出的模型

为了构建功耗预测模型，需要了解 GPU 硬件的复杂特性。此外，还必须了解 CUDA 应用程序的运行时行为。然而，NVIDIA GPU 的公开信息并未披露诸如指令延迟、缓存行为、内存访问细节、调度策略、资源分配等影响 GPU 功耗的关键硬件执行细节。

我们开发了一种基于机器学习的功耗预测模型，它结合了 i) 通过程序分析获取的 CUDA 程序特征；ii) 从供应商提供的 GPU 规格中获取的 GPU 架构特征，以及通过运行一组微基准测试从运行数据中推导出的特征。该方法的概述如图 5.3 所示。

我们将使用第4章第4.1节中描述的抽象 GPU 模型来介绍所提出的功耗预测模型。该模型中的符号用于

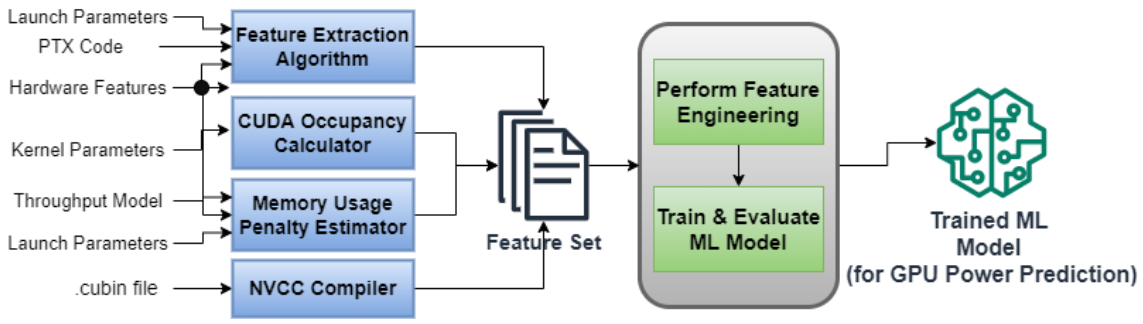


图 5.3：功耗预测建模方法概览

解释为进行预测而收集的一些特征。

我们首先详细讨论如何选择并收集特征，以构建该功耗预测模型。下一节将全面介绍从不同维度考察的特征。

5.4 特征识别与获取

通用图形处理器的功耗受多种因素复杂交互的影响：i）通用图形处理器架构特有的细节，ii）CUDA程序特征及其运行时行为，以及 iii）启动策略（由启动参数决定）。然而，关于 NVIDIA GPU 的公开信息并未披露诸如指令延迟、缓存行为、内存访问细节、调度策略、资源分配等影响图形处理器功耗的关键硬件执行细节。为解决这一问题，我们提出一种基于机器学习的功耗预测模型，该模型使用从 i）GPU 架构特征和 ii）通过程序分析获得的 CUDA程序特征中提取的特征。机器学习算法正被有效地用于预测现代架构的架构特性 [98][105]。

W我们现在将在以下小节中讨论如何推导各种特征。

5.4.1 每线程寄存器数、每线程块共享内存

每线程寄存器数和每线程块共享内存会影响流式多处理器的占用率，进而影响硬件利用率。以往的研究 [29, 135] 表明，这些属性在功耗方面发挥着至关重要的作用。因此，我们在本研究中同时考虑这两个特征。

我们在编译时使用 `nvcc --cubin` 生成了这些值，进而生成一个 .cubin文件 [1]。.cubin文件是一种采用 ELF 格式的 CUDA二进制文件，包含 CUDA可执行代码段以及符号、重定位信息、调试信息等其他详情。它包含有关代码在运行时预计使用多少寄存器和共享内存的信息。

5.4.2 占用率

占用率衡量的是在流式多处理器上实际执行的线程束数量与该流式多处理器允许的最大线程束数量之比。我们将占用率作为模型中的一项特征，以了解线程级并行（TLP）对 GPU功耗的影响。研究也已证明，GPU功耗会受到占用率的影响[20, 136]。本研究采用的占用率值是理论占用率，它体现了内核启动配置和 CUDA设备能力所限定的占用率上限。实际占用率是在代码运行时测得的，这两个值可能略有差异。然而，为了实现通过静态分析进行预测的目标，我们在本研究中采用理论占用率。

由于我们不打算执行代码，因此使用 NVIDIA 的 CUDA 占用率计算器来计算 CUDA 程序的理论占用率。CUDA 占用率计算器使用每个线程块的线程数（block_size）、每线程使用的寄存器数（reg_thread）、一个线程块使用的共享内存量

[1] https://docs.nvidia.com/cuda/cuda-binary-utilities/index.html

（shmem_block），以及所考察设备的 CUDA 计算能力。

我们使用 CUDA 占用率计算器 2。它通过输入以下四项数据，报告特定内核执行时可达到的理论最大占用率：

- 每个线程块的线程数
- 每线程使用的寄存器数
- 一个线程块使用的共享内存量
- 所考虑设备的 CUDA 计算能力

有些架构特征无需了解图形处理器的详细内部信息即可推导得出。我们将详细介绍这些特征。

5.4.3 指令发射周期

NVIDIA GPU 使用取指-译码-调度（FDS）单元向线程发射指令以供执行。自然，FDS 单元也会消耗 GPU 的能量。为了对 FDS 在功耗中的作用进行建模，我们提出了一项名为指令发射周期（$inst\_issue\_cycle$）的特征，用于描述发射的指令批次数。其计算方法是将内核发射的指令总数除以每个周期允许执行的指令数这一硬件限制。我们利用公开的硬件详情（例如每个流式多处理器的线程束调度器数量（nWS）和分派单元数量（nDU））来计算 $inst\_issue\_cycle$。分派单元负责获取给定指令、其线程束 ID 和线程掩码，并计算实际执行该指令的线程 ID。随后，nDU 会检查各功能单元的可用情况，并将每条线程指令发送到相应的功能单元。指令发射取决于启动的线程总数、指令总数以及[2]

https://docs.nvidia.com/cuda/cuda-occupancy-calculator/index.html

以及 GPU 调度器采用的线程束调度策略。我们可以分别使用用户定义的参数和特征提取算法（算法6和7）提取启动的线程总数（总数_线程）和指令总数（总数_指令）。由于很难预测线程束调度器的线程束调度和空闲周期，我们假设线程束始终处于满载状态并执行有用工作。线程束大小（线程束_大小）通常为每个线程束32个线程，只需查询一次图形处理器即可获取。我们使用以下公式计算指令_发射_周期：

$$inst\_issue\_cycle = \frac{total\_threads}{nWS \cdot warp\_size} \cdot \frac{total\_inst}{nDU} \tag{5.1}$$

总计_线程数 $nWS \cdot$ 线程束 p_大小 计算 GPU 上调度的线程束批次数。我们将此值乘以调度的指令批次数 总计¯指令 nDU 。由于 GPU 采用流水线架构，我们假设每个指令批次在一个周期内发出。

5.4.4 内存访问开销

Nagasaka等人 [98]和Shuaiwen [20]的研究表明，GPU内存利用率是功耗的重要影响因素。我们将内存访问开销（最终会影响功耗）建模为惩罚项。为了计算这些惩罚值，我们需要 $\mathcal{TP}(gm)$ 和 $\mathcal{TP}(sm)$ 的吞吐量值。我们利用微基准测试数据，为这两种吞吐量建立了经验模型。有关模型建立与验证的详情，请参见第4章。

5.4.4.1 缓存惩罚

全局内存访问始终会经过 L2缓存，并由所有流式多处理器共享。如果发生缓存未命中，指令执行就会产生显著延迟。缓存惩罚描述了缓存未命中发生后的延迟。通过程序分析对准确的缓存

行为进行建模是不可能完成的任务，因为它取决于多个运行时因素。我们通过利用可用的架构和程序详细信息计算 L2 缓存未命中惩罚，来近似描述图形处理器缓存对全局内存数据的行为。L2_sz 描述了通过设备查询获取的 L2 缓存大小。我们使用特征提取算法 9 和 10 中计算得到的波次以及 全局_指令_流式多处理器。利用全局内存延迟（L(gm)），我们计算缓存惩罚如下：

$$cache\_penalty = \frac{total\_threads \cdot glob\_inst\_sm}{\frac{waves \cdot L2\_sz}{access_{sz}}} \times \mathscr{L}(gm) \tag{5.2}$$

5.4.4.2 全局内存惩罚

由于全局内存使用在功耗中起着至关重要的作用，我们将全局内存惩罚作为一个特征，用于定量确定全局内存使用量。为此，我们首先使用每个 SM 的加载存储单元数量（R[LSU] ）来统计内存访问次数，即 $_{总计-线程}R[LSU]$ 。然后，我们计算每次此类访问所需的时间（ $\frac{access\_sz}{\mathscr{TP}}(gm)$ ）。将访问次数乘以每次访问所需的时间，即可得到一条全局内存指令的惩罚。随后，将该惩罚乘以每个 SM 的全局内存指令数（$glob\_inst\_sm$），即可通过以下公式计算全局内存惩罚：

$$glb\_penalty = \frac{total\_threads}{R[LSU]} \times \frac{access\_sz}{\mathscr{TP}(gm)} \times glob\_inst\_sm \tag{5.3}$$

5.4.4.3 共享内存惩罚

共享内存指令也会显著影响图形处理器的功耗，尤其是在内核执行过程中出现大量存储体冲突时[20]。因此，我们也将共享内存惩罚特征纳入考量，以模拟其对功耗的影响。接着，我们通过统计流式多处理器上共享内存指令的出现次数，计算每条共享内存指令的惩罚值（ $\frac{total\_threads}{R_{[加载存储单元]\cdot 每个流式多处理器}}$ ）。这里，

nSM 表示 GPU架构中的流式多处理器数量，可通过设备查询得出。然后，我们将单条指令的惩罚值乘以每条共享内存指令所需的时间（$\frac{访问大小}{TP^{sz}(共享内存)}$），其中 访问–大小 表示每次共享内存访问的字节数。在最后计算共享内存惩罚的步骤中，我们将特征提取算法生成的每个流式多处理器上的共享内存指令数相乘。~~共享内存_惩罚~~ 计算如下：

$$sm\_penalty = \frac{total\_threads}{R[LSU] \cdot nSM} \times \frac{access\_sz}{\mathcal{TP}(sm)} \times shar\_inst\_sm \tag{5.4}$$

5.4.5 程序特征提取算法

算法9和10所述的程序特征提取算法（PFEA），通过对CUDA代码进行静态分析来模拟图形处理器的执行行为，并为数据集生成额外特征。根据启动配置确定内核代码的运行次数后，计算执行的指令数、指令总延迟及平均延迟。为了提取这些特征，我们修改了第4章中介绍的图形处理器调度算法6、7和8。这两种算法采用相同的图形处理器调度方法。我们使用变量存储指令详情，并将其作为功耗建模的特征。

该算法的工作流程如下。用户将 CUDA 内核 PTX 以及启动参数 线程块_大小 (nT)、网格_大小(nB) 和循环迭代次数 (nLoop) 作为输入提供给特征提取算法 10。根据每个流式多处理器上可驻留的最大线程数 (nTh_m) 以及每个流式多处理器上调度的线程总数 ($nTh\_调度$)，在第 6 行计算每波线程数 ($nTh\_波$)。迭代持续进行，直到 $nTh\_调度$ 为零，从而确保所有线程都已调度并完成执行。CUDA 内核还会进一步划分为控制流图 (CFG)，这是对内核应用进行建模的常用方法[117]。在第 7 行，我们使用 GenerateCFG 对 CFG 执行拓扑排序，以获取排序后的基本

块 (BB')。对于基本块 BB' 中的每个图，我们调用 CollectBBFeatures函数，以提取每个基本块的特征。

在算法 9 的 CollectBBFeatures函数中，对于控制流图（G$_c$f g）代码中的每个基本块（BB），每条指令都按拓扑排序顺序表示为一个顶点。基本块中的每个顶点 v 表示一条指令。算法 9 分别使用 $\mathscr{I}(\mathscr{T}$(v)）（第 6 行）和 $\mathscr{L}(\mathscr{T}$(v)）（第 7 行）计算各指令类型的数量及其延迟，并使用$\mathscr{N}(\mathscr{T}$(v)）（第 15 行）计算全局加载、全局存储指令和分支指令的数量。我们将全局加载和全局存储指令分开统计，因为许多研究人员指出，全局内存指令会显著影响功耗 [98][20]。利用算法统计的已执行指令数量（$\mathscr{I}(\mathscr{T}$(v)).count）和指令总延迟（$\mathscr{L}(\mathscr{T}$(v)).total），我们计算 avg_comp_lat、avg_misc_lat、avg_shar_lat 和 avg_glob_lat。第 18 行中，我们生成包含特征计数 c$_{bb}$ 的数组，并返回算法 10。在算法 10 中，如果存在循环，则将基本块的特征计数乘以循环迭代次数。随后，我们将计算出的控制流图特征加到总特征（f_kernel）中并返回；该总特征包含 CUDA 内核生命周期内收集的全部 n 个特征。

算法 9 收集基本块的特征

1: **过程** COLLECTBBFEATURES($\mathscr{GPU}$, $G = \langle V,E\rangle$, nTW)
2: $c_{bb} = [n-1]$, f=0;
3: V'=拓扑排序(G)
4: **针对** $v \in V'$ **do**
5: 读取($\mathscr{T}(v)$)
6: $\mathscr{I}(\mathscr{T}(v))$.count++;
7: $\mathscr{L}(\mathscr{T}(v))$.总计=$\mathscr{L}(\mathscr{T}(v))$.总计+$\mathscr{L}(\mathscr{T}(v))$;
8: 读取($\mathscr{N}(v)$)
9: $\mathscr{N}(\mathscr{T}(v))$.count++;
10: **对**每个 $\mathscr{T}$ **执行**
11: $c_{bb}[f] = \mathscr{I}(\mathscr{T})$.计数; $f++$;
12: $c_{bb}[f] = \mathscr{L}(\mathscr{T})$.总计; $f++$;
13: **if** $\mathscr{N}\epsilon$(分支，全局加载，全局存储) **then**
14: $c_{bb}[f] = \mathscr{N}(\mathscr{T})$.计数; $f++$;
15: **返回** c_{bb}; ▷ 基本块的特征计数

算法10 程序特征提取算法（PFEA）

过程 特征提取($\mathscr{GPU}$, G_{cfg} = 〈$\mathbb{C}$,$\mathbb{E}$〉, nB, nT, $nLoop$)

$f_{nTh\_调度器}$ = [n]; ▷ 大小为 n 的内核特征数组

$nTh\_调度器$ = $\lceil \frac{nB\ n流}{式多处理器} * nT \rceil$;

波次=0;

当 $nTh\_schd \geq 0$ **执行**

$nTh\_波 = nTh\_schd > nTh_m ?\ nTh_m :\ nTh\_schd$;

BB'=生成控制流图($\mathscr{GPU}$, G_{cfg}, $nTh\_波$);

对于 $G \in BB'$，**执行**

c_{BB} =收集基本块特征($\mathscr{GPU}$,G, nTW);

如果G 存在回边**则**

对于 $f = 0$ 到 $n-1$ **执行**

$c_{BB}[f] = c_{BB}[f] * nLoop$

for $f = 0$ to $n-1$ **do**

$f_{kernel}[f] = f_{kernel}[f] + c_{BB}[f]; f++$;

$nTh\_schd = nTh\_schd - nTh_m$;

波次=波次+1;

$f_{kernel}[n-1]$=波次;

return (f_{kernel}); ▷ 为内核收集的特征数组

表5.4 展示了使用该算法收集的所有特征。

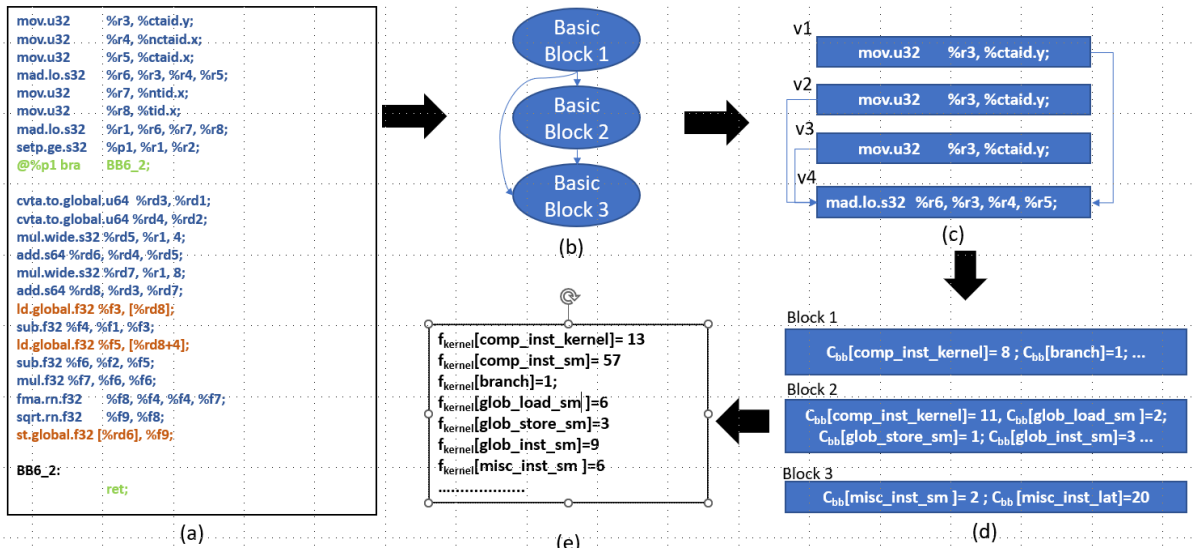


图 5.4：程序特征提取算法示例

表5.4：特征选择所考察的特征。高亮行表示通过数据驱动方法（第5.5节）最终选定的特征。

Name	描述	来源
平均_计算_延迟	平均计算指令延迟	PFEA
avg_glob_lat	平均全局内存指令延迟	PFEA
avg_杂项_lat	平均杂项指令延迟	PFEA
平均_共享_延迟	平均共享内存指令延迟	PFEA
分支	分支指令数	PFEA
计算_指令_内核	内核中的计算指令数	PFEA
计算_指令_流式多处理器	每个流式多处理器的计算指令数	PFEA
计算_延迟_流式多处理器	每个流式多处理器的计算指令总延迟	PFEA
全局_指令_内核	内核中的全局内存指令数	PFEA
全局_指令_流式多处理器	每个流式多处理器的全局内存指令数	PFEA
全局_延迟_流式多处理器	每个流式多处理器的全局内存指令总延迟	PFEA
全局_加载_流式多处理器	每个流式多处理器的全局加载指令数	PFEA
全局_存储_流式多处理器	每个流式多处理器的全局存储指令数	PFEA
misc_inst_内核	内核中的杂项指令数	PFEA
杂项指令流式多处理器_ _	每个流式多处理器的杂项指令数	PFEA
杂项延迟流式多处理器_ _	每个流式多处理器的杂项指令总延迟	PFEA
内核共享内存指令数_ _	内核中的共享内存指令数	PFEA
每个流式多处理器的共享内存指令数_ _	每个流式多处理器的共享内存指令数	PFEA
每个流式多处理器的共享内存指令延迟_ _	每个流式多处理器的共享内存指令总延迟	PFEA
sm_活跃	已激活的流式多处理器数量	PFEA
n_线程束	每个流式多处理器的线程束数	PFEA
波次	在流式多处理器上执行的线程块波次数量	PFEA
总线程数_	启动的线程总数	网格_大小 × 线程块_大小
inst_issue_周期	指令发射周期	公式 5.1
缓存_惩罚	缓存行为导致的延迟	公式 5.2
全局内存_惩罚	全局内存指令导致的延迟	公式 5.3
共享_开销	共享内存指令导致的时间延迟	公式 5.4
占用率	SM 上活动线程束数与最大活动线程束数之比 SM 支持的活动线程束数	CUDA 占用率计算器
reg_线程	每线程寄存器数	Cubin 文件
共享内存_线程块	每线程块共享内存	Cubin 文件
线程块_大小	每个线程块的线程数	由用户提供
网格_大小	块数	由用户提供

5.4.5.1 PFEA 示例说明

我们通过图 5.4 所示的神经网络（神经网络 内核）示例，说明程序特征提取算法如何提取特征。此处以配备 13 个流式多处理器的 Tesla K20 GPU 为例。为了分析 CUDA 应用程序（例如神经网络 内核），我们将其 PTX 代码作为输入（图 5.4(a)）。如算法7第 7 行所示，这些代码被转换为图 5.4(b) 中由基本块构成的控制流图（CFG，G_{cfg}）

控制流图（CFG）常用于对内核应用程序 [117] 进行建模。每个基本块的顶点表示一条 PTX 指令，两个顶点之间的边表示它们之间的数据依赖，如图 5.4(c) 所示。

算法6 CollectBBFeatures函数首先对基本块的顶点进行拓扑排序，以遵循数据依赖关系。算法6分别使用 $\mathscr{I}(\mathscr{T}(v))$（第6行）和 $\mathscr{L}(\mathscr{T}(v))$（第7行）计算每种指令类型的数量及其延迟，并使用 $\mathscr{N}(\mathscr{T}(v))$（第15行）计算全局加载指令、全局存储指令和分支指令的数量。我们将全局加载指令和全局存储指令分别纳入统计，因为许多研究人员指出，全局内存指令会显著影响功耗 [98][20]。

算法7使用特征数组（f_kernel）存储待分析 CUDA 内核的特征。用户将 CUDA 内核 PTX 以及启动参数线程块_大小 (nT)、网格_大小 (nB)和循环迭代次数（nLoop）作为特征提取算法7的输入。假设块数（nB）为78，每个线程块的线程数（nT）为1024。在算法7第3行，我们计算每个 SM 上要调度的线程数，即$nTh_schd = nt \cdot nB\ SM = 6144$。然后，我们根据每个 SM 的最大线程数限制（Tesla K20 为2048个线程）计算每波线程数（nTh_wave）。在此情况下，$nTh\_wave = 2048$（算法7第6行）。

该算法迭代至$nTh_schd = 0$，从而确保所有线程都已调度并完成执行。在此示例中，算法迭代三次（$\lceil nTh\_schd\ nTH\_wave \rceil$），以覆盖所有线程。在每次迭代中，都会针对每个基本块反复调用算法6，以生成基本块特征。如果基本块具有回边，即表示它是一个循环，则将该基本~~块的特征~~计数乘以循环迭代次数$nLoop$（由用户提供），如算法7第12行所示。

在所示示例中，不存在循环。在图5.4(e)所示的最后一步中，我们将每个波次期间收集的特征相加，并计算 CUDA 内核的总特征数。由于根据启动参数共有三个波次，f_{kernel}[com p_inst_sm]表示内核中计算指令的数量（f_{kernel}[com p_inst_kernel]）加总三次（每个波次一次），因此等于57。同样，算法6中的 $\mathscr{L}(\mathscr{T}(v))$表示每种指令类型的延迟（已通过微基准测试针对给定架构计算得出）。该算法持续累加每条指令的延迟，并将其存储在（f_{kernel}[total_com p_latenc y]）中。随后计算平均计算指令延迟（（f_{kernel}[av g_comp_lat]）），其值为f_{kernel}[total_comp_latenc y]/ f_{kernel}[com p_inst_kernel]。

5.4.6 数据集创建

我们从三个热门基准测试套件创建了数据集：Rodinia、 [123],CUDA 软件开发套件 3 和 Tango GPU [137]。我们针对三种 GPU 架构创建了该数据集，分别是 Tesla K20、Tesla M60 和 TeslaV100。三种 GPU 的硬件规格均列于补充文件的第 IV 节。

为创建数据集，我们使用不同的启动配置（网格大小、块大小）运行这些基准测试，并使用 NVIDIA 管理库（NVML） [4] 和统一功耗分析应用程序编程接口（UPPAPI） [138]测量各基准测试的功耗。NVML 是一种基于 C 的编程接口，用于监控和管理 NVIDIA Quadro 和 Tesla 系列 GPU 中的各种硬件状态。NVML 和 UPPAPI 库都使用片上传感器测量功耗。NVML 是 NVIDIA 支持的 nvidia-smi 工具所依赖的底层库，该工具具备线程安全性，支持多个线程同时调用 NVML。你可以在调用 NVML 和 UPPAPI 的函数中调用 CUDA 内核。当内核代码

[3]https://docs.nvidia.com/cuda/cuda-samples/index.html[4]
https://docs.nvidia.com/pdf/NVML_API_参考文献_Guide.pdf

执行时，所消耗的图形处理器功耗会存储在文本文件中。这些库使用片上传感器测量功耗。采集到的功耗值会存储在文件中，并将平均值记录为该内核的测量功耗。表 5.5 展示了一些常见基准测试所采集数据的标准差。

UPPAPI 库[138] 用于计算应用程序在运行时的实际功耗。NVIDIA 提供了命令行工具 nvidia-smi，用于监控图形处理器功耗。不过，要测量应用程序的功耗，我们需要在应用程序本身中插入探针；因此我们使用了 UPPAPI。我们使用 UPPAPI 测得的实际功耗来评估预测模型。UPPAI 库可以计算图形处理器和主机 CPU 的功耗。简而言之，测量方法遵循 [138] 中给出的以下方程：

$$P_{act}(t_i) = Pm(t_i) + C \times \frac{Pm(t_{i+1}) \times Pm(t_{i-1})}{t_{i+1} - t_{i-1}} \tag{5.5}$$

