计 算 机 研 究 与 发 展
Journal of Computer Research and Development
https://crad.ict.ac.cn
62(10): 2622−2634, 2025

面向 GPU 平台的通用 Stencil 自动调优框架

孙庆骁[1] 杨海龙[2]
[1](中国石油大学(北京)人工智能学院 北京 102249)
[2](北京航空航天大学计算机学院 北京 100191)
(qingxiao.sun@cup.edu.cn)

Generalized Stencil Auto-Tuning Framework on GPU Platform

Sun Qingxiao[1] and Yang Hailong[2]
[1](*College of Artificial Intelligence*, *China University of Petroleum-Beijing*, *Beijing* 102249)
[2](*School of Computer Science and Engineering*, *Beihang University*, *Beijing* 100191)

Abstract Stencil computations are widely adopted in scientific applications. Many HPC platforms utilize the high computation capability of GPUs to accelerate Stencil computations. In recent years, Stencils have become more complex in terms of Stencil order, memory accesses, and computation patterns. To adapt Stencil computations to GPU architectures, the academic community has proposed a variety of optimization techniques based on streaming and tiling. Due to the diversity of Stencil computational patterns and GPU architectures, no single optimization technique fits all Stencil instances. Therefore, researchers have proposed Stencil auto-tuning mechanisms to conduct parameter searches for a given combination of optimization techniques. However, existing mechanisms introduce huge offline profiling costs and online prediction overhead, unable to be flexible to arbitrary Stencil patterns. To address the above problems, we propose a generalized Stencil auto-tuning framework GeST, which achieves the ultimate performance optimization of Stencil computations on GPU platforms. Specifically, GeST constructs the global search space through the zero-padding format, quantifying parameter correlations via the coefficient of variation to generate parameter groups. After that, GeST iteratively selects parameter values from the parameter groups, adjusting the sampling ratio according to the reward policy and avoiding redundant execution through Hash coding. The experimental results show that GeST can identify better-performing parameter settings in a short time compared with other state-of-the-art auto-tuning work.

Key words Stencil computation; GPU; auto-tuning; performance optimization; parameter search

摘 要 Stencil 计算在科学应用中得到了广泛采用.许多高性能计算(HPC)平台利用 GPU 的高计算能力来加速 Stencil 计算.近年来,Stencil 计算在阶数、内存访问和计算模式等方面变得更加复杂.为了使 Stencil 计算适配 GPU 架构,学术界提出了各种基于流处理和分块的优化技术.由于 Stencil 计算模式和 GPU 架构的多样性,没有单一的优化技术适合所有 Stencil 实例.因此,研究人员提出了 Stencil 自动调优机制来对给定优化技术组合进行参数搜索.然而,现有机制引入了庞大的离线分析成本和在线预测开销,并且无法灵活地推广到任意 Stencil 模式.为了解决上述问题,提出了通用 Stencil 自动调优框架 GeST,其在 GPU 平台上实现 Stencil 计算的极致性能优化.具体来说,GeST 通过零填充格式构建全局搜索空间,利用变异系数量化参数相关性并生成参数组;之后,GeST 迭代地从参数组选取参数值,根据奖励策略调整采

收稿日期: 2024-07-15; **修回日期**: 2025-01-09
基金项目: 国家自然科学基金项目(62402525, 62322201, 62072018);中央高校基本科研业务费专项资金(2462023YJRC023)
This work was supported by the National Natural Science Foundation of China (62402525, 62322201, 62072018) and the Fundamental Research Funds for the Central Universities (2462023YJRC023).

样比例并通过哈希编码避免冗余执行. 实验结果表明，与其他先进的自动调优工作相比，GeST 能够在短时间内识别出性能更优的参数设置.

关键词 Stencil 计算；GPU；自动调优；性能优化；参数搜索

中图法分类号 TP319

DOI: 10.7544/issn1000-1239.202440612 **CSTR:** 32373.14.issn1000-1239.202440612

Stencil 计算是科学应用最常采用的计算模式之一，出现在图像处理[1]、物理仿真[2]和元胞自动机[3]等领域. Stencil 计算扫描计算网格并处理每个点周围的固定邻居来更新其值，其中邻居沿每个维度的范围称为阶（order）. 例如，盒型 Stencil 用于在图像处理中执行平滑和其他基于邻居像素的计算[4].

近年来，Stencil 计算在阶数、数据访问和计算模式等方面趋向于多样化[5]. Stencil 计算往往具有丰富的并行性，这使得其很适合部署在 GPU 上进行性能加速. 然而，程序员必须确保内存合并、重叠计算和数据访问以及减少同步和消除线程分歧，在并行性和资源利用之间进行权衡[6]. 为了适应 GPU 架构的高计算能力和有限内存带宽，研究工作提出了许多基于流处理和分块的优化技术[7]. 由于计算模式的多样性，单一的优化技术无法适用于所有 Stencil 实例.

Stencil 领域特定语言（domain-specific language, DSL）集成各种优化技术来探索自动代码生成[8]. 尽管 DSL 可有效加速 Stencil 计算，各个优化技术的性能影响仍难以评估. 此外，指定优化组合下能否生成高性能代码取决于 Stencil 计算模式和 GPU 硬件架构. 为了充分提升性能，Stencil DSL 通常采用基于穷尽参数搜索的性能调优策略[9]. 然而，这些策略在适应更多优化选项时扩展性较差，并且在大型优化空间中的搜索过程非常耗时. 另一方面，Stencil 自动调优框架被提出以确定指定优化组合下的最佳参数设置[10]. 尽管这些框架能够缩小参数空间，但它们严重依赖于专家知识. 而且，当优化组合不适用于特定的 Stencil 计算模式时，其性能提升也十分有限.

最新工作基于机器学习选取高性能优化组合，并根据性能预测来指导搜索空间采样，从而避免陷入性能不佳的局部优化空间[11]. 然而，优越设置的相似性会导致不可避免的预测误差，因此仅通过性能预测来确定最优参数设置是不可行的. 另一方面，机器学习方法需要构建庞大的训练集，并引入显著的训练成本，且最终效果因 Stencil 计算模式而异. 即便仅考虑在线开销，对于所含模式较少而无法批量预测的应用来说，前向推理的启动时间也是不可接受的.

为了应对上述挑战，本文提出通用 Stencil 自动调优框架 GeST，其能够高效地识别 GPU 上全局优化空间的高性能参数设置. GeST 首先将 Stencil 优化技术和 GPU 优化选项进行参数化，并基于零填充构建全局搜索空间. 之后，GeST 对全局搜索空间进行随机采样，通过统计学指标量化参数相关性并生成参数组. 最后，GeST 迭代地从参数组中采样参数值并将参数设置的哈希编码存储于字典以避免冗余执行. 本文开展了大量的实验来评估 GeST 框架，以验证其在 Stencil 性能自动调优方面的有效性.

本文的主要贡献包括 4 个方面：

1）详尽分析了 Stencil 计算在 GPU 上的优化技术及其组合的参数范围和性能影响. 此外，设计了检查优化约束的规则，从而仅执行有效的参数设置.

2）提出了参数分组策略，对特定 Stencil 计算模式进行少量的随机采样执行，基于统计学指标量化成对参数的相关性，生成参数组以缩小搜索范围.

3）提出了搜索空间采样机制，迭代地从参数组选取参数值，执行选到的参数设置并根据性能反馈调整采样比例，通过哈希编码存储以避免重复执行.

4）实现了通用 Stencil 自动调优框架 GeST 对 GPU 上的全局优化空间进行高效搜索. 实验结果表明，GeST 取得了更高的调优效率和性能.

1 背　　景

1.1 GPU 架构和执行模型

GPU 由数十到百余个流多处理器（streaming multiprocessor, SM）组成. 如图 1 所示，每个 SM 包含数百个计算核心和其他存储资源（寄存器、共享内存和 L1 高速缓存）. 在 GPU 上执行的代码被称为内核. CPU 主机端启动内核后，GPU 上会创建大量线程，每 32 个线程分到 1 个 Warp 中. 多个 Warp 组成一个线程块（thread block, TB），TB 的大小由内核配置决定. TB 调度器根据轮询策略将 TB 调度给 SM，并在计算资源和硬件限制下最大化 GPU 利用率.

