DuaLip-GPU 技术报告

Gregory Dexter、Aida Rahmattalabi*、Sanjana Garg、Qinquan Song、Ruby Tu、Yuan Gao、Yi Zhang、Zhipeng Wang*、Rahul Mazumder

LinkedIn

arXiv:2603.04621v1 [cs.DC] 4 Mar 2026

1 引言

线性规划（LP）是许多大规模决策系统的基础：这些系统包括排序、分配，以及在生产环境中按固定周期求解的各种匹配问题。该领域的先前工作——ECLIPSE 的岭正则化对偶上升，以及 LinkedIn 在 Scala/Spark 技术栈中开源的*DuaLip* ——表明，一阶方法能够扩展到这些场景，但原有系统与两种固定模式和以 CPU 为中心的运行环境紧密耦合，导致难以表达新的建模形式，也无法有效利用现代加速器。

我们对该求解器进行重新架构，使其能够复用并适配现代硬件。最终成果是一个 Python 库及系统，它协同设计了：（i）用于接入问题逻辑的组合式接口；（ii）适用于具有可分解“简单约束”（将在下文定义）的超大规模线性规划的算法工具包；以及（iii）在实践中充分发挥可用并行性所需的 GPU 执行技术。该库并未采用单一的“调用求解器”入口，而是公开了算子级角色，使用户能够在本地添加新的线性规划建模形式，以及表示简单约束族的投影映射，同时保持求解循环和诊断功能不变[†]。

该架构有意保持通用性：其设计面向广泛的线性规划类别，而非任何单一模式。不过，在本报告中，我们重点关注带有匹配约束 的线性规划，将其作为主要应用领域，因为这类工作负载在我们的生产环境中占主导地位，并暴露出最严苛的算法和系统瓶颈。因此，我们以匹配线性规划作为该库及其实现的贯穿案例。

在这些匹配工作负载上，与此前的分布式 CPU *DuaLip* 相比，我们的设计在达到固定次优性间隙方面实现了数量级的实际运行时间提升，至少更快，同时能够实现从单个 GPU 到多个 GPU 的可预测缩放。

贡献。 本工作结合了受生产环境应用启发的三个方面：

- 以算子为中心的编程模型和库接口。 我们以面向岭正则化对偶上升的以算子为中心的模型，替代 DuaLip 受模式约束的模板化接口（多目标优化与单块匹配）。问题逻辑仅通过三个原语表达：objectives/（封装数据和对偶梯度计算）、projections/（对可分解约束多面体执行分块投影）以及 optimizer/（使用对偶梯度执行对偶上升）；而求解循环、诊断和分布式原语则由各部分共享。这种分离使系统能够以极少的额外代码和纯粹的局部组合支持新的建模形式，例如纳入多个约束族。

- 岭正则化对偶上升的算法改进。基于 ECLIPSE/DuaLip 框架，我们通过 Jacobi 式行归一化、原始变量缩放以及针对正则化参数 γ 的简单延续方案，强化了底层方法。我们分析了这些改动对条件性和一阶优化器收敛的影响。

- 面向超大规模线性规划的图形处理器执行与稀疏布局。 我们展示了如何通过选择与约束结构相匹配的稀疏布局、批量处理投影算子，来实现图形处理器上可用的并行性

*通讯作者：arahmattalabi@linkedin.com, zhipwang@linkedin.com

†本文介绍的线性规划求解器代码可在https://github.com/linkedin/DuaLip获取。

以摊销核函数启动开销，并采用仅传递对偶变量的分布式模式。对于我们研究的结构化匹配工作负载，仅这些系统级选择，在停止准则匹配的条件下，相较于先前的分布式CPU DuaLip，便可实现至少 $10\times$ 的墙钟加速。

这些改动共同将岭正则化对偶上升从一种专用且受模式约束的部署方式，转变为灵活、高性能的求解器架构，适用于现代 GPU 加速流水线。在本报告接下来的部分中，我们将形式化阐述促使我们进行设计的匹配线性规划，介绍以算子为中心的编程模型，提出算法改进，并详细说明 GPU 执行机制，同时展示在超大规模匹配工作负载上的实验。

2 相关工作

我们的工作建立在多个方向的既有研究之上。

用于大规模线性规划的一阶方法。投影一阶方法近期已成为求解超大规模线性规划的实用方案，其中最具代表性的是 PDLP [2] 及其图形处理器实现 cuPDLP [4]。这些求解器具有强有力的理论收敛保证，并在高精度基准测试中展现出有竞争力的性能，测试涵盖路由问题和通用线性规划实例。然而，其实现依赖经过精心设计、针对通用稀疏矩阵-向量运算优化的定制核函数，因此将约束矩阵视为非结构化算子。由此，它们并未显式利用对偶分解或对角块结构，也尚未证明能够扩展至我们目标工作负载所特有的极端问题规模。

相比之下，我们的设定面向超大规模匹配与分配工作负载，这类工作负载具有高度结构化的约束族，且精度要求适中。我们不追求普遍的高精度解，而是着重在可分解的约束结构下快速收敛至具有经济意义的对偶解。我们的方法在对偶更新中显式利用对角块结构和匹配特有结构，从而支持按列进行分布式分解，并通过集合通信高效聚合。这种针对结构的专门设计，使我们的方法有别于通用的一阶线性规划求解器。

大规模匹配与分配。 大量研究致力于大规模加权匹配与指派问题，包括专用二分图匹配求解器和基于网络流的表述。虽然这些方法在典型匹配约束下表现出色，但通常仅适用于固定的约束类别（例如度约束、容量约束），无法自然地纳入生产级分配系统中出现的其他任意线性约束。

相比之下，我们的建模形式在保留一般线性规划灵活性的同时，也保留了匹配结构在对偶中的计算优势。因此，我们无需修改求解器核心，即可对异构约束族进行编码。

岭正则化对偶上升。 Basu 等人 [3]提出的岭正则化对偶形式表明，在原始问题中添加 ℓ_2 正则化会使对偶问题变得光滑，从而能够针对极大规模分配问题进行可扩展的一阶优化。Ramanath等人 [6] 进一步开发了高效的投影算法，并将这些方法集成到可复用的 Scala/Spark 系统（DuaLip）中。

然而，以往的实现与以 CPU 为中心的数据格式和僵化的声明式模式紧密耦合。具体而言，系统只提供少量固定模板（例如单块匹配和多目标优化），因此难以表达具有多个相互作用约束族或更一般块结构的问题，除非修改求解器内部实现。这些局限促使我们重新设计系统，着重提升可组合性、灵活的约束指定能力和 GPU 原生执行能力。

高性能张量框架。 我们的实现利用 PyTorch [5], 等现代数组编程框架，简洁地表达稀疏-稠密操作、批量投影和分布式集合通信。与针对固定 LP 抽象定制的底层内核不同，这种方法能够快速迭代算法变体和结构化表述，同时在现代 GPU集群上保持高性能。

综合来看，这些工作脉络为当前求解器所依托的编程模型、算法设计与分布式执行机制提供了依据。

3 问题设定

工业界的许多大规模优化任务可归约为按周期性节奏求解规模极大的线性规划，例如每日、每周或每次模型更新。示例包括 ECLIPSE 中的电子邮件与通知分配[3]，DuaLip 中的内容与机会分配 [6]，以及近期的市场塑造问题，例如促进非活跃成员 [1]。在每种情况下，随着新的分数、资格信号或需求到来，线性规划实例会发生变化，但建模形式的底层结构保持稳定。

在这类周期性场景中，主要挑战不只是高精度地求解单个大型线性规划，而是在不同运行过程中数据发生变化时，确保稳定性、可扩展性和快速收敛。经典线性规划建模形式本质上并非强凸，而且往往会出现退化。在原始问题中引入 ℓ_2 正则化可带来强凸性，从而得到具有利普希茨连续梯度的光滑对偶目标函数。这使基于梯度的一阶对偶方法能够更快收敛，并形成适用于分布式分解的良态结构。在周期性工业工作负载中，相较于获得原始线性规划的精确顶点解，这些性质往往更为重要。

3.1 预备知识

给定如下形式的线性规划：

$$\min \quad \mathbf{c}^T\mathbf{x} \;\; \text{such that } \mathbf{A}\mathbf{x} \leq \mathbf{b}, \mathbf{x} \in \mathcal{C}, \tag{1}$$