其中，t_i 表示当前时间。t_{i-1} 和t_{i+1} 分别表示前一个和后一个时间点。P 表示功耗读数，其中P_m 是从传感器探针采集的功耗，P_{act} 则是经校正计算得出的功耗。

该方程通过加入电容分量提高了传感器数据的准确率。该分量的计算方法是将一个常数 C （经实验得出为 0.84 ）与所考察点的斜率相乘。文献 [138] 中介绍了对该方法的实证评估。

5.4.7 数据集中存在噪声的可能性

数据集创建可能容易受到噪声的影响。噪声可能会显著影响机器学习算法的结果，包括准确率、模型大小以及构建模型所需的时间 [139]。由于我们处理的不是分类问题，因此不存在标签噪声问题。我们认为，属性噪声在

数据集中的可能性极低，不会影响预测模型，原因如下。

- 架构相关属性是使用可靠/经过认证的工具获得的。因此，这里出现噪声的可能性非常小。

- 程序特征是使用内部开发的工具提取的。其中一些程序特征是代码的结构属性。获取这些特征时不可能引入错误。

- 我们使用了一组基于回归的模型来推导与程序运行时特征相关的属性。例如，缓存未命中惩罚、全局内存访问惩罚等。该数据集可能包含一些由错误信息引起的噪声。如果这些回归模型引入了噪声，那么该噪声将存在于数据集的所有元组中。不可能出现某些数据元组包含此类噪声，而另一些数据元组却不包含此类噪声的情况。

- 我们通过 NVIDIA 开发的 API，从板载传感器测量了 CUDA 内核消耗的功耗。这不太可能是噪声的来源。

该数据集和特征提取代码现已开放 [5]。

5.4.8 其他假设

关于 PFEA 算法和数据收集，还有一些其他假设，列举如下：

- 用于功耗数据采集的 NVML 库由 NVIDIA 设计。因此，预计其结果准确可靠，因为它能够监控和管理 NVIDIA GPU 内部的各种状态。

[5] https://github.com/santonus/energymodel

• 波次的数量取决于执行相应类型指令所需资源的可用情况。因此，我们在线程调度期间计算波次。

• 无论每个线程块包含多少线程，一个线程块都会被调度到一个流式多处理器上。当所有流式多处理器各分配到一个线程块后，下一轮分配将从第一个流式多处理器开始。

• 由于难以预测线程束调度器的空闲周期，我们假设线程束始终处于满载状态并执行有用工作。

• L1缓存与共享内存共存。在Sunpyo Hong的论文中[140], , L1缓存对图形处理器功耗的贡献与共享内存接近。由于我们已经基于共享内存使用情况考虑了五个特征，因此没有将L1缓存单独作为一个特征使用，因为静态地对准确的缓存行为进行建模是一个困难的问题。

• 在所考虑的三种GPU架构中，二级缓存都是末级缓存（LLC）。与L1缓存类似，无法对准确的二级缓存行为进行建模。因此，为了表示二级缓存对功耗的影响，我们采用了用于二级缓存的“缓存惩罚”特征，该特征可预测缓存未命中惩罚。

• 尽管考虑程序中显式同步原语的使用影响是一个非常关键的问题，但由于静态预测同步原语的影响较为困难，我们尚未对此进行考察。由于这一问题无法通过静态方式解决，我们计划在未来的工作中利用运行时数据对其进行研究。

表 5.5：收集到的功耗值的标准差

基准测试	数据点	平均功耗	标准差
vecADD	194	60.57	1.1830
时钟	18	100.29	1.7796
NN	53	48.12	0.2668
粒子滤波器	630	62.22	1.5285
Cifarnet	206176	64.31	0.1359

5.5 特征工程

在本研究考察的所有特征中（参见表5.4），我们系统地选择了可能准确预测功耗的特征。在为模型选择最终特征时，我们采用了以下方法。

5.5.1 派生属性：

我们倾向于使用派生属性，因为这是将专家知识注入模型的方法之一。例如，我们没有使用总延迟和每个流式多处理器的指令总数，而是使用由这两个属性计算得出的平均延迟。因此，我们剔除了 计算_指令_流式多处理器、全局_指令_流式多处理器、杂项_指令_流式多处理器、共享_指令_流式多处理器 （每个流式多处理器的指令总数）以及 计算_延迟_流式多处理器、全局_延迟_流式多处理器、杂项_延迟_流式多处理器、共享_延迟_流式多处理器 （每个流式多处理器的总延迟）特征，只保留了平均延迟。

5.5.2 非相关属性

在我们的模型中使用高度相关特征并不会提升模型性能；另一方面，这些特征可能会掩盖不同特征之间的交互作用。这表明，我们的模型中只应包含高度相关特征中的一个。我们对剩余的二十四个特征进行相关性分析，以观察它们之间的关系。

属性之间的关系。相关性分析用于计算两个变量之间关联的强度和关系的方向。关系强度通过相关系数的值来衡量，其取值介于 +1 和 -1 之间。值为 ± 1 表示两个变量完全相关。相关系数值越接近 0，两个变量之间的关系就越弱。关系的方向由系数的符号表示； + 号表示正相关， –号表示负相关。

我们考虑两种相关系数：皮尔逊系数和肯德尔秩相关系数，接下来将依次讨论。皮尔逊相关系数描述两个特征之间的线性关系。如果两个特征的线性相关性更强，其系数就会更接近 1。完全线性相关的两个特征可以相互表示，这意味着二者所代表的信息没有差异。移除高度相关特征有助于提高模型的准确率，因为这样会减少数据信息损失[141]。

如图 5.5 所示的皮尔逊系数热图所示，我们可以通过阅读热图来完成特征选择：

- 两个变量（x 和 y）之间的相关性可以从行对应于 x、列对应于 y 的单元格中获取，反之亦然。

- 热力图右端的颜色映射表示各系数对应的取值范围，尽管其确切数值已在相应单元格中标出。

- 通常，相关系数的绝对值大于 0.85 时被认为是高度相关，而接近零的值则被认为是相互独立的变量。

因此，我们采用 0.85 作为相关系数阈值，以剔除高度相关的特征。根据皮尔逊相关系数热图，我们删除了

- 流式多处理器_活跃 周期，因为它与线程块_大小 高度相关，相关系数为 1.0。我们选择线程块_大小 ，因为我们通过附录 8 中提供的动态分析观察到了它对特征重要性的影响

- 指令_发射_周期 与全局内存_惩罚 高度相关（0.91），与线程束_数量 （0.85）、波次 （0.85）、网格_大小 （0.85）以及总计_线程数 （0.92）高度相关。指令_发射_周期 大致代表活跃周期；通过附录 8 中提供的动态分析，我们发现活跃周期具有较高的特征重要性。

与皮尔逊系数相比，肯德尔系数对离群值更稳健 [142]。此外，皮尔逊系数假设变量服从正态分布。然而，这一假设并非总是成立。因此，我们也在分析中使用肯德尔系数，以便在皮尔逊分析所揭示的线性关系之外，进一步发现特征之间的单调关系。肯德尔相关系数利用观测值对，根据各对观测值之间的一致性和不一致性来确定关联强度。由于数据集规模较小，我们选择使用肯德尔系数而非斯皮尔曼系数。对于较小的样本量，肯德尔系数的 P 值更准确 [142]。

在根据皮尔逊系数剔除特征后，我们对新的特征集（十八个特征）进行了肯德尔相关分析。肯德尔系数以热力图的形式呈现在图 5.6 中。

根据肯德尔相关系数生成的热图，我们观察到

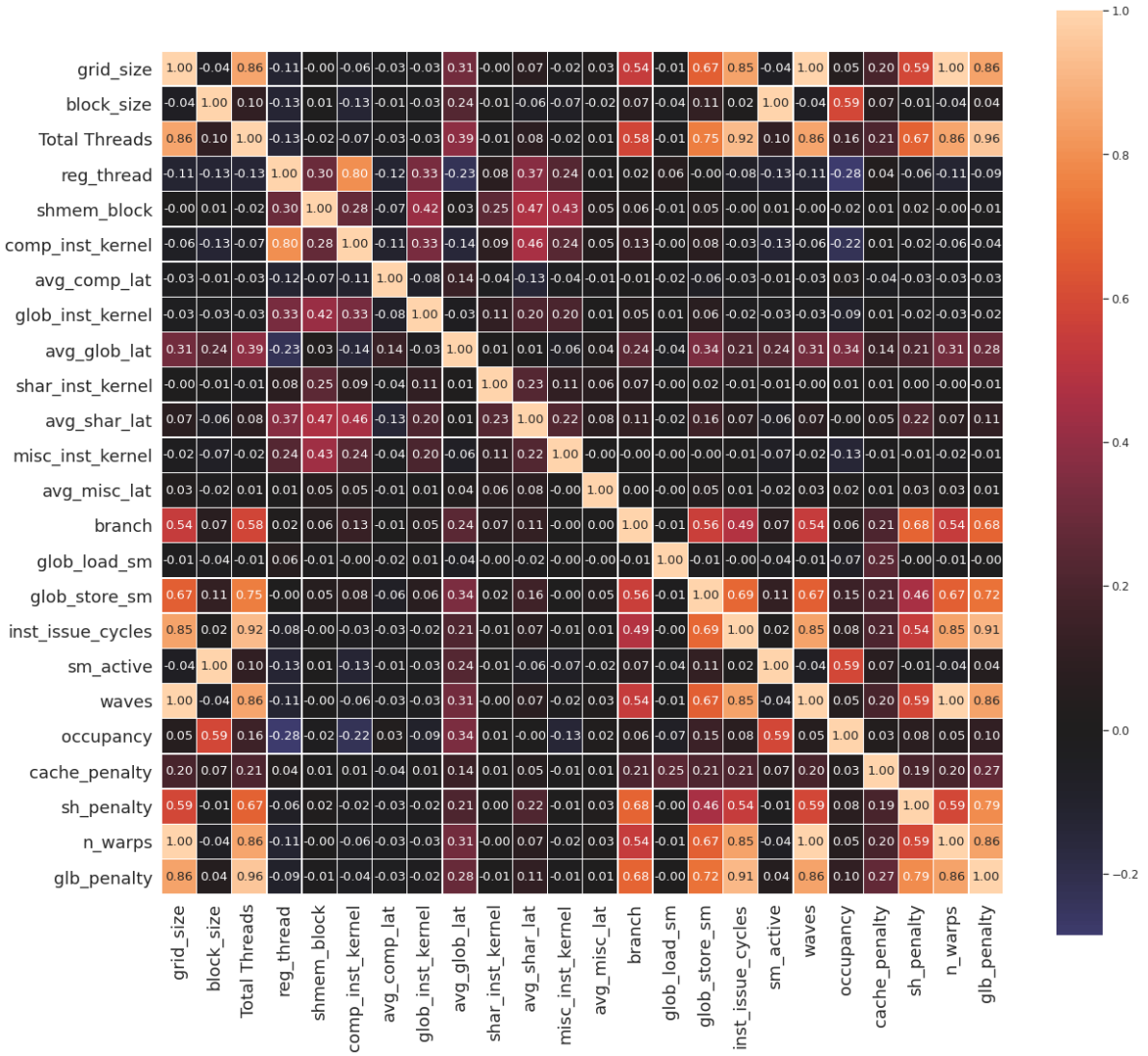


图 5.5：皮尔逊相关系数热图

- 平均_共享_延迟 与共享_指令_内核 高度相关（0.89），且与共享_惩罚 高度相关（0.94）。由于平均_共享_延迟 是一个派生特征，因此我们选择它而不是共享_指令_内核。

因此，我们舍弃这两个特征，并将平均_共享_延迟 保留在最终特征集中。通过相关性分析，我们从二十四个特征中移除了六个高度相关特征。经过相关性分析后，最终的十六个特征预计彼此之间不存在或仅存在较弱的关联和交互作用。

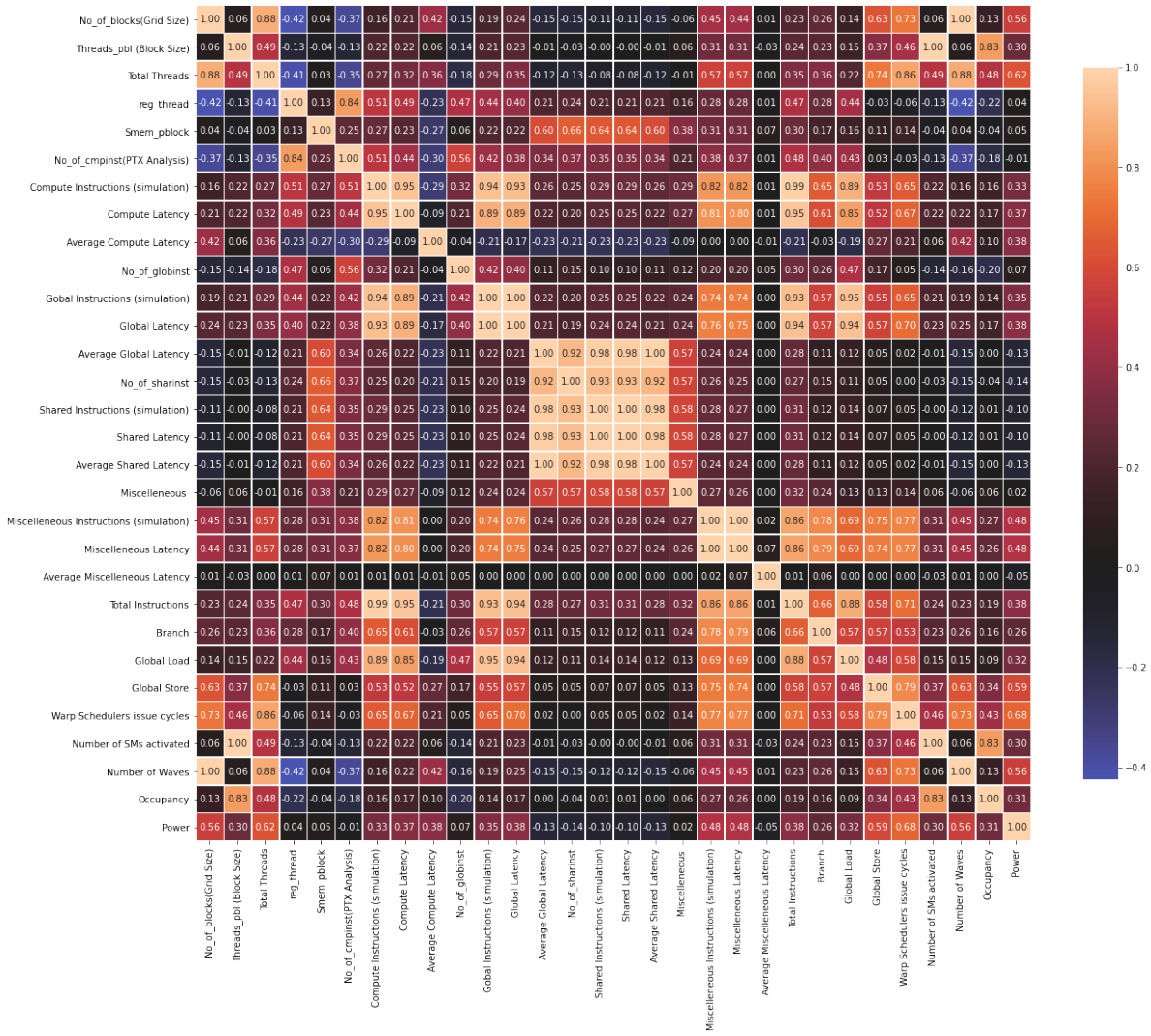


图 5.6：肯德尔相关系数热图

5.5.3 显著影响属性

我们使用包含十六个特征的随机森林回归器进行特征重要性分析，然后系统地移除了对预测准确性没有影响的特征。在此过程中，我们移除了平均_杂项_延迟 特征。

我们还使用 SHAP（沙普利加性解释） [143] 来验证对功耗预测贡献最大的特征。图 5.8展示了 Tesla K20 GPU 的随机森林与 SHAP 特征重要性结果对比。如图所示，两者的结果高度一致，因为排名前四和最后两位的特征相同，其他特征的顺序则略有差异

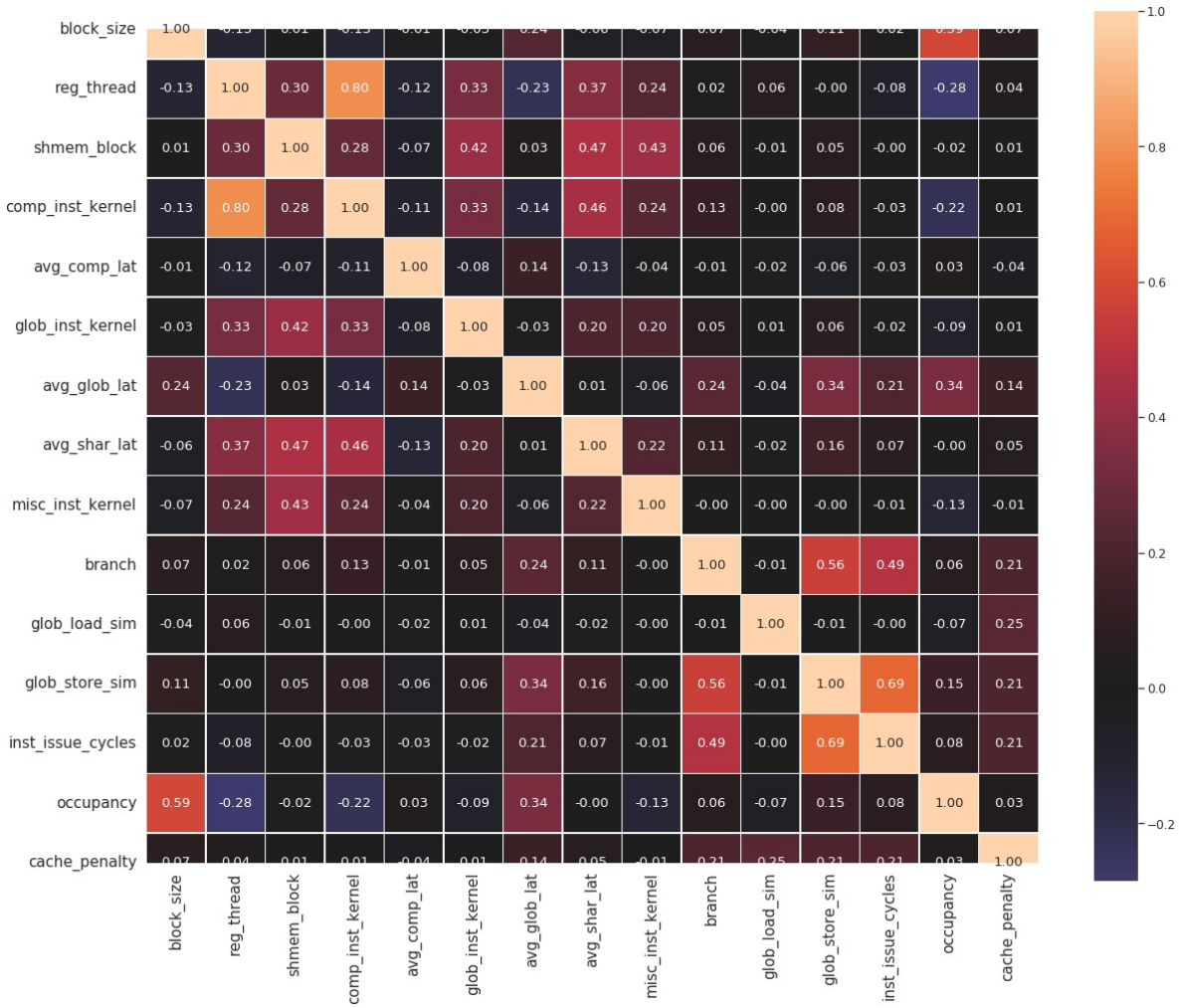


图 5.7：最终特征集的皮尔逊相关系数热图

。如图5.7所示，这些特征的皮尔逊系数结果证实，最终选出的这十五个特征之间的相关性极低。最终的特征集见表5.4。

5.6 模型评估指标

我们将根据 5 折交叉验证分数的均值，使用三个关键指标来验证模型：R^2 分数、均方根误差（RMSE）和平均

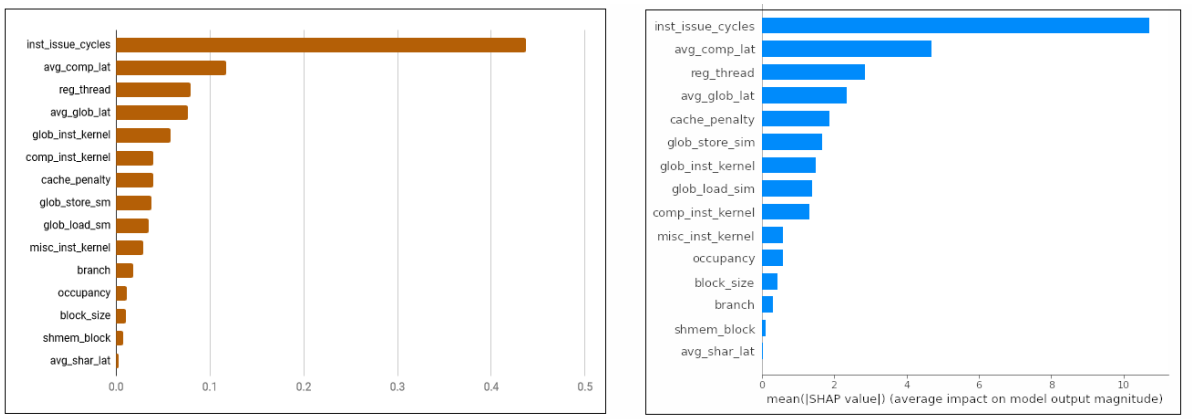


图5.8：使用（a）随机森林和（b）SHAP绘制的特征重要性图

绝对误差（MAE）。我们将详细讨论这些指标。

5.7 模型选择与评估

我们对传统机器学习和神经网络模型进行了系统研究，并根据模型机制和性能确定最终模型。我们首先考察了预测变量（两两之间）以及响应与预测变量之间是否存在线性关系。从预测特征与响应（即功耗）的图中可以明显看出（图5.10），输入特征与GPU功耗之间不存在 线性关系。如果两个变量之间不存在 线性关系，那么随着一个变量变化，另一个变量的增减速率也可能发生变化，从而使数据呈现复杂模式。非线性函数或方法或许能更好地模拟特征与功耗之间的这种复杂趋势。不过，为了排除任何线性回归模型的可能性，我们对数据集进行了多元线性回归。多元线性回归模型的R^2 得分为 -0.1919，这表明该模型对实验数据的拟合效果不佳。

这一观察促使我们进一步探索更复杂的非线性回归技术，如下一节所述。正如本章后文所示，*Catboost* 和 *XGBoost*

最终根据所选特征的性能和相关性，选定这两个最先进模型。

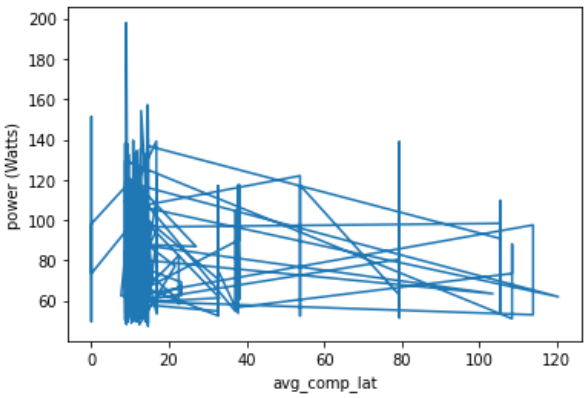
5.7.1 人工神经网络实验与分析

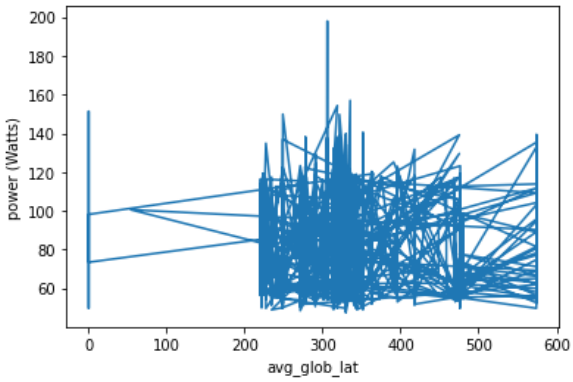
在众多可用的方法和算法中，人工神经网络（ANN） [144] 是最受欢迎的方法之一，常用于对数据中的复杂模式进行非线性关系建模，尤其适用于预测问题。我们通过改变人工神经网络架构、数据集大小和学习率，对人工神经网络进行了广泛实验。我们使用了多种特征选择方法：i）第5.5节和表5.4中介绍的特征工程；ii）包含15个潜在特征的变分自编码器（VAE）；以及iii）自动特征选择（不使用VAE）。