由于 SM 中的计算资源有限[12]，GPU 任务必须进

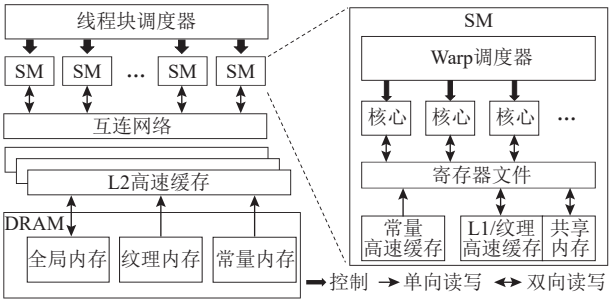


Fig. 1 Basic hardware architecture of GPU

图 1 GPU 基础硬件架构

行微调以实现系统利用率和性能加速比之间的权衡. 某些优化策略(如循环展开)增加寄存器级别的数据重用以提高性能[13]. 然而, 生成的代码受到寄存器压力的高度约束, 甚至可能导致寄存器溢出. 此外, 需要仔细确定参数设置才能获得更好的性能. 例如, 适当的 TB 大小可以在硬件限制下最大化线程级并行(thread-level parallelism, TLP). 但是, 过高的 TLP 可能会导致缓存抖动, 特别是对于诸如 Stencil 计算的内存密集任务[14]. 因此, 仅通过分析建模来进行 GPU 任务的性能调优是不够的[9].

1.2 Stencil 性能优化技术

在具有高计算能力的 GPU 上加速 Stencil 计算已经引起学术界的广泛关注[15-16]. 下面将简要讨论面向 GPU 架构的 Stencil 性能优化技术, 如表 1 所示.

Table 1 Stencil Performance Optimizations Technology on GPU Platforms

表 1 面向 GPU 平台的 Stencil 性能优化技术

优化技术	约束
流处理(ST)	—
块合并(BM)	启用 CM 时无效
循环合并(CM)	启用 BM 时无效
重定时(RT)	仅在启用 ST 时有效
预取(PR)	仅在启用 ST 时有效
时间分块(TL)	—

注: "—"表示没有约束.

1)流处理(streaming, ST). 流处理技术是一种常见的优化, 可以改进数据重用并减少流维度上的计算冗余. 针对 3D 输入网格, 流处理技术的有效实现是采用 2.5D 空间分块. 具体来说, 2D 块的计算在一个维度上流式传输, 每个块的数据被重用以更新下一个块. 然而, 如果问题规模很大, 流处理技术会增大计算粒度从而限制并行性. 为了取得更好的性能, 并发流处理技术[6]将流维度划分为块, 其中 TB 以块粒度并行遍历流维度. 与此同时, 流处理技术可以结合循环展开来增加寄存器级别的数据重用.

2)块/循环合并(block/cyclic merging, BM/CM). 一般来说, 每个 GPU 线程负责单个输出点. 合并多个输出点的计算能够降低内核启动开销并消除冗余内存访问. 研究人员提出了 2 种技术来合并计算, 包括块合并和循环合并. 块合并技术合并多个相邻的输出点, 而循环合并技术则将具有固定距离的每个点合并. 然而, 这 2 种技术都可能增大寄存器压力并降低驻留在每个 SM 上的线程数量. 此外, 全局网格最内层的块合并会破坏内存合并[17]. 合并技术的选择以及要合并点的数量会显著影响 Stencil 计算的性能.

3)预取(prefetching, PR). 在当前迭代更新输出网格后, 流处理会移动位于共享内存中的数据来继续下一次迭代的计算. 由于海量线程的并发执行, 相邻迭代之间必须进行同步以保证结果的正确性. 同步会导致内核之间的串行处理而损失性能. 预取技术通过重叠计算和数据加载来隐藏同步延迟[6]. 具体来说, 在处理当前迭代计算的同时, 用于下一次迭代的数据被加载到寄存器中. 然而, 预取技术可能会耗尽 GPU 上非常有限的寄存器.

4)重定时(retiming, RT). 重定时技术将 Stencil 计算分解为一组子计算, 再累积结果来改进数据重用[18]. 重定时通过同质化 Stencil 访问来平衡内存和寄存器之间的资源使用. 由于寄存器的有效重用, 高阶 Stencil 可以受益于重定时优化. 然而, 重定时技术对于具有低寄存器压力的 Stencil 计算几乎无效.

5)时间分块(temporal blocking, TL). 尽管 Stencil 计算具有跨时间步的数据依赖, 但每个点的依赖范围受到 Stencil 邻接访问模式和自该点上次更新以来经过的时间步数的限制[9]. 时间分块技术合并时间步并避免全局内存访问以利用隐藏的时间局部性, 再通过相邻块的冗余加载来解决沿时间维度的数据依赖. 然而, 时间分块技术可能会导致受寄存器约束的 Stencil 计算的性能下降.

Stencil 优化技术的有效性取决于多种因素, 例如 GPU 架构和 Stencil 计算模式. 在某些约束(见表 1)下, 可以组合优化技术以进一步改进性能. 另外, 参数设置选择不当会导致性能欠佳. 以上优化技术驱动了针对 Stencil 计算的优化和参数联合搜索机制的实现.

1.3 Stencil 自动调优机制

由于 Stencil 模式的多样性, 任何优化都必须进

行微调以最大化性能. Stencil 框架将与性能相关的参数暴露给集成在其框架中的参数调优机制来实现此目的. Halide[8] 应用随机搜索来自动找到良好的流水线规划. Halide 还限制了每个函数的域调度操作数以防止生成代码爆炸. Artemis[6] 通过分层参数调优来裁剪搜索空间. Artemis 首先调整对计算性能影响较大的优化技术，然后选择高性能候选参数. GoPipe[19] 为盒状 Stencil(如卷积)的每个阶段找到最佳任务粒度和调度机制. AN5D[9] 根据性能模型获得 Top-5 的参数集，并基于实际执行选择其中最高性能的参数集. 以上自动调优机制是专为 Stencil 框架定制的，因此它们在评估更多优化技术时的可扩展性较差.

为了克服上述限制，其他工作专注于加快 Stencil 计算的参数调优进程. FAST[20] 训练回归模型来预测最佳解空间的相似度，并建立参数特征和优化解之间的映射. OpenTuner[21] 实现一系列搜索技术(如差分进化和爬山器)来寻找最优解决方案. Garvey 等人[17] 根据先验知识对优化参数进行分组，并在启用随机采样的情况下详尽地搜索每组的参数设置. csTuner[10] 采用统计学和机器学习方法生成参数组和采样设置，随后设计考虑近似性的遗传算法以减少搜索时间. Trümper 等人[22] 对循环嵌套的动静态性能特征进行编码，并将优化参数存储在数据库中以实现迁移调优. 以上自动调优框架通常对预先指定的优化或其组合执行参数搜索，很容易陷入局部优化空间.

除了参数自动调优之外，研究人员还利用性能预测来研究优化技术对 Stencil 计算的影响. Martínez 等人[23] 将内核配置和硬件计数器提供给支持向量机来预测 GFLOPS 和执行时间. Cosenza 等人[24] 利用序数回归来预测代码变体的性能排名，排名质量通过 Kendall 系数进行评估. Nesterenko 等人[25] 对优化组合和领域级属性之间的相互作用进行性能建模. StencilMART[14] 通过张量赋值将 Stencil 模式表示为二元张量，再利用卷积神经网络来预测最佳优化组合. 上述工作[14,23-25] 没有实现性能预测与自动调优的有效融合，且对于不可避免的预测误差也缺乏良好的容错能力. GSTuner[11] 基于零填充统一不同优化组合的特征向量，并根据预测性能来指导全局搜索空间采样和遗传进化搜索. 然而，基于机器学习的性能预测方法都需要预先收集海量性能数据来构成 Stencil 数据集，并且引入了显著的离线训练和在线推理成本，这对于轻量级的 Stencil 计算来说通常是难以接受的.

2 方法设计与实现

2.1 设计概要

本文提出通用 Stencil 自动调优框架 GeST 来高效地确定全局优化空间的高性能参数设置，如图 2 所示. GeST 由 2 个重要组件组成，包括参数分组和搜索空间采样. 参数分组组件通过统计学指标和分组算法聚合强相关的参数，其中 Stencil 数据集来自随机采样过程. 搜索空间采样组件利用奖励策略来选取参数值设置并基于执行反馈来调整采样比例，其中初始参数设置和初始采样比例分别基于 Stencil 数据集和参数组来生成. 与先前方法不同，GeST 不会引入任何离线开销，即使是 Stencil 数据集的收集过程也是通过随机采样在线完成，并且计入整体调优时间内.