其中，$\mathbf{x} \in \mathbb{R}^n$ 是原始变量向量，$\mathbf{c}$ 是原始目标系数向量，$\mathbf{A} \in \mathbb{R}^{m\times n}$ 和 $\mathbf{b}$指定了“复杂”约束，而 $\mathcal{C}$ 表示“简单”约束所构成的多面体。

我们采用 ECLIPSE 的算法方法 [3]，，通过岭正则化扰动线性规划，并使用基于梯度的方法最大化所得的对偶函数。当原始变量位于某些常见多面体中时，这种方法尤其高效，后文将对此进行介绍。具体而言，ECLIPSE 在式 (1) 的原始目标函数中加入岭项 $\frac{\gamma}{2}\|\mathbf{x}\|_2^2$，并求解所得的拉格朗日松弛问题。

令 $\boldsymbol{\lambda} \in \mathbb{R}^m$ 表示复杂约束的对偶值。对偶目标函数定义如下：

$$g(\boldsymbol{\lambda}) = \min_{\mathbf{x}\in\mathcal{C}} \mathbf{c}^T\mathbf{x} + \frac{\gamma}{2}\|\mathbf{x}\|_2^2 + \boldsymbol{\lambda}^T(\mathbf{A}\mathbf{x} - \mathbf{b}) \tag{2}$$

根据 Danskin 定理，$\nabla \mathbf{g}(\boldsymbol{\lambda})$ 可通过 $\nabla \mathbf{g}(\boldsymbol{\lambda}) = \mathbf{A}\mathbf{x}^*_\gamma(\boldsymbol{\lambda}) - \mathbf{b}$ 高效计算，其中 Π 是到 $\mathcal{C}$ 的投影，并且：

$$\mathbf{x}^*_\gamma(\boldsymbol{\lambda}) = \Pi_{\mathcal{C}}\left(\frac{-1}{\gamma}(\mathbf{A}^T\boldsymbol{\lambda} + \mathbf{c})\right).$$

随后，任何仅依赖梯度 $\nabla \mathbf{g}(\boldsymbol{\lambda})$ 的优化算法，都可用于在$\boldsymbol{\lambda} \geq \mathbf{0}$上最大化 $g(\boldsymbol{\lambda})$，并由强对偶性求解扰动后的原始线性规划。我们采用这种算法方法，是因为当 Π 可通过分布式方式应用时，通信开销会大幅降低。这适用于工业界许多重要的超大规模线性规划应用。

3.2 匹配问题建模

尽管我们的方法具有普适性，但我们聚焦于超大规模匹配问题：在这类问题中，用户被指派给物品，并受共同的展示容量约束。形式化地，令决策变量 x_{ij} 表示将来源（例如用户） $i \in [I]$ 指派给目标（例如推广活动或物品） $j \in [J]$。对于每个用户 i，定义块向量

$$\mathbf{x}_i := (x_{i1}, \ldots, x_{iJ})^\top,$$

并将这些块堆叠起来，形成完整的决策向量

$$\mathbf{x} = \left[\mathbf{x}_1; \ldots; \mathbf{x}_I\right].$$

令 $E \subseteq [I] \times [J]$ 表示可行的源–目的地对集合。

耦合约束源于共享的目的地层级要求，例如适用于所有符合某个目标条件的用户的预算、投放节奏、频次或投放限制。例如：

$$\sum_{i\in[I]} a_{ij}x_{ij} \le d_j \qquad \forall j \in [J], \quad \text{(destination capacity)} \tag{3}$$

用矩阵形式，$\mathbf{A}$可写为

$$\mathbf{A} = \begin{bmatrix} a_{11} & \cdots & 0 & a_{21} & \cdots & 0 & a_{I1} & \cdots & 0 \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ 0 & \cdots & a_{1J} & 0 & \cdots & a_{2J} & 0 & \cdots & a_{IJ} \end{bmatrix},$$

这可以视为对角块的水平拼接。因此，$\mathbf{A}$ 具有高度稀疏性：由于块对角结构，部分元素为零；此外，许多系数 a_{ij} 对于不合格配对也为零。正如第6节所述，我们利用这种结构化和非结构化稀疏性这一关键特性，实现高效的矩阵向量乘法。

此外，我们通常施加用户级曝光容量约束，这些约束可以表示为简单的“盒”约束或“单纯形”约束， [6], 并分别作用于每个块$\mathbf{x}_i \in \mathcal{C}_i$。这些约束的扩展形式如下：

$$\sum_{j\in[J]} x_{ij} \le 1 \qquad \forall i \in [I], \quad \text{(per-user capacity)} \tag{4}$$

$$0 \le x_{ij} \qquad \forall (i,j) \in E. \quad \text{(feasibility)} \tag{5}$$

简单约束在上述示例中，约束 (4)、(5) 构成简单多面体（单纯形），可通过高效的投影算法处理。更一般地，在将用户分配给物品时，用户的曝光容量可以用“盒切割”或“单纯形”不等式约束表示，并可应用于每个块$\mathbf{x}_{i*}$，彼此独立地 [6]。我们将这些约束称为简单约束。有关此类多面体及其相应投影过程的详细示例，请参见 [6] 。

利用这种可分结构，可以在原始子问题中隐式处理用户侧约束，而无需为其引入显式对偶变量。因此，在将 I 个用户与J 件物品进行匹配时，显式对偶变量的数量可以从 $I + J$ 减少到 J。这显著降低了通信开销，因为计算 $\nabla \mathbf{g}(\boldsymbol{\lambda})$ 时，只需在各 GPU 之间同步目的地层级对偶变量。

匹配约束下面我们介绍匹配问题中复杂约束 $\mathbf{Ax} \le \mathbf{b}$ 的结构。该结构与 ECLIPSE [3], 中的建模形式十分相似；ECLIPSE 推动了现代的岭正则化对偶上升方法的发展。不过，这种结构比 Scala/Spark DuaLip 中实现的版本更通用，后者最多只允许一个匹配块。这里，我们允许存在任意数量的匹配约束族，每个约束族都以简单的逐元素方式与用户块交互。

在匹配问题中，复杂约束会通过共享目标点将许多用户关联起来——例如适用于所有满足 $(i, j) \in E$ 的 i 的预算、投放节奏、频次或投放约束。由于每个此类约束在每个 $\mathbf{x}^{(i)}$ 上都以逐元素方式进入线性规划，因此约束矩阵在各来源之间呈对角分块结构。

定义 1（匹配问题的复杂约束矩阵）。 给定如上堆叠的决策变量 $\mathbf{x}$ ，复杂约束矩阵 $\mathbf{A} \in \mathbb{R}^{mJ\times IJ}$ 按源分块定义如下：

$$\mathbf{A} = \begin{bmatrix} \mathbf{D}_{11} & \mathbf{D}_{12} & \cdots & \mathbf{D}_{1I} \\ \mathbf{D}_{21} & \mathbf{D}_{22} & \cdots & \mathbf{D}_{2I} \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{D}_{m1} & \mathbf{D}_{m2} & \cdots & \mathbf{D}_{mI} \end{bmatrix}, \qquad \mathbf{D}_{ki} \in \mathbb{R}^{J\times J} \ \textit{diagonal for every } (k,i), \tag{6}$$

其中， $k = 1$、.....、 m 表示不同的约束族。每个对角块$\mathbf{D}_{ki}$都对第 i个变量块$\mathbf{x}^{(i)}$逐元素作用。

这种建模形式刻画了 ECLIPSE 中使用的结构化匹配约束，同时允许多个此类约束族（预算、投放节奏、公平性、频次等）同时出现。$\mathbf{D}_{ki}$ 的对角结构是本文始终加以利用的一项关键性质：它简化了对偶梯度，支持在 GPU 上高效采用稀疏布局并批量进行投影，还能得到精确的条件数结果，例如第 5节中的结果。

4 编程模型与库架构

DuaLip 之前的 Scala Spark 版本 [7] 支持了首批生产部署，但仅支持两种僵化的模式，并且依赖一种扩展成本高昂的对象模型。其运行时也难以适配常用的数值工具和 Python 工作流，因此诊断和调优基本都要手动完成。我们将线性规划求解重新架构为命令式、算子级数组/张量编程模型 （PyTorch 风格，定义即运行），而非任务级的“调用求解器” API。具体而言，关键路径是由稀疏矩阵-向量算子 ($\mathbf{Ax}$、$\mathbf{A}^\top\boldsymbol{\lambda}$）和分块投影构成的显式数据流，由一个小型最大化器负责协调。这一边界经过刻意设计：它展现了主导运行时的内核，使我们能够调整稀疏布局/内核和投影算子，并且可以直接映射到 GPU 执行，而无需重写求解循环。

