用于大规模拓扑优化的切片张量算子

Pedro Zuidberg Dos Martires厄勒布鲁大学计算机科学系，瑞典

Gustavo Assis da Silva厄勒布鲁大学机械工程系，瑞典

arXiv:2611.29021v1 [cs.CE] 21 Nov 2026

摘要

大规模三维拓扑优化受到两方面制约：反复求解控制有限元方程的计算成本，以及这些计算所需的内存。通过专门的底层实现可以控制内存消耗，但开发、优化和扩展此类代码需要投入大量精力，并具备特定硬件专业知识。支持自动微分的高级张量编程框架提供了一种颇具吸引力的替代方案：数值运算可以简洁地表达，而其编译器则能为 CPU 和 GPU 生成高度优化的代码。然而，直接采用张量公式会创建中间数组，迅速耗尽可用内存。我们通过用于无矩阵有限元分析和 Galerkin 多重网格的内存受限张量算子，解决这一限制。细网格刚度作用通过重叠的空间切片计算；在粗网格上，Galerkin 算子则以因子化表示形式保留，该表示由细网格刚度插值系数和预计算局部算子组成。这种表示使其作用无需在每个粗网格上显式构建随空间变化的刚度算子即可计算，同时保留精确的 Galerkin 层次结构。我们在 topax 中实现了这些公式。topax 是一个模块化 JAX 库，支持面向 CPU 和 GPU 的二维及三维基于密度的拓扑优化。在单个 NVIDIA GH200 上，topax 对一个拥有超过 8.09 亿个位移自由度的问题执行 50 次设计迭代，耗时约 24.5 分钟。在通用 GPU 基准测试中，它的速度最高可达最快竞品的 13.6 倍；对于最大规模的问题，其设备内存用量不到竞品的一半。

1 引言

机器学习的迅速发展推动了高性能张量计算软件和硬件基础设施的显著进步。现代数值计算框架，如 JAX [Bradbury 等， 2018] 以及 PyTorch [Paszke 等， 2019], 提供了经过高度优化的计算内核、即时编译和自动微分功能。这些能力也为加速科学计算带来了机遇，前提是数值算法能够通过合适的计算原语来表达。

有限元分析是能够从这一基础设施中获益的一种计算方法。传统实现依赖单元连接关系、稀疏矩阵组装和稀疏线性代数 [Cecka 等， 2011], 这些操作往往涉及不规则的内存访问和索引操作。不过，对于笛卡尔网格，可以利用离散化的规则性，通过结构化张量运算来表达局部数值算子。例如，微分算子的有限差分离散化可以使用卷积核实现 [He 和 Xu， 2019], 而无矩阵有限元公式则可以利用规则的单元连接关系，避免组装全局刚度矩阵 [Martínez-Frutos 和 Herrero-Pérez， 2015]。此类公式可以利用经过优化的 CPU 和 GPU 实现，而无需编写专用的底层代码。

基于密度的拓扑优化对有限元分析提出了尤为严苛的要求 [Sigmund 和 Maute， 2013]。在优化过程中，材料分布不断变化，因此每次设计迭代都需要求解大型线弹性问题。对于高分辨率离散化，这些重复求解构成了主要的计算开销。无矩阵方法通过在不组装全局稀疏矩阵的情况下计算刚度作用，降低了内存需求。结合多重网格预条件，这种方法为大规模拓扑优化提供了有效途径，先前的 GPU 加速实现已展示了这一点 [Dick 等人，2011年，Wu 等人， 2015]。然而，通过张量运算表达有限元算子可能会产生大型中间数组，从而限制能够高效处理的问题规模。在其他张量计算应用中也存在类似的内存挑战，人们已采用分块和索引切片等技术来减少数据移动，并将大型运算分解为较小的计算 [Ziogas 等人，2022，Huang 等人， 2021]。

在这项工 作中，我们通过卷积将细网格刚度作用表示为 d
结构化的张量 张量变换，并在重叠切片中对其进行计算。这避免了显式的单元连接 ty

在算子计算过程中，同时限制中间计算所需的内存。我们将这一方法扩展到多重网格预条件中使用的 Galerkin 粗网格算子。每个粗单元刚度矩阵都分解为预计算局部算子的加权组合，因此无需为每个粗单元显式构造单独的刚度矩阵，即可通过张量收缩直接计算其作用。这些权重由原始细网格材料分布得出，而预计算算子则在每个多重网格层级的各单元间共享。以有界批次处理这些缩并，可进一步降低中间计算的内存需求。

我们在 topax 中实现了这些方法。topax 是一个使用 JAX 开发的模块化拓扑优化库。该库将所提出的算子与自动微分和即时编译相结合，支持在 CPU 和 GPU 上运行，并允许独立选择不同的优化算法、滤波方法、投影函数和求解器组件。

本文的主要贡献如下：

1. 一种适用于笛卡尔网格的基于卷积的无矩阵有限元公式，通过规则的张量运算来表达，可利用经过优化的 CPU 和 GPU 实现。2. 一种张量切片策略，在保留底层有限元算子的同时，限制细网格刚度作用所需的中间内存。3. 一种隐式 Galerkin 公式，可直接根据细网格材料系数和预计算局部算子计算粗网格刚度作用，并结合批量评估策略来控制中间内存消耗。4. topax：一种模块化 JAX 实现，将这些算子集成到多重网格预条件拓扑优化框架中，并支持二维和三维问题。

我们通过数值实验评估所提出公式及其实现，并考察其对计算性能和内存消耗的影响。结果表明，与近期最先进的拓扑优化实现相比，这些方法显著提升了性能，同时也凸显了运行时间与内存需求之间的权衡。在规模最大的实验中，topax 在单个 GPU 上约用 24.5 分钟完成了 50 次设计迭代；该问题包含超过 8.09 亿个位移自由度，峰值内存消耗为 83.4 GiB。

2 预备知识

2.1 基于密度的拓扑优化

本文研究受体积约束的标准柔顺度最小化问题 [Bendsøe 和 Sigmund， 2004]。采用基于密度的方法表示设计域内的设计变量和物理相对密度 [Sigmund 和 Maute， 2013]。在此公式中，设计域采用标准的基于位移的有限元方法进行离散化 [Bathe， 2014], 并将一个设计变量与每个有限元相关联 $\theta_e \in [0,1]$。所有设计变量构成的集合用向量$\boldsymbol{\theta} \in \mathbb{R}^{N_e}$表示，其中 N_e 表示离散化中的单元总数。

设计变量通过基于线性卷积算子的密度滤波器进行处理 [Bourdin，2001，Bruns 和 Tortorelli， 2001] 或 Helmholtz 型滤波器 [Lazarov 和 Sigmund， 2011], 之后还可选择进行投影操作 [Wang 等人， 2011]。此过程得到物理相对密度向量 $\boldsymbol{\rho} \in \mathbb{R}^{N_e}$，其分量 ρ_e 表示单元内分片常数的相对密度，用于有限元分析、约束评估和灵敏度计算。由于物理密度是通过对设计变量进行滤波并可选地进行投影得到的，因此 $\boldsymbol{\rho}$ 可视为 $\boldsymbol{\theta}$的函数。或者，也可以使用灵敏度滤波[Sigmund， 2001]。在此公式中，物理密度直接等同于设计变量， $\boldsymbol{\rho} = \boldsymbol{\theta}$并在将目标函数和约束的灵敏度传递给优化算法之前对其进行滤波。

考虑静态载荷条件下的线弹性行为，柔度最小化问题可表述为

$$
\begin{aligned}
\min_{\boldsymbol{\theta}} \quad & c(\boldsymbol{\theta}) = \boldsymbol{f}^{\mathsf{T}} \boldsymbol{u}(\boldsymbol{\theta}) \\
\text{s.t.} \quad & \frac{1}{N_e} \sum_{e=0}^{N_e-1} \rho_e(\boldsymbol{\theta}) \leq V_f \\
& K(\boldsymbol{\theta}) \boldsymbol{u}(\boldsymbol{\theta}) = \boldsymbol{f} \\
& 0 \leq \theta_e \leq 1, \qquad e \in \{0, \ldots, N_e - 1\}
\end{aligned}
\quad , \tag{1}
$$

其中 c 为结构柔度， $\boldsymbol{u}$ 为全局位移向量， $\boldsymbol{f}$ 为外载荷向量， K 为全局刚度矩阵，且 V_f 为给定的体积分数。这些量对设计变量的依赖关系在公式 1中明确给出。在本文余下部分，若不会造成歧义，则省略此类依赖关系。

算法 1 多重网格预条件共轭梯度

1: 函数 MGPCG(K**,** $\boldsymbol{f}$**,** $\boldsymbol{u}_0$**) 2:** $\boldsymbol{r}_0 \leftarrow \boldsymbol{f} - K\boldsymbol{u}_0$ **3:** $\boldsymbol{z}_0 \leftarrow \mathcal{G}$**(**$\boldsymbol{r}_0$**, 0) 4:** $\boldsymbol{p}_0 \leftarrow \boldsymbol{z}_0$ **5:** $j \leftarrow 0$ **6: 当未满足停止准则时执行 7:** $\boldsymbol{v}_j \leftarrow K\boldsymbol{p}_j$ **8:** $\alpha_j \leftarrow \boldsymbol{r}_j^{\mathsf{T}}\boldsymbol{z}_j / \boldsymbol{p}_j^{\mathsf{T}}\boldsymbol{v}_j$ **9:** $\boldsymbol{u}_{j+1} \leftarrow \boldsymbol{u}_j + \alpha_j \boldsymbol{p}_j$ **10:** $\boldsymbol{r}_{j+1} \leftarrow \boldsymbol{r}_j - \alpha_j \boldsymbol{v}_j$ **11:** $\boldsymbol{z}_{j+1} \leftarrow \mathcal{G}$**(**$\boldsymbol{r}_{j+1}$**, 0) 12:** $\beta_j \leftarrow \boldsymbol{r}_{j+1}^{\mathsf{T}}\boldsymbol{z}_{j+1} / \boldsymbol{r}_j^{\mathsf{T}}\boldsymbol{z}_j$ **13:** $\boldsymbol{p}_{j+1} \leftarrow \boldsymbol{z}_{j+1} + \beta_j \boldsymbol{p}_j$ **14:** $j \leftarrow j + 1$ **15: 结束循环 16: 返回** $\boldsymbol{u}_j$ **17: 结束函数**

单元刚度矩阵采用带惩罚的固体各向同性材料（SIMP）方法 [Bendsøe 和 Sigmund, 1999, Sigmund, 2007],进行插值:

$$K_e = \underbrace{\left(\rho_{\min} + \rho_e^p \left(1 - \rho_{\min}\right)\right)}_{=:w_e} J_e, \tag{2}$$