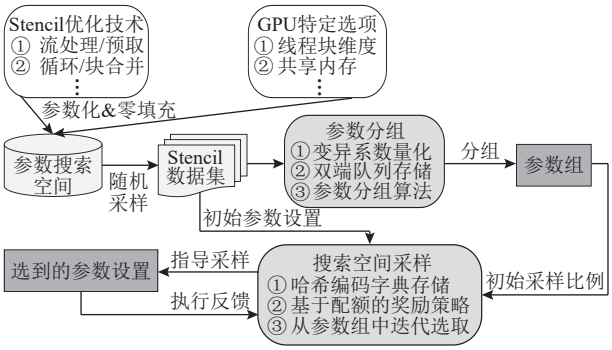


Fig. 2 The design overview of GeST framework

图 2 GeST 框架的设计概要

图 2 展示了 GeST 框架的设计概要. GeST 将 Stencil 优化技术和 GPU 特定选项参数化为搜索空间，其中每个参数设置都被预处理为零填充格式以统一各种优化组合. GeST 对参数搜索空间进行随机采样以获取 Stencil 数据集，其用于指导参数分组以及生成后续采样过程的初始参数设置. GeST 将参数对的量化相关性指标存储在双端队列中，并通过分组算法聚合参数以降低搜索工作量. GeST 从参数组中迭代地采样参数设置，其中每组的采样比例根据基于配额的奖励策略进行调整. GeST 将选取到的设置通过哈希编码存储在字典中以避免冗余的重新执行.

由于 GeST 无需离线性能分析和模型训练，其可以被轻易地推广到任意 Stencil 计算模式和输入网格大小，以及集成更多参数来捕获未来的 Stencil 优化技术. 另一方面，GeST 由于其组件的通用性还可以支持其他通用 GPU 算法的性能自动调优.

2.2 优化空间参数化

首先, 本文需要对后续自动调优流水线的优化技术组合进行参数化. 表 2 以 3D Stencil 计算为例展示了参数优化空间, 其中 M_n 为输入网格第 n 维的长度. GeST 从 1 开始, 以单位步长表示布尔类型参数(如 *useShared*)和枚举类型参数(*SD*). 这种处理方式确保了计算变异系数时涉及到的除法操作的合法性. 与现有工作一致[17], GeST 将数值参数的值限制为 2 的幂. 注意对参数进行分组时, GeST 对数值操作执行 lb 运算以便计算变异系数的输入是连续的. 这保证了不同参数对的相关性的公平比较.

Table 2 Optimization Space of Stencil's Parameters on GPU Platform

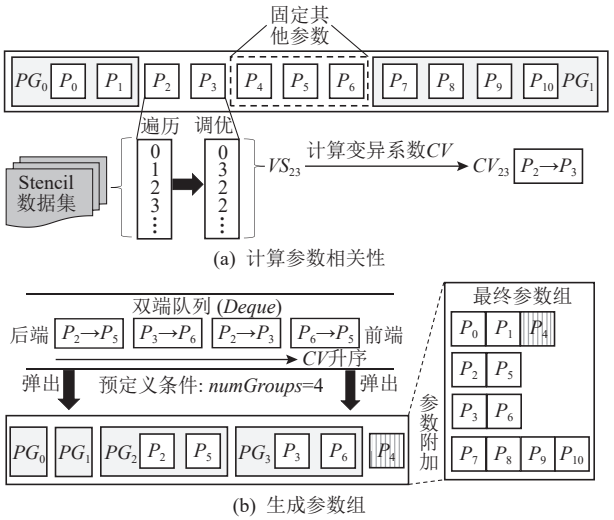
表 2 面向 GPU 平台的 Stencil 参数优化空间

优化技术	参数	范围
线程块维度	TB_x, TB_y, TB_z	[1,1 024], [1,2 024], [1,64]
共享内存	*useShared*	{1,2}
固定内存	*useConstant*	{1,2}
流处理	*useStreaming*	{1,2}
流处理维度	*SD*	{1,2,3}
并发流处理	*SB*	[1,M_{SD}]
循环展开	UF_x, UF_y, UF_z	[1,M_1], [1,M_2], [1,M_3]
循环合并	CM_x, CM_y, CM_z	[1,M_1], [1,M_2], [1,M_3]
块合并	BM_x, BM_y, BM_z	[1,M_1], [1,M_2], [1,M_3]
重定时	*useRetiming*	{1,2}
预取	*usePrefetching*	{1,2}
时间分块	*useTB*	{1,2}

除了表 2 中所示的参数范围外, 优化的参数之间还有一些显式约束. 例如, TB 大小(即 $TB_x \times TB_y \times TB_z$)必须小于或等于 1 024; *TB* 和 *SB* 参数仅在启用流处理优化时有效; 启用并发流时, 流维度中的循环展开因子必须不大于 *SB*. 即便考虑这些显式约束, 总搜索空间仍包含超过 1 亿个参数设置. 此外, GPU 的有限存储资源也带来了隐式约束. 例如, 块合并和循环展开的参数设置受寄存器数量和共享内存大小的限制. GeST 在生成搜索代码之前检查上述约束, 以便在性能调优期间仅执行有效的参数设置.

2.3 参数分组

参数分组的原则是将强相关的参数置于同一组以方便后续的搜索空间采样过程. 如图 3 所示, 考虑到 Stencil 计算的特性, GeST 预先生成 2 个参数组 PG_0 和 PG_1, 其中 PG_0 包含 TB 维度, 而 PG_1 包含循环合并



(a) 计算参数相关性

(b) 生成参数组

Fig. 3 Workflow of parameter grouping

图 3 参数分组的工作流程

和块合并的相关参数. 对于其他单独参数, GeST 需要量化成对参数之间的相关性:

$$CV = \frac{\sigma}{\mu} = \frac{\sqrt{\frac{1}{n}\sum_{i=1}^{n}(x_i - \bar{x})^2}}{\bar{x}}. \tag{1}$$

假设有 N 个单独参数, GeST 每次在固定剩余 $N-2$ 个参数时更改 2 个参数的设置. 以 P_2 和 P_3 为例, 其取值范围分别为 R_2 和 R_3. GeST 从 R_2 中选择 P_2 的设置, 最终得到在 P_2 固定下达到最佳性能的 P_3 设置列表(VS_{23}). 需要注意的是, 如果 R_2 中的某个设置不在 Stencil 数据集内, 则应跳过该设置. 接下来, GeST 计算 VS_{23} 的变异系数 CV_{23}. 变异系数 *CV* 定义为标准差σ与平均值μ的比值, 在统计领域用于反映数据的离散程度, 图 3 中较高的 *CV* 意味着较低的参数相关性. 通过重复上述过程, 最终得到代表参数相关性的 *CV* 序列. 随后, GeST 根据 *CV* 的升序将参数对推入双端队列 *Deque*. GeST 通过算法 1 来实现参数分组.

算法 1. 基于双端队列的参数分组过程.

输入: 参数列表 *paraList*, 初始参数组 *initGroup*, *CV*升序双端队列 *Deque*, 目标参数组数 *numGroups*;

输出: 最终参数组 *paraGroup*.

① *paraGroup*, *gCount* ← *initGroup*, *initGroup.size*;

② while *Deque.size* > 0 do

③ *ftPara*, *bkPara* = *Deque.popright*(); /*右侧弹出*/

④ if *gCount* < *numGroups* then

⑤ if *ftPara* in *paraGroup* then

⑥ *paraGroup.append*([*ftPara*]);

⑦ *gCount* ← *gCount* + 1;

```
⑧       end if
⑨       if bkPara in paraGroup then
⑩           paraGroup.append([bkPara]);
⑪           gCount ← gCount + 1;
⑫       end if
⑬   end if
⑭   if gCount == numGroups then
⑮       break;
⑯   end if
⑰   while Deque.size > 0 do
⑱       ftPara,bkPara = Deque.popleft(); /*左侧
          弹出*/
⑲       if ftPara in paraGroup then
⑳           if bkPara not in paraGroup then
㉑               paraGroup[ftIn].append(bkPara);
㉒           end if
㉓       end if
㉔       if bkPara in paraGroup then
㉕           if ftPara not in paraGroup then
㉖               paraGroup[bkIn].append(ftPara);
㉗           end if
㉘       end if
㉙   end while
㉚ end while
㉛ while paraList.any() not in paraGroup do
㉜   paraGroup[mSize].append(paraList[notIn]);
       /*将剩余单独参数附加到最小规模的参
       数组*/
㉝ end while
```

如算法 1 所示，GeST 首先从 *Deque* 的右侧弹出参数对(行③). 如果某个参数不在已有的参数组中，则创建新的参数组存储该参数(行④~⑬). 当已有参数组数等于目标参数组数时，跳出该循环(行⑭~⑯). 接下来，GeST 从 *Deque* 的左侧弹出参数对(行⑱). 如果仅有 1 个参数不在已有的参数组中，则将该参数合并到已有的参数组中(行⑲~㉘). 最后，GeST 检查是否还有剩余的单独参数，并将这些参数附加到最小规模的参数组中(行㉛~㉝)，以保证参数组的规模平衡. 通过上述过程，GeST 获得了针对特定 Stencil 计算的最终参数组.