该库旨在支持这种通用场景：线性规划满足 (i) 简单约束可在变量块之间自然分解，因此适合采用基于投影的方法；(ii) 稀疏复杂约束的主导操作是重复应用$\mathbf{Ax}$ 和$\mathbf{A}^\top\boldsymbol{\lambda}$。这涵盖了组织内各垂直团队遇到的许多指派、分配和路由问题；编程模型和求解器接口正是针对这一层次设计的。

在这一通用模板中，匹配问题是一个尤为重要且普遍存在的子类。在这类问题中，每个“来源”都有自己的一组变量和简单约束，而跨来源约束（例如预算、投放节奏和频次上限）则通过简单的逐元素项与各组变量相互作用。正如先前研究所示，这会使复杂约束矩阵呈现出独特的对角分块结构。我们在定义1 中将这种结构形式化，并表明它能够实现高效的图形处理器内核、批量投影算子以及有效的预条件。本文重点讨论匹配工作负载，但该库本身并不局限于单个匹配块，甚至也不限于匹配约束；对于任何简单约束可按块分解、复杂约束可通过稀疏算子调用实现的线性规划，相同的抽象均适用。

尽管本文其余部分以匹配线性规划作为贯穿全文的示例，本节中的库接口并未涉及匹配专属概念。该库由三个相互解耦、契约要求极少的组件构成：

组件	**构建自**	**必须提供（单个方法）**
最大化器	算法设置（步长策略、迭代预算、标志）	最大化(obj, 初始_值) → 结果
目标函数	线性规划张量与元数据（$\mathbf{A}$,$\mathbf{b}$,$\mathbf{c}$）加上所提供的投影映射	计算(λ, γ) →目标结果
投影映射	映射 from 原始列/块 to 投影算子（单纯形、盒切割，.....）	投影(块_id, v) → 已投影 v

Table 1：核心对象、由什么构造它们，以及各自暴露的唯一必需方法 s.

一种新的线性规划建模形式实现了一个目标函数类，它封装了线性规划参数，并提供一种从对偶值 $\boldsymbol{\lambda}$ 计算 $\nabla \mathbf{g}(\boldsymbol{\lambda})$ 的方法。在实践中，这需要确保矩阵-向量乘积 $\mathbf{Ax}$ 和 $\mathbf{A}^T\boldsymbol{\lambda}$ 能够在图形处理器上高效地表示和计算。此外，梯度计算还需要将 $\Pi(\mathbf{x}^*_\gamma(\boldsymbol{\lambda}))$ 投影到简单约束多面体 $\mathcal{C}$ 上。简单约束往往具有广泛的适用性，例如逐用户单纯形约束或单位盒约束，因此投影算子可以在新的线性规划建模形式中重复使用。目标函数类封装了线性规划建模形式，并提供计算梯度的方法。这意味着最大化器可以是多种优化算法，用于在对偶可行域 $\boldsymbol{\lambda} \geq \mathbf{0}$ 上对 $\mathbf{g}(\boldsymbol{\lambda})$ 执行对偶上升。

我们认为，求解超大规模线性规划需要达到极高的效率，因此必须能够灵活地在底层修改核心算法原语，尤其是梯度计算，以便在实践中应用于各种建模形式。与此同时，这些修改所需的工程工作量必须尽可能小，并且不同应用之间应能够共享通用原语和合理的默认设置。这种求解器架构的优势在于，主要算法组件都体现在代码结构中，因此修改任何部分都只需进行局部调整。针对具体用例的完整求解器，便是由各个高层组件组合而成，这与 PyTorch 中通过高层组合模型架构、损失函数、优化算法等来构建模型的方式十分相似。

相比之下，基于 Spark Scala 的求解器提供了单一的声明式入口，并支持两种严格模式；随后通过继承层次结构调用必要的算法组件。我们发现，即使某些新约束在算法上表示起来很高效，这种结构也难以支持它们，例如匹配问题中的全局计数约束，$\sum_{ij} x_{ij} \le m$。对于这一约束，计算 $\mathbf{A}\mathbf{x}$ 和 $\mathbf{A}^\top\boldsymbol{\lambda}$ 十分简单，但要将其加入 Spark Scala 求解器中的匹配问题，却需要对整个代码库进行大量修改。

该实现采用原生 Python，以适配常见的数值计算技术栈并支持插桩和迭代。实验日志记录与调优、结构化诊断和标准调试的可选钩子，在不改变这些接口的情况下封装了相同的调用。

5 优化算法

我们的求解器遵循第 3.1 节中的岭正则化对偶上升框架：给定平滑对偶 $g(\boldsymbol{\lambda})$，我们采用一种一阶方法，通过其梯度 g 访问 $\nabla \mathbf{g}(\boldsymbol{\lambda}) = \mathbf{A}\mathbf{x}^*_\gamma(\boldsymbol{\lambda}) - \mathbf{b}$ 以及定义 $x^*_\gamma(\boldsymbol{\lambda})$ 的投影算子。Scala/Spark DuaLip 实现采用 Nesterov 加速梯度下降（加速梯度下降）的一种变体及少量其他优化器来实例化这一框架；为了在大型匹配问题上实现可靠收敛，需要针对具体实例仔细调优，而且对缩放和正则化的选择十分敏感。

在本节中，我们保持相同的对偶形式，但重新设计优化技术栈，从而使单一配置能够在定义 1 的匹配工作负载上稳健运行。具体而言，我们引入三项改动：（i）借助 Jacobi 式行归一化改善对偶问题的条件性，（ii）针对岭参数 γ 采用简单的延续调度，以在曲率与近似误差之间取得平衡，以及（iii）通过原始变量坐标缩放，沿各坐标平衡岭项的平滑。从概念上讲，只要我们能够访问约束矩阵 $\mathbf{A}$ （及其行列统计量）、简单约束投影和平滑对偶目标函数，这些技术即可适用，因此它们与第 4 节的以算子为中心的编程模型兼容。

5.1 算法改进

改进 DuaLip 求解器的一个核心重点，是提升支撑这一系列工作的岭正则化对偶上升法的性能与稳健性。第 3.1 节的基本框架可接纳多种一阶实例化；在实践中，我们在生产环境匹配工作负载上观察到的主要收敛问题来自三个源：（i）$\mathbf{A}\mathbf{A}^\top$ 在对偶中的条件性较差，（ii）岭正则化项所带来的收敛速度与解质量之间的权衡，以及（iii）原始变量的尺度不均一，并与二次正则项产生不良相互作用。我们依次通过 Jacobi 式行归一化、简单的正则化调度和原始变量缩放来加以解决。

雅可比预条件 / 行归一化。我们的第一项改进是通过重新缩放复杂约束来改善对偶问题的条件性。我们假设 $\mathbf{A}$ 是满行秩的。直观地说，当 $\mathbf{A}$ 的某些行的范数远大于其他行时，针对平滑对偶的梯度步在某些方向上过于谨慎，而在另一些方向上又过于激进。

当投影算子未激活（或为恒等算子）时，对偶梯度归约为

$$\nabla \mathbf{g}(\boldsymbol{\lambda}) \;=\; -\frac{1}{\gamma}\big(\mathbf{A}\mathbf{A}^\top\boldsymbol{\lambda} + \mathbf{A}\mathbf{c}\big) \;-\; \mathbf{b},$$

因此海森矩阵为

$$\nabla^2 \mathbf{g}(\boldsymbol{\lambda}) \;=\; -\frac{1}{\gamma}\,\mathbf{A}\mathbf{A}^\top.$$

因此，一阶方法在 g 上的收敛由$\mathbf{A}\mathbf{A}^\top$的条件性决定。

我们应用标准的行缩放变换。令

$$\mathbf{D} = \operatorname{diag}\big(\|\mathbf{A}_{1*}\|_2^{-1}, \dots, \|\mathbf{A}_{m*}\|_2^{-1}\big)$$