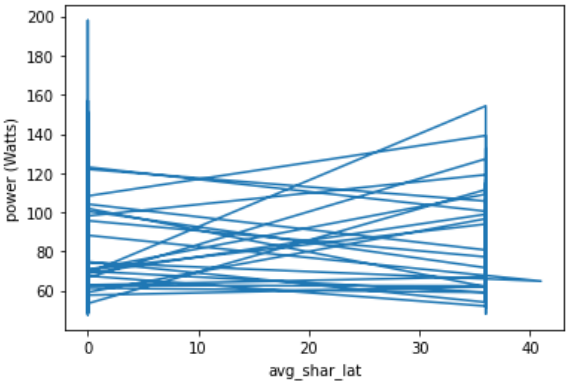
我们使用两种不同的层配置训练了具有三个隐藏层的人工神经网络。为了研究人工神经网络在较大数据集上是否表现更好，我们分别针对较小数据集（990个元组）和较大数据集（1810个元组）进行了实验。人工神经网络实验的详细信息见表5.6。我们的发现如下：

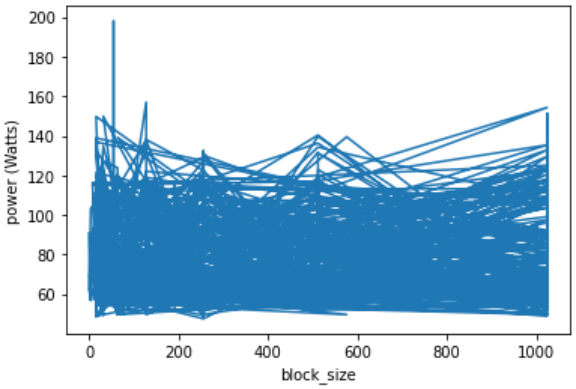
1. 我们观察到，人工神经网络在三层中分别采用 64、64、128 个神经元时效率最高。

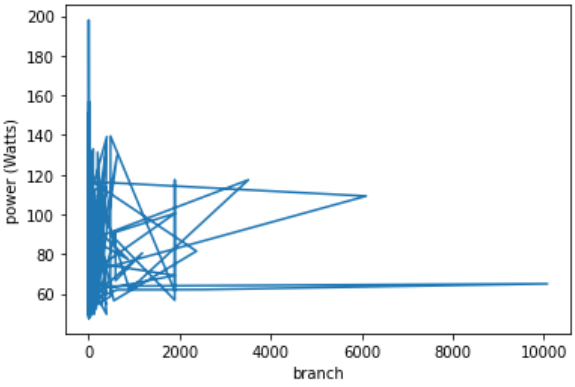
2. 训练人工神经网络模型时，随着训练的进行，降低学习率并观察其对模型性能的影响通常很有用。除了均匀学习率（表5.6的前两行结果）之外，我们还采用了以下自适应学习率方法：i）指数衰减和 ii）反时间衰减学习率调度。对于这三种学习率方法，我们都使用了通过特征工程方法选出的特征。反时间衰减模型在所有训练的人工神经网络模型中精度最高，而且它是唯一一种在较大数据集上表现更好的方法。

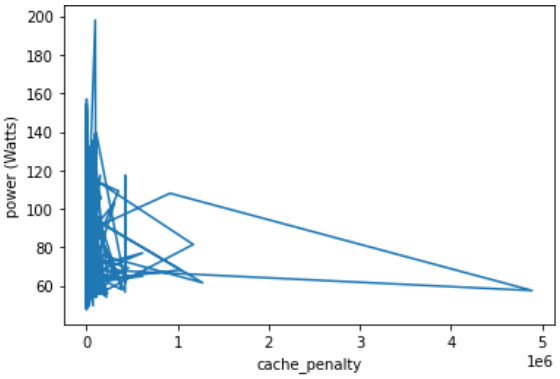

power (Watts)
avg_comp_lat

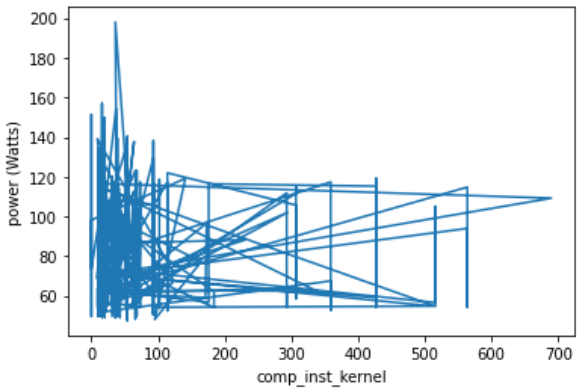

power (Watts)
avg_glob_lat

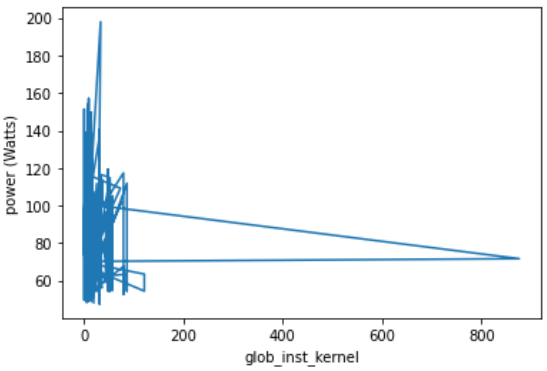

power (Watts)
avg_shar_lat


power (Watts)
block_size


power (Watts)
branch


power (Watts)
cache_penalty
1e6


power (Watts)
comp_inst_kernel


power (Watts)
glob_inst_kernel

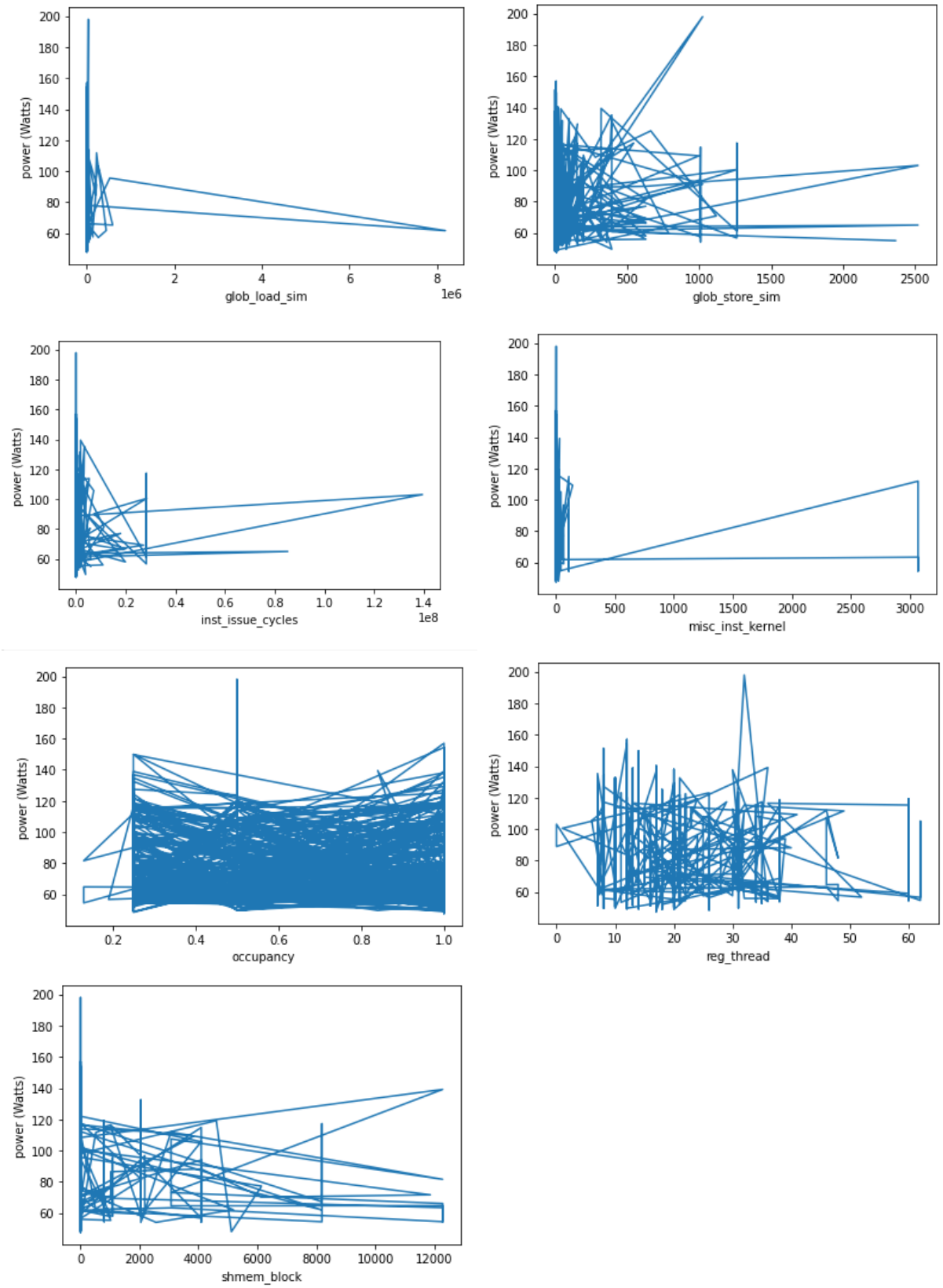


图5.10：特征与功耗之间的关系

3. 从图 5.11 所示两种方法的损失函数图可以看出，随着训练轮数增加，验证损失高于训练损失。这表明出现了过拟合。此外，训练曲线未呈现出均匀的平滑性和单调性，因此损失可能会出现非平滑变化。

4. 使用反时间衰减的、包含 15 个潜在特征的变分自编码器（特征数量与特征工程方法保持一致），其^2 分数低得难以接受，如表5.6所示。增加潜在特征的数量并未带来显著改善。如表 5.6 所示，自动特征选择方法的表现优于基于变分自编码器的方法，其^2 值与基于特征工程的方法相当。

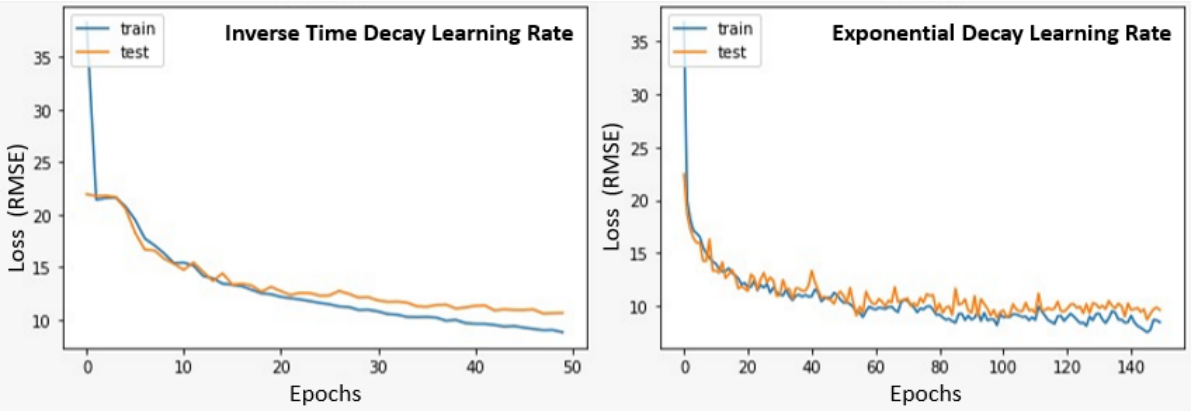


图 5.11：自适应学习率人工神经网络的损失函数图

5.7.2 基于支持向量回归的模型

支持向量回归（SVR） [145] 考虑了数据中非线性关系的存在，并提供了一种高效的预测模式。在对 SVR 的常用内核进行实验后，我们发现径向基函数（RBF）内核能够提供最精确的模型。然而，对于 Tesla K20 GPU，RBF 内核的 R^2 得分为 0.7041，这对于功耗预测模型而言仍然不可接受。

表5.6：人工神经网络模型实验结果

模型参数		较小数据集		较大数据集	
人工神经网络架构（3 个隐藏层）	学习 rate	训练轮次	R^2 分	训练轮次	R^2 分数
通过特征工程选出的15个特征，统一学习率					
5,5,5	0.01	130	0.6226	135	0.5003
64,64,128	0.01	150	0.7011	120	0.6791
使用特征工程选出的 15 个特征，指数衰减学习率					
64,64,128	0.9	175	0.7246	150	0.7057
使用特征工程选出的 15 个特征，逆时间衰减学习率					
64,64,128	0.5	100	0.7418	50	0.7737
用于特征提取的变分自编码器（潜在特征：15），逆时间衰减学习率					
64,64,128	阶梯衰减		0.3270		0.00802
自动特征选择（不使用变分自编码器），统一学习率					
5,5,5	0.01	150	0.6081	150	0.53970
64,64,128	0.01	150	0.7084	120	0.6484

由于 ANN 和支持向量回归等多元线性回归模型无法准确地将功耗映射到输入特征，且表现出过拟合现象，因此我们得出结论：这些模型不足以解决功耗预测问题。此外，神经网络回归器对数据的需求量较大，因此在缺乏充足数据的情况下，通常无法建立足够合理的模型。

5.7.3 基于随机森林的回归器

随机森林（RF） [146] 是一种元估计器，它通过一些有益的改进来集成多个决策树。

- 每个节点可用于分裂的特征数量被限制为总数的一定比例。这确保集成模型不会过度依赖任何单个特征，并能充分利用所有可能具有预测能力的特征。

- 每棵树在生成分裂时都会从原始数据集中随机抽取样本，加入了额外的随机性，从而防止过拟合。

当数据有限且包含大量特征属性时，基于树的回归模型已被证明效率很高 [105][147][148]。下一步，我们考虑基于树的方法，例如决策树，它们能够很好地对非线性关系进行建模。Tesla K20 GPU 上决策树的 R^2 分数为 0.8447，表明基于树的方法前景良好。我们进一步探索更先进的基于树的算法，这些算法通常比决策树表现更好。

随机森林（RF）在特征空间中提供了更高的分辨率，因为其树不会被剪枝，并且会将特征空间划分为更小的区域。RF在每个节点使用行的随机子样本，并选择不同的特征样本进行分裂。由于RF是在多样化的样本上学习得到的，因此具有更高的多样性。RF中的每棵树都会产生一个单独的预测结果，最终预测结果是所有树预测结果的平均值。平均操作确保了训练损失和验证损失的平滑性，并且结合RF的随机性，提高了其准确率并减少了过拟合。

在特定情况下，与人工神经网络（ANN）相比，RF具有一些优势，包括鲁棒性以及成本和时间方面的益处。RF在可解释性方面也特别具有优势 [149] ，尤其是在能耗数据中。根据表5.8所示的结果进行经验观察后发现，随机森林比决策树或线性拟合回归器更为合适。对于Tesla K20，R^2 得分为0.8899。我们考虑了其他流行的基于树的集成机器学习算法，以确定在图形处理器功耗预测中最准确且高效的算法。

集成学习是通过有策略地生成并组合多个模型（如分类器或专家），以解决特定计算智能问题的过程。集成学习主要用于提升模型的（分类、预测、函数逼近等）性能，或降低选用较差模型的可能性。 [150, 151]。这些算法包括：极端随机树回归器[152], 梯度提升回归器 [153], XGBoost回归器 [154, 155], 以及

Catboost [156] 用于预测GPU功耗。这些算法涵盖了用于预测连续变量的多种集成方法。事实证明，XGBoost是预测功耗最高效、最准确的模型。我们首先讨论这些技术，然后介绍相关实验、结果和分析。

5.7.4 极端随机树回归器

极端随机树分类器（Extra Trees分类器）是一种集成学习技术，它将多个彼此不相关的决策树汇集成“森林”，并聚合其结果以输出分类结果。虽然随机森林分类器与极端随机树有相似之处，但两者构建森林中决策树的方式不同。随机森林使用自助法副本对输入数据进行有放回子采样，而极端随机树使用完整的数据样本。随机森林选择最优分割点，而极端随机树随机选择分割点。不过，在选定分割点后，这两种算法都会从所有特征子集中确定最佳子集。因此，极端随机树增加了随机性，但仍然进行优化。

极端随机树算法在计算成本和执行时间方面比随机森林更快。这是因为尽管整个过程相同，但它会随机选择分割点，而不计算最优分割点。如表5.8所示，极端随机树的表现与随机森林回归器接近（Kepler），并略好一些（Maxwell和Volta）。

5.7.5 梯度提升回归器

梯度提升机（GBM）是一种流行的集成方法，其基础是创建多个弱模型，并将它们组合起来，以利用整个组获得更好的性能。决策树被用作弱学习器，尤其是回归树，

它们为划分输出实数，并且其输出可以相加，从而允许将后续模型的输出相加。该输出会校正预测中的残差[157]。

梯度提升树与随机森林之间有两个显著区别：训练方式和决策输出方式。在梯度提升中，我们按顺序训练树，每次训练一棵树，使其纠正前面各棵树产生的误差。而在随机森林中，我们独立构建各棵树。因此，我们可以并行训练 RF，但无法并行训练前者。另一个区别，即它们输出决策的方式，也有所不同，因为随机森林中的各棵树彼此独立。因此，RF 可以按任意顺序确定其输出，而梯度提升树则按照固定顺序运行。因此，其执行顺序不能改变，这限制了梯度提升只能进行顺序评估。与 RF 相比，采用 GBM 可以提高 R^2 分数，这促使我们尝试一种基于 GBM 的更强大技术，即 XGBoost。

5.7.6 XGBoost回归器

XGBoost（极端梯度提升）是一种经过优化的梯度提升算法。不过，两者之间存在一些关键差异，使得 XGBoost 比梯度提升更快、更准确。GBM 不使用正则化[158]，而 XGBoost 采用 L1 和二级缓存正则化，从而提升模型的泛化能力。XGBoost 还具有一项优势：训练速度非常快，并且可以在集群中并行化或分布式运行。从表5.8所示的结果可以看出，XGBoost 是功耗预测问题中最准确的模型。另一种基于梯度提升的高效且强大的技术是 CatBoost。

表 5.7：机器学习模型及其超参数

随机森林	
n_估算器	500
最大_深度	14
最小_样本_分割	2
最大特征数_	auto
最小_叶节点样本数_	1

CatBoost	
深度	8
迭代次数	270
loss_函数	RMSE
日志_级别	静默

决策树	
最大_深度	14
样本_分割	5
准则	mse

XGBoost	
n_估算器	500
最大_深度	6
学习_率	0.05
最大_深度	6
最小_子节点_权重 1	
按树采样的列比例_	0.7
子采样	0.7
静默	1

SVR	
C	200
epsilon	1
伽马	0.5
内核	rbf

极端随机树	
估计器数量_	220
最大_深度	16
最小_样本_分割	2
最大特征数_	auto
最小_样本_叶节点	1

梯度提升	
n_估计器	250
最大_深度	5
学习_率	0.17
最大特征数_	auto
准则	弗里德曼 _mse

5.7.7 CatBoost

CatBoost 在处理高基数类别变量方面改进了梯度提升。对于低基数类别变量，CatBoost 使用独热编码 [159]。CatBoost 使用二叉决策树作为基础预测器 [156, 160]。过去的研究表明，与其他机器学习算法相比，CatBoost 的效率最高 [161, 162]，尤其是在分类问题中 [163, 164]。然而，研究发现，对于使用本研究所考察特征进行的功耗预测，XGBoost 的表现略优于 CatBoost。我们进一步在 5.8 节讨论并验证了 CatBoost 和 XGBoost 的卓越性能。

5.7.8 模型构建与训练：实现细节

如果数据集中各特征的取值尺度相对接近，机器学习模型的效果会好得多。基于树的模型具有尺度不变性。不过，我们仍对数据进行了缩放，因为这是一种能够加快计算的有效技术。这里我们采用 MinMaxScalar 技术，因为它能够保留特征分布的形状，从而保留原始数据中蕴含的信息。

对获得的数据集进行打乱，以消除数据集中的偏斜。

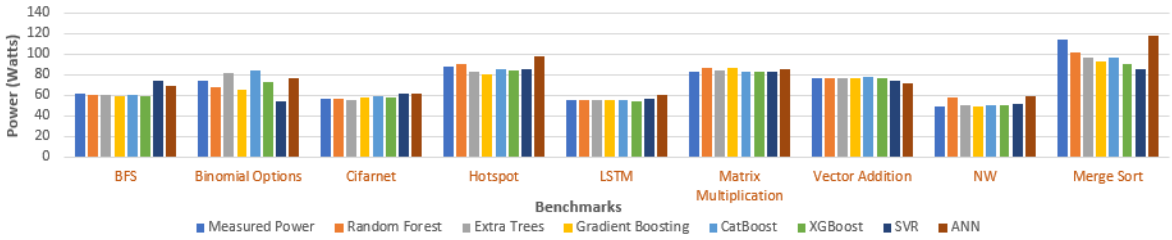
基于树的装袋和提升模型采用 scikit-learn（版本：0.24.1）提供的树回归方法构建。简单交叉验证使用相同的数据进行超参数调优并评估模型性能。这可能会导致对模型的评估产生偏差。因此，我们考虑使用嵌套交叉验证，在所有数据划分上计算每种超参数组合的得分，从而得到最佳参数组合。随后使用该最佳参数组合再次计算所有数据划分上的得分。

这些回归模型的调优参数列于表 5.7 中。$n_$估计器 表示模型中使用的树的数量，$max_$深度 表示树可达到的最大深度，$min_$样本_分割 表示拆分内部节点所需的最小样本数，max_特征表示用于最佳拆分的特征数量，而拆分准则用于衡量拆分质量（可以是均方误差或平均绝对误差）。径向基函数内核的支持向量回归模型使用 scikit-learn 库构建。我们采用前文提到的嵌套交叉验证技术来调整内核系数、正则化参数以及与实际值之间的 e
psilon 距离；对于该距离，不会施加任何惩罚。

我们使用 Keras 库（版本：2.4.3）构建人工神经网络模型。我们采用增量式方法构建人工神经网络模型 [165], ，最终设置了三个全连接隐藏层，神经元数量分别为 64、64 和 128。模型使用 Adam 优化器执行梯度下降。我们采用 dropout 技术和正则化来防止过拟合。我们通过观察训练损失和验证损失，使用嵌套交叉验证来确定训练轮数、批量大小和学习率等最优参数。我们使用 5折交叉验证来划分训练数据和测试数据。

模型验证 我们通过观察训练损失和验证损失，使用嵌套交叉验证来确定训练轮数、批量大小和学习率等最优参数。我们使用 sklearn 的 k折交叉验证（CV）方法[6] 划分训练数据和测试数据。k折交叉验证可确保测试集的先验信息不会“泄漏”到模型中，并使评估指标能够反映泛化性能。我们会留出一个测试集进行最终评估，但在进行交叉验证时不再需要验证集。训练集被划分为 k 个较小的子集；对于 k 折中的每一折，使用其中 $k-1$ 折作为训练数据来训练模型，随后使用剩余部分验证所得模型，也就是说，剩余部分将作为测试集，用于计算准确率等性能指标。我们将 k=5 设为 5折交叉验证。

在5折交叉验证中，随机打乱数据集后，我们将数据集划分为5个折。对于每个不同的折，我们将其中一个折作为测试集，并将其余折作为训练集。随后，我们使用训练集对模型进行拟合，并在测试集上进行评估。此外，我们运行模型5次，并报告性能的平均值和标准差。正如第7节所示，在全部5次运行中，性能均较为稳健，因为性能与平均性能之间的标准差较低。



Fi图 5.12：热门基准测试的功耗实测值与预测值

[6]https://scikit-learn.org/stable/modules/cross_验证.html

表5.8：不同 GPU 架构下机器学习算法的验证分数

架构	回归模型 R^2 得分		RMSE	MAE
Tesla K20	随机森林	0.8899	7.9265	4.4177
	极端随机树	0.8893	7.9438	3.9346
	梯度提升	0.9025	7.4657	3.7849
	XGBoost	**0.9101**	**7.1592**	**3.4816**
	CatBoost	**0.9067**	**7.3001**	**3.8308**
	决策树	0.8447	9.3899	5.007
	SVR	0.7041	13.0078	8.4063
	ANN	0.7011	12.1048	7.4597
Tesla M60	随机森林	0.9398	4.5547	2.2427
	极端随机树	0.9557	3.9610	1.9362
	梯度提升	0.9415	4.3645	2.0169
	XGBoost	**0.9544**	**3.9172**	**1.7588**
	CatBoost	**0.9548**	**3.89382**	**1.8252**
	决策树	0.8762	6.6781	3.7151
	SVR	0.6021	12.0735	8.7283
	ANN	0.7314	8.8217	5.9993
Tesla V100	随机森林	0.9415	8.8127	3.7447
	极端随机树	0.9526	7.9428	3.1989
	梯度提升	0.9595	7.4883	3.2475
	XGBoost	**0.9646**	**6.9968**	**2.9028**
	CatBoost	**0.9543**	**7.8954**	**3.1766**
	决策树	0.8978	10.9053	4.4655
	SVR	0.6435	22.6876	13.5012
	ANN	0.7134	18.3762	11.0164