其中， J_e 是固体材料的单元 e 刚度矩阵， p 是惩罚指数，而 $\rho_{\min}$ 是为防止奇异刚度矩阵而引入的小正参数。全局刚度矩阵通过标准的有限元组装 [Bathe, 2014]进行定义。

$$K = \sum_{e=0}^{N_e-1} L_e^{\mathsf{T}} K_e L_e, \tag{3}$$

其中， L_e 是连接全局自由度排列与单元自由度排列的布尔定位矩阵。特别地，单元位移向量 $\boldsymbol{u}_e$ 由全局位移向量按照下式获得： $\boldsymbol{u}_e = L_e \boldsymbol{u}$。公式3 以代数方式定义了全局刚度矩阵，并不意味着要显式组装该矩阵；后续章节介绍的无矩阵公式将避免这一过程。

2.2 多重网格预条件共轭梯度求解器

对于大规模拓扑优化问题，使用直接法求解方程1中的有限元平衡问题，由于矩阵分解所需的内存和计算成本较高，计算开销十分昂贵。迭代法（例如共轭梯度法（CG））为求解由此产生的大型稀疏对称正定线性系统提供了另一种选择。然而，CG 的收敛性取决于刚度矩阵的条件数；在基于密度的拓扑优化中，由于设计域内材料刚度变化幅度较大，条件数可能尤其不利。

因此，采用预条件来降低线性系统的有效条件数，从而改善 CG 的收敛特性。特别是，几何多重网格法非常适用于大规模有限元问题，因为它将能够降低高频误差分量的平滑操作与处理低频误差的粗网格校正相结合 [Trottenberg 等人, 2001]。在本文中，我们在 CG 迭代中使用几何多重网格 V 循环作为预条件器，从而得到多重网格预条件共轭梯度（MGPCG）方法 [Amir 等人, 2014],其总结见算法1。

在算法 1中， $\boldsymbol{r}_j$、 $\boldsymbol{z}_j$ 和 $\boldsymbol{p}_j$ 分别表示第 j次迭代时的残差、预条件残差和搜索方向。预条件操作 $\mathcal{G}(\boldsymbol{r}_j, 0)$ 采用算法 2中定义的递归几何多重网格 V-循环进行计算。通常使用相对残差范数评估收敛情况。当 $\|\boldsymbol{r}_j\|_2 / \|\boldsymbol{f}\|_2 \leq \varepsilon$时，退出循环，其中 ε 表示预先设定的容差。实际应用中，还会设定循环迭代次数的上限，以避免在未达到期望容差时产生过多的计算工作量。

在算法 2 中， ℓ 表示当前的多重网格层级，其中 $\ell = 0$ 对应最细层， $\ell = L - 1$对应最粗层；其中 L 是 V 循环中的层级总数。向量 $\boldsymbol{r}^{(\ell)}$ 和 $\boldsymbol{z}^{(\ell)}$ 分别表示层级 ℓ上的右端项和修正量，而 $K^{(\ell)}$ 是相应的刚度矩阵。算子$\mathcal{S}^{(\ell)}$ 表示平滑过程，在粗网格校正之前和之后执行。算子 $P^{(\ell)}$将向量从层级 $\ell+1$ 延拓到层级 ℓ，而其转置 $\left(P^{(\ell)}\right)^{\mathsf{T}}$ 用于将向量从层级 ℓ 限制到

算法 2 递归几何多重网格 V-循环 le

函数 $\mathcal{G}(\boldsymbol{r}^{(\ell)},\ \ell)$
  如果 $\ell = L-1$ **则**
    $\boldsymbol{z}^{(\ell)} \leftarrow \left(K^{(\ell)}\right)^{-1}\boldsymbol{r}^{(\ell)}$ ▷ 直接求解
  else
    $\boldsymbol{z}^{(\ell)} \leftarrow \mathbf{0}$
    $\boldsymbol{z}^{(\ell)} \leftarrow \mathcal{S}^{(\ell)}(\boldsymbol{r}^{(\ell)},\ \boldsymbol{z}^{(\ell)})$ ▷ 预平滑
    $\tilde{\boldsymbol{r}}^{(\ell)} \leftarrow \boldsymbol{r}^{(\ell)} - K^{(\ell)}\boldsymbol{z}^{(\ell)}$
    $\boldsymbol{r}^{(\ell+1)} \leftarrow (P^{(\ell)})^{\mathsf{T}}\tilde{\boldsymbol{r}}^{(\ell)}$ ▷ 限制
    $\boldsymbol{z}^{(\ell+1)} \leftarrow \mathcal{G}(\boldsymbol{r}^{(\ell+1)},\ \ell+1)$ ▷ 递归调用
    $\boldsymbol{z}^{(\ell)} \leftarrow \boldsymbol{z}^{(\ell)} + P^{(\ell)}\boldsymbol{z}^{(\ell+1)}$ ▷ 延拓
    $\boldsymbol{z}^{(\ell)} \leftarrow \mathcal{S}^{(\ell)}(\boldsymbol{r}^{(\ell)},\ \boldsymbol{z}^{(\ell)})$ ▷ 后平滑
  结束 if
  返回 $\boldsymbol{z}^{(\ell)}$
结束函数

层级 $\ell+1$。常见的平滑方案包括高斯–赛德尔和阻尼雅可比迭代 [Trottenberg 等人，2001]。后者尤其适合 GPU 实现，因为其更新可以独立且高效地并行计算。在层级 $L-1$ 上，线性系统通过稀疏直接求解器求解，通常采用稀疏 Cholesky 分解。各层级相关的刚度矩阵通过伽辽金粗网格构造递归定义 [Trottenberg 等人，2001]，

$$K^{(\ell+1)} = \left(P^{(\ell)}\right)^{\mathsf{T}} K^{(\ell)} P^{(\ell)}, \qquad \ell \in \{0,\dots,L-2\}, \tag{4}$$

从细网格刚度矩阵 $K^{(0)}$ 开始。采用结构化的二进多重网格层次结构，在该结构中，每个层级的网格沿各空间方向均按二分之一的比例进行粗化。因此，在二维中，层级 $\ell+1$ 的每个单元包含层级 ℓ 的四个单元；在三维中则包含八个单元。

3 切片无矩阵线性算子

在基于密度的拓扑优化中，大规模有限元系统的重复求解构成了主要的计算开销。尽管 MGPCG 方法避免了对细网格刚度矩阵进行分解，但每次迭代都需要重复计算矩阵–向量乘积 $K\boldsymbol{u}$。对于大型问题，显式组装并存储全局刚度矩阵 K 可能会带来巨大的内存需求。无矩阵有限元方法通过直接根据单元级贡献计算刚度作用来避免这种存储开销 [KronbichlerandKormann， 2012]。

高效的 GPU 执行还受益于能够展现并行性并避免不规则内存访问的规则计算结构。我们考虑采用双线性四边形单元离散的二维均匀笛卡尔网格，以及采用三线性六面体单元离散的三维均匀笛卡尔网格。规则的连接关系使有限元矩阵–向量乘积能够通过类似卷积的张量运算来表达，从而在算子应用过程中无需显式的单元到节点和节点到单元连接数组。为便于说明，我们在线性运算 $K\boldsymbol{u}$的表述中，首先详细介绍二维情形。随后将其扩展到三维，过程十分直接。

考虑由 $N_x \times N_y$ 个双线性四边形单元组成的结构化笛卡尔网格。方程

$\rho_{\min}+\rho_{ij}^{p}\ 1-\rho_{\min}$ $i\ j$ 2 e $W \in \mathbb{R}^{N_x\times N_y}$ $W_{ij} =$ 中出现的逐元素刚度插值系数收集在张量 中，其中 ( )。下标 (,) 表示结构化网格中的单元索引（参见方程中的下标），其中 $i \in \{0,\dots,N_x-1\}$ 和 $j \in \{0,\dots,N_y-1\}$。 2

对于基于张量的公式，节点位移场可方便地表示为 $U \in \mathbb{R}^{2\times(N_x+1)\times(N_y+1)}$。第一个维度区分两个位移分量，其余维度对应结构化网格的节点索引。因此，U 由以下位移场构成U_x，$U_y \in \mathbb{R}^{(N_x+1)\times(N_y+1)}$，其中 U_x 和 U_y 分别汇集了 x 方向和 y 方向上的节点位移分量。全局位移向量 $\boldsymbol{u}$ 与张量 U 表示相同的节点位移场，只是位移分量的排列遵循不同的索引约定。图 1 展示了这些表示方式，并突出了逐单元刚度系数与双分量节点位移场之间的区别。

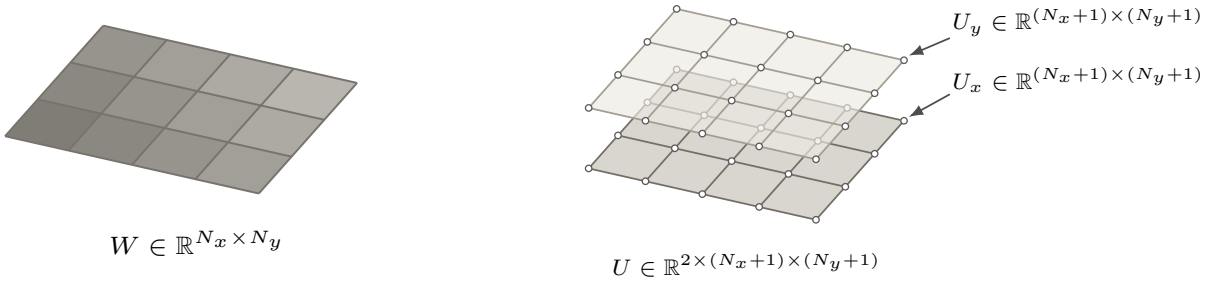

图 1：结构化四边形网格上的逐单元和节点张量表示。左：单元刚度插值系数 $W \in \mathbb{R}^{N_x \times N_y}$。右：双分量节点位移场 $U \in \mathbb{R}^{2\times(N_x+1)\times(N_y+1)}$。

3.1 基于张量的无矩阵算子

考虑结构化网格中任意一个四边形单元 e，其索引为 $(i,\ j)$。为方便起见，将其刚度插值系数 W_{ij} 记为 w_e。每个单元有四个节点，每个节点有两个位移自由度。我们将这八个自由度收集到单元位移向量 $\boldsymbol{u}_e \in \mathbb{R}^8$中。在整个公式推导中，局部自由度采用分量优先排列：四个 x- 位移分量排在前面，随后是四个 y-位移分量；节点先沿x方向、再沿 y 方向排列。

由于 网格是均匀的，且所有单元具有相同的几何形状和固体材料属性， e 矩阵 $J_e \in \mathbb{R}^{8\times8}$ 在方程 2 中对所有 e均相同。因此，我们省略下标 e 并写作

$$K_e = w_e J. \tag{5}$$

单元刚度矩阵根据下式将单元位移向量映射为单元力向量

$$\boldsymbol{f}_e = K_e \boldsymbol{u}_e = w_e J \boldsymbol{u}_e, \tag{6}$$

其中，$\boldsymbol{f}_e \in \mathbb{R}^8$ 收集与单元自由度相关的八个力贡献。虽然 J 对所有单元都相同，但刚度插值系数 w_e 和单元位移向量 $\boldsymbol{u}_e$ 会随网格变化。