对所有具有非零范数的行（具有 $\|\mathbf{A}_{r*}\|_2 = 0$ 的行是冗余的，可以删除，或在$\mathbf{D}_{rr} = 1$下保持不缩放），并定义行缩放系统

$$\mathbf{A}' = \mathbf{D}\mathbf{A}, \qquad \mathbf{b}' = \mathbf{D}\mathbf{b}.$$

由于$\mathbf{D}$具有正对角元素，行缩放精确保持可行集：

$$\{\,\mathbf{x} : \mathbf{A}\mathbf{x} \le \mathbf{b}\,\} \;=\; \{\,\mathbf{x} : \mathbf{A}'\mathbf{x} \le \mathbf{b}'\,\}.$$

此外，

$$\mathbf{A}'\mathbf{A}'^\top \;=\; \mathbf{D}(\mathbf{A}\mathbf{A}^\top)\mathbf{D}, \qquad \mathbf{D}^2 = \operatorname{diag}(\mathbf{A}\mathbf{A}^\top)^{-1} \;\text{(on nonzero rows)},$$

因此，行归一化正是对偶海森矩阵–$\nabla^2 \mathbf{g}(\boldsymbol{\lambda}) \;=\; \frac{1}{\gamma}\,\mathbf{A}\mathbf{A}^\top$的雅可比预条件。 –

对于定义 1中的**匹配约束矩阵**，A 是各个源的对角子块的水平拼接，因此 $\mathbf{A}\mathbf{A}^\top$ 是（近似）对角矩阵之和（每个源对应一个），所以在实际中接近对角矩阵。强制要求 $\operatorname{diag}(\mathbf{A}'\mathbf{A}'^\top) = \mathbf{I}$ 会使谱高度聚集；在理想的对角/正交情形下，它得到$\mathbf{A}'\mathbf{A}'^\top$ =$\mathbf{I}$ ，且条件数为 1。

下面的引理在一个关于匹配块的简单统计模型下，将这一直觉形式化。

引理 *5.1*。设 $A \;=\; [A_1 \cdots A_I] \in \mathbb{R}^{mJ \times IJ}$，其用户块 $A_i \in \mathbb{R}^{mJ \times J}$在 i上独立同分布，且如定义 *1* 所示按行呈对角结构。设 $D^{\exp} = diag\big(\mathbb{E}\|A_{1*}\|_2^2, \dots, \mathbb{E}\|A_{m_2*}\|_2^2\big)^{-1/2}$，并令 $\widetilde{A} \;= D^{\exp} A$。则 $diag\big(\mathbb{E}[\widetilde{A}\widetilde{A}^\top]\big) = I$。此外，如果对于 $r \neq s$,

$$\mathbb{E}\big[\langle \mathbf{A}_{r*}, \mathbf{A}_{s*} \rangle\big] \;\le\; \eta \sqrt{\mathbb{E}\|\mathbf{A}_{r*}\|_2^2\, \mathbb{E}\|\mathbf{A}_{s*}\|_2^2} \quad with\ \eta \in [0, 1),$$

then

$$\kappa\big(\mathbb{E}[\widetilde{\mathbf{A}}\widetilde{\mathbf{A}}^\top]\big) \;\le\; \frac{1 + (m-1)\eta}{1 - (m-1)\eta}.$$

因此，在温和的跨行相关性假设下，对于匹配工作负载，行归一化几乎能使$\mathbf{A}\mathbf{A}^\top$的特征值在期望意义下趋于一致，从而稳定对偶一阶更新。

正则化衰减 上述预条件处理了简单约束处于非活跃状态时的主导因素：在这种情况下，$\mathbf{A}\mathbf{A}^\top$ 的条件数决定了一阶方法的收敛速度。然而，岭参数 γ 还会通过平滑对偶的光滑性 （以及有效强凸性）单独发挥作用。

接下来，我们考虑如何在优化过程中设置正则化超参数。虽然 γ 不会影响 $\mathbf{A}\mathbf{A}^\top$的条件性，但会显著影响目标函数的光滑性。一方面，为避免扰动原始线性规划，人们倾向于取较小的 γ。Basu 等人的引理 2 [3] 保证，总存在一段足够小的 γ 取值范围，使得能够精确恢复未平滑线性规划的一个解。另一方面，梯度的 Lipschitz 常数仅能由 $\|\mathbf{A}\|_2^2/\gamma$ 界定（见 [3]中的引理 3），因此，若 γ 过小，就会导致对偶问题条件性不佳，在实践中进展缓慢。

为了平衡收敛速度与解的保真度，我们采用一种简单的延续方案：将 γ初始化为适中的较大值（以确保前期稳定、快速地推进），随后随着算法逐步逼近良好的对偶解，按照预先设定的计划和速率逐渐衰减。由于 γ 会直接影响对偶目标函数的光滑性，我们将 AGD 的最大步长与 γ 的衰减幅度成比例地缩小，以在各个过渡点保持稳定。总体而言，这种方法能够加快前期迭代的收敛速度，同时确保最终解不会偏离真实的线性规划解太远。

原始变量缩放 ECLIPSE 和 DuaLip 所采用的岭正则化表述有一个局限：二次平滑项 $\frac{\gamma}{2}\|\mathbf{x}\|_2^2$ 隐含地假设 $\mathbf{x}$ 的所有坐标都处于相近的尺度。当 $\mathbf{x}$ 中的某些分量相差几个数量级时，正则化要么会主导较小的坐标，要么会对较大的坐标几乎不起作用。对于适中的 γ，项 $\frac{\gamma}{2}\|\mathbf{x}\|_2^2$ 可能会扭曲目标几何，使最优解偏离真实的线性规划最优解；而对于极小的 γ，平滑作用则会失效，对偶景观也会失去快速收敛所需的曲率。

一种简单的补救方法是对原始坐标进行对角缩放。令$\mathbf{v} \in \mathbb{R}^n_{>0}$ 为正缩放因子组成的向量，并定义缩放变量：

$$\mathbf{z} = \mathbf{D}_v\,\mathbf{x}, \qquad \mathbf{D}_v := \mathrm{diag}(\mathbf{v}),$$

使得$\mathbf{x} = \mathbf{D}_v^{-1}\mathbf{z}$。将其代入正则化原问题，得到

$$\min_{\mathbf{z}\in\mathbf{D_v}^{-1}\mathcal{C}} (\mathbf{D}_v^{-1}\mathbf{c})^\top\mathbf{z} + \tfrac{\gamma}{2}\|\mathbf{z}\|_2^2 \quad \text{s.t.} \quad \mathbf{A}\mathbf{D}_v^{-1}\mathbf{z} \le \mathbf{b},$$

其中

$$\mathbf{D_v}^{-1}\mathcal{C} := \{\mathbf{z}\in\mathbb{R}^n : \mathbf{D_v}^{-1}\mathbf{z} \in \mathcal{C}\}.$$

等价地，

$$\min_{\mathbf{x}\in\mathcal{C}} \mathbf{c}^\top\mathbf{x} + \tfrac{\gamma}{2}\|\mathbf{D}_v\mathbf{x}\|_2^2 \quad \text{s.t.} \quad \mathbf{A}\mathbf{x} \le \mathbf{b}。$$

因此，得到两种代数上等价的视角：（i）将正则项推广为 $\frac{\gamma}{2}\|\mathbf{x}\|^2_{\mathbf{D}_v^2} = \frac{\gamma}{2}\,\mathbf{x}^\top\mathbf{D}_v^2\mathbf{x}$，即用 $\gamma\mathbf{I}$ 替换为 $\gamma\mathbf{D}_v^2$；或（ii）通过 $\mathbf{D}_v$ 对原始空间进行缩放，令

$$\mathbf{c}' = \mathbf{D}_v^{-1}\mathbf{c}, \qquad \mathbf{A}' = \mathbf{A}\mathbf{D}_v^{-1}.$$

通过变量替换，两种视角都会得到相同的优化问题。根据原始坐标的典型量级（或 $\mathbf{A}$ 的列范数）选择$\mathbf{v}$，可以平衡岭项中各坐标的贡献，并对平滑对偶的有效曲率进行归一化。这种缩放可防止正则项在某些坐标上占主导或趋于消失，从而使对偶目标函数在任何适度的 γ 下都具有良好的条件数。