2.4 搜索空间采样

GeST 采用了一种简明但高效的搜索空间采样机制，无需大量离线分析来训练性能模型，这得益于其基于配额的奖励策略. 如图 4 所示，经过参数分组

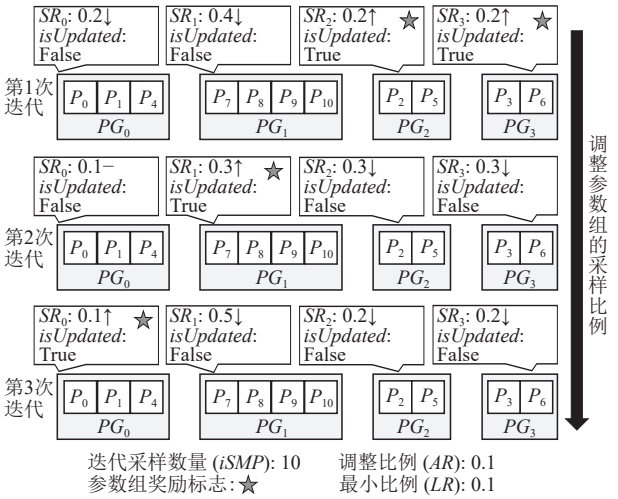


Fig. 4 Quota-based reward policy

图 4 基于配额的奖励策略

后，Stencil 计算的搜索空间已大幅缩小，此时不适合使用遗传算法或模拟退化法等方法来解决优化问题. 例如，遗传算法需要庞大的种群来进行交叉和迁移，这意味着每次迭代都要测试大量的参数设置，对于小搜索空间而言代价过高，几轮迭代几乎等同于穷尽搜索. 若选择较小的种群规模，则会因个体数量过少而陷入局部最优. 因此，GeST 在每次迭代中遍历参数组，并根据执行性能反馈动态调整采样比例，从而确保搜索过程沿着更优参数设置的方向进行.

算法 2 说明了基于奖励策略的搜索空间采样过程，其原则是增加能够带来性能提升的参数组的采样比例，以奖励它们的性能潜力. 首先，GeST 根据每个参数组包含的组合数量来初始化每个参数组的采样比例. 在计算参数组的组合数量时，GeST 会检查优化参数之间的显式约束. 接下来，GeST 迭代地从参数组中采样设置，直到达到预定义的最大采样数量 *mSMP*(行②). 具体来说，GeST 在每次迭代之前重置奖励参数组列表 *upList* 和总采样比例 *totSR*(行③). GeST 遍历参数组，并从每个参数组的有效设置 *validSET* 中执行不放回的随机采样(行⑤~⑦). 有效设置是指满足约束条件且不与已采样设置重复的设置. 为了实现不放回的随机采样，GeST 将每个参数设置编码为唯一的哈希值并存储在字典中，以便快速查找不重复的设置. 如果某个参数组采样到的设置表现优于当前最佳设置，则更新最佳设置，并将该参数组添加到 *upList* 中(行⑧~⑩).

算法 2. 基于奖励策略的搜索空间采样过程.

输入：总参数设置 *totSET*，最佳参数设置 *optSET*，参数组 *paraGroup*，最大采样数量 *mSMP*，迭代采样数

量$iSMP$, 采样比例SR, 调整比例AR, 最低比例LR;

输出: 采样参数设置字典$sampledDict$.

① $numGroups, numSMP \leftarrow paraGroup.size, 0$;
② while $numSMP <= mSMP$ do
③ $upList, totSR \leftarrow [], 0$; /*重置奖励参数组列表*/
④ for id in range [1, $numGroups$] do
⑤ $validSET \leftarrow totSET(optSET[paraGroup_{id}])$;
⑥ $iSET \leftarrow validSET.sample(iSMP \times SR_{id})$;
⑦ if $iSET.isUpdate$ ==True then
⑧ $upList.append(id)$;
⑨ $optSET.update()$; /*更新最佳参数设置*/
⑩ end if
⑪ $sampledDict.append(iSET)$; /*字典存储*/
⑫ $numSMP \leftarrow numSMP + iSET.size$;
⑬ end for
⑭ for id in range [1, $numGroups$] do
⑮ if id in $upList$ then
⑯ continue; /*跳到下一个参数组*/
⑰ end if
⑱ if $SR_{id} \geqslant LR + AR$ then
⑲ $SR_{id} \leftarrow SR_{id} - AR$; /*降低采样比例*/
⑳ $totSR \leftarrow totSR + SR_{id}$;
㉑ end if
㉒ end for
㉓ for id in $upList$ do
㉔ $SR_{id} \leftarrow (1 - totSR)/upList.size()$; /*将剩余的比例配额均分给奖励参数组*/
㉕ end for
㉖ end while

GeST 将奖励参数组的设置限制在单次迭代内, 而不是在整个循环内进行管理, 以避免某个参数组由于在早期表现突出而主导采样过程. GeST 根据奖励结果调整每个参数组的采样比例. 具体来说, 将其他参数组的采样比例减去固定的调整比例 AR, 然后将剩余的采样比例配额均分给奖励参数组(行⑮~㉗). 由于最优参数设置往往被较差的设置包围[26], 最优参数设置也可能由平均表现平庸的参数组所决定. 为此, GeST 为每个参数组的采样比例设置了下限(LR), 以确保在之前迭代中没有突出表现的参数组仍有机会获得奖励, 从而进行“反击”. 基于配额的奖励策略实现了对 Stencil 优化空间的全局探索, 在保证搜索广度的同时提高了采样质量.

3 实验与结果分析

3.1 实验设置

1)硬件平台和 Stencil 程序. 如表 3 所示, 本文在 1 台配备 Intel Xeon E5-2680 v4 CPU 和 2 块 NVIDIA Tesla V100 GPU 的服务器上进行评测. 实验运行在 Ubuntu 16.04 操作系统上, 使用的编译器为 GCC v7.5 和 NVCC v10.1. 本文选取了 21 个典型 Stencil 程序用于实验测试, 这些程序涵盖了不同的阶数、维度和形状. 本文将 2D Stencil 和 3D Stencil 程序的输入网格大小分别设置为 $8\,192^2$ 和 512^3.

Table 3 Hardware Specifications

表 3 硬件规格

服务器参数	CPU	GPU
型号	Intel Xeon E5-2680 v4	NVIDIA Tesla V100
频率/GHz	2.4	1.5
核心	28	13 440 (80 SMs)
缓存	32 KB L1, 256KB L2, 35 MB L3	6 MB L2
内存	378 GB DDR4	32 GB HBM2
带宽/GBps	76.8	900

2)搜索方法和实现细节. 本文将 GeST 与 5 种流行的参数调优方法进行比较, 包括 GSTuner[11], AN5D[9], Artemis[6], OpenTuner[21], RandSearch. RandSearch 在参数搜索空间中进行不放回随机采样, 并在每次采样后更新最佳参数设置. GSTuner 通过训练后的回归模型指导搜索空间采样, 采样比例设置为 10%. 对于 GeST 和 GSTuner, AR 和 LR 均设置为 0.1. 为了支持全局优化探索, 本文将 AN5D, Artemis, OpenTuner 的参数设置预处理为零填充格式. GSTuner 和 OpenTuner 均采用遗传算法, 其具有 2 个子种群, 每个子种群包含 16 个个体; 交叉率和突变率分别设置为 0.8 和 0.005.

3)对比指标. 衡量参数调优有效性的关键指标是找到最佳设置所需的时间. 因此, 本文将 GeST 与上述方法进行等时搜索质量的比较, 即所有方法都在固定的时间(100 s)内运行. 等时比较能够充分验证调优过程的收敛性. 为了降低随机性的影响, 本文将每种方法运行 10 次并给出平均结果.