现在考虑未加权单元力贡献，

$$\mathbf{h}_e = J \boldsymbol{u}_e. \tag{7}$$

收集这 些贡献在所有单元上的值，得到张量 $H \in \mathbb{R}^{8\times N_x \times N_y}$，其中 H_{aij} 表示 第a个未加权 对于索引为 $(i,\ j)$ 的单元的力贡献。

对于索引为 $(i,\ j)$ 的单元 e，规则网格连接关系使其单元位移向量的每个分量都可以直接从节点位移张量 U 中提取，如下所示

$$(\boldsymbol{u}_e)_{(4b+2\delta_x+\delta_y)} = U_{b,i+\delta_x,j+\delta_y}, \tag{8}$$

其中，$b \in \{0,1\}$ 表示位移分量，$\delta_x \in \{0,1\}$ 和 $\delta_y \in \{0,1\}$ 表示单元 e 的四个节点之一。$\boldsymbol{u}_e$ 的八个分量对应于这些位移，并遵循既定的局部自由度排序。为将单元刚度作用表示为张量形式，将实体单元刚度矩阵 $J \in \mathbb{R}^{8\times8}$ 重塑为 $C \in \mathbb{R}^{8\times2\times2\times2}$。保留 J 的行索引 $a \in \{0,\dots,7\}$ 作为 C 的第一个索引，而其列索引则按照与单元位移向量各分量相同的索引方式进行重塑：$C_{ab\delta_x\delta_y} = J_{a,\ 4b+2\delta_x+\delta_y}$。未加权单元力贡献可表示为

$$H_{aij} = \sum_{b=0}^{1}\sum_{\delta_x=0}^{1}\sum_{\delta_y=0}^{1} C_{ab\delta_x\delta_y} U_{b,i+\delta_x,j+\delta_y}. \tag{9}$$

图 2 展示了无权单元刚度作用的张量表示。左侧，实体单元刚度矩阵 J被重塑为八个系数张量 C_0、.....、$C_7 \in \mathbb{R}^{2\times2\times2}$。右侧，这些系数作用于节点位移场，为通过索引$(i,\ j)$识别的每个单元生成 H 的八个分量。

张量 H 包含未加权单元力贡献。为了考虑依赖密度的材料刚度，将每项贡献乘以相应单元的刚度插值系数，

$$\widehat{H}_{aij} = W_{ij} H_{aij}, \tag{10}$$

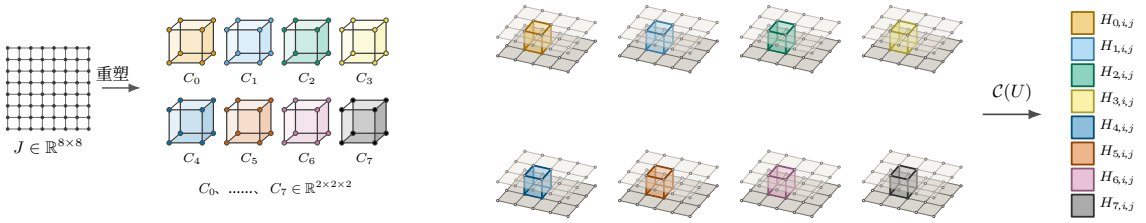

图2：无权单元刚度作用的张量表示。左：将实体单元刚度矩阵$J \in \mathbb{R}^{8\times8}$重塑为八个系数张量 C_0，……，$C_7 \in \mathbb{R}^{2\times2\times2}$。右：将这些系数应用于节点位移场，为每个单元生成八个局部力贡献。

从而得到加权单元力贡献 $\widehat{H} \in \mathbb{R}^{8\times N_x\times N_y}$。

最后一步是将这些贡献组装到全局节点力场中。由于相邻单元共享节点，每个内部节点都会接收来自周围四个单元的贡献。为了表示这一组装过程， H 的第一个^维按照 $\boldsymbol{u}_e$ 和 $\boldsymbol{f}_e$共有的局部自由度排序进行重塑。我们将所得张量记为 $\widehat{\mathcal{H}} \in \mathbb{R}^{2\times2\times2\times N_x\times N_y}$，其中条目 $\widehat{\mathcal{H}}_{b\delta_x\delta_y ij}$ 包含与分量 $b \in \{0,1\}$ 及局部单元节点（δ_x， δ_y） $\in \{0,1\}^2$ 相关联的加权单元力贡献，该单元由（i， j）索引。

单元索引为（i， j）的单元，其单元局部节点（δ_x， δ_y）对应于索引为（m， n）的全局节点，其中 $m = i + \delta_x$ 和 $n = j + \delta_y$。因此，全局节点（m， n）处与单元局部节点（δ_x， δ_y）相关的贡献，来自索引为（$m - \delta_x$， $n - \delta_y$）的单元。组装后的节点力场$F \in \mathbb{R}^{2\times(N_x+1)\times(N_y+1)}$ 定义为

$$F_{bmn} = \sum_{\delta_x=0}^{1}\sum_{\delta_y=0}^{1} \widetilde{\mathcal{H}}_{b\delta_x\delta_y,\, m-\delta_x,\, n-\delta_y}, \tag{11}$$

~ ^其中， $\mathcal{H}$ 表示将 $\mathcal{H}$ 在单元索引域之外进行零延拓。此延拓涵盖了边界节点，因为这些节点接收的贡献少于来自四个单元的贡献。

图 3 展示了两个位移分量的组装过程。每个节点都会接收四个局部力贡献，这些贡献以节点周围的方块表示。在内部节点处，所有四个贡献均来自网格内的单元。在边界节点处，缺失的贡献以零替代，并用白色方块表示。箭头表示这些贡献如何累积到相应的节点位置。



图3：将加权单元力贡献组装到节点力场中。左：x分量的组装及其对应的颜色图例。右： y分量的组装及其对应的颜色图例。彩色方块表示网格中各单元的贡献，白色方块表示通过填充引入的零值贡献。箭头表示在相应节点处累加贡献。

我们将公式 9 中定义的张量运算记为 $\mathcal{T}$，将公式 11 中定义的组装运算记为 $\mathcal{A}$。将这些运算与逐元素刚度插值相结合，即可得到完整的无矩阵刚度作用：

$$K\boldsymbol{u} = \mathrm{vec}\left(\mathcal{A}\left(W \odot \mathcal{T}(U)\right)\right), \tag{12}$$

其中 $\odot$ 表示 对八个局部力贡献进行广播的逐元素乘法，并且
vec 将组装后的力张量映射

3.2 基于卷积的无矩阵算子

上一小节推导出的张量运算 $\mathcal{T}$具有一种特别高效的计算实现。每个分量 H_{aij} 取决于由 (i， j) 标识的单元中四个节点的位移，并且在整个网格中始终采用相同的系数 $C_{ab\delta_x\delta_y}$。该运算具有二维卷积的结构，包含两个输入通道、八个输出通道，空间内核尺寸为 2×2。

清单 1 卷积 -无矩阵有限元算子 维二维实现，使用 JAX 基本算子。

```
导入 jax.numpy 作为jnp
从 jax 导入 lax
从 einops 导入rearrange, reduce

定义矩阵_无_2维(u    , J, w):
    # 根据实体单元刚度矩阵构造核
    核 =J.重塑(8,          2, 2, 2)

    # 计算无权单元刚度作用
    h = lax.卷积_通用_膨胀(
        u[无    , ...],
        核    ,
        窗口_步幅=(1      , 1),
        填充="VALID",
        维度_编号=("NCHW"          , "OIHW" , "NCHW"),
    )[0]

    # 应用密度相关刚度系数
    h_加权 = w * h

    # 将贡献分配到单元节点
    h_加权 = rearrange(h_加权,          "(c i j) x y ->c (x i) (y j)", c=2,   i=2, j=2)

    # 在共享节点处组装贡献
    h_加权 = jnp.pad(h_加权,          ((0, 0), (1, 1), (1, 1)))
    f = reduce(h_加权,      "c (x i) (y j) -> c x y", "sum", i=2, j=2)

    返回 f
```

两个输入通道分别对应节点位移场 U_x 和 U_y，二者共同构成位移张量 U。八个输出通道对应未加权单元力向量的八个分量，每个通道都包含网格中每个单元的一个分量。系数张量 $C \in \mathbb{R}^{8\times2\times2\times2}$提供卷积核权重。对 $U \in \mathbb{R}^{2\times(N_x+1)\times(N_y+1)}$ 应用单位步幅且不进行填充的卷积，可得到 $H \in \mathbb{R}^{8\times N_x\times N_y}$。

这种解释使我们能够用一次卷积运算取代重复的逐元素矩阵–向量乘积。结合逐元素加权和结构化组装，所得公式无需构造全局稀疏矩阵或存储单元连接数组，即可重现组装后的有限元刚度矩阵的作用。通过使用高性能数值计算库（如 JAX）提供的标准计算原语来表达刚度作用，我们可以利用针对不同架构优化的硬件实现，包括 CPU 上的 SIMD 指令和专用 GPU 内核，而无需开发自定义底层代码。

清单 1 展示了使用 JAX [Bradbury et al., 2018] 和 Einops [Rogozhnikov, 2022]实现的简洁方案。节点位移张量 U 和逐单元刚度系数 W 分别对应数组 u 和 w。实体单元刚度矩阵 J 以 J 的形式提供，并在第 7 行重塑为卷积核$C \in \mathbb{R}^{8\times2\times2\times2}$。JAX 原语 lax.conv_general_dilated 对公式 9 进行计算，生成未加权单元力贡献数组 h。随后乘以 w，引入由公式 10 定义的与密度相关的刚度插值系数。最后，rearrange、jnp.pad 和 reduce 组合实现公式 11 中的组装运算。rearrange 操作将局部力贡献放置到对应的单元节点上。填充扩展所得张量，以计入计算域之外的贡献；而归约则对与每个全局节点相关联的贡献求和。所得数组 f 表示组装后的节点力场 $F \in \mathbb{R}^{2\times(N_x+1)\times(N_y+1)}$。

3.3 处理边界条件

在有限元分析中，狄利克雷边界条件规定了选定自由度处的位移。在拓扑优化中，这些条件通常表示阻止一个或多个方向位移的支承。其余未受约束的自由度则通过求解平衡方程确定。

对于齐次狄利克雷边界条件，规定的位移为零。因此，我们可以将平衡方程限制在不受约束的自由度上。令 N_{dof} 表示自由度总数，N_{free} 表示不受约束的自由度数量。引入限制矩阵 $R \in$

清单 2 使用不受约束自由度的二进制掩码，应用带齐次狄利克雷边界条件的无矩阵刚度算子。

```
定义 矩阵_无矩阵_2d_狄利克雷(u,J,w,free):u  =
free* uf  = 矩阵_free_2d(u, J,w)返回 free *
f
```