6 图形处理器系统/算法改进

现在，我们转而讨论在现代硬件上高效运行该求解器所需的系统设计。在生产环境中，新建模形式必须能够可靠收敛，同时将工程投入降至最低，并避免大规模重新调优超参数。我们采用的、源自先前工作的岭正则化对偶上升算法尤其适合这一场景：其平滑处理提高了重复求解时的稳定性，并且每次迭代都能在各个区块间轻松并行化，只需对对偶变量进行同步。因此，该方法天然适用于分布式图形处理器执行，也便于以较低开销部署新的稳定建模形式。

定义 1 中的匹配线性规划展现出大规模稀疏线性算子和分块投影，这些都非常适合在图形处理器上运行；但原有的 Scala/Spark DuaLip 技术栈依赖 Java 虚拟机/CPU 运行环境，难以利用适合加速器的布局或通信模式。本节的目标是通过基于 PyTorch 的稀疏与分布式原语，尽可能获得定制 C++/CUDA 实现（例如 PDLP 风格的系统）所能带来的大部分性能优势，同时保持实现轻量。为此，我们选择符合匹配约束结构的稀疏张量布局，通过批处理投影避免产生过小的内核，并采用简单的多图形处理器执行模式，使不同建模形式能够复用相同代码。

稀疏张量计算 实践中，定义 1 中的约束矩阵具有两种稀疏性来源：（i）块对角形式带来的结构化稀疏；（ii）对角块内部的非结构化稀疏，因为受外部因素影响，只有一部分源-目的地对符合条件。

我们可以高效地对这类约束矩阵进行如下操作。设 $\mathbf{A}$ 如定义 1 所示，且仅包含一个匹配约束族，即

$$\mathbf{A} = [\mathbf{D}_1, \ldots, \mathbf{D}_I], \quad \mathbf{x} = [\mathbf{x}_{(1)}; \ldots; \mathbf{x}_{(I)}]$$

我们通过将约束矩阵存储为压缩稀疏列（CSC）格式，并按目标 j 对列排序，来利用结构化稀疏和非结构化稀疏。这样，对应于给定来源的所有变量都能在内存中连续存放。稀疏张量表示的每一列 $\mathcal{T}$ 都满足 $\mathcal{T}[:][i] = \mathrm{diag}(\mathbf{D}_i)$。

在数值精度固定的情况下，CSC 恰好存储了我们需要的结构：每列一个整数指针、每个非零元素一个整数行索引，以及每个非零元素一个值。此外，这使我们能够立即调用逐元素算术运算，通过经过优化的 GPU 张量运算执行对偶上升算法。相比之下，Scala 版本采用的元组/序列方案会产生指针和装箱开销，缓存局部性较差，并且在数学内核运行前还需进行转换和排序；这些都会增加内存流量和运行时间，却没有带来任何额外信息。

批量投影算子 CSC 布局张量中的列 $\mathcal{T}$与 Basu 等人的目标分区相对应。 [3], 这种布局利用块对角结构时非常适合稀疏矩阵-向量乘积，但如果在图形处理器上一次启动一列来执行逐切片投影，效率仍然很低（内核规模很小、启动开销大、占用率低）。因此，我们保留 CSC 张量表示，但改为对稠密填充批次执行投影。

批处理方案采用对数间隔分桶：我们按照切片长度将列分组到若干区间 $[2^{t-1}, 2^t)$。对于每个桶，我们将相关切片收集到一个稠密块中（填充至该桶的上界），应用一个批量投影内核（例如单位盒钳制或单纯形投影），再将结果散布回原处。几何分桶将每个桶内的填充浪费限制在两倍以内，因此临时内存仍处于实际非零元素数量的一个较小常数倍范围内，而 GPU 内核启动次数仅为 $1 + \lfloor \log_2 s_{\max} \rfloor$，其中 $s_{\max}$ 是最大切片长度。

这样无需放弃紧凑的 CSC 存储格式，就能将许多不规则、低占用率的调用转化为少量高占用率的合并访问操作。它还可以与 PyTorch 的批量稠密内核无缝集成，而先前的元组序列风格会妨碍高效打包，并且在数学运算前需要额外转换。

分布式 GPU 通信 我们使用带有 nccl 后端的 torch.distributed，并在所有图形处理器上采用单个进程组。$\mathcal{T}$（以及 $\mathbf{c}$，保持一致）的列按照 CSC 格式矩阵的均衡列划分分配到各设备上，而 $\boldsymbol{\lambda}$ 和约束向量 $\mathbf{b}$则在每台设备上复制。0 号进程在 CPU 上生成并划分完整数据集，然后通过集合散播分发每个进程秩的列分区，接着广播共享约束向量 $\mathbf{b}$。对于固定的 $\boldsymbol{\lambda}$，每台设备都会计算 $\mathbf{x}^*_\gamma(\boldsymbol{\lambda})$ 的本地切片及其对偶量的贡献，并利用简单约束分解（逐个 i 投影和按目标可分离的块）避免计算过程中的跨设备依赖。

每步的通信量仅取决于对偶维度 $|\boldsymbol{\lambda}|$，与非零元素数量或每 GPU 列划分无关。每次迭代按以下步骤进行：

1. 每个进程秩在自己的 CUDA 设备上并发计算局部梯度贡献、局部目标值和局部正则化惩罚。2. 通过归约（SUM，至 0 号进程）汇总梯度（$|\boldsymbol{\lambda}|$ 个浮点数）和两个标量（目标函数、正则化）。3. 0 号进程执行加速梯度上升更新，生成新的迭代变量 $\boldsymbol{\lambda}_1$ 和 $\boldsymbol{\lambda}_2$。4. 通过两次广播调用（从 0 号进程发出），将更新后的 $\boldsymbol{\lambda}_1$ 和 $\boldsymbol{\lambda}_2$ 向量分发给所有其他进程；这些向量构成加速梯度下降更新所用的动量状态。

因此，每步通信包括一次归约和两次广播操作，每次操作的数据量为 $|\boldsymbol{\lambda}|$，外加 O（1）次标量归约——与稀疏度和每 GPU 列划分无关。同样的模式适用于单节点多 GPU 和多节点部署。

7 实验

我们从三个方面评估所提出的 DuaLip PyTorch 实现：（i）与生产环境中的 Scala 实现的数值一致性；（ii）系统级性能和多 GPU 扩展性；以及（iii）预条件和正则化延续等算法改进的影响。

我们使用合成匹配数据，以便对问题规模和稀疏度进行可控缩放。数据生成流程和完整的实验设置详见附录 B。

实现一致性 我们首先验证 PyTorch 实现与原始 Scala 求解器之间的数值等价性。图 1 比较了单图形处理器和多图形处理器设置下，加速梯度下降（AGD）迭代过程中的对偶目标函数轨迹。

所有配置下的轨迹几乎完全重合，证实了实现层面的一致性。为量化差异，图 2 报告了相对于 Scala 求解器的对偶目标函数相对误差。在所有实验中，相对误差在前 100 次迭代内降至 1% 以下，并在此后继续下降。这些结果表明，分布式 PyTorch 实现忠实复现了生产系统的优化动态。

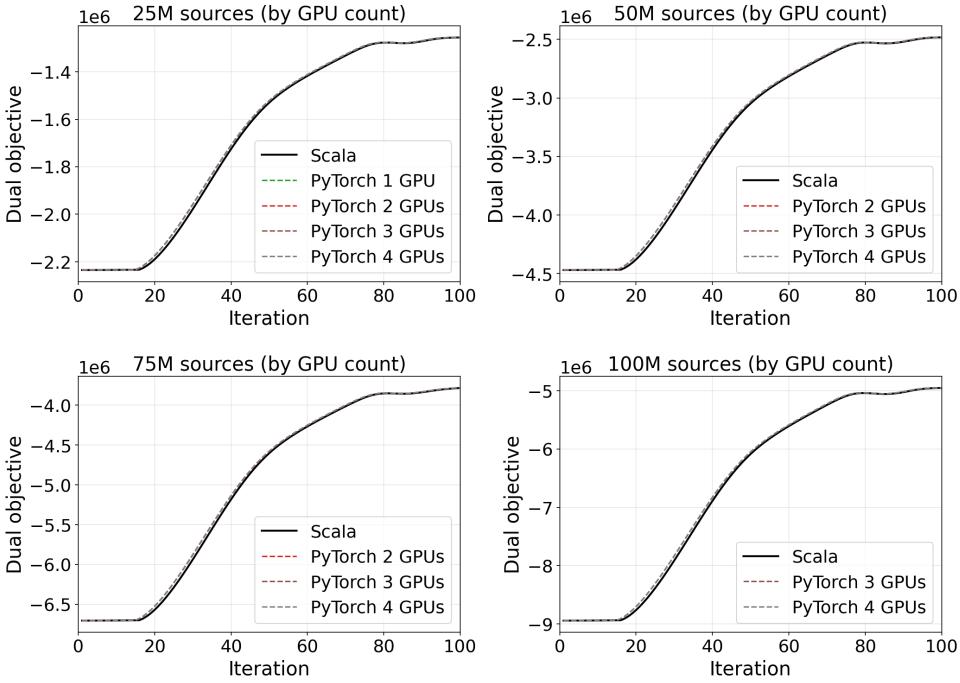

图 1：Scala–DuaLip（ PyTorch）一致性。每个面板展示了对偶目标函数随 AGD 迭代次数的变化，比较对象为 Scala 和 PyTorch 实现 。轨迹几乎完全重合，证实了数值等价性。