3.2 自动调优性能

图 5 显示了不同参数调优方法在典型 Stencil 计算模式的等时比较, 其中横轴表示搜索时间, 纵轴表示 Stencil 计算的最短执行时间. 需要注意的是, 对于

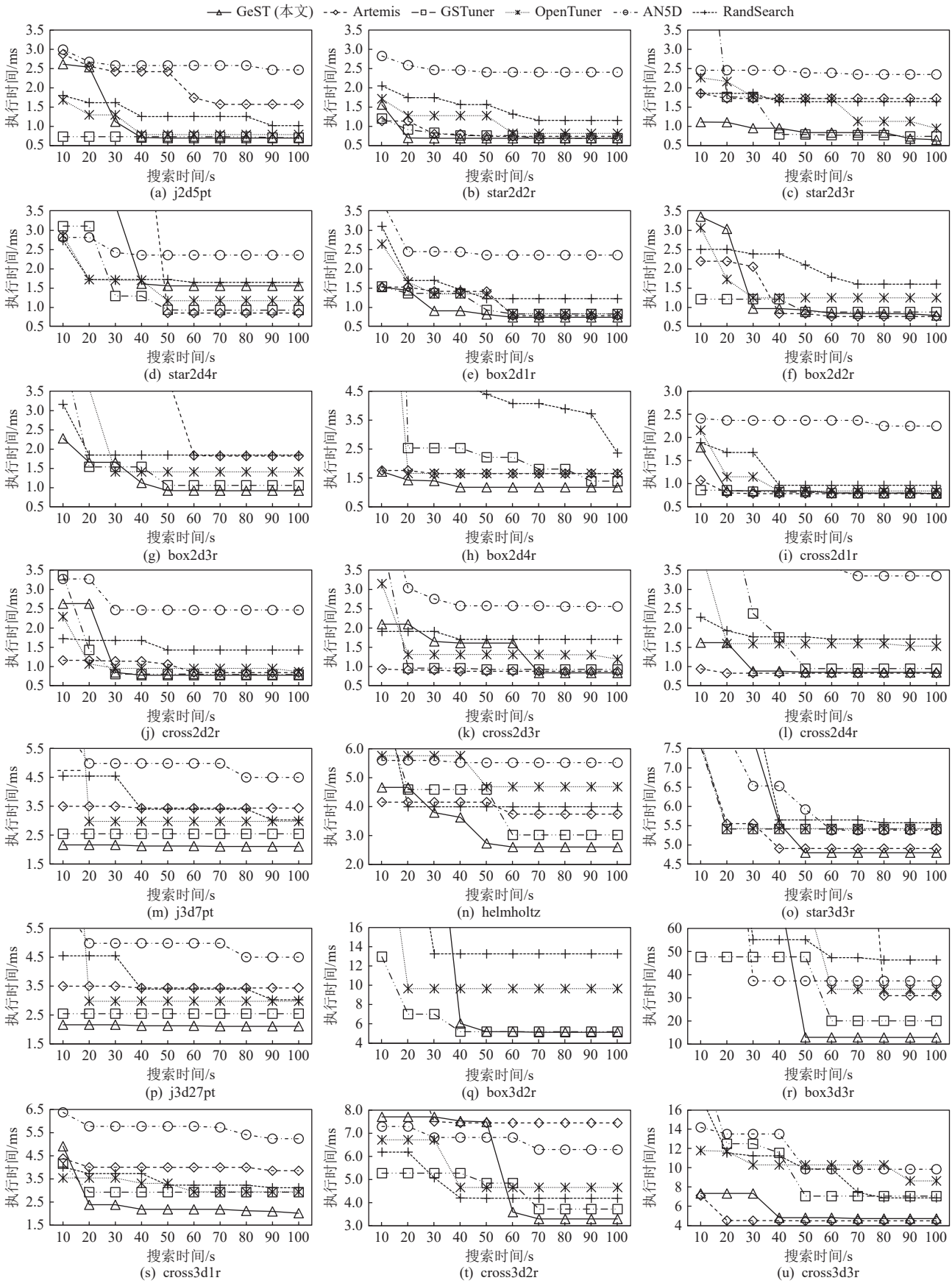


Fig. 5 Auto-tuning efficiency comparison of typical Stencil patterns on V100 GPU

图 5 V100 GPU 上典型 Stencil 模式的自动调优效率对比

GeST, 图 5 中已包含获取 Stencil 数据集所需的时间. 如图 5 所示, GeST 的收敛速度相对较快, 能够在短时间内找到高性能的参数设置. 这表明 GeST 的奖励策略确保了参数组中采样设置的高质量. 尽管 Artemis 的固定参数组同样显著降低了搜索空间, 但其激进的分组策略使其难以捕捉到高性能的设置. 相比之下, OpenTuner 和 RandSearch 缺乏降低搜索空间的机制, 因此在短时间内无法实现收敛.

图 6 显示了参数调优方法相对于 RandSearch 的归一化等时性能. 从图 6 中可以看出, GeST 在大多数 Stencil 计算模式下表现最佳, 其最佳比例达到 71.4%. 在等时评估中, GeST 相对于 GSTuner, AN5D, Artemis, OpenTuner, RandSearch 的平均加速比分别为 1.1, 3.3, 1.6, 1.5, 1.8. 这证明 GeST 采用的参数分组策略和搜索空间采样机制能够有效地推广到各种 Stencil 计算模式. 相比之下, AN5D 和 Artemis 的性能不稳定, 这是因为 AN5D 的激进参数范围缩减和 Artemis 的基于经验的分层调优都难以全面覆盖高性能设置. 例如, 在 box2d3r 下, Artemis 的最终执行时间甚至是 GeST 的 2.9 倍. OpenTuner 由于种群规模较小, 容易陷入局部最优. 选择较小的种群规模是为了充分利用遗传算法, 否则在短时间内迭代次数太少则更接近于随机搜索. RandSearch 具有很强的随机性, 缺乏沿执行时间降低方向的指导性搜索, 平均结果未能体现出良好的性能表现. 尽管 GSTuner 的性能接近 GeST, 但其需要训练回归模型来指导采样过程, 而收集训练数据的离线分析成本较高. 总之, 与其他参数调优方法相比, GeST 能够以更快的速度识别更优的参数设置.

3.3 其他硬件架构

为了证明 GeST 方法的通用性, 本文在配备 2 块 NVIDIA Tesla A100 GPU 的平台上进行实验评估. 具体来说, 本文在新的硬件架构上重新采用参数自动调优方法来搜索高性能配置. 图 7 显示了在 A100 GPU 上归一化为 RandSearch 的等时性能. 结果表明, 对于大多数 Stencil, GeST 的表现优于其他参数调优方法. GeST 在 61.9% 的 Stencil 程序上表现最佳, 而 GSTuner 和 Artemis 的最佳表现比例为 28.6% 和 9.5%. 此外, GeST 相对于 GSTuner, AN5D, Artemis, OpenTuner, RandSearch 分别实现了 1.0, 3.3, 2.2, 1.3, 1.5 的加速比. GSTuner 和 GeST 在平均性能上接近, 但在某些情况(如 box2d2r 和 cross3d1r)下存在较大差异, 可根据不