5.8 最终模型分析

在图 5.12中，我们展示了本研究所考虑的所有机器学习模型对属于全部三个基准测试套件的一些常用基准测试的实测功耗与预测功耗。为了便于展示，我们选择了一小部分基准测试在图 5.12中呈现，而未展示本研究中考虑的全部基准测试。我们在图 5.13中展示了相同基准测试的预测误差。

结果分析：实验结果表明，在所研究的三种 GPU 架构下，XGBoost 模型的表现均优于其他所有算法。CatBoost 的损失曲线与 XGBoost 的曲线相似，都呈现单调下降趋势。各模型的损失函数图表明，这两个模型训练良好，因为二者

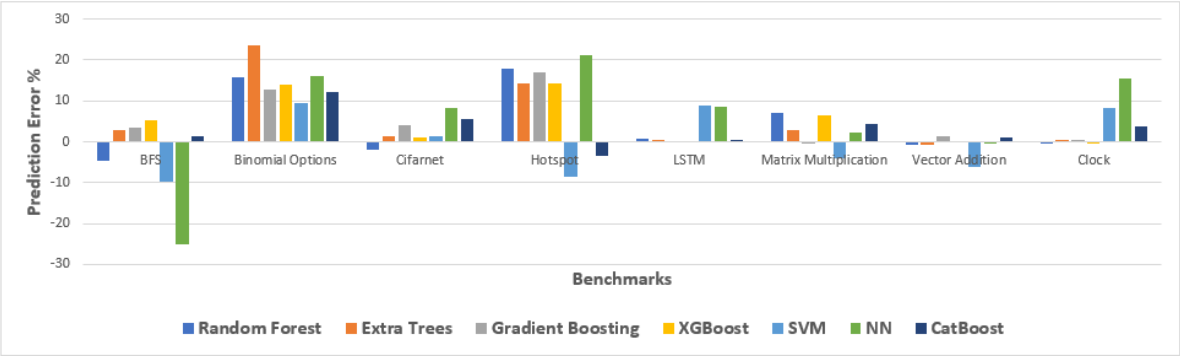


图 5.13：热门基准测试的功耗预测误差

都呈现平滑、单调的下降趋势，如图 5.14 所示。测试损失曲线与训练损失曲线紧密贴合，因此模型能够很好地泛化到新数据。这与人工神经网络和其他非树型回归器形成鲜明对比（见图 5.11）。训练损失与测试损失之间的数值差异足够小，足以证明与包括 CatBoost 在内的其他任何回归器相比，XGB 的拟合能力更强。因此，我们在表5.10中评估并比较这两种基于树的模型的性能（为简洁起见，表中展示的是 Tesla K20 GPU 的结果）。

我们进一步根据损失曲线（图 5.11 和图 5.14）分析 CatBoost 和 XGBoost 模型与基于人工神经网络的模型之间的表现差异。人工神经网络很早就出现过拟合，约在 20 个训练轮次时。在 20 个训练轮次时，逆衰减人工神经网络的验证 R^2 分数为 0.6943（训练损失：12.51，验证损失：13.13），而指数衰减人工神经网络的验证 R^2 分数为 0.5839（训练损失：11.77，验证损失：12.97）。超过 20 个训练轮次后，情况会进一步恶化，损失曲线也会波动（最终训练损失：*8.82*，验证损失：*11.656*）。CatBoost 模型约在 60 个训练轮次时开始过拟合，此时 $R^2 = 0.839$（训练损失：7.397，验证损失：7.55），远高于人工神经网络。XGBoost 明显优于所有模型。它在 165 个训练轮次时达到了人工神经网络的最终 R^2 分数（0.773）（训练损失：8.802，验证损失：8.86），且没有出现过拟合。当 XGBoost 在约 200 个训练轮次时开始轻微过拟合，其验证 R^2 分数达到 0.848（训练损失：6.04，验证损失：7.54），显著高于人工神经网络模型的最终 R^2 分数。在基于树的方法中，决策树

表 5.9：5 次运行后全部三种架构的 XGBoost 结果

架构 R^2score		RMSE	MAE	MAPE
Kepler	0.901 ± 0.0083	$7.4919 \pm 0.0.3194$	3.6740 ± 0.1126	4.5784 ± 0.1809
Maxwell	$0.9418 \pm 0.0.0056$	4.4000 ± 0.1702	1.8780 ± 0.0634	2.9278 ± 0.0857
Volta	0.9553 ± 0.0056	7.7712 ± 0.4081	3.0521 ± 0.1365	3.9900 ± 0.0732

也会较早出现过拟合，即在深度=4时，其验证 $R^2 = 0.641$（训练损失：11.79，验证损失：14.21）。此外，请注意，损失会振荡，并非平滑衰减。达到一定训练轮数后，损失也会趋于饱和（逆衰减为 50 轮，指数衰减为 140 轮），这与图 5.14 不同；在图 5.14 中，基于树的方法表现出平滑、单调递减的损失曲线。我们在 Tesla V100 上运行 XGBoost 模型五次，以测试其稳健性，观察到的结果列于表 5.9。这证明模型表现良好并非偶然，而且无论测试数据和训练数据如何变化，预测准确率都保持一致。

我们将响应（即 CUDA 内核消耗的功耗）与程序特征联系起来，并通过特征重要性了解这些特征的贡献。如表5.10所示，我们可以得出结论：*inst_issue_*周期、*avg_comp_lat*、*reg_*线程、*glob_inst_*内核和*glob_*存储_流式多处理器 是排名前五的特征，在两个模型中都具有较高的特征重要性。由于我们在第5.5节所述的特征选择过程中使用了 Shapley，这些特征也被 Shapley 标记为重要特征。

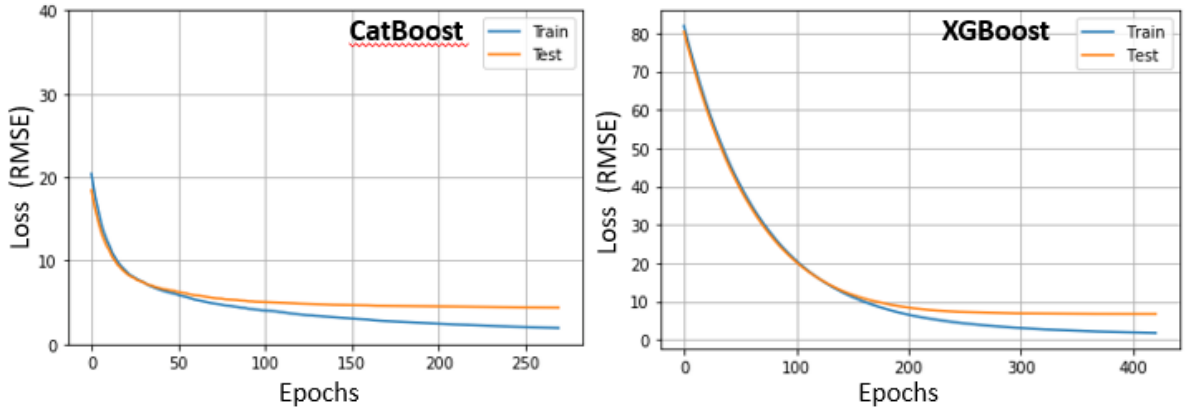


图 5.14：最精确的基于树的方法的损失函数图

表5.10：XGBoost 与 CatBoost 评估

评估指标		**XGBoost**	**CatBoost**
R^2 **分数**		0.9101	0.9067
RMSE		7.1592	7.3001
MAE		3.4816	3.8308
较小数据集	**内存占用**	323.23 兆字节	241.52 兆字节
较大数据集		162.68 兆字节	214.86 兆字节
较小数据集	**执行 time**	3.29 秒	10.3秒
较大数据集		6.4s	11.8秒
特征重要性排名（一致的跨越全部三个图形处理器架构）		1. glob_存储_流式多处理器 2. 指令_发射_周期 3. 平均_计算_延迟 4. 其他指令内核_ _ 5. 全局_指令_内核 6. 寄存器_线程 7. 计算_指令_内核 8. 平均_全局_延迟 9. 缓存_惩罚 10. 平均_共享_延迟 11. 占用率 12. 共享内存 线程块_ 13. 分支 14. 全局_加载_流式多处理器 15. 线程块_size	1. 平均_计算_延迟 2. 指令_发射_周期 3. reg_线程 4. glob_inst_内核 5. comp_inst_内核 6. 平均_全局_延迟 7. 其他指令内核_ _ 8. 全局_存储_流式多处理器 9. 分支 10. 全局_加载_流式多处理器 11. 缓存_惩罚 12. 占用率 13. 平均_共享_延迟 14. 块大小_ 15. 共享内存 _线程块

现在，我们考察这些影响响应的重要特征，并尝试解释预测变量逐一变化时响应变量所呈现的变化。我们还假定这些重要特征之间的两两交互较少。在通过 XGBoost 拟合的非线性模型中，我们固定所有（重要）特征的取值，仅改变当前考察的一个特征。随着该特征取值变化、其他特征保持不变，我们绘制响应变量的变化。这将衡量图 5.15 中每个关键特征对预测功耗的影响。正如预期，这些排名靠前的特征确实能够有效解释响应变量的变化。需要指出的是，由于没有可用的闭式回归模型，这种方法直观上是检验响应变量对排名靠前的预测变量变化所作反应的合理方式。

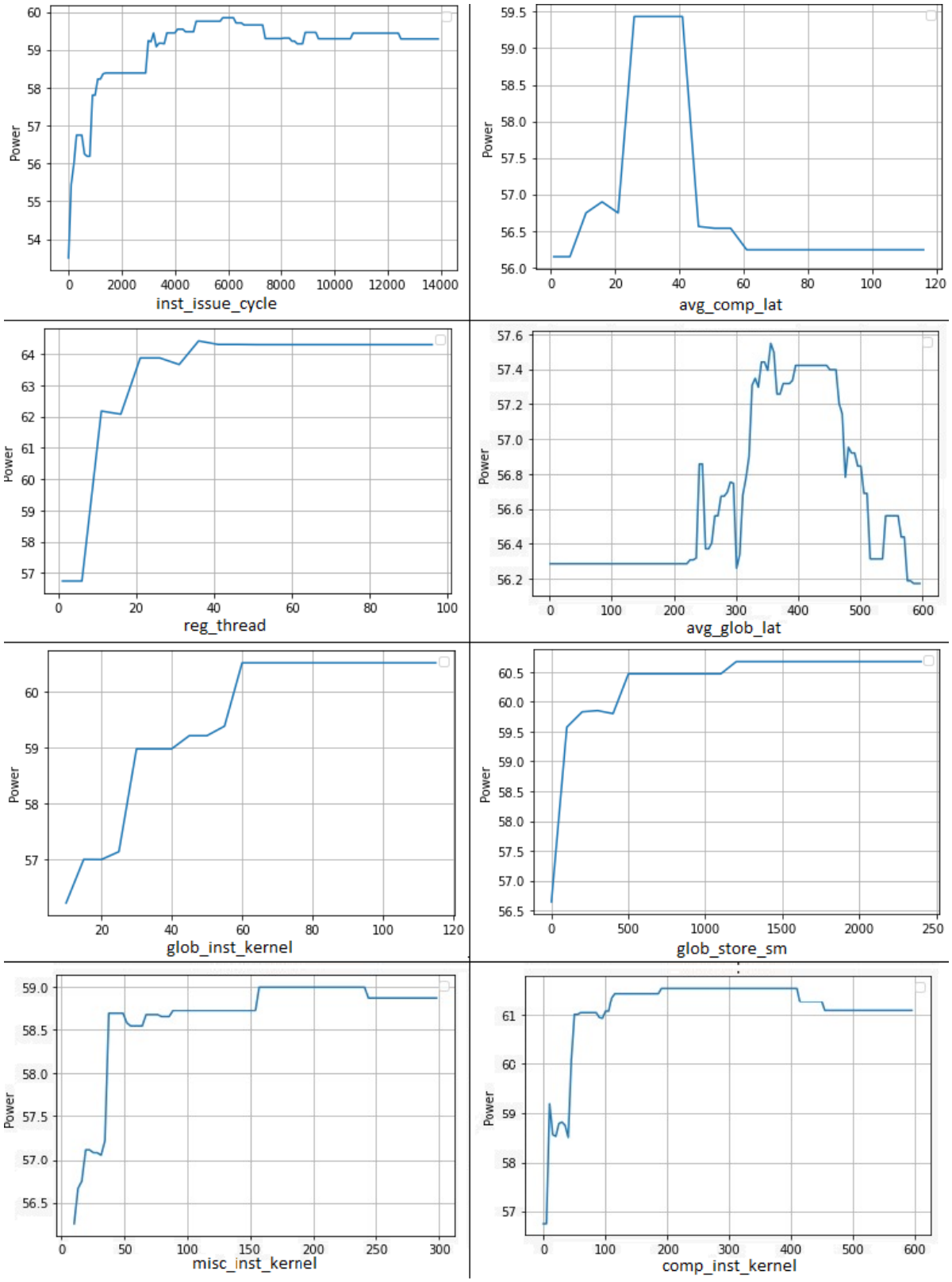


图 5.15：关键程序特征与功耗关系图

5.8.1 从架构视角分析重要特征

特征重要性分析发现，指令_发射_周期 对所有三种架构的功耗影响最大。由于 FDS 单元用于发射指令，指令_发射_周期可以合理地表示 FDS 的激活次数。Hong 等人 [29] 指出，FDS 单元的功耗高于其他组件，因为所有指令都会使用该单元。因此，我们发现该特征是最重要的特征，这无疑与相关发现 [29]一致。由于开发人员难以控制特定任务所发射的指令数量，一个可行的选择是专注于编写高效代码，以更少的指令完成任务。

特征 平均_计算_延迟 是所有已执行计算指令的平均延迟，包括 maddf、divf 等浮点指令。从图 5.15 可以观察到，当平均_计算_延迟 的值超过十个周期时，它会影响功耗预测。当应用程序包含浮点指令时，就会出现这种情况，因为其他整数计算指令（加法、sub）的延迟明显较低（补充文件中的第III节）。我们的结果与.....不同Hong 等人 [29] 的研究发现，该研究认为指令和浮点单元的功耗之间不存在显著差异。

下一个重要特征表示每线程寄存器数（寄存器_线程）。这与 Greener [135] 报告的研究结果一致。随着寄存器文件大小增加，漏电功耗也会增加；因此，建议限制寄存器的使用，以构建高能效应用程序。

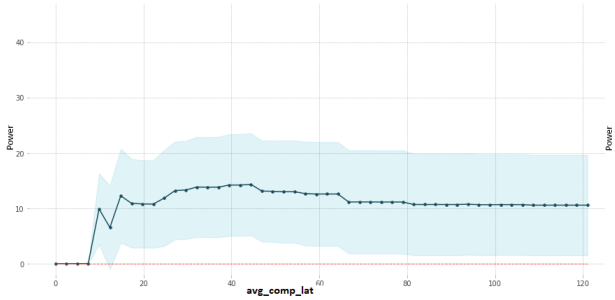
每个内核的全局内存指令数（全局_指令_内核）和全局内存平均延迟（平均_全局_延迟）也会影响功耗预测，尤其是全局存储指令（全局_存储_*SM*），其影响与全局_指令_内核一致。Nagasaka等人

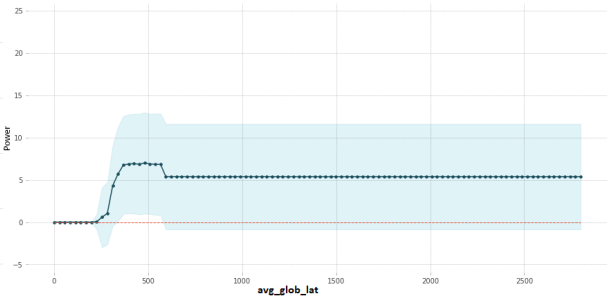
[98] 得出结论，获取全局内存访问计数对于准确进行 GPU 功耗建模至关重要。根据 Hong 等人的运行时硬件单元分析， [29],全局内存指令消耗的功耗相当可观，远高于浮点基准测试。这支持了我们的发现：全局内存对功耗有显著影响。为减少全局内存导致的功耗增长，可以尽可能多地使用共享内存来替代全局内存指令。观察全局_存储_*SM* 相较于全局_加载_*SM* 的特征重要性差异表明，向全局内存写入比从中读取更耗电。

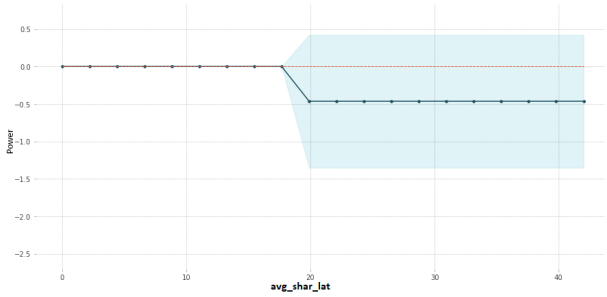
在所有采用的特征重要性技术中，线程块_大小 和 共享内存_线程块 始终是最不重要的特征。观察与共享内存相关特征的特征重要性（共享内存_线程块，平均_共享_延迟）可以发现，与其他所有指令类型相比，共享内存的使用对功耗预测的影响最小。关于线程块_大小，我们可以得出结论：只要启动的总线程数（网格_大小 × 线程块_大小）较多，用于计算指令_发射_周期的线程块_大小 就没那么重要。

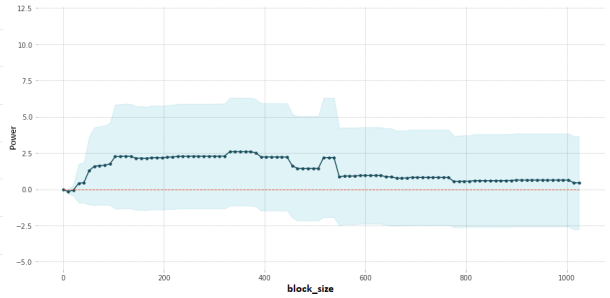
5.8.2 异常值检测

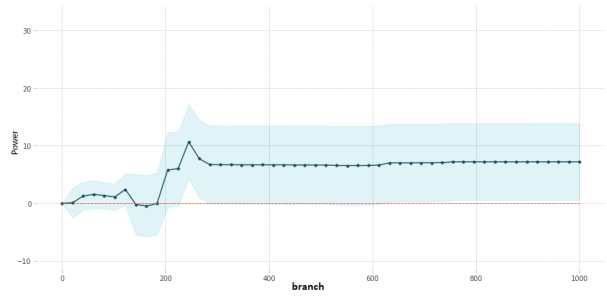
由于异常值会对底层数据及其关系产生误导性呈现，因此会影响统计度量和数据分布。因此，从训练数据中移除异常值对于建模是必要的，这样可以更好地拟合数据，进而提高预测能力和结果准确率。对于根据程序特征构建的功耗预测模型，如果某个 CUDA 基准测试的功耗与其他具有相似特征的数据点相比偏低或偏高，就可能出现异常值。例如，向量加法基准测试包含三种启动配置，即网格大小、块大小（c1：（256,1024），c2：（512,512），c3：（1024, 256））。总

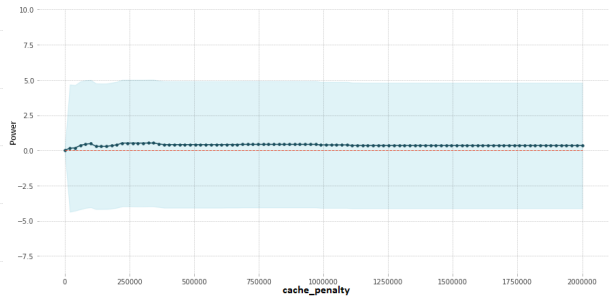

Power
avg_comp_lat

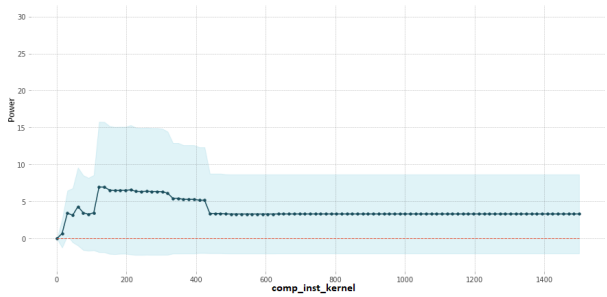

Power
avg_glob_lat

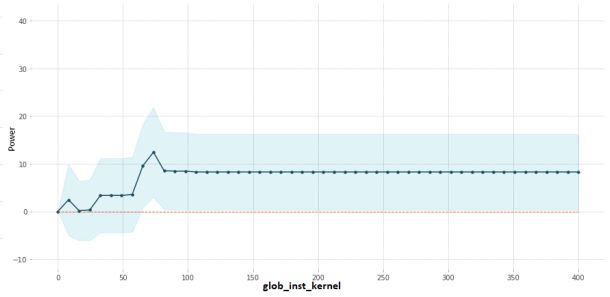

Power
avg_shar_lat

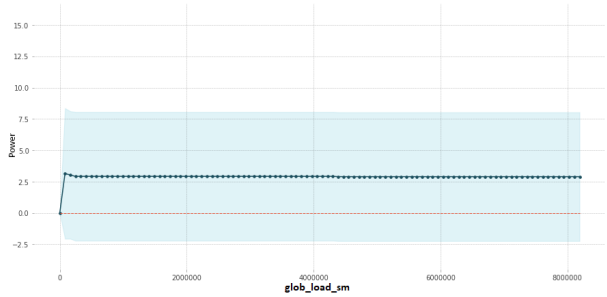

Power
block_size

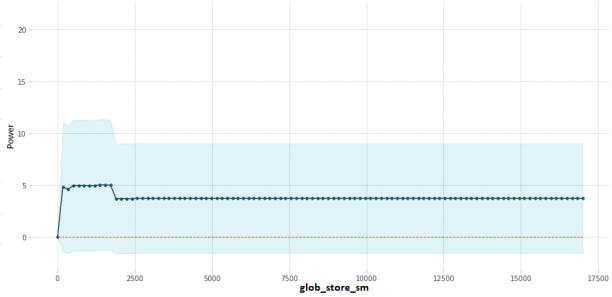

Power
branch


Power
cache_penalty


Power
comp_inst_kernel


Power
glob_inst_kernel


Power
glob_load_sm


Power
glob_store_sm

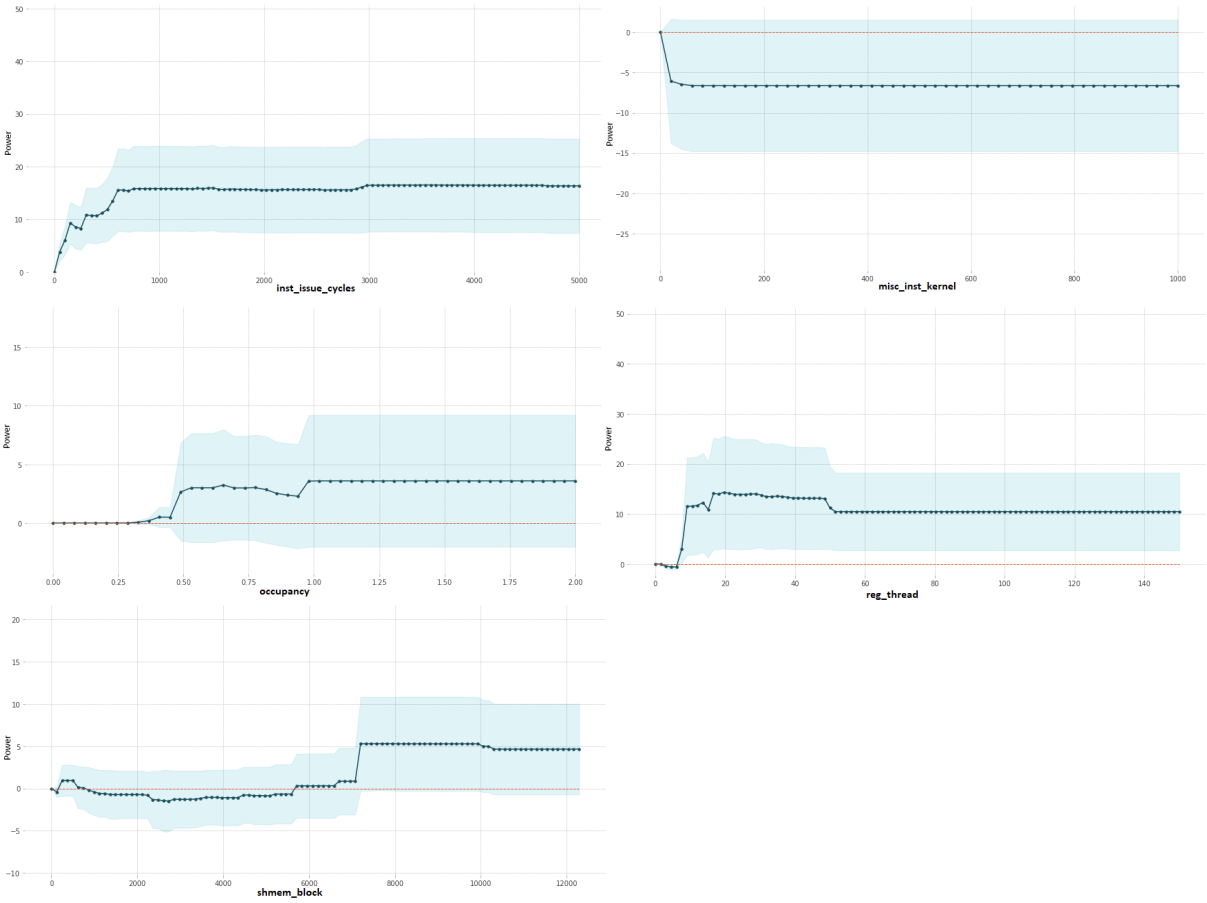


图 5.17：XGBoost 所有特征的部分依赖图（PDP）

三个配置的线程数相同，因此这三个配置的大多数特征也相同。然而，观察发现，c1 的功耗为 109.91 W，c2 为 112.08 W，c3 为 112.49304 W。在这种情况下，我们的异常值检测方法应将 c1 标记为异常值。