$\mathbb{R}^{N_{\text{free}} \times N_{\text{dof}}}$，它提取向量中的非受约束条目，约化刚度算子的形式为

$$K_f = RKR^{\mathsf{T}}. \tag{13}$$

限制矩阵 R 提供了一种约化空间表示，其中移除了受约束自由度，仅保留 N_{free} 非受约束自由度。尽管约化空间公式是标准做法，但显式移除受约束自由度会破坏基于卷积的无矩阵运算所需的规则全网格张量布局。因此，我们在张量表示中保留所有自由度，并使用二进制掩码 $M \in \{0,1\}^{2\times(N_x+1)\times(N_y+1)}$施加齐次约束。在非受约束自由度处， M 的条目等于一；在受约束自由度处则等于零。在刚度运算前后应用掩码可得

$$F = M \odot \mathcal{A}\left(W \odot \mathcal{T}(M \odot U)\right), \tag{14}$$

其中，内层掩码在受约束自由度处施加零位移，而外层掩码会在组装后移除相应的力条目。

与约化空间形式不同，掩码形式在位移和力的表示中保留所有 N_{dof} 自由度。不过，受约束条目始终为零，不参与迭代求解。限制在无约束子空间内时，掩码刚度作用等价于约化算子 RKR^{T}。

在全空间中，掩码刚度算子在受约束自由度处的行和列均为零，因此是奇异的。因此，共轭梯度法在无约束子空间内初始化并应用，同时保持解、右端项、残差和搜索方向中的受约束条目为零。清单2 展示了相应的 JAX 实现。二进制掩码由数组`free`表示，并且只需进行两次逐元素乘法即可施加齐次约束。

3.4 对无矩阵算子进行切片

尽管基于卷积的形式无需组装和存储全局刚度矩阵，但在整个计算域上计算算子会产生大小随单元数量增长的中间张量。具体而言，单元力贡献张量在二维中每个单元包含八个条目，在三维中每个单元包含二十四个条目。为限制这些中间量所需的内存，我们将节点力场划分为多个切片，并分别计算。

考虑一个包含 $N_x \times N_y$ 个单元的二维网格。我们沿 x 方向将节点位置划分为多个切片，每个切片包含 B_{slice} 个连续的节点平面，并沿 y 方向覆盖整个域。假设 $N_x + 1$ 能被 B_{slice} 整除，我们将切片编号为 $\sigma \in \{0, \ldots, (N_x+1)\ /B_{\text{slice}} - 1\}$。切片 σ 包含编号为

$$\nu \in \{\sigma B_{\text{slice}}, \ldots, (\sigma+1)B_{\text{slice}} - 1\}. \tag{15}$$

有限元刚度作用的局部性使得每个切片的计算只需与相邻切片少量重叠。给定位置上的节点力仅取决于紧邻单元的贡献。因此，计算切片 σ 内的节点力需要用到编号为以下平面上的单元贡献：

$$\epsilon \in \{\sigma B_{\text{slice}} - 1, \ldots, (\sigma+1)B_{\text{slice}} - 1\}. \tag{16}$$

反过来，这些单元又取决于编号为以下平面上的节点位移：

$$\nu \in \{\sigma B_{\text{slice}} - 1, \ldots, (\sigma+1)B_{\text{slice}}\}. \tag{17}$$

因此，计算 B_{slice} 个输出平面需要 $B_{\text{slice}} + 2$ 个节点位移平面和 $B_{\text{slice}} + 1$ 个单元刚度系数平面。这些额外平面提供了所需的重叠区域，以便在切片边界处计算刚度作用。在计算域边界处，所需的重叠由零填充提供，这与前文介绍的组装运算一致。仅保留分配给 B_{slice} 的输出平面，因此所得切片彼此不重叠，且可拼接起来以还原完整的节点力场。

代码清单 3 二维切片无矩阵算子的简化实现。该区域沿 x 方向划分为 B_{slice} 个节点平面的切片。注释中标明了张量形状。

```
导入 jax.numpy 作为jnp

定义 矩阵_无矩阵_2算子_切片(u , J, w, B_切片):
    # u: (2,    Nx+1, Ny+1)
    # w: (Nx,    Ny)

    # 沿 x 方向填充输入张量
    u_填充 =jnp.填充(u,   ((0, 0), (1, 1), (0, 0)))
    w_填充 =jnp.pad(w,    ((1, 1), (0, 0)))

    切片 = []

    对于 sigma 在 范围(u。形状[1] //B_切片):
        i  =sigma * B_切片

        # 提取重叠输入区域
        u_局部 = u_填充[:, i:i+B_切片+2, :]                    # (2,    B_切片+2,   Ny+1)
        w_局部 = w_填充[i:i+B_切片+1, :]                       # (B_切片+1,    Ny)

        # 计算无矩阵算子
        f_局部 = 矩阵_自由_2d(u_局部, J, w_局部)               # (2,    B_切片+2,    Ny+1)

        # 保留指定的输出平面
        切片。append(f_local[:,  1: -1,  :])                  # (2,    B_切片,    Ny+1)

    # 恢复完整的节点力场
    返回jnp。concatenate(切片, axis=1)                        # (2,    Nx+1,  Ny+1)
```

列表3 展示了使用列表 1中介绍的无矩阵算子进行切片的过程。首先，沿切片方向对节点位移和逐单元刚度张量进行填充。对于每个切片，我们提取重叠的输入区域，计算无矩阵算子，并仅保留分配给该切片的节点力平面。拼接这些输出即可还原完整的节点力场。

虽然列表 3 展示了切片过程，但其实现尚未针对 GPU 执行进行优化。具体而言，将输出切片收集到 Python 列表中，需要一直保留中间结果，直到完成拼接。更高效的实现会使用 JAX 控制流原语替代 Python 循环，例如 `lax。scan` 或 `lax。map`。这些原语在编译后的计算中表达迭代过程，从而无需将循环展开为一长串操作即可处理切片。结合动态切片后，它们能够使无矩阵算子处理大型域，同时限制中间张量的大小。

为估算内存节省量，考虑一个包含 $N_x \times N_y$ 个单元且切片大小为 B_{slice} 的网格。在整个域上计算无矩阵算子会产生包含 $\mathcal{O}(N_x N_y)$ 个条目的中间张量。相比之下，计算一个切片只需要包含 $\mathcal{O}(B_{\text{slice}} N_y)$ 个条目的中间张量。因此，假设按顺序处理切片，这些中间量所需的峰值内存大约减少了 N_x / B_{slice} 倍。当 $B_{\text{slice}} \ll N_x$ 时，这种减少对于大型网格尤其显著。
尽管 GPU 编程通常倾向于并行执行，但依次处理切片并不一定会显著增加运行时间。对于足够大的问题，每个切片都包含足够多的单元，可在卷积和张量运算中充分发挥并行性。此外，与所访问的数据量相比，无矩阵刚度作用所需的计算量相对较少，因此其执行主要受内存带宽而非算术吞吐量限制。因此，只要切片足够大，依次处理切片仍可保持较高的 GPU 利用率。由此可以减少中间内存需求，同时对整体算子求值时间影响甚微。

3.5 扩展到三维

无矩阵形式可自然扩展到由三线性六面体单元构成的结构化网格。每个单元有八个节点，每个节点具有三个位移自由度，因此总计有二十四个局部自由度。因此，实体单元刚度矩阵满足 $J \in \mathbb{R}^{24\times 24}$。对于包含$N_x \times N_y \times N_z$ 个单元的网格，我们用张量 $U \in \mathbb{R}^{3\times(N_x+1)\times(N_y+1)\times(N_z+1)}$表示节点位移场，而逐单元刚度系数组成 $W \in \mathbb{R}^{N_x\times N_y\times N_z}$。

按照　　二维情形中引入的分量优先排列，三个位移分量　　ts
进行排列　先按分量排序，再按单元局部节点排序。在每个分量内，八个节点 a　　re

先沿 x 方向排序，再沿 y 方向排序，最后沿 z 方向排序。实体单元刚度矩阵重塑为 $C \in \mathbb{R}^{24\times3\times2\times2\times2}$。未加权单元力贡献于是满足

$$H_{aijk} = \sum_{b=0}^{2}\sum_{\delta_x=0}^{1}\sum_{\delta_y=0}^{1}\sum_{\delta_z=0}^{1} C_{ab\delta_x\delta_y\delta_z} U_{b,i+\delta_x,j+\delta_y,k+\delta_z}, \tag{18}$$

其中， $b \in \{0,1,2\}$ 表示位移分量， $a \in \{0,\dots,23\}$ 遵循 $\boldsymbol{u}_e$ 和 $\boldsymbol{f}_e$采用的通用局部自由度排序。

与二维情况相同，此操作对应于一次卷积，此时有三个输入通道、二十四个输出通道，空间内核尺寸为 $2\times2\times2$。如4所示，只需对 JAX 实现稍作修改即可扩展到三维。在步长为 1 且不进行填充的情况下，卷积会为每个单元生成一组局部力贡献。

代码清单 4 使用 JAX 基本算子的基于三维卷积的单元刚度作用。

```
# J: (24, 24)核 = J。重塑(24,3, 2,
2, 2)

    # u: (3, Nx+1, Ny+1, Nz+1) h = lax.conv_general_
dilated( u[None, ...] ,核, window_strides=(1, 1, 1),
padding="VALID", dimension_numbers=("NCDHW", "
OIDHW", "NCDHW"), )[0]# h: (24, Nx, Ny,Nz)
```

所得张量 $H \in \mathbb{R}^{24\times N_x\times N_y\times N_z}$ 包含未加权单元力贡献。与二维情况一样，这些贡献会先乘以逐单元刚度系数，然后组装到全局节点力场中。组装遵循相同的索引原则，每个内部节点都会接收来自周围八个单元的贡献。

切片策略可以直接扩展到三维。我们沿一个空间方向将域划分为切片块，同时保留另外两个方向上的完整范围。例如，当沿 x方向进行切片时，负责 B_{slice} 个节点平面的切片块需要 $B_{\text{slice}}+2$ 个位移平面和 $B_{\text{slice}}+1$ 个单元权重平面。局部算子在这些重叠的输入上进行计算，并且只保留分配给它的 B_{slice} 个输出平面。将所得切片块拼接起来，即可恢复完整的节点力场。

在三维情况下，切片尤其有益，因为中间单元力张量每个单元包含二十四个条目。对于包含 $N_x\times N_y\times N_z$ 个单元的网格，未切片计算需要包含 $\mathcal{O}(N_xN_yN_z)$个条目的中间张量。沿 x方向进行切片后，相应的每个切片块需求可降至 $\mathcal{O}(B_{\text{slice}}N_yN_z)$，同时保持相同的无矩阵刚度作用。

4 切片 Galerkin 算子

4.1 逐单元 Galerkin分解

考虑多重网格层次结构中层级 ℓ 的单元 e，并令 $\boldsymbol{u}_e$ 表示与单元 e 的四个节点相关联的位移向量。在二分粗化下，层级 ℓ 的每个单元沿每个空间方向包含 2^ℓ 个细网格单元，因此在二维中总共有 4^ℓ 个细网格单元。我们用 $s \in \{0,\dots,4^\ell-1\}$ 为这些细单元编索引。对于每个细单元 s，令 $Q_s^{(\ell)}$ 表示用于计算粗单元位移场在细单元 s 节点处取值的插值矩阵。由此得到的细单元位移向量为

$$\boldsymbol{u}_s = Q_s^{(\ell)}\boldsymbol{u}_e. \tag{19}$$

位于 e 内的细单元所对应的弹性势能为 [Bathe, 2014]

$$\mathcal{E}_e = \frac{1}{2}\sum_{s=0}^{4^\ell-1} \boldsymbol{u}_s^{\mathsf{T}} K_s \boldsymbol{u}_s. \tag{20}$$