系统性能与多 GPU 扩展 接下来，我们比较基于 Spark 的 Scala DuaLip 与 PyTorch GPU 实现的每次迭代运行时间。使用固定的随机种子，我们生成与 Scala 求解器输入模式兼容的相同问题实例，并在可比配置下评估两个系统。

我们未将 PDLP/cuPDLP 纳入此次运行时间比较，因为现有的 GPU 实现仅在单设备上运行，不支持我们最大实例所需的多 GPU 分片机制。由于我们此处的重点是在分布式执行下扩展到极大的问题规模，这样的比较无法反映预期的部署场景。

表 2 报告了每次加速梯度下降迭代的平均时间。对于中等规模的问题（2500万源点），单个图形处理器相较 Scala 已带来一个数量级的提升。随着问题规模增大，必须跨多个 GPU 进行模型分片，以满足内存限制。除了能够支持更大的实例外，多 GPU 并行化还显著缩短了迭代时间，实现了超过 $10\times$ 相对于 Scala 的加速比，并在图形处理器环境下实现近线性扩展。

图 3 进一步展示了缩放行为。随着 GPU 数量增加，求解时间持续缩短，所实现的加速比也趋近理想的线性趋势（4 个 GPU 上为 3。$86\times$ ，而理想值为 $4\times$）。这表明

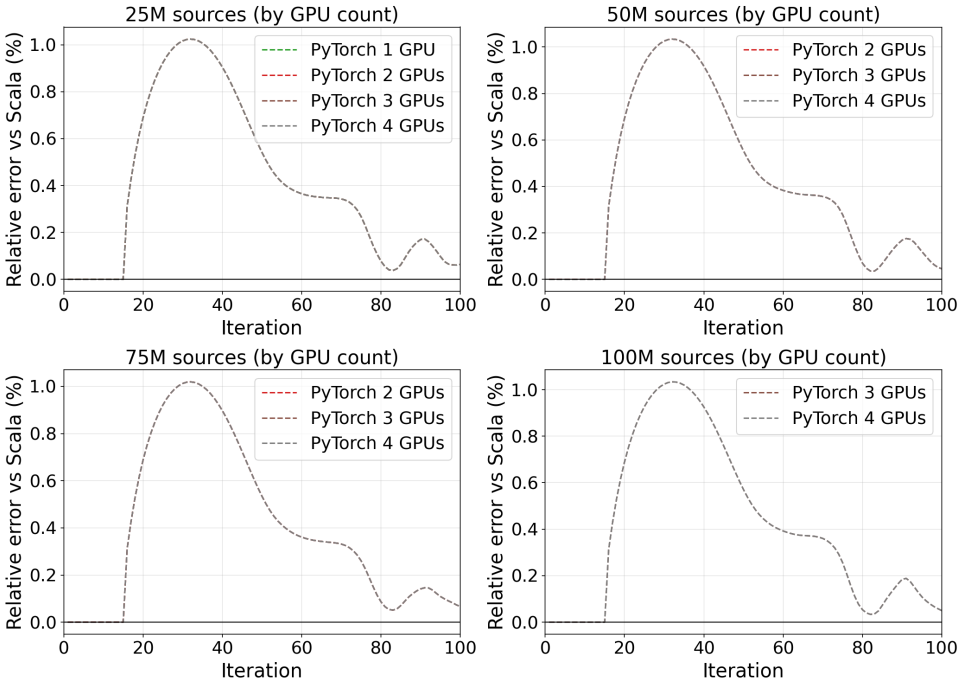

图 2：与 Scala 求解器相比，对偶目标函数的相对误差。所有设置下，误差均在前 100 次迭代内降至 1% 以下。

		DuaLip（PyTorch）			
来源	**Scala 1 个图形处理器**	**2 个 GPU**	**3 个 GPU**	**4 个 GPU**	
25M	2.46	0.27	0.14	0.09	0.07
50M	3.44	-	0.27	0.18	0.13
75M	2.63	-	0.43	0.29	0.21
100M	3.33	-	-	0.37	0.27

表 2：每次加速梯度下降迭代的平均时间（秒）。目标点数量为 $= 10{,}000$，稀疏度为 $= 0.001$。多 GPU 分片支持更大规模的实例，并且相比基于 Spark 的 Scala 实现带来了显著的加速。

求解器能有效并行执行梯度计算，同时将通信开销降至最低。

算法改进 现在，我们将分别考察第 5 节中引入的两项算法改进的影响：对角预条件和正则化延续。

预条件。 图 4 绘制了 $\log(|L - \hat{}L|)$，其中 L 是对偶目标函数，而 $\hat{}L$ 是收敛参考值。采用对角预条件后，收敛速度显著加快，尤其是在迭代初期。这证实了缩放对偶变量能够缓解由约束幅值不均一引起的病态性。

正则化延续。 图5 评估了岭参数 γ。从较大的 γ 开始，可以稳定并加速早期优化；逐步衰减则可确保最终解接近未正则化的线性规划最优解。将 γ 从 0.16 衰减至 0.01（每 25 次迭代减半），比使用固定正则化水平更快收敛。

可复现性 用于复现所有合成基准测试实验的代码可在以下地址获取：
https://github.com/linkedin/DuaLip。

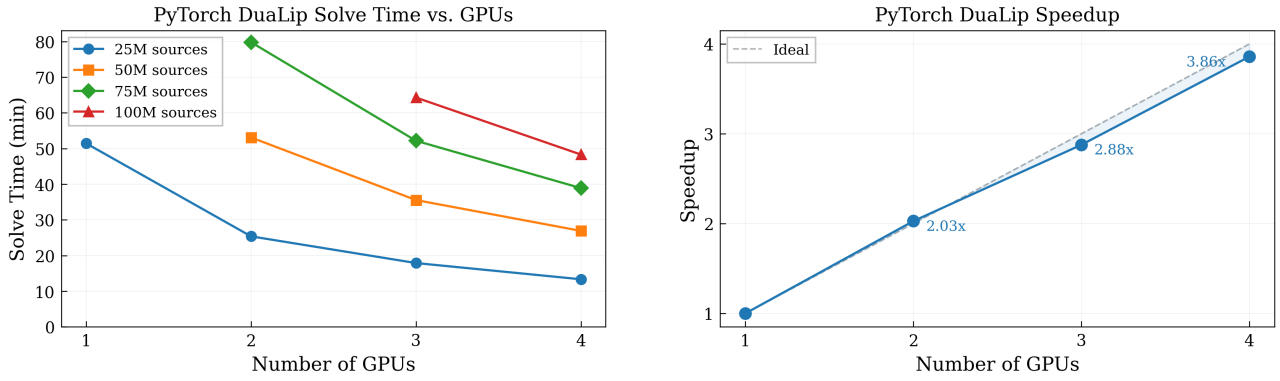

图 3：不同 GPU 数量下的扩展表现。（左）问题规模为 2500 万至 1 亿源点时，求解时间随 GPU 数量的变化。（右）相对于单个 GPU 的加速比，接近理想的线性扩展。