如果存在此类离群值，它们可能会影响模型的准确率。因此，我们希望进行异常值检测，以避免任何可能影响模型预测准确率的异常值。如前所述，特征与预测变量之间不存在任何线性关系。此外，由于特征数量庞大，无法通过目视检查进行异常值检测。因此，我们采用 HDBSCAN （带噪声的应用程序层次密度空间聚类） [166] ，这是一种基于密度的

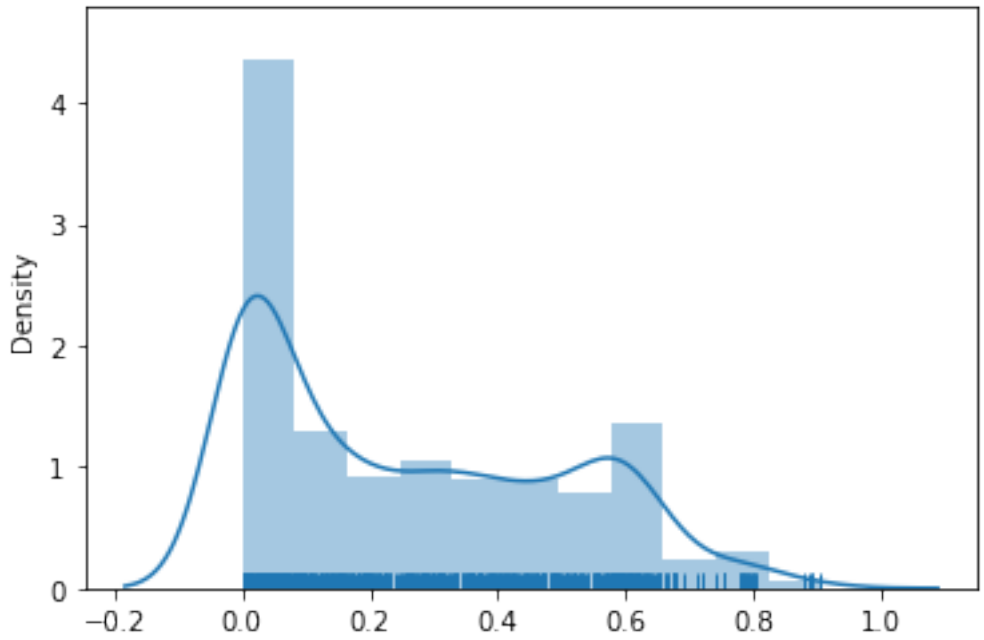


图 5.18：使用 HDBSCAN 的离群点得分密度

聚类算法 [167, 168]。HDBSCAN 算法会在不同的 epsilon 值下执行 DBSCAN，并整合结果，以找到在 epsilon 范围内稳定性最佳的聚类。因此，HDBSCAN 能够发现密度各异的簇（与 DBSCAN 不同）。这使得 HDBSCAN 对参数选择更加稳健。

HDBSCAN 库支持 GLOSH 异常值检测算法，并在 HDBSCAN 聚类类中实现该算法 [169]。图 5.18 展示了使用 HDBSCAN 得到的异常值密度分数。该算法检测出 97 个异常值。然而，剔除异常值后，使用 Tesla K20 数据集的 XGBoost 的 R^2 分数低于原始数据集的结果（R^2 分数：0.8942；原始数据集的R^2 分数：0.9101）。剔除异常值后，CatBoost 的结果为R^2 分数：0.8937，而剔除异常值之前的分数更高（R^2 分数：0.9067）。因此，我们得出结论：剔除异常值并不能帮助构建更稳健的模型。所有机器学习技术在异常值检测后的结果列于表 5.11。

表 5.11：异常值剔除后机器学习技术的验证得分

模型	R^2得分	均方根误差	MAE
随机森林	0.8777	10.2092	1.8252
极端随机树	0.8768	8.2609	4.4506
梯度提升	0.8875	9.7673	3.9698
XGBoost	0.8942	5.9352	3.6002
CatBoost	0.8937	6.8369	3.9394
决策树	0.8451	10.8890	4.9394
SVR	0.6747	18.39	11.015
ANN	0.5737	14.084	9.46

5.8.3 使用并行小类分析预测模型 s

并行计算侏儒代表了并行应用程序的通信和计算模式 [14]。为进一步分析所构建模型的功耗预测结果，我们为所研究的每类侏儒选取了一组基准测试。我们使用多种启动配置运行这些基准测试，并利用训练好的机器学习模型预测其功耗。Tesla K20 设备上每类侏儒对应的平均绝对误差（MAE）见表5.12。

最佳情况：观察发现，所提出的模型在预测属于稠密线性代数、动态规划和非结构化网格 Dwarf 的基准测试功耗时效果最佳。稠密线性代数基准测试属于计算密集型应用，而动态规划和非结构化网格则受内存延迟限制。因此，我们认为该模型适用于计算密集型和内存密集型内核。

最差情况： 我们观察到，属于图遍历、分支定界和结构化网格负载的基准测试具有较高的平均绝对误差。分支定界基准测试的性能上限取决于具体问题。结构化网格基准测试具有较高的空间局部性。属于图遍历负载的基准测试包含间接表查找，且计算量较少。我们得出结论，对于具有这些特征的基准测试，我们的模型可能无法给出精确结果。

表5.12：并行计算侏儒的平均绝对误差

模型	分支定界	图遍历	稠密线性代数	结构化 Grid	非结构化 Grid	动态编程
随机森林	11.31	13.17	4.45	14.31	0.33	8.33
极端随机树	16.17	19.06	3.66	13.52	0.18	0.3
梯度提升	19.51	14.52	4.6	25.74	0.43	0.05
XGBoost	21.14	14.85	3.91	24.03	0.3	0.96
CatBoost	14.42	23.4	7.22	6.29	3.28	2.65
SVR	29.34	25.86	5.89	3.69	1.37	1.96
ANN	15.91	32.91	4.74	4.8	5.96	10.04

5.9 讨论

在本章中，我们介绍一种基于程序分析并采用机器学习算法的 GPU功耗预测模型。我们利用硬件细节生成本研究所考察的特征，无需运行应用程序。本研究考察的机器学习模型包括随机森林回归器、极端随机树回归器、梯度提升回归器、XGBoost回归器、CatBoost 回归器、支持向量回归和人工神经网络。我们在三台 GPU 机器上测试了这些模型，以验证我们的假设：Tesla K20、Tesla M60 和 Tesla V100。我们发现，与支持向量回归和人工神经网络的结果相比，基于树的回归方法能够生成最准确的功耗预测模型。我们发现指令发射周期 是各模型中最重要的特征，因为它代表所有指令都会使用的 FDS单元；因此，无论其他基准测试参数如何，它都是一个恒定的贡献因素。包含浮点指令和全局内存指令的基准测试也会消耗大量能量；因此，必须限制这些指令的使用，才能构建更节能的应用程序。应用程序中寄存器的使用也至关重要，为提高能效，应尽可能减少寄存器的使用。因此，通过“数据驱动”方法获得的最终“重要特征”集合能够很好地解释响应（功耗）的变化，且与领域专业知识相符。本研究推动了 GPU 功耗建模领域的进一步探索。我们可以利用本研究预测 GPU 的能耗。还可以进一步深入探究如何根据本研究提出的观测结果重构程序。本.....

重构应当以一种能够在不牺牲性能的情况下降低功耗的方式实现。未来的工作还包括分析并改进对属于图遍历、分支定界和结构化网格负载的基准测试的预测。

CHAPTER 6

工具设计

Design

用于分析和预测 GPU 加速应用程序性能的实现级工具十分匮乏。尽管已有针对 GPU 的性能建模方法，但其复杂性使得人们几乎不可能在编码时使用这些方法快速分析应用程序的性能，并获得易于使用且可读的反馈。这就是为什么尽管 GPU 是许多高性能计算领域中显著的性能提升工具，性能预测仍然依赖于大量基准测试，而性能瓶颈分析仍然是一个非系统化、依赖经验的过程。

GPU 开发人员的主要目标是对代码进行充分调优，以获得最佳性能，并高效利用 GPU 上可用的各种资源。然而，在高性能计算基础设施上运行的应用程序不仅应实现最佳性能，同时还需要以尽可能低的能耗完成运行。开发者必须配备相应的工具，以便在启动应用程序之前了解 GPU 的能耗情况。研究人员正在致力于开发工具，以帮助开发者实现可持续计算 [11, 12, 170]。基于类似的理念，即，

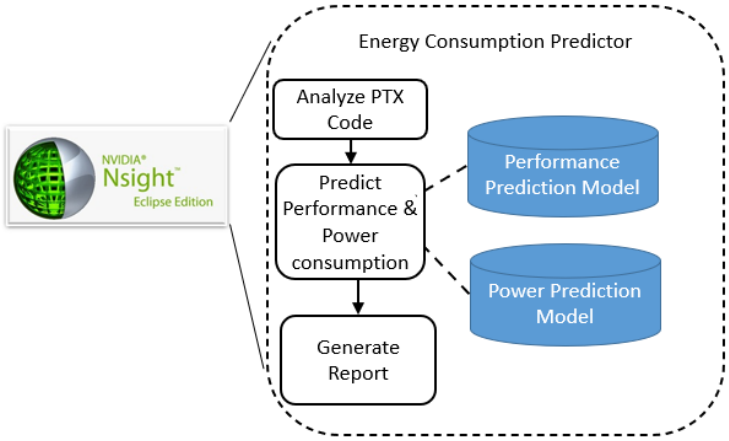


图 6.1：能耗估算工具概览

为了理解并降低能耗，我们提出了一种能耗预测工具，帮助开发人员了解 GPU 应用程序的性能和功耗，并可用于事前能耗分析。

我们开发了能耗估算工具，这是一个作为 Eclipse 插件构建的预测工具，无需运行应用程序即可预测其性能和功耗。该插件工具还提供详细报告，说明哪些特征会影响应用程序的功耗。能耗估算工具插件已针对三种 GPU 架构进行了验证：Kepler、Maxwell 和 Volta。我们相信，本论文介绍的工具将为 GPU 应用程序的开发以及了解应用程序的功耗和性能特征提供有益指导。下一节将讨论该工具的架构和使用细节。

6.1 工具架构与实现

本文介绍的工具以插件形式构建，利用 NVIDIA NSight IDE 编译 CUDA程序。该插件会分析 `nvcc`编译器生成的 PTX代码，并生成供预测模型使用的特征。图 6.1 展示了能耗估算工具的执行流程。功耗预测模型会生成

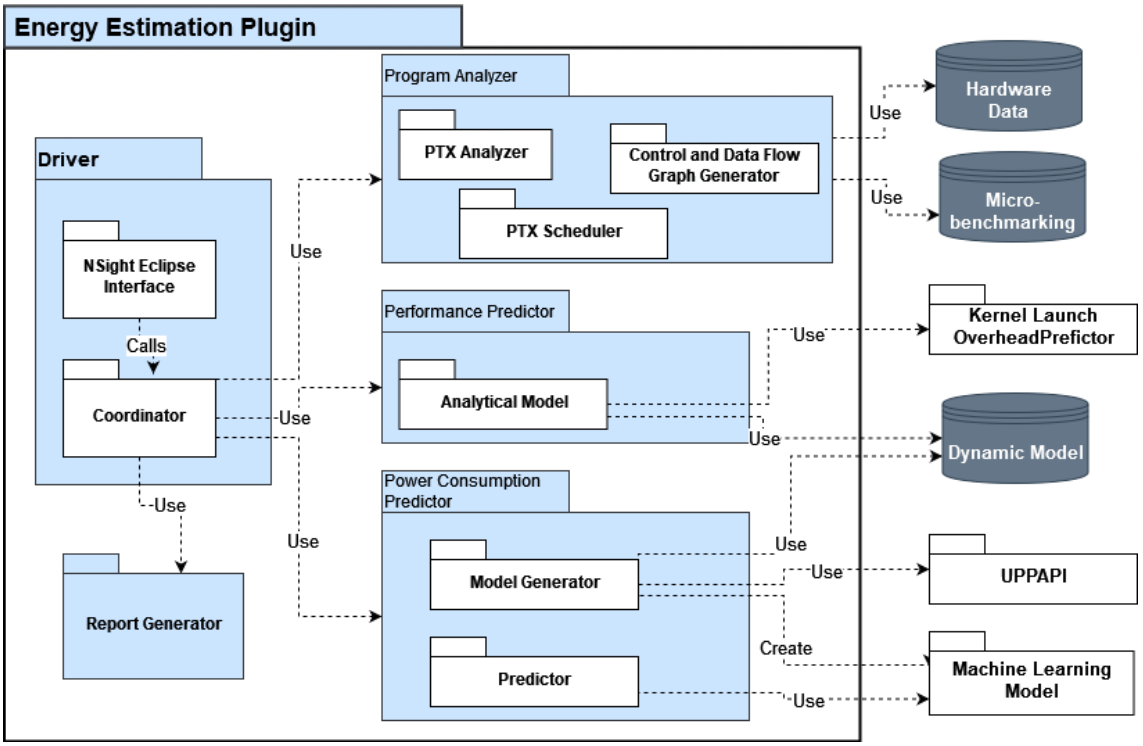


图 6.2：能耗估算工具的功能架构

一份功耗报告，其中包含对功耗预测有贡献的特征详情。图 6.2 详细展示了能耗估算工具的功能架构。该插件由图 6.2 所示的多个模块组成。下面将逐一详细讨论这些模块。

6.1.1 程序分析器

程序分析器由三个子模块组成。首先，PTX 分析器会解析 PTX 代码并提取指令信息。然后，控制与数据流生成器利用这些信息构建控制数据流图（CDFG） [116]。最后，使用 PTX 调度器模块生成 CDFG 图，以提取表示图形处理器运行时行为的特征。

PTX 调度器使用表示设备硬件配置和执行限制的设备特征。调度器还需要应用程序特征-

无法通过静态分析推导的特征。这些包括网格大小、块大小和循环迭代次数。PTX 调度器使用第5章中介绍的算法生成一组特征。用户需要提供 GPU 的计算能力 、每个线程块的线程数以及使用 CUDA 占用率计算器计算的占用率 值。PTX 分析器模块根据 PTX 代码和微基准测试数据生成所有其他特征。

6.1.2 执行时间预测器

该模型利用 PTX 分析器提供的详细信息对指令进行调度，以计算 CUDA 内核的性能。该模块结合内核启动开销模型和动态分析模型来预测执行时间。有关我们性能预测方法的详细信息，请参见第4章第4.4节。

6.1.3 功耗预测器

该模块负责预测 CUDA 内核消耗的功耗。我们采用机器学习方法，利用程序分析器模块提取的特征构建并训练预测模型。有关功耗预测模型的详细信息，请参阅第5章。

6.1.4 报告生成器

根据采用所选机器学习技术进行的预测，插件分别以 μ 秒和瓦特为单位，报告所研究应用程序的性能和功耗。能量通过将两个模块得到的结果相乘来计算。

我们进一步使用 LIME 工具 [171] 分析功耗预测。LIME 是一种与模型无关的工具，可向用户详细说明为何得出此预测，或哪些

变量促成了该预测。我们坚信，这能让应用程序开发者更深入地了解应用程序的功耗，从而重构代码，使其更加节能。

根据所选的回归技术以及用户提供的其他应用程序详情，能耗估算工具会预测 GPU应用程序的性能和功耗。图 6.4 展示了结果窗口。我们利用 LIME 进一步剖析功耗预测工具所做的预测，并生成详细报告。我们将在第 6.2.5 节通过一个案例研究，进一步详细讨论能耗估算工具的工作原理。

6.2 插件使用详情

6.2.1 先决条件

NVIDIA Nsight Eclipse IDE [1] 启动后，插件会调用一个方法，检查用户的计算机上是否安装了`NVCC` 。如果没有安装，插件会冻结并提示用户安装`NVCC`。如果已安装`NVCC` ，插件便会将一些可执行文件提取到临时位置。这些可执行文件用于查询用户硬件并在其上运行微基准测试。提取这些文件的方法会在 Eclipse 启动时调用。图 6.3 展示了插件窗口。

6.2.2 插件设置

现在，用户可以进入“插件设置”阶段。在此阶段，系统会运行可执行文件，以查询并运行针对用户图形处理器的微基准测试。首次执行时，这一阶段可能需要一些时间。这些可执行文件的输出会以文本文件的形式存储在用户的计算机上。首次完成后，准备

[1]https://developer.nvidia.com/nsight-eclipse-edition

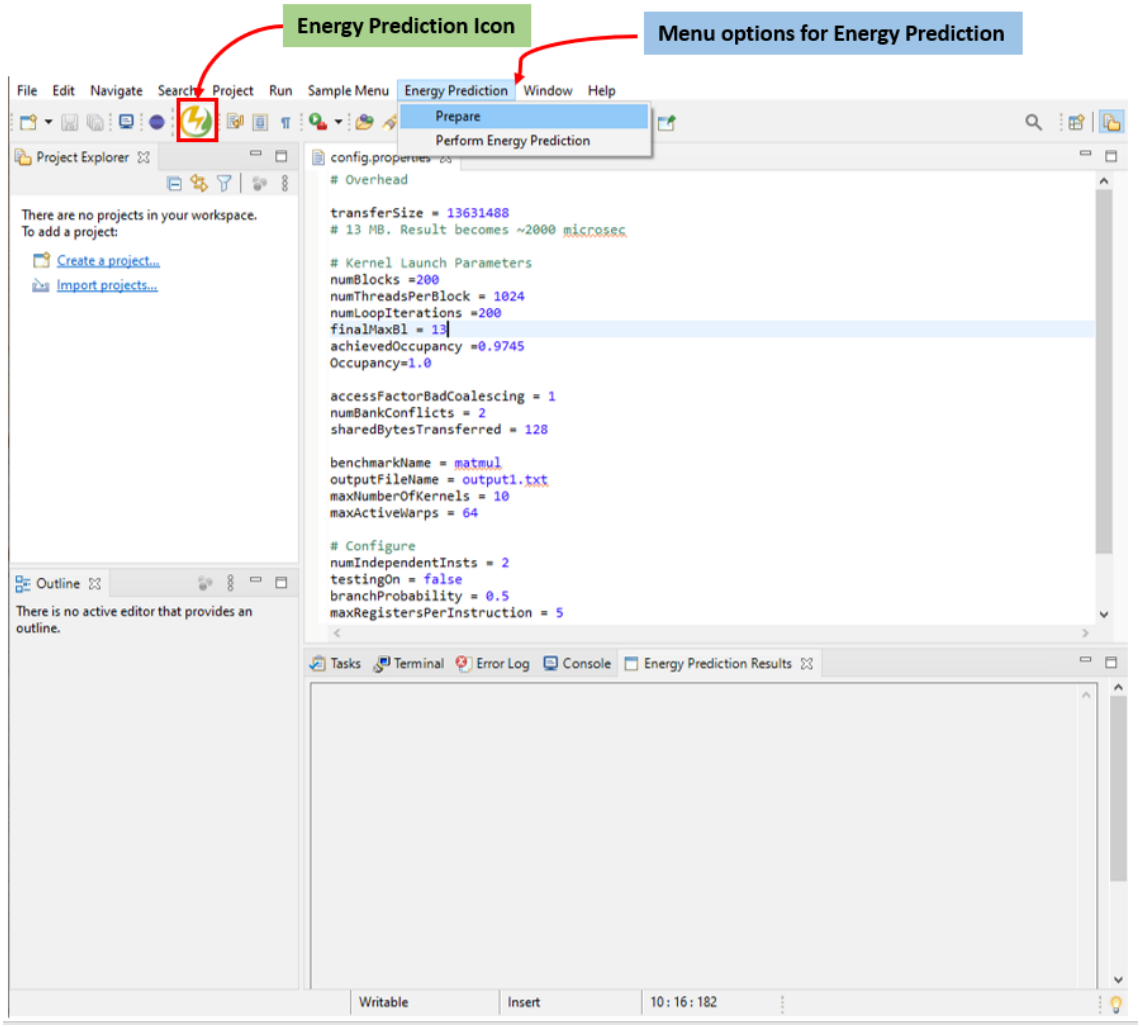


图 6.3：能耗估算工具

阶段只需读取这些文本文件，因此速度更快。请注意，插件会在此阶段生成一些模型，但不会将其存储为文本文件。因此，每次用户想使用该插件时，都必须至少“设置”一次。

6.2.3 用户输入

完成“准备”阶段后，用户可以在配置文件中提供基准测试所需的详细信息。这些信息包括图 6.3 所示的启动参数和循环迭代次数。我们还记录了其他一些

我们认为这些信息对插件未来的发展具有参考价值。

6.2.4 结果生成

填写完所需参数后，用户可以在菜单栏中选择“能耗估算>执行能耗估算”来进行能耗估算。用户也可以点击菜单栏中的插件图标来执行此任务。结果将显示在自定义控制台中。图 6.4 展示了在窗口中生成的结果。对于每项功耗预测结果，都会生成 LIME 分析图，并在对话框中显示生成图表的位置，如图 6.6 所示。

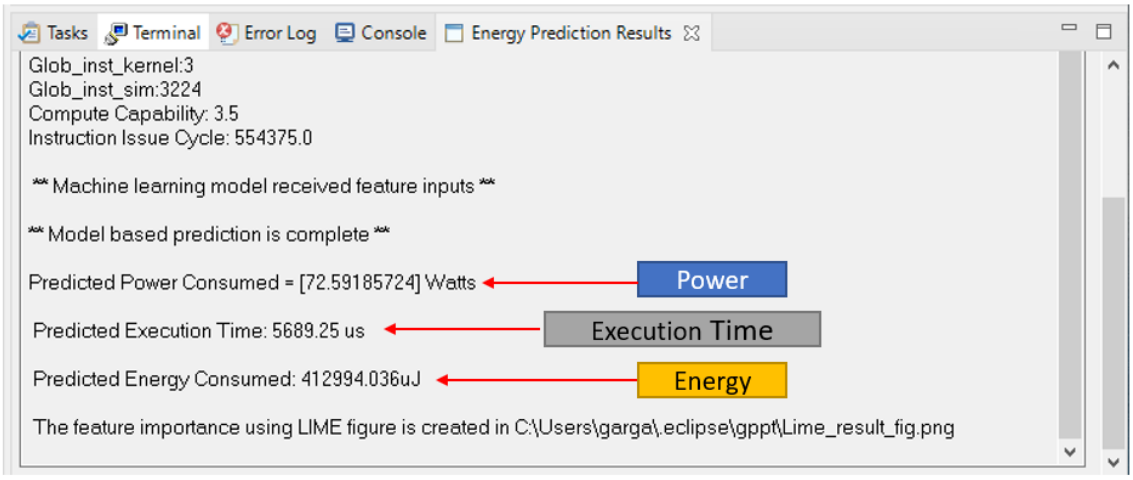


图 6.4：能耗估算结果

6.2.5 案例研究：矩阵乘法

矩阵乘法是 GPU 计算中常见的问题，应用于多个领域。它包含全局和共享内存指令，是一种内存受限应用。我们已在第4章中展示了矩阵乘法的执行时间预测结果。表 6.1 展示了在所研究的三种架构上运行矩阵乘法时的能耗预测结果，每次运行使用 1024 个线程块，每个线程块包含 1024 个线程。