将方程 19代入方程20可得

$$\mathcal{E}_e = \frac{1}{2}\sum_{s=0}^{4^\ell-1} (Q_s^{(\ell)}\boldsymbol{u}_e)^{\mathsf{T}} K_s (Q_s^{(\ell)}\boldsymbol{u}_e) = \frac{1}{2}\boldsymbol{u}_e^{\mathsf{T}}\left(\sum_{s=0}^{4^\ell-1}(Q_s^{(\ell)})^{\mathsf{T}} K_s Q_s^{(\ell)}\right)\boldsymbol{u}_e. \tag{21}$$

与 Jones 和 Vassilevski [2001], 的 ℓ层级刚度矩阵 $K_e^{(\ell)}$单元聚合 Galerkin 构造一致，该矩阵的定义使其二次型能够重现插值后的细网格单元位移场的弹性势能： $\mathcal{E}_e = \frac{1}{2}\boldsymbol{u}_e^{\mathsf{T}} K_e^{(\ell)} \boldsymbol{u}_e$。将此表达式与方程 21 对每个 $\boldsymbol{u}_e$ 相等，可得

$$K_e^{(\ell)} = \sum_{s=0}^{4^\ell - 1} (Q_s^{(\ell)})^{\mathsf{T}} K_s Q_s^{(\ell)} = \sum_{s=0}^{4^\ell - 1} w_{e,s} (Q_s^{(\ell)})^{\mathsf{T}} J Q_s^{(\ell)}, \tag{22}$$

其中，第二个等式由 $K_s = w_{e、s} J$ 以及 $w_{e、s}$ 得出；s 是包含于单元 e 中的细网格单元 s 的刚度插值系数。

每个细网格单元贡献中与几何和插值相关的部分均独立于材料分布。因此，我们引入固定的层级相关矩阵 $G_s^{(\ell)} := \left(Q_s^{(\ell)}\right)^{\mathsf{T}} J Q_s^{(\ell)}$. 随后， ℓ层级的单元刚度矩阵（参见公式22）可写为

$$K_e^{(\ell)} = \sum_{s=0}^{4^\ell - 1} w_{e,s} G_s^{(\ell)}. \tag{23}$$

在最细层级， $\ell = 0$，单元 e 仅包含其自身。在这种情况下， $Q_0^{(0)}$ 是单位矩阵， $G_0^{(0)} = J$，并且我们将 $w_{e、0}$ 与 w_e相对应。公式 23 因此可简化为 $K_e^{(0)} = w_e J = K_e$，这恢复了公式 5中的细网格 SIMP 插值。因此，细网格公式就是 Galerkin 表示的第 0 层级实例。在更粗层，每个单元刚度矩阵都由 4^ℓ 固定的局部算子与相应的细网格刚度插值系数组合而成。

在多重网格文献中，Galerkin 算子的逐单元分解已得到充分研究，尤其是在基于单元聚合的方法中 [Jones 和 Vassilevski， 2001]。此前的工作还展示了在 GPU 上应用无矩阵多重网格方法，包括将其用于有限元分析和高分辨率拓扑优化 [Dick 等人、2011年、Wu 等人， 2015]。在此，我们利用这种分解，直接根据原始细网格刚度插值系数计算 Galerkin 刚度作用。我们不再显式计算并存储每个粗单元的 $K_e^{(\ell)}$，而是保留其因子化表示，并利用相应权重$w_{e,\ s}$作用于预计算矩阵 $G_s^{(\ell)}$。这样既无需显式构造粗单元刚度矩阵，又能保留不同网格层级之间的 Galerkin 关系。在下一小节中，我们将说明如何通过张量变换和缩并实现这一操作。

4.2 Galerkin 算子的无矩阵计算

上一小节推导的分解使我们能够计算 Galerkin 刚度作用，而无需逐个显式构造粗单元刚度矩阵。相反，我们保留预计算算子 $G_s^{(\ell)}$，并利用相应的细网格刚度插值系数组合其作用。

考虑层级 ℓ 上的笛卡尔网格，其中包含 $N_x^{(\ell)} \times N_{(\ell)\ y}$ 个单元。我们将与每个粗单元相关联的细网格刚度插值系数收集到张量 $W^{(\ell)} \in \mathbb{R}^{N_x^{(\ell)} \times N_y^{(\ell)} \times 4^\ell}$ 中，其中 $W_{ijs}^{(\ell)}$ 表示粗单元（i， j）内细网格子单元 s 的系数。该张量通过重新排列原始细网格刚度插值系数得到，不进行平均或以其他方式修改其数值。层级 ℓ 上的节点位移场表示为 $U^{(\ell)} \in \mathbb{R}^{2\times(N_x^{(\ell)}+1)\ \times\ (N_y^{(\ell)}+1)}$。与细网格公式中一样，规则网格连接关系使得我们能够直接从节点张量中提取单元位移向量。将这些向量收集起来得到 $E^{(\ell)} \in \mathbb{R}^{8\times N_x^{(\ell)} \times N_y^{(\ell)}}$， 其中 $E_{qij}^{(\ell)}$ 是由（i， j）索引的单元的位移向量中的第 q 个分量。于是，局部 Galerkin 刚度作用可表示为

$$H_{aij}^{(\ell)} = \sum_{s=0}^{4^\ell - 1} W_{ijs}^{(\ell)} \sum_{q=0}^{7} G_{saq}^{(\ell)} E_{qij}^{(\ell)}, \tag{24}$$

其中， $a \in \{0, \ldots, 7\}$ 和 $q \in \{0, \ldots, 7\}$ 分别按照通用的局部自由度排序对输出项和输入项进行索引，且 $G_{saq}^{(\ell)} = \left(G_s^{(\ell)}\right)_{aq}$。张量 $H^{(\ell)} \in \mathbb{R}^{8\times N_x^{(\ell)} \times N_y^{(\ell)}}$ 包含由此得到的单元力贡献。

不同于细网格公式先应用单个参考单元矩阵、再进行标量加权，公式 24 在每个单元内组合了多个固定算子。不过，该计算完全由常规张量运算构成。具体而言，加权矩阵—向量乘积及其求和可以通过单次张量缩并完成。

列表 5说明了使用 JAX 和 Einops 原语再次展示该过程。预先计算的伽辽金矩阵 a re 以如下形式提供 G，而 w 包含原始细网格刚度插值系数。rearrange 操作 n

代码清单 5 使用预计算局部矩阵和细网格刚度插值系数，简化了二维 Galerkin 刚度算子的无矩阵应用。

```
1 import jax.numpy as jnp 2 from jax import lax 3 from einops importrearrange, reduce, einsum 4 5
def galerkin_matrix_free_2d(u, G, w, l): 6 # u: (2, Mx+1, My+1) 7 # G: (4**l, 8, 8) 8 # w: (Nx, Ny) 9 # l: 层级
编号（整数） 1 r = 2**l 2 Mx, My = u.shape[1] - 1, u.shape[2] - 1 4 # 按粗单元对细网格系数分组 5
weights = rearrange(w, "(x i) (y j) -> (x y) (ij)", i=r, j=r) # (Mx*My,4**l) 7 # 提取粗单元局部位移 8
patches =lax.conv_general_dilated_patches( 9 u[None, ...] ,0 filter_shape=(2, 2), 1 window_strides=(1,
1), 2 padding="VALID", 3 dimension_numbers=("NCHW", "OIHW", "NCHW"), 4 )[0]5 local_u =
rearrange(patches, "a x y -> (x y) a") # (Mx*My,8) 7 # 应用加权伽辽金算子 8 local_f = einsu
10
m(weights, G, local_u, "e s, s a q,e q -> e a") # (Mx*My,8) 0 # 将局部贡献组装到共享节点 1 local_f =
1
rearrange( 2 local_f, 3 "(x y) (c i j)-> c (x i) (y j)", 4 x=Mx, y=My, c=2, i=2, j=25 ) 6 local_f =jnp.pad(local_
12
f, ((0, 0), (1, 1), (1, 1))) 7 f = reduce(local_f, "c (x i) (y j) -> c x y", "sum", i=2, j=2) # (2,Mx+1, My+1) 9
1
return f
16
1
1
1
2
2
2
2
2
2
26
2
2
29
3
3
3
3
3
3
3
3
38
3
```

按粗单元和子单元对这些系数进行分组。在第 18 行，我们使用 JAX 原语执行卷积，以提取局部位移向量；第28 行则计算公式24。随后，使用为细网格算子引入的相同张量变换来组装所得的局部力贡献。因此，Galerkin 刚度作用既不需要构造全局稀疏矩阵，也不需要显式生成各个粗单元刚度矩阵。

4.3 Galerkin 算子的批处理与切片

代码清单 5 同时对所有粗单元执行张量缩并。然而，对于大型问题，中间数组可能需要大量内存。因此，我们将粗单元划分为较小的批次，并依次计算其局部刚度作用。与为细网格算子引入的空间切片不同，这种分解不需要重叠输入区域，因为不同粗单元的局部力贡献可以独立计算。

对于包含 $N_x^{(\ell)} \times N_y^{(\ell)}$ 个粗单元的二维网格，完整的张量收缩涉及 $4^\ell \times N_x^{(\ell)} \times N_y^{(\ell)}$ 个子单元贡献。每次处理 B_{batch} 个粗单元，可将单批次评估的贡献数量减少至 $B_{\text{batch}} \times 4^\ell$。总计算工作量保持不变，但同时处理的子单元贡献数量约减少 $(N_x^{(\ell)} \times N_y^{(\ell)})/B_{\text{batch}}$ 倍。这限制了与张量收缩相关的临时存储量，不过仍必须存储输入系数和生成的力贡献。

清单6 展示 了使用 `lax.map` 进行批量评估的方法。代码将第 28 of
清单5，w 同时保持后续组装不变。函数 apply_element 计算权重 ed

清单 6 使用 JAX 对无矩阵 Galerkin 刚度作用进行批量评估。批次大小 B_{batch} 控制同时处理的粗单元数量。

```
def apply_单元(inputs):
    w_e, u_e = 输入
    返回 einsum(w_e,     G, u_e,   "s, s a q, q -> a")

# 按批次计算局部刚度作用
局部_f  = lax。map(
    应用_单元,
    （权重,    局部_u），
    批次 -大小=B -批次 ，
)                  # local_f: (Mx*My,     8)
```