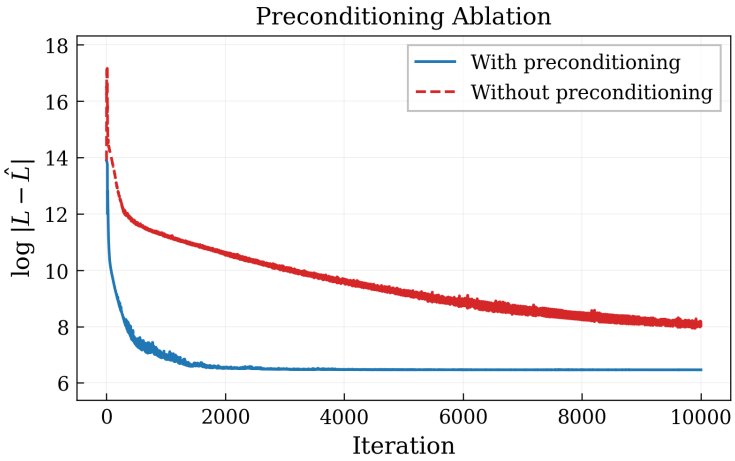

图 4：对角预条件的影响。对于一个 2500 万源点实例（1 万个目标点，稀疏度为 0.1%），我们报告 $\log(|L - \hat{L}|)$。预条件显著改善了早期的收敛。

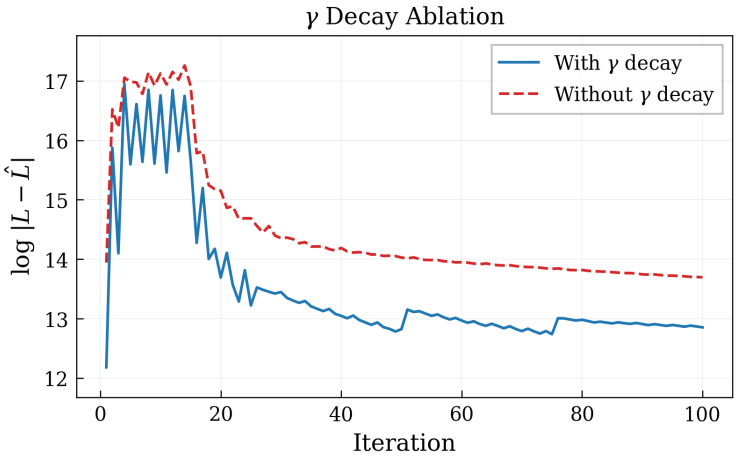

图 5：正则化延续的影响。在优化过程中逐渐减小 γ，可加快收敛，同时保持解的保真度。

8 讨论与局限性

本报告概述了在工业环境中解决超大规模匹配问题所需的主要架构选择、算法改进和 GPU 实现策略。下一步自然是进一步在匹配问题之外的标准线性规划基准测试上评估核心对偶上升法。此外，由于适用于超大规模匹配线性规划的现实世界公开数据集较为稀缺，目前的评估依赖于大型合成实例。建立标准化的基准测试套件和数据源，将有助于对超大规模匹配线性规划开展更加全面且可复现的实证研究。

参考文献

[1] Ayan Acharya、Siyuan Gao、Borja Ocejo、Kinjal Basu、Ankan Saha、Keerthi Selvaraj、Rahul Mazumdar、Parag Agrawal 和 Aman Gupta。在边构建市场中推广非活跃成员。载于*ACM Web Conference 2023* 伴随论文集，第 945–949 页，2023 年。

[2] David Applegate、Mateo Díaz、Oliver Hinder、Haihao Lu、Miles Lubin、Brendan O'Donoghue 和 Warren Schudy。使用原始-对偶混合梯度法进行实用的大规模线性规划。神经信息处理系统进展，34:20243–20257，2021 年。

[3] Kinjal Basu、Amol Ghoting、Rahul Mazumder 和 Yao Pan。Eclipse：面向 Web 应用的超大规模线性规划求解器。发表于 国际机器学习大会，第 704–714 页。PMLR，2020 年。

[4] Haihao Lu 和 Jinwen Yang。cuPDLP.jl：用于 Julia 线性规划的重启原始-对偶混合梯度法的图形处理器实现。运筹学，73(6):3440–3452，2025 年。

[5] Adam Paszke、Sam Gross、Francisco Massa、Adam Lerer、James Bradbury、Gregory Chanan、Trevor Killeen、Zeming Lin、Natalia Gimelshein、Luca Antiga 等。PyTorch：一种命令式风格的高性能深度学习库。神经信息处理系统进展，32，2019 年。

[6] Rohan Ramanath、S Sathiya Keerthi、Yao Pan、Konstantin Salomatin 和 Kinjal Basu。互联网市场全局推断的高效算法。*arXiv* 预印本 *arXiv:2103.05277*，2021 年。

[7] Rohan Ramanath、Sathiya S. Keerthi、Kinjal Basu、Konstantin Salomatin、Pan Yao、Amol Ghoting 和 Miao Cheng。DuaLip：基于对偶分解的线性规划求解器，版本 2.0.0，2022 年 12 月。

A 补充理论

A.1 引理的证明 5.1

证明。则

$$\mathbf{A}\mathbf{A}^\top = \begin{bmatrix}\mathbf{A}_1 & \cdots & \mathbf{A}_I\end{bmatrix}\begin{bmatrix}\mathbf{A}_1^\top \\ \vdots \\ \mathbf{A}_I^\top\end{bmatrix} = \sum_{i=1}^{I}\mathbf{A}_i\mathbf{A}_i^\top.$$

取期望并利用独立同分布块可得 $\mathbb{E}[\mathbf{A}\mathbf{A}^\top] = \sum_{i=1}^{I}\mathbb{E}[\mathbf{A}_i\mathbf{A}_i^\top] = I\,\mathbf{M}$，其中 $\mathbf{M} = \mathbb{E}[\mathbf{A}_1\mathbf{A}_1^\top]$。对于归一化矩阵，

$$\mathbb{E}[\widetilde{\mathbf{A}}\widetilde{\mathbf{A}}^\top] = \mathbf{D}_{\exp}\,\mathbb{E}[\mathbf{A}\mathbf{A}^\top]\,\mathbf{D}_{\exp} = I\,\mathbf{D}_{\exp}\,\mathbf{M}\,\mathbf{D}_{\exp}.$$

根据 $\mathbf{D}_{\exp}$ 的定义，（r，r）元素等于 $I\,\mathbb{E}\|\mathbf{A}_{r*}\|_2^2 / \mathbb{E}\|\mathbf{A}_{r*}\|_2^2 = 1$，因此 $\mathrm{diag}(\mathbb{E}[\widetilde{\mathbf{A}}\widetilde{\mathbf{A}}^\top]) = \mathbf{I}$。对于 $r \neq s$，

$$\left|\mathbb{E}[\widetilde{\mathbf{A}}\widetilde{\mathbf{A}}^\top]_{rs}\right| = \frac{\left|\mathbb{E}\langle\mathbf{A}_{r*},\mathbf{A}_{s*}\rangle\right|}{\sqrt{\mathbb{E}\|\mathbf{A}_{r*}\|_2^2\,\mathbb{E}\|\mathbf{A}_{s*}\|_2^2}} \le \eta.$$

因此，$\mathbb{E}[\widetilde{\mathbf{A}}\widetilde{\mathbf{A}}^\top]$ 的每一行对角元素均为1，且非对角元素绝对值之和至多为$(m-1)\eta$。根据格尔施戈林圆盘定理，所有特征值均位于 $[1-(m-1)\eta,\ 1+(m-1)\eta]$中，因此得到 $\kappa \le \frac{1+(m-1)\eta}{1-(m-1)\eta}$. □

A.2 原始可行性收敛

我们扩展了 [3] 和 [6]的理论分析。具体而言，我们将 ℓ_2-范数原始不可行性界定为对偶目标次优性的函数，从而可以应用标准的一阶收敛理论，根据第3.1节提出的对偶上升法，得到原始可行性保证。

引理 A.1. 设 $\boldsymbol{\lambda}^*$ 为凹对偶函数 g 的唯一最大化点，该函数与线性规划 *1*相关联。若 $\boldsymbol{\lambda} \geq \mathbf{0}$，则原始不可行性的上界为

$$\left\|\left(\mathbf{A}\mathbf{x}_\gamma^*(\boldsymbol{\lambda}) - \mathbf{b}\right)_+\right\|_2 \leq \sqrt{2L\big(g(\boldsymbol{\lambda}^*) - g(\boldsymbol{\lambda})\big)}\,, \tag{7}$$