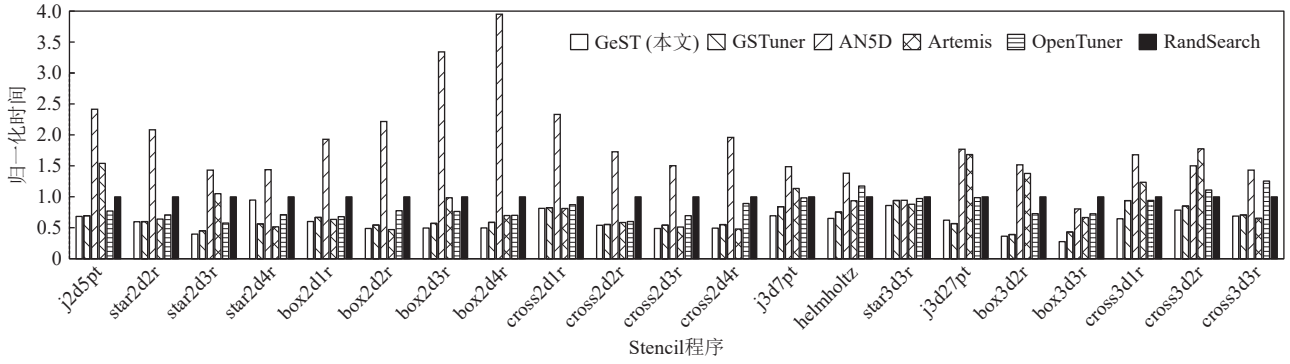


Fig. 6 Auto-tuning performance comparison of parameters on V100 GPU

图 6 V100 GPU 上的参数自动调优性能对比

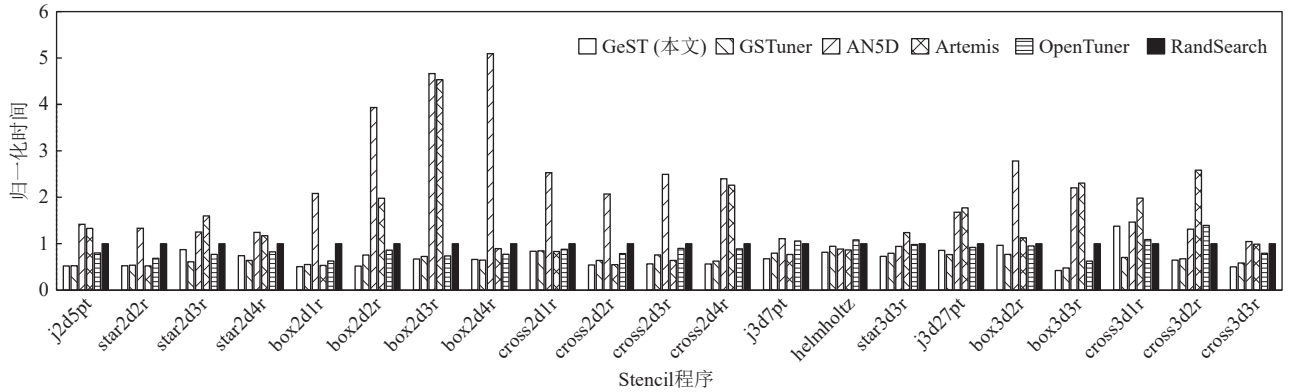


Fig. 7 Auto-tuning performance comparison of parameters on A100 GPU

图 7 A100 GPU 上的参数自动调优性能对比

同场景灵活选择使用. GeST 在 box2d3r 下分别比 AN5D 和 Artemis 快了 6.9 倍和 6.7 倍，这是因为激进的搜索空间剪枝导致性能不稳定. OpenTuner 由于优化空间较大，往往陷入局部最优. RandSearch 在极少数情况（如 box3d2r）下与 GeST 性能相当，但在其他情况下明显不如 GeST. 需要注意的是，参数分组算法和基于奖励的采样策略无需离线性能分析和模型训练，因此 GeST 能够灵活应用于各种硬件平台并且保持稳定的参数调优质量.

3.4 其他网格大小

考虑到不同场景下 Stencil 计算的网格大小可能会有所区别，本文探讨了 GeST 参数调优组件在其他网格大小下的适配性. 本文随机选取了 6 个 Stencil 计算模式，其中 2D 网格大小设置为 $2\,048^2$ 和 $4\,096^2$，3D 网格大小设置为 128^3 和 256^3. 图 8 显示了不同网格大小下的参数调优性能对比，纵轴表示归一化为 RandSearch 的时间. 从图 8 中可以看出，GeST 在大多数 Stencil 下表现优于其他参数调优方法. 具体来说，对于 $2\,048^2$ 和 128^3 的输入网格，GeST 相比 GSTuner, AN5D, Artemis, OpenTuner, RandSearch 分别取得了 1.3, 2.7, 1.4, 1.6, 1.8 的加速比. 而对于 $4\,096^2$ 和 256^3 的输入网格，GeST 的加速比分别为 1.1, 2.9, 1.6, 1.6, 1.9. 结果表明，无论网格大小如何变化，GeST 都能够很好地适应不同的搜索空间范围，并体现卓越的参数自动调优性能，而无需对框架组件进行额外调整.

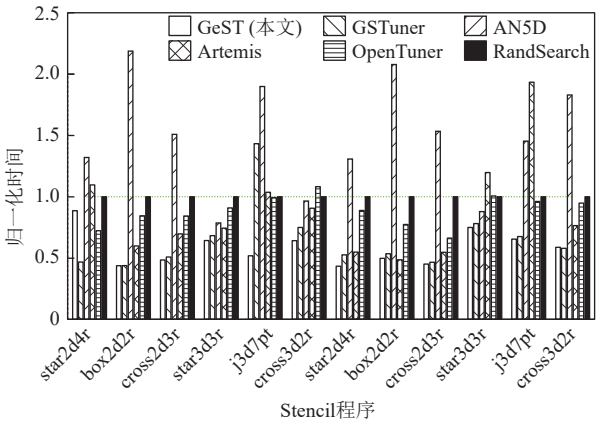


Fig. 8 Auto-tuning performance comparison of parameters under other grid sizes

图 8 其他网格大小下的参数自动调优性能对比

3.5 参数敏感性分析

1）Stencil 数据集规模. 图 9 显示了不同 Stencil 数据集规模的参数调优性能对比，纵轴表示归一化为 RandSearch 的时间，sd_X 指代 Stencil 数据集共有 X 个实例. 如图 9 所示，归一化时间都小于 1，证明 GeST

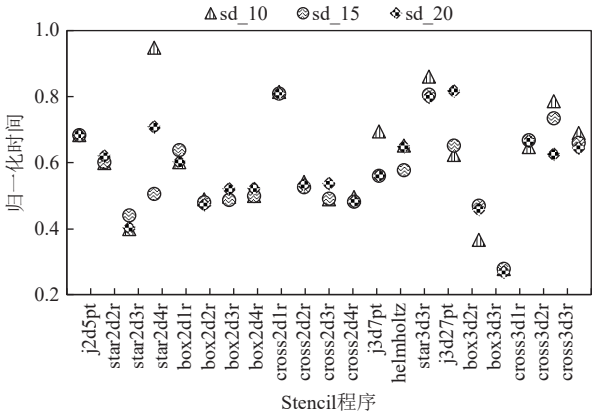


Fig. 9 Auto-tuning performance comparison of parameters with different Stencil dataset sizes

图 9 不同 Stencil 数据集规模的参数调优性能对比

在所有情况下的表现均优于 RandSearch. 值得注意的是，GeST 在收集 Stencil 数据集时，其过程实际上是随机搜索，因此这样的结果充分证明了基于配额的奖励策略能够使 GeST 快速地向性能更优的设置收敛. 整体来看，sd_10 在超过半数情况（52.4%）下表现最差. 这表明 GeST 需要一定规模的 Stencil 数据集才能实现合理的参数分组. 另一方面，sd_15 的表现相对 sd_20 更为稳定，在 42.9% 的情况下表现最佳. 因此，应将 Stencil 数据集规模控制在合理范围（如 15~20 个实例），以便在限定时间内迭代更多搜索步骤以提升最终性能.

2）参数组数量. 图 10 显示了不同参数组数量的参数自动调优性能对比，其中 pg_X 指代 GeST 设置了 X 个参数组. 如图 10 所示，GeST 在大多数情况下的表现优于 RandSearch. 整体来看，pg_3 在 52.4% 的情况下表现最差. 这是因为只有足够多的参数组才能进行细粒度的采样比例调整，以充分挖掘每个参

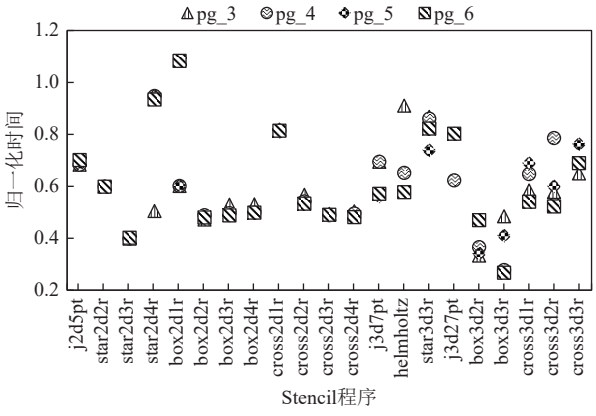


Fig. 10 Auto-tuning performance comparison with different numbers of parameter groups

图 10 不同参数组数量的自动调优性能对比

数的性能潜力. pg_6 的性能极其不稳定，在 box2d1r 下甚至比 RandSearch 更差. 原因在于，当设置的参数组过多时，性能影响相近的参数可能会被置于不同组，无法发挥参数联合调优的快速性能提升优势. pg_4 和 pg_5 的性能表现相对稳定，但 pg_5 的表现更为出色，在 47.6% 的情况下取得最佳表现. 鉴于参数组数量是影响调优效率的关键因素，针对未来 Stencil 计算中可能涉及到的更多优化参数，可以根据实际需求灵活调整参数组数量，从而平衡搜索范围和搜索速度.

3.6 开销分析

GeST 的处理开销可划分为参数分组开销和搜索空间采样开销 2 部分. 参数分组开销主要包括基于变异系数计算的相关性量化和基于双端队列结构的分组算法. 搜索空间采样开销则涉及基于奖励策略的参数设置选择和基于哈希编码与字典结构的设置去重存储. 需要注意的是，GeST 的处理开销已纳入固定时间的实验评估中. 图 11 显示了处理开销在整个参数调优流水线时间中的归一化结果. 如图 11 所示，处理开销几乎可以忽略不计，平均只占参数调优流水线的 0.97‰. 这意味着参数调优的大部分时间都花在启动和执行 Stencil 计算上. 由于 GeST 的轻量级和通用特性，它能够高效地对大规模 Stencil 应用中的多样化计算进行批量调优，从而提升整个应用的性能.