Galerkin 刚度作用于单个粗单元的`lax。map` 会对最多 mo 个批次执行此函数 st B_{batch} 单元。这使得无需展开完整计算即可执行张量收缩。
最终得到的局部力贡献与未切片形式具有相同的形状，并通过从第 31 行开始的张量操作进行组装，见代码清单5。与细网格切片方法一样，按顺序处理各批次并不会消除 GPU 并行性，因为每个批次内的缩并仍可并行计算。只要批次大小足够大，就能在不显著降低 GPU 利用率的前提下，减少中间内存需求。

向三维的扩展很直接。每个六面体单元有二十四个位移自由度，而层级 ℓ 的单元包含 8^ℓ 个原始细网格单元。因此，Galerkin 算子由 8^ℓ 个大小为 24×24 的固定矩阵表示，局部刚度作用则变为

$$\boldsymbol{f}_e^{(\ell)} = \sum_{s=0}^{8^\ell - 1} w_{e,s} G_s^{(\ell)} \boldsymbol{u}_e. \tag{25}$$

对于包含 $N_x^{(\ell)} \times N$ (ℓ) y $\times N$ (ℓ) z 个粗单元的三维网格，计算完整的张量收缩需要 $8^\ell \times N$ ($_x^\ell \times N_y^{(\ell)} \times N$) (ℓ) z 个子单元贡献。每次处理 B_{batch} 个粗单元，可将每批计算的贡献数减少到 $B_{\text{batch}} \times 8^\ell$。与二维情况一样，这能减少张量收缩所需的临时存储，而不改变总计算工作量。

在实现中，有界批处理与沿 x 方向对粗网格进行空间切片相结合。每个切片块覆盖网格在 y 和 z 方向上的完整范围，而细网格权重的分组、局部位移的提取、Galerkin收缩以及节点贡献的组装均在切片块内局部完成。这样可避免为整个粗网格生成相应的单元局部中间数组。在三维情况下，这一优势尤为显著，因为每个局部 Galerkin 算子都包含 24×24 个条目。通过以有界批次计算加权贡献，我们无需生成空间变化的粗单元刚度矩阵，同时可限制中间计算所需的内存，并保留 Galerkin 刚度作用。

5 topax 库

我们使用 JAX 实现了所提出的无矩阵算子和 Galerkin 算子，topax 是一个模块化的拓扑优化库，采用 JAX 开发 [Bradbury 等人， 2018]。该库为基于密度的拓扑优化提供了灵活的框架，可在不修改底层有限元实现的情况下组合不同的数值方法。它支持二维和三维笛卡尔四边形和六面体网格，并可在 CPU 和 GPU 上运行。图 4 展示了该库的整体架构，包括拓扑优化工作流以及用于有限元分析的多重网格层次结构。该实现以开源软件形式发布，并可从 https://github.com/pedrozudo/topax 获取。

5.1 库架构

topax 将拓扑优化问题的主要组成部分拆分为可互换的模块。优化算法可以独立于有限元求解器进行选择，而滤波与投影操作则可根据问题需求进行配置。具体而言，该库既支持基于卷积的密度滤波和 Helmholtz 型滤波，也支持使用不同的投影函数将滤波后的设计变量映射为物理密度。

线性 求解器同样采用模块化设计。有限元平衡方程采用多重网 id
预条件 ...共轭梯度法，并支持配置平滑过程以及替换稀疏直接求解器 er

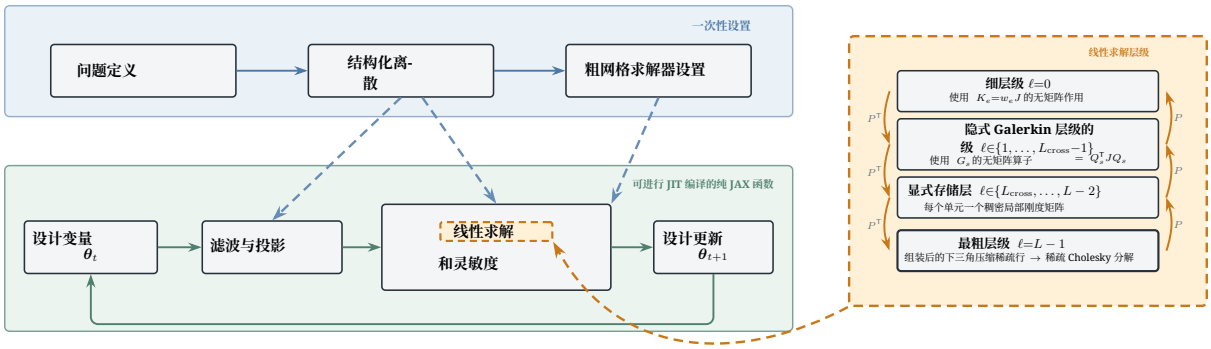

图 4：topax 拓扑优化工作流概览。静态离散化数据和稀疏求解器资源只需构建一次（蓝色方框）；滤波与投影、线性求解、灵敏度评估和设计更新则构成经编译的设计迭代（绿色方框）。

用于最粗层级。这种分离方式使各个数值组件能够被替换或扩展，而无需改变整体优化流程。

该实现依赖 JAX 来执行张量运算、自动微分和即时编译。通过使用标准的 JAX 基本算子来表达数值计算，同一实现可面向不同的硬件架构，包括 CPU 和 GPU。结构化离散和基于张量的算子针对二维和三维空间维度进行了构造，使主要数值算法能够共用同一实现。

5.2 多重网格层次结构的构建

多重网格层次结构会根据网格层级采用不同形式的刚度算子表示。最细层使用第3节提出的切片卷积无矩阵算子。在后续层级中，则使用第4节介绍的隐式张量缩并来计算 Galerkin 刚度作用。虽然隐式 Galerkin 表示无需为每个粗单元构造单独的刚度矩阵，但预计算算子的数量会随粗化层级增加。在二维中，层级 ℓ的每个单元需要 4^ℓ 个算子，而在三维中，相应的数量为 8^ℓ。因此，在层次结构较深的层级上计算加权贡献的开销会越来越大。为此，topax引入了一个可配置的交叉层级 L_{cross}。对于层级 $\ell \in \{1,\dots,L_{\text{cross}}-1\}$，Galerkin 作用采用隐式方式计算
，使用原始细网格刚度插值系数和预计算算子 $G_s^{(\ell)}$（参见第4节）。在层级 L_{cross}，我们使用公式 23为每个单元显式构造单独的局部刚度矩阵。也就是说，当问题经过充分粗化后，我们会为每个单元构造一个单独的刚度矩阵。

这些矩阵构造完成后，其余层级通过递归的逐单元 Galerkin 投影得到。在二维中，每个父单元将其四个子单元的刚度矩阵合并；在三维中，则合并八个子单元的刚度矩阵。与隐式形式不同，这种表示方式为每个单元存储一个刚度矩阵，但避免了反复累加所有原始细网格子单元的贡献。通过局部矩阵–向量乘积应用这些显式存储的单元矩阵，随后组装得到的节点力。因此，中间层级仍无需组装全局稀疏刚度矩阵。交叉层级由此决定隐式 Galerkin 收缩的计算成本与存储各单元刚度矩阵所需内存之间的权衡。可根据问题规模和可用硬件内存进行选择。

全局刚度矩阵仅在最粗层级显式存储，并以压缩稀疏行（CSR）格式组装。所得线性系统使用稀疏 Cholesky 分解求解，采用 CHOLMOD [Chen 等人， 2008] 在 CPU 上使用 cuDSS[1] 在 NVIDIA GPU 上使用。稀疏模式和符号分解可在初始化期间确定，而数值分解则会随着优化过程中材料分布的变化而更新。

5.3 优化流程

拓扑优化工作流如图4所示。从设计变量出发，所选的滤波方法生成滤波密度场，随后通过选定的投影函数将其转换为物理密度。根据 SIMP 模型，这些密度决定单元的刚度插值系数。对于每种材料分布，该库都会更新刚度系数和

[1] https://developer.nvidia.com/cudss

相应的多重网格数据。然后，使用第2.2节介绍的 MGPCG 方法求解有限元平衡方程。

求解平衡方程后，将计算目标函数和约束函数及其灵敏度。JAX 可对数值运算进行自动微分，无需手动实现相应的灵敏度表达式，即可获得所需导数。需要注意的是，线性求解过程中的微分通过伴随方法实现，不会展开共轭梯度循环。随后，将得到的目标函数和约束灵敏度传递给选定的优化算法，以更新设计变量。此过程会重复进行，直至达到规定的收敛准则。

如图 4所示，该库区分了仅依赖问题定义的操作与优化过程中必须重复执行的操作。具体而言，结构化离散和稀疏求解器资源的初始化与重复执行的优化计算分开进行。在整个优化过程中，网格结构和最粗层刚度矩阵的稀疏模式保持不变，而其随材料变化的数值则会随着设计的演进而更新。这样一来，滤波、投影、有限元分析、灵敏度评估和设计更新便可集成到编译后的 JAX 工作流中，同时仍能灵活配置各个数值组件。

6 相关工作

6.1 大规模拓扑优化

Amir 等人 [2014] 将多重网格预条件引入拓扑优化，奠定了一种后来成为许多大规模实现基础的方法。尽管他们的实验仅限于规模相对较小的问题，但后续工作展示了基于多重网格的方法具有可扩展性。尤其是，Aage 等人 [2015] 开发了一个基于 PETSc 的大规模并行拓扑优化框架 [Balay 等人，1997], 为利用分布式内存计算实现十亿体素级拓扑优化奠定了基础 [Aage 等人，2017，Baandrup 等人，2020]。

无矩阵有限元公式通过避免组装和存储全局刚度矩阵，为减少大规模拓扑优化中的内存消耗提供了另一种途径。近期，Wang 等人[2025] 将无矩阵有限元分析与几何多重网格预条件相结合，并在 MATLAB 中实现了该方法，可在配备 64 GB 内存的工作站上处理多达 1.28 亿个六面体单元的问题。他们的方法以节省内存的分块方式计算基于单元的矩阵运算，同时采用非二进制粗化来降低多重网格层次结构的内存需求。相比之下，我们的公式利用卷积和张量变换来发挥结构化网格连接关系的优势，并保留二进制 Galerkin 层次结构，同时隐式计算粗网格算子。

6.2 GPU 加速

GPU 加速已作为提升拓扑优化计算效率的一种手段得到广泛研究。Schmidt 和 Schulz [2011] 展示了在图形处理硬件上实现拓扑优化的早期方案。Dick 等人 [2011] 开发了一种用于弹性模拟的无矩阵几何多重网格求解器，该求解器采用六面体有限元，并通过专用 CUDA 实现利用规则网格连接关系。在此基础上，Wu 等人 [2015] 开发了一种 GPU 加速的高分辨率拓扑优化系统，通过即时构建最细网格模板并采用高效内存多重网格技术，降低了内存需求。Martínez-Frutos 等人 [2017] 进一步研究了采用多层次粒度的基于 GPU 的拓扑优化，并针对有限元分析、灵敏度计算、滤波和设计更新开发了专用 CUDA 内核。Ratnakar 等人 [2021] 研究了一种基于 GPU 的无矩阵共轭梯度求解器，并采用定制的节点连接关系存储。最近，Yang 等人 [2026b] 开发了用于三维 SIMP 拓扑优化的融合式 gather–矩阵乘法–scatter 内核，通过将单元级运算合并到单个 CUDA 内核中，减少了中间内存流量。与这些专用 GPU 实现不同，我们的公式通过卷积和结构化张量变换表达刚度作用，从而能够利用经过硬件优化的数值内核，而无需定制 CUDA 内核。