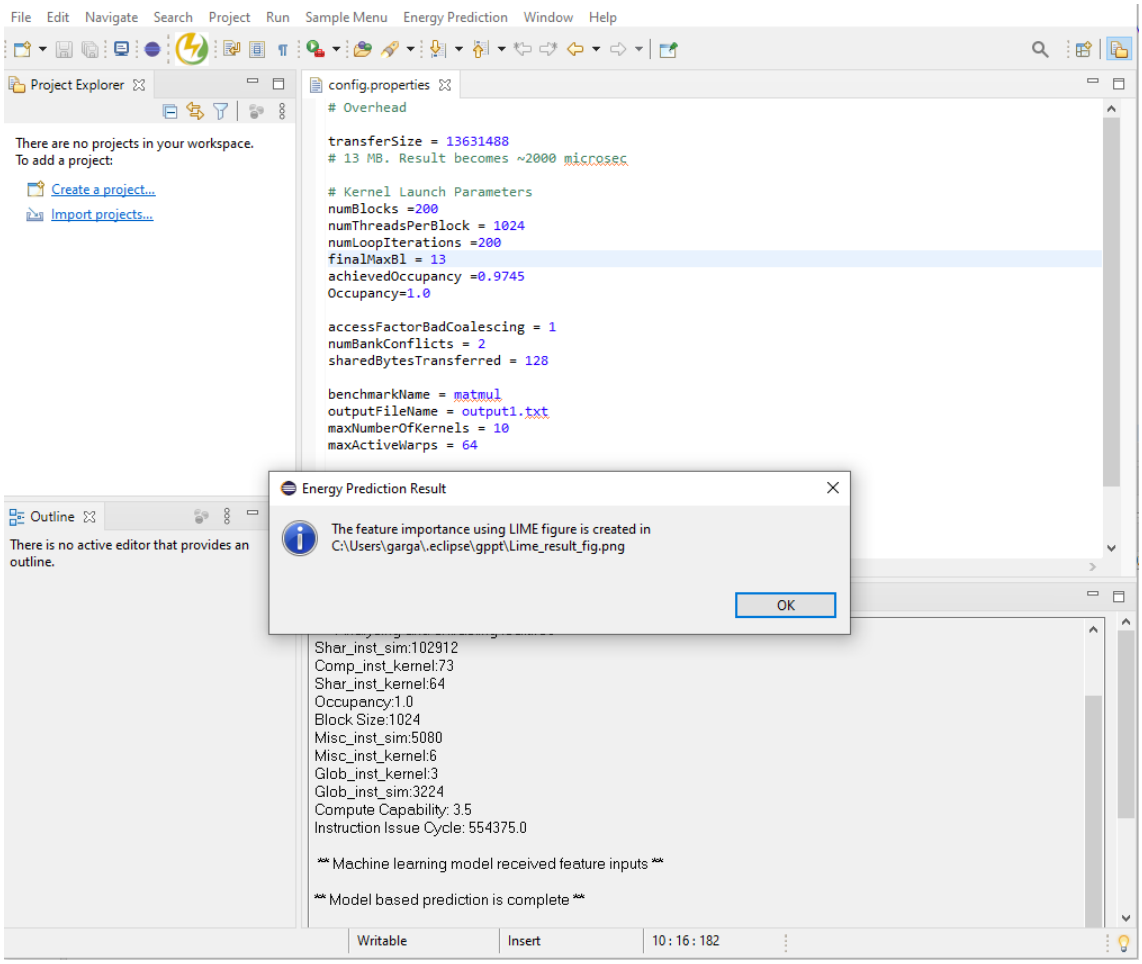


图 6.5：LIME 插件中的输出生成

表 6.1：不同架构下矩阵乘法的结果

	实际执行			预测结果		
	执行时间（微秒）	功耗(W)	能量（μJ）	执行时间（微秒）	功耗（W）	能量（μJ）
Tesla K20	6997.75	83.17	582002.86	5689.25	83.28	473800.74
Tesla M60	4917.4	84.99	417929.82	5617.52	86.21	484286.39
Tesla V100	8514.9	137.26	1168755.17	8945.25	138.16	1235875.74

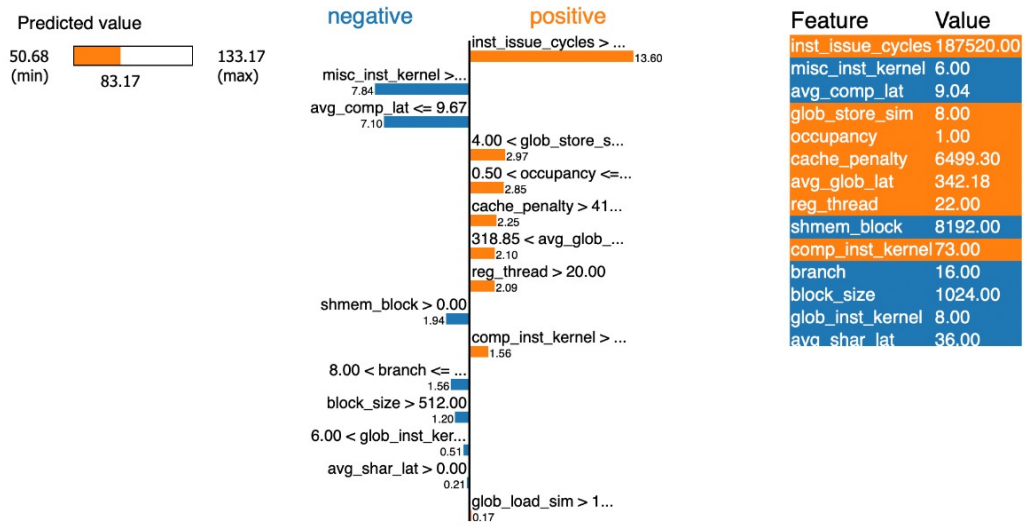


图 6.6：使用 LIME 进行功耗预测分析

我们还在图 6.6 中展示了 XGBoost 在 Kepler 架构上进行矩阵乘法预测的 LIME 报告。LIME 详细说明了哪些特征对预测产生了影响。如图 6.6 所示，y 轴表示特征，x 轴表示各特征的相对影响程度。正值（橙色）表示该特征支持或提高预测值。相反，负值（蓝色）表示该特征会对预测值产生负面影响或使其降低。在此案例中，从 shmem_block 和 avg_shar_lat 特征的贡献可以看出，共享内存指令对功耗的影响并不显著。为开发功耗更低的应用程序，开发者需要尽可能多地使用共享内存指令，而非全局内存指令。avg_glob_lat 和 glob_store_sim 对功耗预测的贡献，体现了全局内存指令对功耗的影响。每线程寄存器数也对功耗预测有显著贡献；因此可以得出结论，在此基准测试中，减少寄存器使用量有助于降低功耗。减少已执行指令的数量至关重要，因为如第5章所示，inst_issue_cycles 特征对功耗预测的贡献最大。

表 6.2：所选基准测试的性能与功耗结果比较

序号	基准测试	内核名称	实际性能	预测性能	实际功耗	预测功耗
1	黑洞	BlackScholes	547.12	819.18	65.40066	67.28
2	CFD	CFD初始化	29.15	34.86	68.0969	73.34959
3	CFD	CFD计算步骤F	55.39	71.28	89.8506	91.64792
4	CFD	CFD通量	820.19	1,519.40	48.1210	55.588272
5	可分离卷积	可分离卷积 C_	1,165.50	985.57	61.4910	102.682365
6	可分离卷积	可分离卷积 R_	1,665.50	2,801.80	59.7500	109.073555
7	高斯	GaussF2	326.63	147.93	99.92654	101.66327
8	LUD	LUDPeri	15.95	20.24	63.539	61.495842
9	矩阵乘法	矩阵乘法	6,997.75	5,825.05	95.5720	95.3723
10	归并排序	MergeMS_generateSample	183.33	150.60	89.7283	116.52137
11	归并排序	mergeRanksAndIndices	1,114.10	910.58	59.2251	76.29441
12	归并排序	mergeSortSharedKernel	1,114.10	910.58	113.4631	69.39528
13	NW	NW	22.24	16.99	49.6300	50.31136
14	Scan	共享内存独占扫描	361.83	678.01	122.45502	126.607155
15	Srad	Srad	11.26	14.84	56.3509	57.872116
16	转置	Tpose_optimized	77.35	64.32	75.8094	76.264854
17	转置	Tpose_粗粒度	74.88	58.65	76.7434	76.43441
18	转置	Tpose_朴素方法	62.15	53.03	84.9380	88.74651
19	转置	Tpose_合并的	78.05	234.03	77.0491	83.31207
20	卷积纹理	convolutionRowsKernel	620.85	794.89	65.556	72.84

6.3 功耗与执行时间预测结果比较

本节将执行时间预测误差较大的结果与对应的功耗预测结果进行了比较。为此，我们选取了 20 个执行时间预测误差超过 15% 的基准测试。图 6.7 展示了这些结果的比较，表 6.2 列出了比较结果的详细信息。如图所示，除 Convolution Seperable 基准测试外，其他所有基准测试的执行时间预测误差与功耗预测误差均不一致。与执行时间预测模型相比，功耗预测模型似乎表现更好。为了解两种模型均无法对 Convolution Separable 基准测试作出准确预测的原因，我们对该基准测试进行了详细分析，发现对于 convolutionRowsKernel，已发射指令（18420093）与已执行指令（11943936）之间的差异相当大。这可能是由于存在模型未能预测到的硬件级优化。与执行时间预测模型相比，功耗预测模型似乎表现更好。

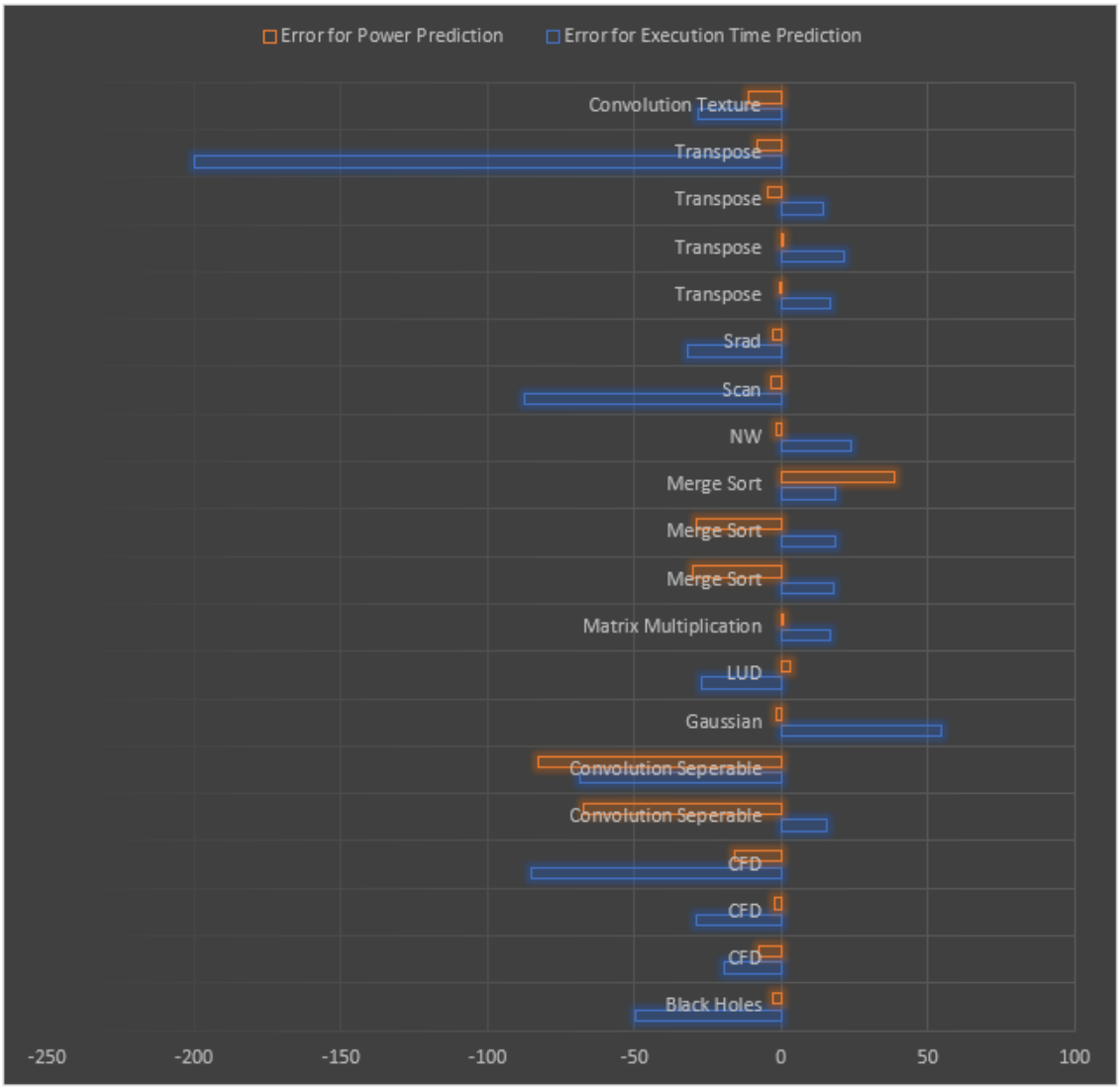


图 6.7：功耗与执行时间预测结果对比

6.4 讨论

我们介绍了一款 Eclipse 插件，它结合了本论文开发的执行时间和功耗预测模型，用于预测 CUDA 内核的能耗。此能耗预测插件可在此处获取[2]。它为 GPGPU 开发人员提供了易于使用的设计辅助工具，可帮助他们在不执行应用程序的情况下了解其能耗。该插件的设计使其能够

[2]https://github.com/santonus/energymodel

扩展到多种架构，并分析不同类型的 CUDA 基准测试。我们坚信，这款工具是推动 GPU 成为可持续计算替代方案的创新一步，而且还可以进一步扩展，用于查找代码中的能耗热点。随后应重构这些能耗热点，以构建节能型应用。

CHAPTER 7

结论与未来方向

Through

在本学位论文中，我们尝试通过静态分析和一次性动态分析来回答有关

GPU 应用程序性能和功耗预测的研究问题。现在，我们将总结本研究工作的意义，并提出未来研究的方向，以此作为结论。

7.1 总结

通过研究应用程序的性能和功耗，可以分析其能耗。因此，我们将研究目标分为两个子问题。在第一个子问题中，我们基于对 CUDA 内核进行静态和动态分析，提出了一种适用于 NVIDIA GPU 架构的解析式性能预测模型。静态分析包括分析 PTX 代码，以构建控制流和数据流信息。此外，我们还执行动态分析，以一次性获取运行时行为特征，例如指令延迟、内存吞吐量和内核启动开销。利用通过静态和动态分析获得的信息，我们

对内核进行调度，以估算其执行时间。

第二个子问题是提出一种基于程序分析的GPU功耗预测模型。我们利用硬件细节生成本研究考察的特征，而无需运行应用程序。我们采用机器学习算法从这些特征中进行学习并作出预测。本研究考察的机器学习模型包括随机森林回归器、极端随机树回归器、梯度提升回归器、*XGBoost*回归器、*CatBoost* 回归器、支持向量回归和人工神经网络。我们在三种图形处理器设备 Tesla K20、Tesla M60 和 Tesla V100 上测试了这些模型，以验证我们的假设。我们发现，与支持向量回归和人工神经网络相比，基于树的回归方法能够构建出最准确的功耗预测模型。研究发现，*inst_issue_*周期 是各模型中最重要的特征，因为它代表所有指令都会使用的 FDS 单元；因此，无论其他基准测试参数如何，它始终是一个恒定的影响因素。包含浮点指令和全局内存指令的基准测试也会消耗大量功耗，因此必须限制这些指令的使用，才能构建更节能的应用程序。应用程序中寄存器的使用也至关重要，为提高能效，应尽可能减少寄存器的使用。因此，通过“数据驱动”方法得到的最终“重要特征”集能够很好地解释响应（功耗）的变化，并且与领域专业知识相符。

我们坚信，可以进一步扩展这些来自性能和功耗预测的观测值，构建一个能够找出能耗热点的模型。随后，应用程序开发者可以重构代码，为 GPU 构建节能型应用程序。

7.2 未来工作的方向

以下是我们预测方法的一些可能改进方向：

- 我们的方法不进行循环迭代预测。目前，我们依赖

用户提供循环迭代次数，因为内核中的大多数循环往往是数据依赖且非仿射的 [124]。如何预测非仿射基准测试的循环迭代次数仍有待解决。

- 线程束中的分支发散是误差来源，因为需要累加所有已执行指令的延迟。这里假设每个线程束中的每个分支至少会被一个线程执行。这意味着，执行该分支的线程会执行指令，而其余线程则执行空操作。如果某个分支在一次线程束执行期间从未被执行，最终的指令执行时间可能不准确。

- 我们尝试使用一个简单的参数化方程对缓存惩罚进行建模。缓存未命中的建模颇具挑战性，迄今为止，我们尚未发现任何成熟的先验缓存未命中模型。在未来的工作中，我们计划更深入地研究图形处理器中的缓存未命中现象。

- 由于寄存器使用量会显著影响内核的功耗和性能 [135, 172]，因此需要通过分析使用不同数量寄存器的内核来了解其影响。我们计划在未来的工作中，考虑寄存器利用率对功耗和性能预测的影响。

- 我们通过全局内存惩罚因子对合并内存访问的影响进行了建模。虽然我们考虑了内存访问合并，但模型中尚未考虑非合并内存访问。此外，我们也尚未探究是否可以仅通过对内核进行静态分析来确定合并或非合并的访问模式。

- 在功耗建模中加入更多架构特定属性，例如在功耗预测中使用 CUDA 内核动态分析所生成的特征。例如，纳入

内存访问模式、共享内存库冲突的详细信息，以及能够更好地捕捉缓存行为的模型。

• 我们需要研究并分析如何利用本研究中提出的观测值对程序进行重构。这种重构应当能够有效降低能量消耗，同时不影响性能。与此同时，这种重构不应对性能产生负面影响，否则将违背使用性能加速型机器的初衷。

• 我们需要分析并行计算侏儒的通信与计算模式同性能和功耗预测之间的关系。例如，在功耗预测中，应研究并改进对属于图遍历、分支定界和结构化网格负载的基准测试的预测。分支定界基准测试的性能上限因问题而异。结构化网格基准测试具有较高的空间局部性。属于图遍历侏儒的基准测试包含间接表查找，且计算量较少。我们需要深入研究具有这些特征的基准测试，并制定改进预测的方法，将其纳入模型。

• 我们可以研究一种用于执行时间预测的混合建模方法，包括在图形处理器上运行一次基准测试并收集其运行时参数。我们可以利用这些数据预测该基准测试在任何其他启动配置下的执行时间。

• GPU 在构建注重性能的机器学习库（NVIDIA cuDNN [173], Tensorflow [174] ）方面非常受欢迎，而这些库有时会在大数据应用程序中连续运行数天。我们可以针对流行的机器学习专用库，为 GPU 构建能耗预测模型。这将对可持续机器学习计算作出重要贡献。

• 我们尚未考虑应用程序使用显式同步原语的少数情况。这些作用于线程束或线程块的强制同步操作可能会影响分析方法预测运行时间的能力。我们希望在模型中对这种行为进行建模。

• 新一代 CUDA 硬件引入了更多架构和编程特性，例如张量核心、多内核执行、特殊指令等。我们需要研究并纳入这些特性的影响。

• 我们希望采用自适应学习方法，利用强化学习和迁移学习等机器学习技术进行功耗预测。通过这些技术，系统将在应用程序运行时持续学习。

• 除了 NVIDIA GPU 之外，还有许多其他流行的多核和众核架构，例如 Xeon Phi 和 AMD GPU。我们可以将所提出的方法应用于其他多核和众核架构，并观察其可用性和有效性。

• 最后，基于静态分析的功耗和性能模型通常面临一个问题：我们利用 PTX 代码进行先验预测。然而，在使用 PTX 代码时，我们依赖一个有时难以成立的假设，即列表中的所有指令都会执行。静态分析预测模型的缺点是无法事先知道将执行多少条指令。理论结果已证明这一点无法解决——例如，图灵关于停机问题的研究证明，无法判定给定程序和输入是否会永远运行。因此，我们需要了解如何基于 PTX 代码，更准确地确定基于静态分析的模型中已执行指令的数量。

参考文献

[1] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, 等，“并行计算研究领域：伯克利视角”，技术报告，技术报告 UCB/EECS-2006-183，电气工程与计算机科学系，加州大学伯克利分校，2006年。[2] E. Gianniti, L. Zhang 和 D. Ardagna，“基于 GPU 的深度学习应用程序性能预测”，载于国际计算机体系结构与高性能计算研讨会（*SBAC-PAD*），2018年。[3] M. Lattuada, E. Gianniti, D. Ardagna 和 L. Zhang，“GPU 即服务系统中深度学习应用程序训练的性能预测”，集群计算，第 25 卷，第 1279–1302 页，2022年。[4] S. Huang, S. Xiao 和 W. Feng，“图形处理器在科学计算中的能效”，载于*IEEE* 并行与分布式处理国际研讨会（*IPDPS*），2009年。[5] A. Romero，“Tesla AI Day 2021 回顾——第 3 部分：Project Dojo。Tesla 的新超级计算机。”
h
t
t
p
s://
t
o
w
a
r
d
s
d
a
t
a
s
cience.co
m/tesla-ai-day-2021-review-part-3-project-dojo-teslas-new-supercomputer-715d102dbb29，2021年。[6] R. A. Bridges, N. Imam 和 T. M. Mintz，“理解 GPU 功耗：性能分析、建模与仿真方法综述”，*ACM* 计算调查，第 49 卷，第 41:1–41:27 页，2016。[7] S. Mittal 和 J. S. Vetter，“GPU 能效分析与改进方法综述”，*ACM* 计算调查，第 47 卷，第 19:1–19:23 页，2014。[8] M. Pi Puig, L. De Giusti 和 M. Naiouf，“GPU 是非绿色计算设备吗？”，计算机科学与技术杂志，第 18 卷，第 e17 页，2018年。[9] C. Collange, D. Defour 和 A. Tisserand，“从软件角度看 GPU 的功耗”，载于计算机科学讲义，第 5544 卷，第 914–923 页，2009年。[10] M. Couto, T. Carção, J. Cunha, J. P. Fernandes 和 J. Saraiva，“检测 Android 应用程序中的异常能耗”，载于巴西编程语言研讨会，第 77–91 页，2014年。

[11] M. Couto, D. Maia, J. a. Saraiva 和 R. Pereira，“论能源债务：管理不断演进的软件的能耗”，载于第三届国际技术债务会议论文集，2020。[12] G. S. Foundation，“软件碳强度（SCI）规范。”https://greensoftware.foundation/projects/，2021年。[13] S. Madougou, A. Varbanescu, C. de Laat 和 R. van Nieuwpoort，“GPGPU 性能建模工具综述”，并行计算，第 56 卷，第 18–33 页，2016。[14] K. Asanovic, R. Bodik，等，“并行计算领域概览”，*ACM* 通讯，第 52 卷，第 56–67 页，2009年。[15] F. Nielson, H. R. Nielson 和 C. Hankin，程序分析原理。施普林格科学与商业媒体，2004年。[16] A. A. Kleimenov 和 N. N. Popova，“一种预测 GPU 程序执行时间的方法”，*Computational nanotechnology*，第 8 卷，第 38–45 页，2021年。[17] S. Sahni 和 V. Thanvantri，“性能指标：聚焦运行时”，*IEEE* 并行与分布式技术：系统与应用，第 4 卷，第 43–56 页，1996年。[18] K. Zhou, X. Meng, R. Sai 和 J.Mellor-Crummey，“GPA：基于指令采样的 GPU 性能顾问”，载于*2021 IEEE/ACM* 代码生成与优化国际研讨会（*CGO*），2021年。[19] W. S. Hui Chen，“功率测量与性能分析：最新技术水平”，能源感知与绿色计算手册，2012年。[20] S. Song, C. Su, B. Rountree 和 K. Cameron，“新兴 GPU 架构上功耗性能效率的简化且准确模型”，载于国际并行与分布式处理研讨会（*IPDPS*），2013年。[21] S. Benedict, R. R.S. 和 S. A. Alex，“使用动态回归模型预测 CUDA 应用程序的能耗与性能”，载于印度软件工程会议（*ISEC*），2016。[22] A. Bakhoda, G. L. Yuan, W. W. L. Fung, H. Wong 和 T. M. Aamodt，“使用详细的 GPU 模拟器分析 CUDA 工作负载”，载于*IEEE* 系统与软件性能分析国际研讨会（*ISPASS*），2009年。[23] J. Leng, T. Hetherington, A. ElTantawy, S. Gilani, N. Kim, T. Aamodt 和 V. Janapa Reddi，“GPUWattch：实现 GPGPU 能源优化”，*ACM SIGARCH*计算机体系结构新闻，第 41 卷，第 487–498 页，2013年。