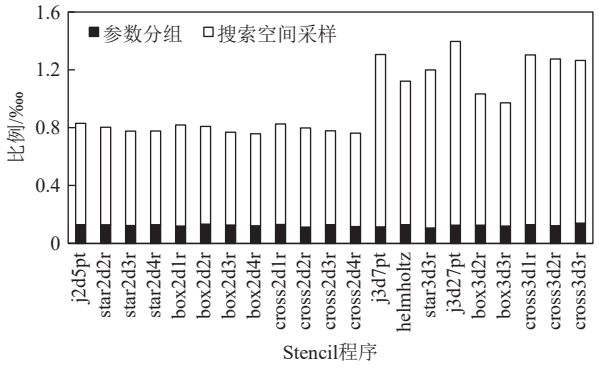


Fig. 11 Time breakdown of processing overhead

图 11 处理开销的时间分解

4 相关工作

4.1 Stencil 领域特定语言

基于 Stencil 计算的常规模式，研究工作将优化技术集成到 DSL 中来实现自动代码转换. PATUS[27] 允许用户通过类 C 的语法定义 Stencil 并选择预定义和自定义的优化策略. Physis[28] 将用户编写的 Stencil 代码转换为 GPU 集群的可扩展实现. Forma[4] 提出 DSL 专用于具有 Stencil 操作的图像处理应用. Grosser 等人[7] 将六边形分块和 wavefront 分块在 GPU 上相结合以实现混合分块方法. Hagedorn 等人[5] 使用 LIFT 原语来实现 Stencil 代码和优化技术. Rawat 等人[29] 使用有向无环图进行阻塞、流式传输和资源管理来优化性能. StencilFlow[30] 提供从高级程序描述到分布式硬件的端到端分析和映射. 曹杭等人[31] 以“自然块”的新概念标识不同形状的特征并针对星型 Stencil 设计 2 层密铺方案. Bisbas 等人[32] 实现由 MLIR 方言组成的编译组件以用于在分布式 Stencil 计算中进行信息传递. 上述工作[27-32] 普遍缺乏对性能自动调优的有效支持，而 GeST 能够集成到 DSL 中以确定目标优化组合的最佳设置，且无需引入离线成本.

4.2 GPU 性能自动调优

由于识别最佳内核变体对于程序员或代码生成器来说极具挑战性，大量研究致力于通过自动调优技术来最大化 GPU 程序的性能. 孙相征等人[33] 设计对角线稀疏存储格式并根据应用矩阵的特点进行自适应 SpMV 性能优化. Li 等人[34] 预先计算临界点并选择全局最优来解决并发性和寄存器使用之间的冲突. Pfaffe 等人[35] 将分层在线自动调优与多面体并行化相结合以降低搜索复杂度并提高收敛速度. Ansor[36] 对优化组合进行采样并利用进化搜索和成本模型来精调张量程序. LLaMA[37] 通过动态运行基于成本的优化器并配置单独的操作调用来遍历空间. GTuner[38] 通过图注意网络共同学习结构和统计特征以找到最佳的代码实现. 曹坤等人[39] 提取 OpenCL 应用的特性并基于改进交叉熵策略的调度技术以提高设计点寻优效率. PARALiA[40] 通过硬件抽象来优化通信并利用自动调优和性能建模来提高 BLAS 例程的性能. PrecTuner[41] 整合混合精度代码生成器和自动调优器以预测定义参数的最佳性能值. Li 等人[42] 提出基于分析特征的机器学习预测模型和动态梯度下降搜索策略来降低自动调优时间. 上述工作能够用于优化 Stencil 计算，而 GeST 因其轻量级的全局搜索机制也可灵活扩展到其他目标程序.

5 结束语

本文提出了一种面向通用 Stencil 计算的性能自动调优框架 GeST. 首先，GeST 通过随机搜索获取 Stencil 数据集，基于变异系数指标量化参数之间的相

关性，从双端队列弹出参数对来生成参数组. 之后，GeST 利用基于配额的奖励策略来指导参数组的迭代搜索，并通过哈希编码和字典存储来避免冗余设置采样. 实验结果表明，与当前最先进的方案相比，GeST 能够在更短时间内识别到高质量的参数设置.

未来的工作包括以下 3 个方面：1)将 GeST 应用于 Stencil 计算的其他硬件环境，例如分布式异构系统和边缘计算设备；2)将 GeST 扩展到拥有更大搜索空间的应用领域，例如深度学习中的张量优化；3)针对 GPU 上的 Stencil 计算，建立分析模型以指导性能优化并明确性能上界. 为了实现上述目标，需要对 GeST 的方法和框架进行充分解耦并形成插件化的设计，使其能够快速嵌入目标场景，实现非侵入式的参数自动调优. 在此基础上，构建 Roofline 模型以深入分析性能瓶颈，并评估不同优化策略在各种 Stencil 模式下所能达到的性能极限.

作者贡献声明：孙庆骁提出了算法思路和实验方案，完成实验并撰写论文；杨海龙提出了指导意见并修改论文.

参 考 文 献

[1] Mullapudi R T, Vasista V, Bondhugula U. PolyMage: Automatic optimization for image processing pipelines[C]//Proc of the 20th Int Conf on Architectural Support for Programming Languages and Operating Systems. New York: ACM, 2015: 429−443

[2] Gamell M, Teranishi K, Heroux M A, et al. Local recovery and failure masking for Stencil-based applications at extreme scales[C/OL]//Proc of the 27th Int Conf for High Performance Computing, Networking, Storage and Analysis. New York: ACM, 2015[2024-07-15]. https://dl.acm.org/doi/abs/10.1145/2807591.2807672

[3] Sano K, Hatsuda Y, Yamamoto S. Multi-FPGA accelerator for scalable Stencil computation with constant memory bandwidth[J]. IEEE Transactions on Parallel and Distributed Systems, 2013, 25(3): 695−705

[4] Ravishankar M, Holewinski J, Grover V. Forma: A DSL for image processing applications to target GPUs and multi-core CPUs[C]//Proc of the 8th Workshop on General Purpose Processing Using GPUs. New York: ACM, 2015: 109−120

[5] Hagedorn B, Stoltzfus L, Steuwer M, et al. High performance stencil code generation with LIFT[C]//Proc of the 16th ACM/IEEE Int Symp on Code Generation and Optimization. New York: ACM, 2018: 100−112

[6] Rawat P S, Vaidya M, Sukumaran-Rajam A, et al. On optimizing complex Stencils on GPUs[C]//Proc of the 33rd IEEE Int Parallel and Distributed Processing Symp. Piscataway, NJ: IEEE, 2019: 641–652

[7] Grosser T, Cohen A, Holewinski J, et al. Hybrid hexagonal/classical tiling for GPUs[C]//Proc of the 12th ACM/IEEE Int Symp on Code Generation and Optimization. New York: ACM, 2014: 66−75

[8] Ragan-Kelley J, Barnes C, Adams A, et al. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines[J]. ACM SIGPLAN Notices, 2013, 48(6): 519−530

[9] Matsumura K, Zohouri H R, Wahib M, et al. AN5D: Automated stencil framework for high-degree temporal blocking on GPUs[C]//Proc of the 18th ACM/IEEE Int Symp on Code Generation and Optimization. New York: ACM, 2020: 199–211

[10] Sun Qingxiao, Liu Yi, Yang Hailong, et al. csTuner: Scalable auto-tuning framework for complex Stencil computation on GPUs[C]// Proc of the 23rd IEEE Int Conf on Cluster Computing. Piscataway, NJ: IEEE, 2021: 192−203

[11] Sun Qingxiao, Liu Yi, Yang Hailong, et al. Adaptive auto-tuning framework for global exploration of Stencil optimization on GPUs[J]. IEEE Transactions on Parallel and Distributed Systems, 2024, 35(1): 20−33

[12] Sun Qingxiao, Liu Yi, Yang Hailong, et al. QoS-aware dynamic resource allocation with improved utilization and energy efficiency on GPU[J]. Parallel Computing, 2022, 113: 102958

[13] Rawat P S, Rastello F, Sukumaran-Rajam A, et al. Register optimizations for Stencils on GPUs[C]//Proc of the 23rd ACM SIGPLAN Symp on Principles and Practice of Parallel Programming. New York: ACM, 2018: 168−182

[14] Sun Qingxiao, Liu Yi, Yang Hailong, et al. StencilMART: Predicting optimization selection for Stencil computations across GPUs[C]//Proc of the 36th IEEE Int Parallel and Distributed Processing Symp. Piscataway, NJ: IEEE, 2022: 875–885