Galerkin 粗网格算子的表示会给基于 GPU 的多重网格求解器带来额外的计算和内存需求。尽管 Dick 等人 [2011], Wu 等人 [2015], 和 Martínez-Frutos 等人 [2017]采用了 Galerkin 多重网格，但其实现会显式构造并存储粗网格模板或系数矩阵。具体而言，Martínez-Frutos 等人 [2017] 指出，与这些系数相关的内存需求和全局内存访问是其实现的局限。最近，Yang 等人 [2026a]开发了一种用于三维拓扑优化的 Galerkin 多重网格求解器，将无矩阵细网格计算与显式构造的粗网格算子相结合。其第一个粗层通过局部 Galerkin 聚合获得，而后续层级则使用稀疏 Galerkin 乘积构造。

我们的方法 在这些 Galerkin 算子的表示和计算方式上有所不同。我们不使用局部 Galerkin 乘积 仅用于构造显式存储的粗单元矩阵或节点模板的投影，我们保留 al a

由细网格刚度插值系数和预计算局部算子组成的因子化表示。在多重网格 V 循环中，通过批量张量收缩直接计算这种表示，无需在选定的隐式层级上显式生成粗单元刚度矩阵。在足够粗的层级上，算法会切换为显式存储的算子，从而能够在内存消耗与重复算子计算的成本之间进行可配置的权衡。

6.3 高级编程与可微编程

高级编程框架日益广泛地用于开发硬件加速的拓扑优化方法，而无需专门的底层实现。Träff 等人 [2023] 展示了使用 OpenMP 实现的 GPU 加速拓扑优化方法 [Dagum 和 Menon, 1998] 以及 Futhark [Hen-riksen et al., 2017], 结果表明，高级并行编程框架无需专门的 CUDA 内核也能实现有竞争力的性能。他们的研究还突显了内存消耗方面的挑战，尤其是在 Futhark 实现中，编译器生成的中间数组会大幅增加内存使用量。近期，Nobari 和 Ahmed [2026]推出了 pyFANTOM，这是一个用于拓扑优化的模块化 Python 框架，配备独立的 CPU 和 GPU 后端。其 GPU 实现采用 CuPy [Okutaet al., 2017] 并使用专门的 CUDA 内核进行无矩阵有限元分析和几何多重网格，而非采用高级张量运算。

自动微分是采用高级数值计算框架的另一动因，因为它简化了灵敏度分析，并有助于开发灵活的优化方法。Nørgaard 等人 [2017]研究了自动微分在拓扑优化中的应用。随后，Chandrasekhar 等人 [2021] 推出了 AuTO，这是一个基于 JAX 的教学框架，展示了如何通过稠密有限元矩阵组装和直接线性求解器，利用自动微分进行结构优化。其他基于 Python 的框架则封装现有数值求解器，以支持可微分的有限元分析。Xue 等人 [2023]开发了 JAX-FEM，将自动微分与稀疏矩阵组装及现有线性求解器后端相结合。类似地，Jia etal. [2024] 推出了 FEniTop，这是一个基于 Python 的 SIMP 拓扑优化框架，封装了 FEniCSx [Baratta etal., 2023] 以及 PETSc，以进行有限元组装和线性系统求解，并使用符号微分进行灵敏度分析。近期研究还探索了自动微分在结构形状优化 [Wu、 2023],水平集拓扑优化 [Neofytou et al.、2024], 神经拓扑优化 [Sanuet al.、 2026],多物理场优化 [Padhy et al., 2026]等领域中的应用。

尽管这些方法展示了高级编程和自动微分的优势，但底层数值运算的效率对于大规模拓扑优化仍至关重要。Träff 等人 [2023] 报告称，其 Futhark 实现由于编译器生成了中间数组，内存消耗有所增加。在可微框架中，AuTO 会组装稠密刚度矩阵，而 JAX-FEM 和 FEniTop 则依赖稀疏矩阵组装。尽管与稠密矩阵相比，稀疏组装显著减少了存储需求，但这些方法仍需承担显式组装刚度算子带来的内存开销，因此在应用于超大规模三维问题时受到限制。相比之下，topax 通过卷积和结构化张量变换来表达细网格刚度作用，并使用批量张量收缩计算 Galerkin 粗网格算子。切片和批处理限制了中间数组的内存需求，而 JAX 则提供了对硬件优化计算内核、即时编译和自动微分的支持。由此得到的公式兼具高级可微编程环境的灵活性，以及在 CPU 和 GPU 上的高计算性能。

7 二维验证实验

本节介绍文献中的两个二维拓扑优化示例，用于验证实现并评估 topax 在二维情况下的计算可扩展性。所有二维实验均在一台配备 NVIDIA GeForce RTX 4090 GPU（拥有 24GiB 设备内存）的工作站上进行。

7.1 采用线性衰减权重的密度滤波器

我们首先考虑Sigmund [2007]研究的 MBB 梁算例。原始问题定义在一个矩形设计域上，其尺寸为 120×40，厚度均匀，为 $t = 1$，并采用 120×40个单元进行离散化。假设满足平面应力条件。左边界受到水平位移约束，右下角受到垂直位移约束。左上角施加大小为 1 的向下集中载荷。设定的体积分数为 0。5，SIMP 惩罚指数为 $p = 3$，杨氏模量为 $E = 1$，泊松比为 $\nu = 0.3$，最小相对刚度为 10^{-9}。除原始离散化外，我们还考虑几何缩放网格 360×120， 750×250， 1500×500，3000×1000，以及 5400×1800。这些规模更大的算例旨在评估计算扩展性和最终设计质量；它们并非针对固定物理域开展的网格收敛研究。单元尺寸

在每个网格中均固定为 1×1。因此，随着单元数量增加，物理设计域也随之扩大，同时保持 3:1 的纵横比。

使用标准密度滤波器对设计变量进行正则化，该滤波器采用线性衰减权重，由Bourdin [2001] 提出，并由Sigmund [2007]加以研究。在topax所用的结构化网格上，该滤波器可直接映射为 JAX `lax.conv_general_dilated` 原语。所有网格的滤波半径均固定为 $R = 3.5$ 个单元宽度。更具体地说，滤波器会纳入形心与被滤波单元的形心之间欧几里得距离不超过3.5倍单元宽度的单元。这些单元的贡献随距离线性递减，并在达到滤波半径时降为零。对于本文考虑的单位尺寸正方形单元，这会形成一个 7×7具有圆形支撑域的包围模板。所有网格均使用相同的模板。因此，不同网格尺寸下的这些实验对应的是几何缩放后的问题，而不是对固定物理域进行逐步网格细化。此外，不应用 Heaviside 投影或其他阈值化操作。因此，滤波场直接表示用于 SIMP 插值、有限元分析、体积约束以及优化设计可视化的物理密度。

优化问题采用移动渐近线法（MMA）求解 [Svanberg, 1987]。前 3 个网格使用两个多重网格层级，而最大的 3 个网格使用三个层级。在这两种配置中，细网格刚度算子均采用第 3.4 节所述的切片策略，以无矩阵方式应用，从而限制中间张量所需的内存。在两层配置中，第二层是最粗层；其刚度矩阵会被组装，并直接求解相应的系统。在三层配置中，中间 Galerkin 算子无需组装或存储其全局刚度矩阵即可计算。其逐单元贡献按照第 4.3 节所述，以有界批次进行处理。仅组装最粗层刚度矩阵，并直接求解其系统。

满足两个停止条件中的任意一个时，优化即终止。第一个条件是设计变量的最大变化不超过 0.01。第二个条件用于检测柔顺度是否进入平台期，并在
仅在至少进行 500 次设计迭代后。在迭代 $k \geq 500$ 时，若$|C^{(k)} - C^{(k-100)}| / |C^{(k)}| \leq 10^{-5}$，则认为柔顺度已趋于平稳。该平台判据将当前柔顺度与 100 次迭代之前的柔顺度进行比较，并在判据启用后于每次迭代时进行评估。设计迭代次数上限设为 10 000 次。

图 5 展示了终止时的物理密度场。该 120×40 情形与 Sigmund [2007]报告的参考设计一致。较大网格上得到的布局与粗网格结果不同，并包含更多、更精细的结构构件。由于网格尺寸增大时滤波半径仍以单元单位保持固定，因此施加的长度尺度相对于设计域尺寸有所减小。因此，这些情形并不构成对固定物理问题的逐次网格细化。

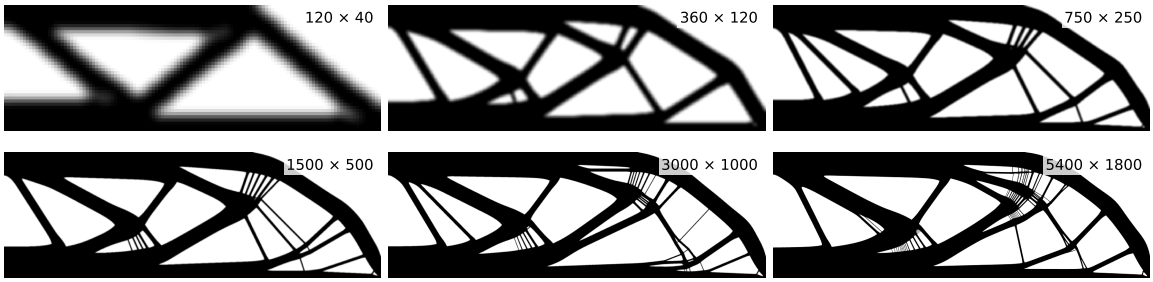

图 5：物理 密度场，采用线性衰减权重的密度滤波器得到。网格尺 ze
从左到右 从上到下递增。不应用投影。

对于原始 120×40问题，topax 得到的柔顺度为218.1546，而218.3 是Sigmund [2007]报告的数值。这对应的相对差异约为 -0.067%。topax 运行在经过 452 次迭代后满足设计变化准则。相比之下，Sigmund 报告的参考结果是在经过 1000 次迭代后得到的。

物理密度场中间密度的程度通过非离散度度量（M_{nd}）进行量化，该度量由Sigmund [2007]提出：

$$M_{\mathrm{nd}} = \frac{100}{N_e} \sum_{e=0}^{N_e - 1} 4\rho_e \left(1 - \rho_e\right). \tag{26}$$

物理密度为零或一时，对 M_{nd}没有贡献；而物理密度为0.5时，贡献最大。因此， $M_{\mathrm{nd}} = 0\%$ 表示完全离散的设计，而数值越大，表示中间密度的占比越高。对于原始问题，topax 得到 $M_{\mathrm{nd}} = 23.933\%$，而 24.02% 是Sigmund报告的数值。这对应的相对差异约为-0.362%。