[24] Y. Wang 和 N. Ranganathan，“一种面向 GPU 的指令级能耗估算与优化方法”，载于国际计算机与信息技术会议（*CIT*），2011年。[25] N. Corporation，“性能分析器用户指南。”http://docs.nvidia.com/cuda/profiler-users-guide/，2022年。[26] B. Purnomo、N. Rubin 和 M. Houston，“Ati 流性能分析器：一种用于优化 Ati Radeon 图形处理器上 OpenCL 内核的工具”，载于*ACM SIGGRAPH* 海报，2010年。[27] G. Malhotra、S. Goel 和 S. R. Sarangi，“GPUTejas：一种面向 GPU 架构的并行模拟器”，载于国际高性能计算会议（*HiPC*），*IEEE*，2014年。[28] S. Malik 和 F. Huet，“虚拟云：出租已租用的资源”，载于互联网技术与安全交易国际会议（*ICITST*），2011年。[29] S. Hong 和 H. Kim，“一种集成的图形处理器功耗与性能模型”，载于第*37*届年度国际计算机体系结构研讨会（*ISCA*）论文集，*ACM*，2010年。[30] M. J. Ikram、M. E. Saleh、M. A. Al-Hashimi 和 O. A. Abulnaja，“研究块大小变化对 GPU 内核功耗与能耗的影响”，《超级计算杂志》，第1–21页，2022年。[31] D. Boughzala、L. Lefèvre 和 A.-C. Orgerie，“使用 SimGrid 预测 CUDA 内核的能耗”，载于国际计算机体系结构与高性能计算研讨会（*SBAC-PAD*），2020年。[32] N. Corporation，“CUDA C++ 编程指南（设计指南）v11.7.0。”https://docs.nvidia.com/cuda/pdf/CUDA_C_编程_指南.pdf，2022年。[33] H. Wong、M.-M. Papadopoulou、M. Sadooghi-Alvandi 和 A. Moshovos，“通过微基准测试揭示 GPU 微体系结构”，载于*IEEE* 系统与软件性能分析国际研讨会（*ISPASS*），2010年。[34] M. Burtscher、I. Zecena 和 Z. Zong，“利用 K20 内置传感器测量 GPU 功耗”，载于通用图形处理器处理研讨会论文集，*ACM*，GPGPU-7，（美国纽约），第28:28–28:36页，2014年。[35] M. Amaris、R. Camargo、M. Dyab、A. Goldman 和 D. Trystram，“使用机器学习和解析建模进行 GPU 执行时间预测的比较”，*IEEE* 网络计算与应用国际研讨会（*NCA*），2016。[36] K. Fan、B. Cosenza 和 B. Juurlink，“面向能耗与性能的可预测 GPU 频率调节”，载于国际并行处理会议（*ICPP*），2019年。

[37] J. Guerreiro、A. Ilic、N. Roma 和 P. Tomas，“面向多域电压频率缩放的通用图形处理器功耗建模”，发表于国际高性能计算机架构研讨会（*HPCA*），2018年。[38] J. Lim、N. B. Lakshminarayana、H. Kim、W. Song、S. Yalamanchili 和 W. Sung，“使用 McPAT 对 GPU架构进行功耗建模”，发表于*ACM*电子系统设计自动化汇刊（*TODAES*），第19卷，第1–24页，2014年。[39] J. Lucas、S. Lal、M. Andersch、M. Alvarez-Mesa 和 B. Juurlink，“单芯片如何造成巨额电费”，发表于系统与软件性能分析国际研讨会（*ISPASS*），2013年。[40] A. Kerr、E. Anger、G. Hendry 和 S. Yalamanchili，“Eiger：用于统计性能模型自动化综合的框架”，发表于高性能计算国际会议（*HIPC*），2012年。[41] R. Kumar、D. Marinov、D. Padua、M. Parthasarathy、S. Patel、D. Roth、M. Snir 和 J. Torrellas，“伊利诺伊大学的并行计算研究：UPCRC议程”，伊利诺伊大学，香槟市，2008年。[42] S. Wienke、T. Cramer、M. S. Müller 和 M. Schulz，“量化生产力——迈向高性能计算开发工作量估算”，发表于高性能计算、网络、存储与分析国际会议（*SC*），2015年。[43] N. Corporation，“CUDA示例 v11.7.0。”`https://docs.nvidia.com/cuda/cuda-samples/index.html`，2022年。[44] N. Bell 和 J. Hoberock，“Thrust：面向 CUDA 生产力的库”，载于*GPU*计算宝典（翡翠版），第359–371页，爱思唯尔，2012年。[45] E. Rubin、E. Levy、A. Barak 和 T. Ben-Nun，“MAPS：利用设备级内存抽象优化大规模并行应用程序”，发表于*ACM*架构、代码与优化汇刊，第11卷，第44:1–44:22页，2014年。[46] Y.-P. You、H.-J. Wu、Y.-N. Tsai 和 Y.-T. Chao，“VirtCL：用于 OpenCL 设备抽象与管理的框架”，发表于*ACM SIGPLAN*并行编程原理与实践研讨会（*PPoPP*），2015年。[47] M. Steuwer、M. Friese、S. Albers 和 S. Gorlatch，“引入并实现用于多GPU系统编程的 allpairs骨架”，发表于《国际并行编程杂志》，第42卷，第601–618页，2013年。[48] R. Marques、H. Paulino、F. Alexandre 和 P. D. Medeiros，“用于协调GPU计算的算法骨架框架”，发表于欧洲并行处理会议，2013年。[49] J. Kurzak、P. Luszczek、M. Faverge 和 J. Dongarra，“适用于配备加速器的多核系统的部分选主元LU分解”，发表于*IEEE*并行与分布式系统汇刊（*IEEE TPDS*），第24卷，第1613–1621页，2013年。

[50] M. J. Anderson、D. Sheffield 和 K. Keutzer，“利用 GPU 寄存器解决小型线性代数问题的预测模型”，载于国际并行与分布式处理研讨会（*IPDPS*），2012年。[51] D. Li 和 M. Becchi，“在 GPU 上部署图算法：一种自适应解决方案”，载于 *IEEE* 国际并行分布式处理研讨会（*IPDPS*），2013年。[52] S. Sarkar、S. Mitra 和 A. Srinivasan，“通过 GPU 内核的复用与重构设计复杂应用程序”，载于 国际并行与分布式处理及应用研讨会（*ISPA*），2012年。[53] R. Collobert、S. Bengio 和 J. Mariethoz，“Torch：一个模块化机器学习软件库”，技术报告 *IDIAP-RR 02-46*，*IDIAP*，2002年。[54] A. Betts、N. Chong、A. F. Donaldson、J. Ketema、S. Qadeer、P. Thomson 和 J. Wickerson，“GPU 内核验证技术的设计与实现”，*ACM*编程语言与系统汇刊，第37卷，第10:1–10:49页，2015年。[55] G. Li 和 G. Gopalakrishnan，“GPU 内核函数的可扩展基于SMT的验证”，载于 *ACM SIGSOFT*国际软件工程基础研讨会（*FSE*），2010年。[56] I. Sergey、A. Nanevski 和 A. Banerjee，“细粒度并发程序的机械化验证”，载于 *ACM SIGPLAN*编程语言设计与实现会议（*PLDI*），2015年。[57] K. Kothapalli、R. Mukherjee、M. S. Rehman、S. Patidar、P. Narayanan 和 K. Sri- nathan，“CUDA GPGPU平台的性能预测模型”，载于国际高性能计算会议（*HiPC*），2009年。[58] J. S. Kirtzic，*GPU*架构的并行算法设计模型。博士论文，德克萨斯大学达拉斯分校，2012年。[59] A. Resios，使用参数化模型进行*GPU*性能预测。硕士论文，乌得勒支大学，2011年。[60] L. Cheng 和 S. Reiji，“考虑指令级并行性和线程级并行性的 GPU 执行时间预测解析模型”，载于并行、分布式与协作处理夏季联合研讨会，2011年。[61] Y. Zhang 和 J. D. Owens，“GPU架构的定量性能分析模型”，载于*IEEE* 高性能计算机体系结构国际研讨会（*HPCA*），2011年。[62] I. Baldini、S. J. Fink 和 E. Altman，“使用机器学习根据 CPU 运行情况预测 GPU 性能”，载于 *IEEE*国际计算机体系结构与高性能计算研讨会（*SBAC-PAD*），2014年。

[63] Y. Zhang、Y. Hu、B. Li 和 L. Peng，《ATI GPU 的性能与功耗分析：一种统计方法》，载于*IEEE* 网络、体系结构与存储国际会议（*NAS*），2011年。[64] N. Ardalani、C. Lestourgeon、K. Sankaralingam 和 X. Zhu，《利用 CPU 代码预测 GPU 性能的跨架构性能预测（XAPP）》，载于国际微体系结构研讨会（*MICRO*），2015年。[65] G. Chapuis、S. Eidenbenz 和 N. Santhi，《通过并行离散事件仿真和常识进行 GPU 性能预测》，载于性能评估方法与工具国际会议，2016年。[66] C. Lehnert、R. Berrendorf、J. P. Ecker 和 F. Mannuss，《GPU 架构上 SpMV 内核的性能预测与排名》，载于欧洲并行处理会议，2016年。[67] E. Konstantinidis 和 Y. Cotronis，《面向计算受限和内存受限 GPU 内核的实用性能模型》，载于并行、分布式与基于网络的处理国际会议（*PDP*），2015年。[68] T. Remmelg、B. Hagedorn、L. Li、M. Steuwer、S. Gorlatch 和 C. Dubach，《用于模板计算 GPU 性能预测的高级硬件特征提取》，载于图形处理器通用处理年度研讨会，2020年。[69] S. Fortune 和 J. Wyllie，《随机存取机中的并行性》，载于*ACM* 计算理论年度研讨会（*STOC*），1978年。[70] K. W. Ryu，《网络模型上的高效并行算法》，技术报告，论文，马里兰大学帕克分校，美国，1990年。[71] L. G. Valiant，《一种用于并行计算的桥接模型》，*ACM* 通讯，第33卷，第103–111页，1990年。[72] D. Culler、R. Karp、D. Patterson、A. Sahay、K. E. Schauser、E. Santos、R. Subramonian 和 T. von Eicken，《LogP：迈向现实的并行计算模型》，载于*ACM SIGPLAN* 并行编程原理与实践研讨会（*PPOPP*），1993年。[73] P. Gibbons、Y. Matias 和 V. Ramachandran，《队列读—队列写异步 PRAM 模型》，《理论计算机科学》，第196卷，第3–29页，1998年。[74] P. B. Gibbons、Y. Matias 和 V. Ramachandran，《队列读—队列写 PRAM 模型：并行算法中竞争的考量》，工业与应用数学学会（*SIAM*）《计算杂志》，第733–769页，1999年。

[75] M. Amaris、D. Cordeiro、A. Goldman 和 R. Y. de Camargo，“一种用于预测 GPU 应用程序执行时间的简单 BSP 模型”，载于*IEEE* 高性能计算国际会议（*HiPC*），2015年。[76] W. Jia、K. A. Shaw 和 M. Martonosi，“Stargazer：基于自动回归的 GPU 设计空间探索”，载于*IEEE* 系统软件性能分析国际研讨会，2012年。[77] G. Wu、J. L. Greathouse、A. Lyashevsky、N. Jayasena 和 D. Chiou，“利用机器学习进行 GPGPU 性能和功耗估算”，载于*IEEE* 高性能计算机体系结构国际研讨会（*HPCA*），2015年。[78] B. C. Lee、D. M. Brooks、B. R. de Supinski、M. Schulz、K. Singh 和 S. A. McKee，“并行应用程序性能建模中的推断与学习方法”，载于并行编程原理与实践研讨会（*PPoPP*），2007年。[79] Y. Arafa、A. Badawy、G. Chennupati、N. Santhi 和 S. Eidenbenz，“PPT-GPU：可扩展 GPU 性能建模”，*IEEE* 计算机体系结构快报，第55–58页，2019年。[80] S. Hong 和 H. Kim，“一种兼顾内存级并行性和线程级并行性的 GPU 架构解析模型”，载于计算机体系结构国际研讨会（*ISCA*），2009年。[81] J. Sim、A. Dasgupta、H. Kim 和 R. Vuduc，“用于识别 GPGPU 应用程序潜在收益的性能分析框架”，*ACM SIGPLAN* 并行编程原理与实践研讨会，2012年。[82] J. Huang、J. H. Lee、H. Kim 和 H. S. Lee，“GPUMech：基于区间分析的 GPU 性能建模技术”，载于*IEEE/ACM* 微体系结构国际研讨会，2014年。[83] G. F. Diamos、A. R. Kerr、S. Yalamanchili 和 N. Clark，“Ocelot：用于异构系统中大规模同步应用程序的动态优化框架”，载于并行架构与编译技术国际会议（*PACT*），2010年。[84] S. S. Baghsorkhi、M. Delahaye、S. J. Patel、W. D. Gropp 和 W.-m. W. Hwu，“一种面向 GPU 架构的自适应性能建模工具”，载于*ACM SIGPLAN* 并行编程原理与实践研讨会（*PPoPP*），2010年。[85] V. Volkov，理解 *GPU* 上的延迟隐藏。博士论文，电气工程与计算机科学系，加州大学伯克利分校。UCB/EECS-2016-143，电气工程与计算机科学系，加州大学伯克利分校。UCB/EECS-2016-143，2016年。

[86] S. Williams, A. Waterman 和 D. Patterson, 《屋顶线：一种用于多核架构的直观可视化性能模型》，*ACM* 通讯，第 52 卷，第 65–76 页，2009年。[87] E. Konstantinidis 和 Y. Cotronis, 《利用微基准测试和硬件指标分析对 GPU 内核性能进行估算的定量屋顶线模型》，《并行与分布式计算杂志》，第 107 卷，第 37–56 页，2017年。[88] A. Li、Y. Tay、A. Kumar 和 H. Corporaal, 《Transit：一种用于多线程机器的可视化解析模型》，载于国际高性能并行与分布式计算研讨会 *(HPDC)*，2015年。[89] J. Lemeire、J. G. Cornelis 和 L. Segers, 《用于分析 GPU 特性的微基准测试：占用率屋顶线与流水线模型》，载于*Euromicro* 并行、分布式和基于网络处理国际会议 *(PDP)*，2016。[90] S. Sarkar 和 S. Mitra, 《一种在为 GPU 转换应用程序时优化分支发散的基于性能分析的引导方法》，载于第八届印度软件工程会议论文集，第 176–185 页，2015年。[91] V. Volkov 和 J. W. Demmel, 《用于调优稠密线性代数的 GPU 基准测试》，载于高性能计算、网络、存储与分析国际会议 *(SC)*，2008年。[92] X. Li 和 R. Taylor, 《AMD GPU 微基准测试套件》，载于国际并行处理会议研讨会 *(ICPP)*，2010年。[93] X. Mei 和 X. Chu, 《通过微基准测试剖析 GPU 内存层次结构》，*IEEE* 并行与分布式系统汇刊，第 28 卷，第 72–86 页，2017年。[94] Z. Jia、M. Maggioni、B. Staiger 和 D. P. Scarpazza, 《通过微基准测试剖析 NVIDIA Volta GPU 架构》，载于*arXiv*，2018年。[95] J. Lucas 和 B. Juurlink, 《Mempower：数据感知型 GPU 内存功耗模型》，载于计算机系统架构 – *ARCS 2019* (M. Schoeberl、C. Hochberger、S. Uhrig、J. Brehm 和 T. Pionteck 编)，(Cham)，第 195–207 页，施普林格国际出版，2019年。[96] M. Andersch、J. Lucas、M. A. LvLvarez-Mesa 和 B. Juurlink, 《GPU 吞吐量微架构中的延迟问题》，载于*IEEE* 系统与软件性能分析国际研讨会 *(ISPASS)*，2015年。[97] X. Ma、M. Dong、L. Zhong 和 Z. Deng, 《基于 GPU 的计算的统计功耗分析与建模》，载于*ACM SOSP* 功耗感知计算与系统研讨会 *(HotPower)*，2009年。

[98] H. Nagasaka、N. Maruyama、A. Nukada、T. Endo 和 S. Matsuoka，《使用性能计数器对 GPU 内核进行统计功耗建模》，载于国际绿色计算会议（*GREENCOMP*），2010年。[99] W. Jia、E. Garza、K. A. Shaw 和 M. Martonosi，《使用回归树进行 GPU 性能与功耗调优》，*ACM*体系结构与代码优化汇刊（*TACO*），第12卷，第1–26页，2015年。[100] V. M. Weaver、M. Johnson、K. Kasichayanula、J. Ralph、P. Luszczek、D. Terpstra 和 S. Moore，《使用 PAPI 测量能量与功耗》，载于国际并行处理会议研讨会（*ICPPW*），2012年。[101] K. Singh、M. Bhadauria 和 S. A. McKee，《通过性能计数器进行实时功耗估算与线程调度》，*ACM SIGARCH*计算机体系结构新闻，第37卷，第46–55页，2009年。[102] Q. Zhao、H. Yang、Z. Luan 和 D. Qian，《Poigem：面向程序设计的 CUDA 程序指令级 GPU 能量模型》，载于国际并行处理算法与体系结构会议（*ICA3PP*），2013年。[103] Y. Abe 等，《GPU加速系统的功耗与性能分析》，载于*USENIX*功耗感知计算与系统会议，2012年。[104] J. W. Sheaffer、D. Luebke 和 K. Skadron，《一种灵活的图形体系结构仿真框架》，载于*ACM SIGGRAPH/EUROGRAPHICS*图形硬件会议，2004年。[105] J. Chen、B. Li、Y. Zhang、L. Peng 和 J. Peir，《GPU功耗研究的树结构分析》，载于*IEEE*计算机设计国际会议（*ICCD*），2011年。[106] W. Song、S. Mukhopadhyay 和 S. Yalamanchili，《Energy Introspector：面向多核体系结构的集成功耗-可靠性-热建模的并行、可组合框架》，载于*IEEE*系统与软件性能分析国际研讨会（*ISPASS*），2014年。[107] G. Wang、Y. Lin 和 W. Yi，《内核融合：提升多线程 GPU 功耗效率的有效方法》，载于*IEEE/ACM*绿色计算与通信国际会议，2010年。[108] J. Pool、A. Lastra 和 M. Singh，《图形处理器的能量模型》，载于*IEEE*计算机设计国际会议（*ICCD*），2010年。[109] N. Corporation，《并行线程执行指令集架构7.7版》。`https://docs.nvidia.com/cuda/parallel-thread-execution/index.html`，2022年。[110] C.-A.-M. Hristea 等，《多处理器内存层次结构性能微基准测试》。博士论文，麻省理工学院，1997年。

[111] M.-M. Papadopoulou、M. Sadooghi-Alvandi 和 H. Wong，《GT200 GPU微基准测试》，多伦多大学电气与计算机工程系计算机组技术报告，2009年。[112] R. H. Saavedra-Barrera，《CPU性能评估与执行时间预测：窄谱基准测试方法》。博士论文，电气工程与计算机科学系，加州大学伯克利分校，1992年。[113] C. Z. R. Meltzer 和 C. Cecka，《C2070微基准测试》，发表于 *GPU*技术大会的海报，2013年。[114] Y. Arafa、A.-H. A. Badawy、G. Chennupati、N. Santhi 和 S. Eidenbenz，《NVIDIA GPGPU 的低开销指令延迟表征》，载于*2019 IEEE*高性能极限计算会议（*HPEC*），第1–8页，IEEE，2019年。[115] J. G. Cornelis 和 J. Lemeire，《流水线性能模型：一种适用于 GPU 的通用可执行性能模型》，载于*Euromicro*并行、分布式与网络处理国际会议（*PDP*），第260–265页，IEEE，2019年。[116] F. E. Allen，《控制流分析》，载于*ACM*编译器优化研讨会，1970年。[117] M. S. Hecht 和 J. D. Ullman，《可约流图的特征》，《*ACM*学报（*JACM*）》，第21卷，第367–375页，1974年。[118] W. W. Peterson、T. Kasami 和 N. Tokura，《关于while、repeat 和 exit语句的能力》，*ACM* 通讯，第16卷，第503–512页，1973年。[119] S. R. Kosaraju，《结构化程序分析》，《计算机与系统科学杂志》，第9卷，第232–255页，1974年。[120] W. A. Wulf、R. K. Johnsson、C. B. Weinstock、S. O. Hobbs 和 C. M. Geschke，《优化编译器的设计》，载于卡内基梅隆大学，匹兹堡，1975年。[121]N. Corporation，《NVIDIA CUDA占用率计算器 v11.7.0》。`https://docs.nvidia.com/cuda/cuda-occupancy-calculator/index.html`，2022年。[122] N. Corporation，《NVIDIA Kepler GK110/GK210架构白皮书1.1版》。`https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/documents/NVIDIA-Kepler-GK110-GK210-Architecture-Whitepaper.pdf`，2014年。[123] S. Che、M. Boyer、J. Meng、D. Tarjan、J. W. Sheaffer、S.-H. Lee 和 K. Skadron，《Rodinia：异构计算基准测试套件》，载于*IEEE*国际工作负载表征研讨会（*IISWC*），2009年。

[124] P. Feautrier, “仿射调度问题的一些高效解法。I. 一维时间，”《国际并行编程杂志》，第21卷，第313–347页，1992年。[125] S. Cook, “第9章——优化应用程序，”载于《CUDA编程》（S. Cook 编），第305–440页，Morgan Kaufmann，2013年。[126] M. Lam, “软件流水线：一种适用于VLIW机器的高效调度技术，”载于ACM SIGPLAN编程语言设计与实现大会（PLDI），1988年。[127] N. Corporation, “NVIDIA新一代CUDA计算架构：Kepler GK110，有史以来速度最快、能效最高的高性能计算架构，”NVIDIA，2012年。[128] B. R. Rau和C. D. Glaeser，“用于高性能科学计算的若干调度技术及一种易于调度的水平架构，”载于微程序设计年度研讨会（MICRO 14），1981年。[129] S. Hong和H. Kim，“一种兼顾内存级并行性和线程级并行性的GPU架构解析模型，”载于计算机体系结构国际研讨会，2009年。[130] A. C. I. Malossi、Y. Ineichen、C. Bekas、A. Curioni和E. S. Quintana-Ortí，“多线程架构上稀疏矩阵-向量乘法的性能与能耗感知特性分析，”载于国际并行处理会议研讨会，2014年。[131] G. Ruetsch和P. Micikevicius，“优化CUDA中的矩阵转置。”`https://www.cs.colostate.edu/cs675/MatrixTranspose.pdf`，2009年。[132] L. Breiman、J. H. Friedman、R. A. Olshen和C. J. Stone，《分类与回归树》。Routledge，2017年。[133] G. Alavani、K. Varma和S. Sarkar，“使用静态分析预测CUDA内核执行时间，”载于*2018 IEEE*国际并行与分布式处理应用会议（*ISPA*），*IEEE*，第948–955页，2018年12月。[134] G. Alavani、J. Desai和S. Sarkar，“一种估算CUDA内核功耗的方法，”载于*2020 IEEE*国际并行与分布式处理应用、大数据、云计算、可持续计算、通信、社会计算与网络会议（*ISPA/BDCloud/SocialCom/SustainCom*），第984–991页，2020年。[135] V. Jatala、J. Anantpur和A. Karkare，“Greener：一种提升寄存器文件能效的工具，”*arXiv*，2017年。

[136] A. Alnori 和 K. Djemame，《云计算中图形处理器的整体资源管理》，《理论计算机科学》电子札记，第 340 卷，第 3–22 页，2018年。[137] A. Karki 等，《Tango：适用于各种加速器的深度神经网络基准测试套件》，载于*IEEE* 系统与软件性能分析国际研讨会（*ISPASS*），2019年。[138] M. Chadha、A. Srivastava 和 S. Sarkar，《面向高性能计算协处理器的统一功耗与能量测量 API》，载于国际性能计算与通信会议（*IPCCC*），2016。[139] S. Gupta 和 A. Gupta，《处理机器学习数据集中的噪声问题：系统综述》，《普罗赛迪亚计算机科学》，第 161 卷，第 466–474 页，2019年。[140] S. Hong，《面向能效型通用图形处理器计算的性能与功耗建模》。博士论文，佐治亚理工学院，2012年。[141] H.Zheng 和 Y. Wu，《结合天气相似度分析与特征工程的 XGBoost 模型：短期风电功率预测》，《应用科学》，第 9 卷，第 15 期，第 3019 页，2019年。[142] H. Abdi，《肯德尔秩相关系数》，《测量与统计百科全书》，第 2 卷，第 508–510 页，2007年。[143] S. M. Lundberg 和 S.-I. Lee，《解释模型预测的统一方法》，载于神经信息处理系统国际会议（*NIPS*），2017年。[144] T. Hill、L. Marquez、M. O’Connor 和 W. Remus，《用于预测和决策的人工神经网络模型》，《国际预测学杂志》，第 10 卷，第 5–15 页，1994年。[145] D. Parbat 和 M. Chakraborty，《基于 Python 的支持向量回归模型：预测印度 COVID-19 病例》，《混沌、孤子与分形》，第 138 卷，第 109942 页，2020。[146] G. Biau，《随机森林模型分析》，《机器学习研究期刊》，第 13 卷，第 1063–1095 页，2012年。[147] S. Basak、S. Kar、S. Saha、L. Khaidem 和 S. R. Dey，《使用基于树的分类器预测股市价格走势》，《北美经济与金融杂志》，第 47 卷，第 552–567 页，2019年。[148] L. Khaidem、S. Saha 和 S. R. Dey，《使用随机森林预测股市价格走势》，*arXiv* 预印本 *arXiv:1605.00003*，2016。