其中 $L = \|\mathbf{A}\|_2^2/\gamma$ 是 ∇g 的 *Lipschitz* 常数，并且对于向量 $\mathbf{v}$，算子 $(\mathbf{v})_+$ 表示与零取逐分量最大值，即

$$(\mathbf{v}_+)_i = \max(v_i, 0).$$

证明。 设 g 为具有 L-Lipschitz 梯度的凹函数，并定义

$$\boldsymbol{\lambda}_+ \;=\; \Pi_{\mathbb{R}_+^m}\left(\boldsymbol{\lambda} + \frac{1}{L}\nabla\mathbf{g}(\boldsymbol{\lambda})\right), \qquad \mathbf{d} := \boldsymbol{\lambda}_+ - \boldsymbol{\lambda}, \qquad \nabla_+\mathbf{g}(\boldsymbol{\lambda}) := \max(\nabla\mathbf{g}(\boldsymbol{\lambda}), \mathbf{0}).$$

注意， $\nabla_+\mathbf{g}(\boldsymbol{\lambda}) = (\mathbf{A}\mathbf{x}_\gamma^\star(\boldsymbol{\lambda}) - \mathbf{b})_+$。由 g 的 L-光滑性与凹性，

$$g(\boldsymbol{\lambda}_+) - g(\boldsymbol{\lambda}) \;\geq\; \langle\nabla\mathbf{g}(\boldsymbol{\lambda}), \mathbf{d}\rangle - \frac{L}{2}\|\mathbf{d}\|_2^2. \tag{8}$$

设 $\mathbf{y} := \boldsymbol{\lambda} + \frac{1}{L}\nabla\mathbf{g}(\boldsymbol{\lambda})$。由于 $\boldsymbol{\lambda}_+ = \Pi_{\mathbb{R}_+^m}(\mathbf{y})$，投影最优性条件给出

$$\langle\mathbf{y} - \boldsymbol{\lambda}_+, \mathbf{u} - \boldsymbol{\lambda}_+\rangle \leq 0 \qquad \forall\mathbf{u} \in \mathbb{R}_+^m.$$

取$\mathbf{u} = \boldsymbol{\lambda}$（由于 $\boldsymbol{\lambda} \geq 0$，这是有效的），可得

$$\left\langle\boldsymbol{\lambda} + \frac{1}{L}\nabla\mathbf{g}(\boldsymbol{\lambda}) - \boldsymbol{\lambda}_+,\, \boldsymbol{\lambda} - \boldsymbol{\lambda}_+\right\rangle \leq 0.$$

代入$\mathbf{d} = \boldsymbol{\lambda}_+ - \boldsymbol{\lambda}$ 并化简，

$$\left\langle\frac{1}{L}\nabla\mathbf{g}(\boldsymbol{\lambda}) - \mathbf{d},\, -\mathbf{d}\right\rangle \leq 0 \;\;\Longrightarrow\;\; \langle\nabla\mathbf{g}(\boldsymbol{\lambda}), \mathbf{d}\rangle \geq L\|\mathbf{d}\|_2^2.$$

将其代入（8）可得

$$g(\boldsymbol{\lambda}_+) - g(\boldsymbol{\lambda}) \;\geq\; L\|\mathbf{d}\|_2^2 - \frac{L}{2}\|\mathbf{d}\|_2^2 = \frac{L}{2}\|\mathbf{d}\|_2^2.$$

接下来，注意到

$$L\mathbf{d} = L(\boldsymbol{\lambda}_+ - \boldsymbol{\lambda}) = L\left(\Pi_{\mathbb{R}_+^m}\left(\boldsymbol{\lambda} + \frac{1}{L}\nabla\mathbf{g}(\boldsymbol{\lambda})\right) - \boldsymbol{\lambda}\right) = \max\big(\nabla\mathbf{g}(\boldsymbol{\lambda}),\, -L\boldsymbol{\lambda}\big),$$

因此 $\|L\mathbf{d}\|_2 \geq \|\nabla_+\mathbf{g}(\boldsymbol{\lambda})\|_2$。所以

$$g(\boldsymbol{\lambda}_+) - g(\boldsymbol{\lambda}) \;\geq\; \frac{L}{2}\|\mathbf{d}\|_2^2 = \frac{1}{2L}\|L\mathbf{d}\|_2^2 \;\geq\; \frac{1}{2L}\|\nabla_+\mathbf{g}(\boldsymbol{\lambda})\|_2^2.$$

最后，

$$g(\boldsymbol{\lambda}^\star) - g(\boldsymbol{\lambda}) \;\geq\; g(\boldsymbol{\lambda}_+) - g(\boldsymbol{\lambda}) \;\geq\; \frac{1}{2L}\|L\mathbf{d}\|_2^2 \;\geq\; \frac{1}{2L}\|(\mathbf{A}\mathbf{x}^\star(\boldsymbol{\lambda}) - \mathbf{b})_+\|_2^2.$$

□

B 实验设置

合成线性规划构造。 我们首先生成一个稀疏二部交互图，然后在其边上分配取值和约束系数，从而构造合成实例。给定目标“请求”数量 I、“资源”数量 J以及目标稀疏度，我们为每个资源 j抽取一个对数正态“广度”参数，将其归一化以获得概率 p_j，并抽样得到关联请求数 $K_j \sim$，该数量服从 Poisson($p_j I\nu$) 分布，并截断至I，其中 ν 是每行期望的非零元素数量。对于每个资源 j，我们随后选取 K_j 个不同的请求，并创建边（$i,\ j$）。在每条边上，我们抽取一个特定于资源的取值尺度 v_j、一个特定于请求的响应因子 u_i以及乘性噪声 ε_{ij}，并定义一个非负取值系数

$$c_{ij} = \min\bigl(v_j u_i \varepsilon_{ij},\, c_{\max}\bigr).$$

约束系数取为这些值的缩放版本，

$$a_{ij} = s_j c_{ij},$$

其中每种资源的缩放因子 s_j 也取自对数正态分布。通过这种构造，得到一个稀疏矩阵 $\mathbf{A}$ ，其各行在非零元素的数量和幅值上均有所不同（幅值往往相差数个数量级），以及一个具有相同稀疏模式的对应价值矩阵$\mathbf{C}$ 。

源容量和右端项。 右端项源容量 b_j 的选取使得非平凡比例的约束处于活跃状态。我们不将 b_j 设为与总和 $\sum_i a_{ij}$ 成正比，而是通过贪心分配来近似每种资源在每个请求至多分配给一种资源的单纯形约束下所能获得的最大可行负载：对于每个请求 i，我们找出关联边中a_{ij} 最大的边，并将该值分配给对应资源。对各请求的这些贡献求和，即得到每种资源的“贪心负载” ℓ_j ，并据此设定

$$b_j = \rho_j\bigl(\ell_j + \varepsilon\bigr),$$

其中， ρ_j 从 $[0.5, 1.0]$ 中均匀抽取， $\varepsilon > 0$ 为一个较小的常数。这样可以确保某些资源约束在最优解中处于紧约束状态，而其他约束则保持松弛。传递给求解器的最终线性规划数据是$\mathbf{A}$的稀疏 CSC 表示、对应的价值矩阵（调整符号以符合我们的最小化约定），以及源容量向量$\mathbf{b}$。

优化算法。 我们采用 Nesterov 加速梯度上升法的一个变体来优化正则化对偶目标，并遵循 [7] (`AcceleratedGradientDescent.scala`) 中的实现，直接将其移植到 PyTorch。

该方法根据近期的梯度信息持续估计局部 Lipschitz 常数，并据此在每次迭代时自适应地选择步长。这使算法能够响应局部曲率的变化，同时保留加速结构。

为确保数值稳定性，步长会被限制在一个最大允许值以内。如果该上限设置得过于激进，方法可能会因曲率低估而发散；如果设置得过于保守，收敛速度则会显著变慢。实践中，这一上限对于平衡稳健性与收敛速度至关重要。

除非另有说明，所有实验均使用固定的超参数：最大步长为 = 1e-3，初始步长为 = 1e-5，岭参数为 $\gamma = 0.01$ 。为确保结果具有可比性，这些值在不同问题规模下均保持不变。优化在达到固定迭代次数后终止。