[15] Yuan Liang, Cao Hang, Zhang Yunquan, et al. Temporal vectorization for Stencils[C/OL]//Proc of the 33rd Int Conf for High Performance Computing, Networking, Storage and Analysis. New York: ACM, 2021[2024-07-15]. https://dl.acm.org/doi/abs/10.1145/3458817.3476149

[16] Li Kun, Yuan Liang, Zhang Yunquan, et al. Reducing redundancy in data organization and arithmetic calculation for Stencil computations[C/OL]//Proc of the 33rd Int Conf for High Performance Computing, Networking, Storage and Analysis. New York: ACM, 2021[2024-07-15]. https://dl.acm.org/doi/abs/10.1145/3458817.3476154

[17] Garvey J D, Abdelrahman T S. Automatic performance tuning of Stencil computations on GPUs[C]//Proc of the 44th Int Conf on Parallel Processing. Piscataway, NJ: IEEE, 2015: 300−309

[18] Stock K, Kong M, Grosser T, et al. A framework for enhancing data reuse via associative reordering[C]//Proc of the 35th ACM SIGPLAN Conf on Programming Language Design and Implementation. New York: ACM, 2014: 65−76

[19] Oh C, Zheng Zhen, Shen Xipeng, et al. GoPipe: A granularity-oblivious programming framework for pipelined Stencil executions on GPU[C]//Proc of the 29th Int Conf on Parallel Architectures and Compilation Techniques. New York: ACM, 2020: 43−54

[20] Luo Yulong, Tan Guangming, Mo Zeyao, et al. FAST: A fast Stencil autotuning framework based on an optimal solution space model[C]//Proc of the 29th Int Conf on Supercomputing. New York: ACM, 2015: 187−196

[21] Ansel J, Kamil S, Veeramachaneni K, et al. OpenTuner: An extensible

framework for program autotuning[C]//Proc of the 23rd Int Conf on Parallel Architectures and Compilation Techniques. New York: ACM, 2014: 303−316

[22] Trümper L, Ben-Nun T, Schaad P, et al. Performance embeddings: A similarity-based transfer tuning approach to performance optimization[C]//Proc of the 37th Int Conf on Supercomputing. New York: ACM, 2023: 50−62

[23] Martínez V, Dupros F, Castro M, et al. Performance improvement of stencil computations for multi-core architectures based on machine learning[J]. Procedia Computer Science, 2017, 108: 305−314

[24] Cosenza B, Durillo J J, Ermon S, et al. Autotuning Stencil computations with structural ordinal regression learning[C]//Proc of the 31st IEEE Int Parallel and Distributed Processing Symp. Piscataway, NJ: IEEE, 2017: 287−296

[25] Nesterenko B, Yi Qing, Lin P H, et al. Modeling optimization of stencil computations via domain-level properties[C]//Proc of the 13th Int Workshop on Programming Models and Applications for Multicores and Manycores. New York: ACM, 2022: 35−44

[26] Roy R B, Patel T, Gadepally V, et al. BLISS: Auto-tuning complex applications using a pool of diverse lightweight learning models[C]//Proc of the 42nd ACM SIGPLAN Int Conf on Programming Language Design and Implementation. New York: ACM, 2021: 1280−1295

[27] Christen M, Schenk O, Burkhart H. PATUS: A code generation and autotuning framework for parallel iterative stencil computations on modern microarchitectures[C]//Proc of the 25th IEEE Int Parallel and Distributed Processing Symp. Piscataway, NJ: IEEE, 2011: 676−687

[28] Maruyama N, Nomura T, Sato K, et al. Physis: An implicitly parallel programming model for Stencil computations on large-scale GPU-accelerated supercomputers[C/OL]//Proc of the 23rd Int Conf for High Performance Computing, Networking, Storage and Analysis. New York: ACM, 2011[2024-07-15]. https://dl.acm.org/doi/abs/10.1145/2063384.2063398

[29] Rawat P S, Vaidya M, Sukumaran-Rajam A, et al. Domain-specific optimization and generation of high-performance GPU code for Stencil computations[J]. Proceedings of the IEEE, 2018, 106(11): 1902−1920

[30] de Fine Licht J, Kuster A, De Matteis T, et al. StencilFlow: Mapping large Stencil programs to distributed spatial computing systems[C]//Proc of the 19th IEEE/ACM Int Symp on Code Generation and Optimization. Piscataway, NJ: IEEE, 2021: 315−326

[31] Cao Hang, Yuan Liang, Huang Shan, et al. A parallel star Stencil algorithm based on tessellating[J]. Journal of Computer Research and Development, 2020, 57(12): 2621−2634 (in Chinese)
(曹杭, 袁良, 黄珊, 等. 一种基于空间密铺的星型 Stencil 并行算法[J]. 计算机研究与发展, 2020, 57(12): 2621−2634)

[32] Bisbas G, Lydike A, Bauer E, et al. A shared compilation stack for distributed-memory parallelism in Stencil DSLs[C]//Proc of the 29th ACM Int Conf on Architectural Support for Programming Languages and Operating Systems. New York: ACM, 2024: 38−56

[33] Sun Xiangzheng, Zhang Yunquan, Wang Ting, et al. Auto-tuning of SpMV for diagonal sparse matrices[J]. Journal of Computer Research and Development, 2013, 50(3): 648−656 (in Chinese)
(孙相征, 张云泉, 王婷, 等. 对角线稀疏矩阵的 SpMV 自适应性能优化[J]. 计算机研究与发展, 2013, 50(3): 648−656)

[34] Li Ang, Song S L, Kumar A, et al. Critical points based register-concurrency autotuning for GPUs[C]//Proc of the 53rd Design, Automation & Test in Europe Conf & Exhibition. Piscataway, NJ: IEEE, 2016: 1273−1278

[35] Pfaffe P, Grosser T, Tillmann M. Efficient hierarchical online-autotuning: A case study on polyhedral accelerator mapping[C]//Proc of the 33rd Int Conf on Supercomputing. New York: ACM, 2019: 354−366

[36] Zheng Lianmin, Jia Chengfan, Sun Minmin, et al. Ansor: Generating high-performance tensor programs for deep learning[C]//Proc of the 14th USENIX Symp on Operating Systems Design and Implementation. Berkeley, CA: USENIX Association, 2020: 863−879

[37] Romero F, Zhao M, Yadwadkar N J, et al. LLaMA: A heterogeneous & serverless framework for auto-tuning video analytics pipelines[C/OL]//Proc of the 12th ACM Symp on Cloud Computing. New York: ACM, 2021[2024-07-15]. https://dl.acm.org/doi/abs/10.1145/3472883.3486972

[38] Sun Qi, Zhang Xinyun, Geng Hao, et al. GTuner: Tuning DNN computations on GPU via graph attention network[C]//Proc of the 59th ACM/IEEE Design Automation Conf. New York: ACM, 2022: 1045−1050

[39] Cao Kun, Long Saiqin, Li Zhetao. Lifetime-driven OpenCL application scheduling on CPU-GPU MPSoC[J]. Journal of Computer Research and Development, 2023, 60(5): 976−991 (in Chinese)
(曹坤, 龙赛琴, 李哲涛. CPU-GPU MPSoC 中使用寿命驱动的 OpenCL 应用调度方法[J]. 计算机研究与发展, 2023, 60(5): 976−991)

[40] Anastasiadis P, Papadopoulou N, Goumas G, et al. PARALiA: A performance aware runtime for auto-tuning linear algebra on heterogeneous systems[J]. ACM Transactions on Architecture and Code Optimization, 2023, 20(4): 1−25

[41] Xu Jinchen, Song Guanghui, Zhou Bei, et al. A holistic approach to automatic mixed-precision code generation and tuning for affine programs[C]//Proc of the 29th ACM SIGPLAN Annual Symp on Principles and Practice of Parallel Programming. New York: ACM, 2024: 55−67

[42] Li Chendi, Xu Yufan, Saravani S M, et al. Accelerated auto-tuning of GPU kernels for tensor computations[C]//Proc of the 38th ACM Int Conf on Supercomputing. New York: ACM, 2024: 549−561



Sun Qingxiao, born in 1996. PhD, associate professor. Member of CCF. His main research interests include heterogeneous runtime system and performance optimization.

孙庆骁, 1996 年生. 博士, 副教授. CCF 会员. 主要研究方向为异构运行时系统、性能优化.



Yang Hailong, born in 1985. PhD, professor. Distinguished member of CCF. His main research interests include high performance computing and intelligent computing system.

杨海龙, 1985 年生. 博士, 教授. CCF 杰出会员. 主要研究方向为高性能计算、智能计算系统.