[149] M. W. Ahmad、M. Mourshed 和 Y. Rezgui，《树与神经元：随机森林与人工神经网络在高分辨率建筑能耗预测中的比较》，《能源与建筑》，第 147 卷，第 77–89 页，2017年。[150] X. Dong、Z. Yu、W. Cao、Y. Shi 和 Q. Ma，《集成学习综述》，《计算机科学前沿》，第 14 卷，第 241–258 页，2020。[151] C. Zhang 和 Y. Ma，《集成机器学习：方法与应用》。施普林格，2012年。[152] P. Geurts、D. Ernst 和 L. Wehenkel，《极端随机树》，《机器学习》，第 63 卷，第 3–42 页，2006年。[153] J. H. Friedman，《贪心函数逼近：梯度提升机》，《统计年刊》，第 29 卷，第 1189–1232 页，2000年。[154] T. Chen 和 C. Guestrin，《XGBoost：一种可扩展的树提升系统》，发表于国际知识发现与数据挖掘会议（*ACM SIGKDD*），2016。[155] X. Zhang、C. Yan、C. Gao、B. A. Malin 和 Y. Chen，《通过 XGBoost 回归预测医疗数据中的缺失值》，《医疗信息学研究杂志》，第 4 卷，第 383–394 页，2020。[156] A. V. Dorogush、V. Ershov 和 A. Gulin，《CatBoost：支持类别特征的梯度提升》，*arXiv*，第 abs/1810.11363 卷，2018年。[157] A. Natekin 和 A. Knoll，《梯度提升机教程》，《神经机器人学前沿》，第 7 卷，第 21 页，2013年12月。[158] Y. Tian 和 Y. Zhang，《机器学习正则化策略综合综述》，《信息融合》，第 80 卷，第 146–166 页，2022年。[159] P. Rodríguez、M. A. Bautista、J. Gonzàlez 和 S. Escalera，《超越独热编码：低维目标嵌入》，《图像与视觉计算》，第 75 卷，第 21–31 页，2018年。[160] L. Prokhorenkova、G. Gusev、A. Vorobev、A. V. Dorogush 和 A. Gulin，《CatBoost：基于类别特征的无偏提升》，发表于神经信息处理系统国际会议（*NIPS*），第 6639–6649 页，2018年。[161] A. Ibrahim、R. L.、M. Muhammed、R. Abdulaziz 和 G. A.，《CatBoost分类器与其他机器学习方法的比较》，《国际高级计算机科学与应用杂志》，第 11 卷，第 738–748 页，2020。[162] A. Samat、E. Li、P. Du、S. Liu 和 J. Xia，《通过并行化mRMR集成子空间特征选择实现超光谱图像分类的GPU加速CatBoost森林》，《*IEEE*应用地球观测与遥感精选主题期刊》，第 14 卷，第 3200–3214 页，2021年。

[163] E. Al Daoud，《使用房贷数据集比较 XGBoost、LightGBM 和 CatBoost》，《国际计算机与信息工程杂志》，第 13 卷，第 6–10 页，2019年。[164] J. T. Hancock 和 T. M. Khoshgoftaar，《面向大数据的 CatBoost：跨学科综述》，《大数据杂志》，第 7 卷，第 1–45 页，2020。[165] M. Jahn，《面板数据设定下的人工神经网络回归模型：经济增长预测》，《经济建模》，第 91 卷，第 148–154 页，2020。[166] R. J. G. B. Campello、D. Moulavi 和 J. Sander，《基于层次密度估计的基于密度的聚类》，发表于《知识发现与数据挖掘进展》（J. Pei、V. S. Tseng、L. Cao、H. Motoda 和 G. Xu 编），第 160–172 页，2013年。[167] M. Ester、H.-P. Kriegel、J. Sander 和 X. Xu，《一种用于在大型空间数据库中发现含噪声簇的基于密度的算法》，发表于国际知识发现与数据挖掘会议（*KDD*），1996年。[168] H.-P. Kriegel、P. Kröger、J. Sander 和 A. Zimek，《基于密度的聚类》，《*Wiley* 跨学科综述：数据挖掘与知识发现》，第 1 卷，第 231–240 页，2011年。[169] R. J. Campello、D. Moulavi、A. Zimek 和 J. Sander，《用于数据聚类、可视化和异常值检测的层次密度估计》，《*ACM* 数据知识发现汇刊（*TKDD*）》，第 10 卷，第 1 期，第 1–51 页，2015年。[170] S. Pawanr、G. K. Garg 和 S. Routroy，《机床瞬态能耗预测模型的开发》，《*CIRP*会议论文集》，第 98 卷，第 678–683 页，2021年。[171] M. T. Ribeiro、S. Singh 和 C. Guestrin，《“我为什么应该相信你？”：解释任意分类器的预测》，发表于国际知识发现与数据挖掘会议（*ACM SIGKDD*），第 1135–1144 页，2016年。[172] V. Jatala、J. Anantpur 和 A. Karkare，《通过资源共享提升GPU性能》，发表于国际高性能并行与分布式计算研讨会（*HPDC*），2016。[173] S. Chetlur、C. Woolley、P. Vandermersch、J. Cohen、J. Tran、B. Catanzaro 和 E. Shelhamer，《cuDNN：深度学习的高效原语》，*arXiv*预印本 *arXiv:1410.0759*，2014年。[174] B. Pang、E. Nijkamp 和 Y. N. Wu，《使用TensorFlow进行深度学习：综述》，《教育与行为统计学杂志》，第 45 卷，第 2 期，第 227–248 页，2020。

[175] V. Volkov, 《用于研究 GPU 性能模型的微基准测试》，第 *23* 届 *ACM SIGPLAN* 并行编程原理与实践研讨会，第 53 卷，第 1 期，第 421–422 页，2018年。

[176] N. Ding 和 S. Williams, 《面向 GPU 的指令屋顶线模型》，发表于*IEEE/ACM* 高性能计算机系统性能建模、基准测试与仿真 *(PMBS)*，2019年。

附录 A

基于动态分析的模型

动态模型利用在应用程序运行时收集的数据来预测应用程序的执行时间和功耗。每次需要进行预测时都使用利用运行时数据的动态模型，会违背能效预测的初衷。本研究开发了一个动态模型，作为初步研究，以了解应用程序特征对功耗和能量的重要性。

本研究将 GPU 运行时事件视为因变量，并结合每个线程块的线程数（block_size）、每线程寄存器数（reg_t）、GPU 时钟频率（ν_{gpu}）和内存时钟频率（ν_{mem}）来构建模型。

GPU 事件

NVIDIA GPU 事件本质上是 NVIDIA 硬件向 NVIDIA CUPTI 库报告的各种指标。它们捕获各种运行时细节，例如内存事务的数量、已执行指令的数量等。GPU 事件涵盖了多种指标，可捕获有关算法及其实现的详细信息。因此，通过使用事件，可以创建应用程序无关的功耗和执行时间模型。以下将列出并描述这些事件。

1. fb_subp0_read_sectors：子分区 0 的 DRAM 读请求数，每进行一次 32 字节访问便递增 1。2. fb_subp1_read_sectors：子分区 1 的 DRAM 读请求数，每进行一次 32 字节访问便递增 1。3. fb_subp0_write_sectors：子分区 0 的 DRAM 写请求数，每进行一次 32 字节访问便递增 1。4. fb_subp1_write_sectors：子分区 1 的 DRAM 写请求数，每进行一次 32 字节访问便递增 1。5. l2_subp0_write_sector_misses：L2 缓存切片 0 中的写未命中数。每进行一次 32 字节访问便递增 1。

6. l2_subp1_write_sector_misses：L2 缓存切片 1 中的写未命中数。每进行一次 32 字节访问便递增 1。7. l2_subp2_write_sector_misses：L2 缓存切片 2 中的写未命中数。每进行一次 32 字节访问便递增 1。8. l2_subp3_write_sector_misses：L2 缓存切片 3 中的写未命中数。每进行一次 32 字节访问便递增 1。9. l2_subp0_read_sector_misses：L2 缓存切片 0 中的读未命中数。每进行一次 32 字节访问便递增 1。10. l2_subp1_read_sector_misses：L2 缓存切片 1 中的读未命中数。每进行一次 32 字节访问便递增 1。11. l2_subp2_read_sector_misses：L2 缓存切片 2 中的读未命中数。每进行一次 32 字节访问便递增 1。12. l2_subp3_read_sector_misses：L2 缓存切片 3 中的读未命中数。每进行一次 32 字节访问便递增 1。13. l2_subp0_write_l1_sector_queries：从 L1 发往 L2 缓存切片 0 的写请求数。每进行一次 32 字节访问便递增 1。14. l2_subp1_write_l1_sector_queries：从 L1 发往 L2 缓存切片 1 的写请求数。每进行一次 32 字节访问便递增 1。15. l2_subp2_write_l1_sector_queries：从 L1 发往 L2 缓存切片 2 的写请求数。每进行一次 32 字节访问便递增 1。16. l2_subp3_write_l1_sector_queries：从 L1 发往 L2 缓存切片 3 的写请求数。每进行一次 32 字节访问便递增 1。17. l2_subp0_read_l1_sector_queries：从 L1 发往 L2 缓存切片 0 的读请求数。每进行一次 32 字节访问便递增 1。18. l2_subp1_read_l1_sector_queries：从 L1 发往 L2 缓存切片 1 的读请求数。每进行一次 32 字节访问便递增 1。19. l2_subp2_read_l1_sector_queries：从 L1 发往 L2 缓存切片 2 的读请求数。每进行一次 32 字节访问便递增 1。20. l2_subp3_read_l1_sector_queries：从 L1 发往 L2 缓存切片 3 的读请求数。每进行一次 32 字节访问便递增 1。21. l2_subp0_read_l1_hit_sectors：在 L2 缓存切片 0 中命中的 L1 读请求数。每进行一次 32 字节访问便递增 1。22. l2_subp1_read_l1_hit_sectors：在 L2 缓存切片 1 中命中的 L1 读请求数。每进行一次 32 字节访问便递增 1。

23. l2_subp2_read_l1_hit_sectors：在 L2 缓存切片 2 中命中的 L1 读请求数量。每次 32 字节访问，该计数增加 1。24. l2_subp3_read_l1_hit_sectors：在 L2 缓存切片 3 中命中的 L1 读请求数量。每次 32 字节访问，该计数增加 1。25. l2_subp0_total_read_sector_queries：对 L2 缓存切片 0 的总读请求数。这包括来自 L1、纹理缓存和系统内存的请求。每次 32 字节访问，该计数增加 1。26. l2_subp1_total_read_sector_queries：对 L2 缓存切片 1 的总读请求数。这包括来自 L1、纹理缓存和系统内存的请求。每次 32 字节访问，该计数增加 1。27. l2_subp2_total_read_sector_queries：对 L2 缓存切片 2 的总读请求数。这包括来自 L1、纹理缓存和系统内存的请求。每次 32 字节访问，该计数增加 1。28. l2_subp3_total_read_sector_queries：对 L2 缓存切片 3 的总读请求数。这包括来自 L1、纹理缓存和系统内存的请求。每次 32 字节访问，该计数增加 1。

29. l2_subp0_total_write_sector_queries：对 L2 缓存切片 0 的写请求总数。包括来自 L1、纹理缓存和系统内存的请求。每次 32 字节访问，此计数加 1。

30. l2_subp1_total_write_sector_queries：对 L2 缓存切片 1 的写请求总数。包括来自 L1、纹理缓存和系统内存的请求。每次 32 字节访问，此计数加 1。

31. l2_subp2_total_write_sector_queries：对 L2 缓存切片 2 的总写请求数。包括来自 L1、纹理缓存和系统内存的请求。每次 32 字节访问时，此计数增加 1。32. l2_subp3_total_write_sector_queries：对 L2 缓存切片 3 的总写请求数。包括来自 L1、纹理缓存和系统内存的请求。每次 32 字节访问时，此计数增加 1。33. gld_inst_32bit：所有线程块中的所有线程已执行的 32 位全局加载指令总数。34. gst_inst_32bit：所有线程块中的所有线程已执行的 32 位全局存储指令总数。35. thread_inst_executed：所有线程已执行的指令数，不包括重放。对于每条指令，计数按执行该指令的线程束中的线程数递增。

36. gld_request：已执行的加载指令数，这些指令未指定状态空间，因此使用通用寻址；在多处理器上按线程束递增。可能包括全局、局部和共享状态空间中的加载操作。37. gst_request：已执行的存储指令数，这些指令未指定状态空间，因此使用通用寻址；在多处理器上按线程束递增。可能包括对全局、局部和共享状态空间的存储操作。38. active_cycles：多处理器至少有一个活动线程束时的周期数。39. l1_global_load_transactions：来自 L1 缓存的全局加载事务数。每笔事务计数增加 1。40. l1_global_store_transactions：来自 L1 缓存的全局存储事务数。每笔事务计数增加 1。41. uncached_global_load_transaction：未缓存的全局加载事务数。每笔事务计数增加 1。42. global_store_transaction：全局存储事务数。每笔事务计数增加 1。

结果

使用随机森林进行能耗预测时的特征重要性如图 1 所示，功耗预测的特征重要性如图 2 所示。运行时内存和着色器频率是功耗预测中最重要的特征。由于这两个特征只有在执行代码后才能获得，因此我们无法在基于静态分析的模型中使用它们。特征“active_cycles”对功耗预测也有显著贡献。我们尝试在静态分析方法中将此特征建模为 inst_issue_cycles。inst_issue_cycles 也是基于静态分析的模型中最重要的特征之一。

在动态分析中，全局内存指令也对功耗有显著影响，这一点在基于静态分析的模型中同样可见。每个线程块的线程数（block_size）对功耗的贡献并不显著，这也是基于静态分析的模型中常见的观察结果。

我们采用了三种常用的机器学习方法来构建执行时间、功耗与能耗的预测模型。这些算法包括线性回归、随机森林和XGBoost，其验证分数如表1所示。根据三项指标的R^2 分数，随机森林和XGBoost在准确率方面均表现出较高的效率。与第5章5.3节所 प्रस्तुत的基于静态分析的模型相比，此处描述的动态分析模型更加准确。由于所使用的特征是应用程序执行过程中由CUPTI库捕获的运行时行为，因此这一结果符合预期。然而，本论文的主要目标是

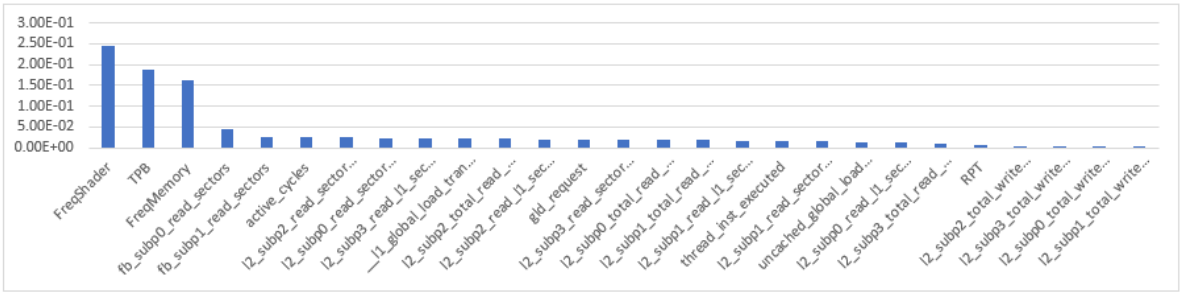


图 1：能耗预测的特征重要性

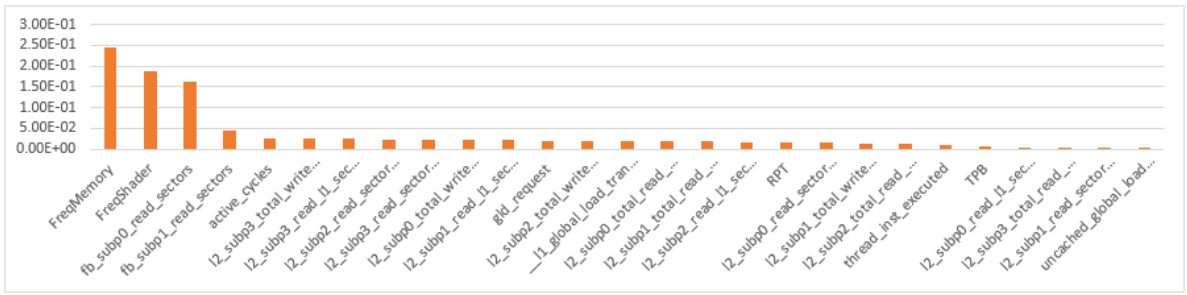


图 2：功耗预测的特征重要性

在不执行CUDA应用程序的情况下预测其性能和功耗。这项动态分析研究使我们对运行时特征与功耗之间的关系有了一些认识。研究观察到，这项动态分析研究的观测值对所提出的基于静态分析的模型同样成立。

表 1：机器学习算法的验证分数

	能量		功耗		执行时间	
	R^2分数	均方根误差	R^2得分	均方根误差	R^2分数	均方根误差
线性回归	0.79719	3366204	0.2978	8203.235	0.4327	79.38
随机森林	0.9979	568.011	0.9732	1873.3	0.9999	0.0003
XGBoost	0.9975	1250.51	0.9730	2654.86	0.9993	0.0012

微基准测试

内核启动开销代码片段

```
__global__void emptyKernel(){// 未执行任何指令}// 使用不同的()
```

清单 1：内核启动开销

指针追逐基准测试代码片段

```
int i, x=0; for(i = 0; i <内层循环迭代次数; i++) { outerStart =时钟(); startClo
cl
st
ou
y
j
*d
```

代码清单 2：指针追逐基准测试

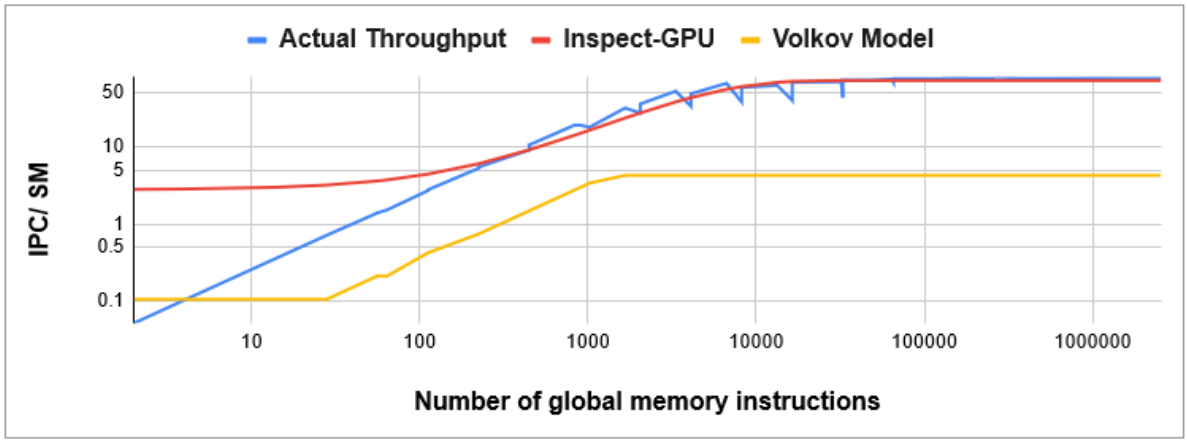


图3：吞吐量模型与 Volkov 模型

全局内存吞吐量模型的有效性

本文提出的全局内存吞吐量模型有助于理解图形处理器的性能。全局内存指令是影响图形处理器性能的最关键特征之一。为了证明所提出吞吐量模型的有效性，我们开展了两项研究来分析其有效性，具体如下文所述。

与 Volkov 等人的吞吐量模型比较

我们将自己的全局内存模型与 Volkov 开发的一种广受欢迎的模型进行比较 [85, 175]，该模型曾用于预测性能，并与其他一些现有方法进行比较 [175]。Volkov [85]采用 IPC/SM 表示吞吐量，其中 IPC 是每周期指令数。我们通过以下步骤将自己的模型与 Volkov 模型进行比较：

1. 我们将实际吞吐量以及使用模型计算得出的吞吐量转换为 IPC/SM。为此，先将以 GB/s 为单位的吞吐量转换为 B/s，再将该值除以每条指令的字节数（128 B） [85]，得到每秒指令数。要得到每周期指令数，再将该值除以 GPU 时钟频率。最后，将每周期指令数除以图形处理器上的流式多处理器数量，得到 IPC/SM。

2. Volkov 使用其研究中提出的方程 [85] 为我们提供每个线程束的内存吞吐量。我们将该吞吐量值乘以一个线程束中的线程数（32），将其转换为每条指令的吞吐量，从而得到 IPC/SM。我们在图 3 中绘制了实际吞吐量、我们的模型所提出的吞吐量以及 Volkov 模型的吞吐量。

屋顶线模型用于将应用程序的性能（P）和算术强度（I）与平台的峰值性能和内存带宽 [87]联系起来。屋顶线是一种面向吞吐量的性能模型。因此，我们使用屋顶线模型。

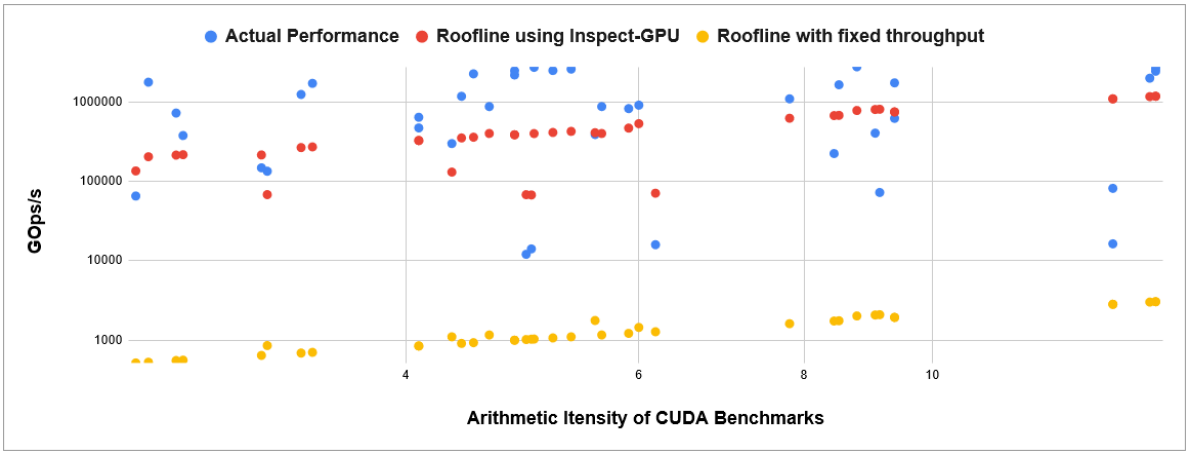


图 4：使用屋顶线模型进行比较

以观察使用我们的内存吞吐量模型进行的性能预测是否足够接近实际性能。在屋顶线模型图中，x 轴表示算术强度，y 轴表示性能。这两个参数均采用对数刻度。屋顶线公式 $P = min(\pi, I \cdot \beta)$ 用于根据机器峰值性能（π）、机器峰值带宽（β）以及应用程序的算术强度（I），限定 P(GOps/s) 的上界。

我们使用了 39 个属于 NVIDIA CUDA 工具包 [1] 和 Rodinia 测试基准 [123] 的 CUDA 内核。对于每个基准测试，我们测量其实际性能、使用屋顶线模型预测的性能，以及使用我们的方法预测的性能。对于每个基准测试，我们首先计算其算术强度 I，即计算指令数量与内存指令数量之比。接下来，我们对每个基准测试执行以下操作：

- 我们使用供应商规格中提供的吞吐量（带宽）（β），并利用屋顶线公式计算在该固定吞吐量下的计算性能（P）。
- 我们使用所提出的全局内存吞吐量模型计算 β，并使用屋顶线公式计算预测性能。
- 我们通过将基准测试的指令总数除以其执行时间（微秒），计算其实际性能（GOps/s）。

图 4 中以算术强度表示了每个 CUDA 基准测试。图中根据每个基准测试的算术强度绘出了实际性能（蓝点）、使用我们的吞吐量模型预测的性能（红点）以及理论值（黄点）。

[1] https://docs.nvidia.com/cuda/cuda-samples/index.html

要点

让我们将 Volkov 等人提出的流行线程束吞吐量模型与我们的方法进行比较。由于我们的模型是指数模型，如图3所示，当访问的内存指令数量较少时，我们模型预测的吞吐量（IPC/SM）与实际值相差较大。不过，随着指令数量增加，模型的表现良好。无论访问的指令数量是多少，我们模型预测的吞吐量都比 Volkov 模型的预测值更接近实际值。主要原因是，所提出的模型基于实际程序执行结果。随着指令数量增加，已准备好执行（其数据已获取）的指令数量也会增加，因此吞吐量随之提高，而 Volkov 模型无法捕捉这一点。

接下来，我们来看看另一种基于屋顶线模型的常用性能预测方法。如图 4 所示，显而易见，与采用供应商提供的理论吞吐量相比，使用我们的内存吞吐量模型预测的性能更接近实际性能。屋顶线模型是一种简单的性能预测模型，可能无法捕捉像图形处理器这样的多核架构的所有复杂性。许多研究人员对这一屋顶线模型进行了改进，以开发稳健的性能预测模型 [87, 176]。我们使用这一屋顶线模型来说明，如图 4 所示，采用所提出的吞吐量进行性能预测，相比采用供应商规格中提供的固定吞吐量，预测效果有所提升。要开发稳健且准确的性能预测模型，需要不止一项指标。在不同粒度层级上，还需要延迟、吞吐量、并发性、启动开销、延迟隐藏、资源分配等其他指标，涵盖从单条指令到整个应用程序的层级。为测量所有这些指标，本文提出的方法和研究结果将为研究人员提供切实帮助。