表1 汇总了计算结果。报告的每次设计迭代平均耗时不包括编译和不计时的预热迭代。MGPCG 平均迭代次数指每次设计迭代中执行刚度求解时的迭代次数。GPU 内存使用量峰值通过在每次运行期间以0.2s为间隔采样设备内存，并减去实验开始前立即测得的基线值来确定。已禁用 JAX 的 GPU 内存预分配，因此报告的数值反映实验期间分配量的增量峰值。完整的测量流程和实验配置记录在随topax分发的实验脚本中。

表1：几何缩放 MBB梁问题的优化和计算结果。

mesh	单元	层级	设计迭代	平均每次迭代时间 [秒]	平均 MGPCG 迭代次数	峰值内存 [GiB]	柔顺度	非离散性 [%]
120×40	4,800	2	452	0.010	14.42	0.611	218.1546	23.933
360×120	43,200	2	3,581	0.022	13.07	0.666	196.7388	13.310
750×250	187,500	2	4,636	0.050	12.40	0.875	192.5300	8.262
1500×500	750,000	3	7,850	0.208	16.32	1.053	191.3681	4.622
3000×1000	3,000,000	3	10,000	0.549	17.93	2.361	192.5512	2.713
5400×1800	9,720,000	3	10,000	1.416	19.26	6.395	194.5227	1.661

平均设计迭代时间从 4800 个单元时的 0.010s 增加到 9.72 百万个单元时的 1.416s。在同一范围内，基线校正后的 GPU 显存峰值消耗从 0.611GiB 增加到 6.395GiB。因此，最大规模的问题也能轻松容纳在本实验所用的 24GiB NVIDIA GeForce RTX 4090 中。刚度 MGPCG 迭代次数的平均值仍处于适中水平，从约 14.42 增加到 19.26。

非离散性从 23.933% （原始问题）降至 1.661% （最大规模问题）。图 5所示的物理密度场逐渐变得更加清晰，这可以解释为：随着网格尺寸增大，滤波模板在单元单位下保持不变。因此，滤波半径及其相关过渡区域相对于完整设计域逐渐缩小。因此， M_{nd} 的下降反映的是滤波半径与设计域尺寸之间比例的变化，而不是固定物理问题的网格收敛。

3000×1000 和 5400×1800 案例均达到规定的 10000 次设计迭代上限，但都未满足任一停止准则。因此，报告的柔顺度值对应于最终的优化迭代值，且不应将所得拓扑结构视为已根据规定准则收敛。尽管如此，这些实验表明，topax 能够在消费级 GPU 硬件上优化一个包含近千万个单元的二维问题，同时保持适中的 MGPCG 迭代次数和低于设备可用容量的内存占用。

7.2 Helmholtz 型密度滤波器

接下来，我们考虑 Lazarov 和 Sigmund [2011]所用的悬臂梁示例。设计域的尺寸为4×1 ，厚度为 $t = 0.01$，并假设满足平面应力条件。大小为0。2 的向下集中载荷施加在右边界的中点，左边界则完全固定。规定的体积分数为 0。5，SIMP 惩罚指数为 $p = 3$，杨氏模量为 $E = 1$，泊松比为 $\nu = 0.2$，最小相对刚度为 10^{-9}。

使用由Lazarov 和 Sigmund[2011]提出的亥姆霍兹型密度滤波器对设计变量进行正则化。对于每个网格，滤波半径在物理尺寸下固定为 $r = 0.02$ 。因此，与几何缩放的 MBB 梁示例（参见第 7.1 节）不同，这些算例代表同一物理问题的逐级细化。未应用投影或阈值化操作，而是将滤波场直接用作 SIMP 插值、体积约束和可视化中的物理密度。
Lazarov 和 Sigmund 考虑的三个网格包含 400×100、 800×200 和 1200×300 个单元。此外，我们还考虑了包含 1600×400、 3200×800 和 6400×1600 个单元的网格，其中最大问题包含10.24 百万个单元。采用 Bendsøe 和 Sigmund [提出的最优性准则法（OC） 2004], 更新设计变量，移动限值为 0.2。前三个网格采用两个多重网格层级，最大的三个网格采用三个层级。使用与上一小节所述相同的无矩阵切片、隐式 Galerkin 批处理和直接粗网格求解策略。刚度系统以及正向和伴随亥姆霍兹系统均使用 MGPCG 求解。报告的设计迭代时间包括所有求解；表 2 中报告的平均 MGPCG 迭代次数则特指刚度求解。我们采用上一小节所述的停止准则以及 GPU、计时和内存测量流程。

图 6 展示了在六种网格分辨率下获得的物理密度场。优化后的拓扑结构具有相同的整体布局，在不同网格分辨率下，结构构件和开孔的排列方式也相同。所得设计之间的有限差异表明，对于本例，网格依赖性得到了有效控制，这符合采用具有恒定滤波半径的亥姆霍兹型密度滤波器时的预期 r。由于未应用投影，在每种分辨率下，实体区域和空域之间的平滑过渡仍然清晰可见。

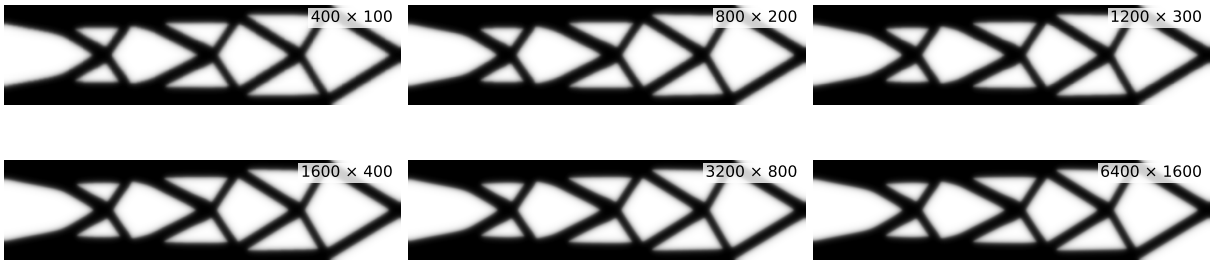

图 6：物 理密度场，使用亥姆霍兹密度滤波器并采用 $r = 0.02$获得。网格分辨率逐渐提高 es 从左到 右、从上到下。未应用投影。

为匹配 Lazarov 和 Sigmund [2011], 所采用的柔顺度约定，报告的比较值通过将原始二维柔顺度乘以厚度 $t = 0.01$获得。对于原论文中报告的三种网格，topax 得到的柔顺度值分别为 16.5620、16.5369 和 16.5329，而相应的参考值分别为 16.6556、16.6690 和 16.6785。相应的相对差异约为 -0.56%、-0.79% 和 -0.87%。

表 2 总结了计算结果。计时和内存数据遵循第 7.1 节中介绍的定义。六种分辨率下，非离散度度量均保持在接近 26% 的水平。不同于几何缩放的 MBB梁问题中观察到的递减值，这一近似恒定的数值是意料之中的，因为在网格细化过程中，物理域和物理滤波半径均保持不变。

表 2：网格细化悬臂梁问题的优化和计算结果。

mesh	单元	层级	设计迭代	平均每次迭代时间 [秒]	平均 MGPCG 迭代次数	峰值内存 [GiB]	柔顺度	非离散性 [%]
400×100	40,000	2	1,305	0.037	10.19	0.822	16.5620	26.418
800×200	160,000	2	2,897	0.075	9.17	1.002	16.5369	26.121
1200×300	360,000	2	5,773	0.116	8.45	1.344	16.5329	26.048
1600×400	640,000	3	5,927	0.219	8.72	1.199	16.5382	26.026
3200×800	2,560,000	3	6,819	0.445	8.09	2.445	16.5531	25.999
6400×1600	10,240,000	3	6,548	1.136	7.43	7.398	16.5714	25.996

全部六个优化问题均满足规定的停止准则。 400×100 和 800×200案例根据设计变化准则终止，其余案例则根据柔度平台准则终止。

平均设计迭代时间从 40 000 个单元时的 0.037 s 增至 10.24 百万个单元时的 1.136 s。在相同范围内，基线校正后的 GPU 显存峰值消耗从 0.822 GiB 增至 7.398 GiB。刚度 MGPCG 迭代次数的平均值保持在约 7.4 至 10.2 之间，网格细化并未导致其系统性增加。因此，最大规模案例展示了如何在普通硬件（NVIDIA GeForce RTX 4090）上使用 topax 求解包含一千多万个有限元的拓扑优化问题，同时保留了使用相同物理滤波半径、在较低网格分辨率下获得的整体结构布局。

8 大规模三维数值实验

8.1 GPU 上的大规模 Galerkin 多重网格

我们将 topax 与两个已发布的 GPU 实现进行比较，这两个实现使用 Galerkin 粗网格算子构建多重网格层次结构：Futtop[2], Träff 等人的 Futhark 实现 [2023], 以及 pyFANTOM[3] [Nobari 和 Ahmed, 2026]。

基准测试采用 Träff 等人 [2023]研究的三维线弹性悬臂梁。长方体设计域的固定尺寸为 $2 \times 1 \times 1$，并通过逐渐加密的网格进行离散。在一端面上的两条带状区域内，所有位移分量均受到约束；总大小为 1 的分布载荷沿负 z方向作用于靠近另一端的半圆形区域上。载荷和边界条件遵循原始基准测试；其完整定义可在所引用的论文以及随本工作附带的源代码中查阅。

我们考虑 11 种网格，规模从 $64 \times 32 \times 32$到 $640 \times 320 \times 320$ 个三线性六面体单元不等，对应 65 536 至 65 536 000 个单元。给定的体积分数为 0。1。SIMP 惩罚指数取 $p = 3$,

[2]https://github.com/topopt/futtop [3] https://github.com/ahnobari/pyFANTOM

杨氏模量为 $E = 1$，泊松比为 $\nu = 0.3$，最小相对刚度为 10^{-6}。使用权重线性衰减、半径为1。5 个单元宽度的密度滤波器对设计变量进行正则化。不应用投影或阈值化操作。三种实现均采用最优性准则法更新设计，移动限值为 0。2。每个问题均运行 20 次设计迭代。实验设置参照Träff 等人 [2023]。

运行时间和内存扩展

图 7 比较了 20 次设计迭代的运行时间、GPU 的峰值设备内存消耗，以及刚度 MGPCG 迭代次数的平均值。 topax 是在所有考察分辨率下运行最快的实现。相较于速度最快的竞争实现，其加速比介于 6.1 至 13.6。对于包含 65.5 百万个单元的最大问题，topax 完成 20 次设计迭代耗时 99.6s，相比之下，Futtop 耗时 720.0s ，pyFANTOM 耗时 9595.8s 。这分别对应于 7.2 和 96.4 的加速比。
在较大分辨率下，设备内存结果呈现出相同的总体趋势。对于包含 65.5 百万个单元的最大网格，topax 的设备内存峰值消耗为 33.5 GiB，Futtop 为 76.5 GiB，pyFANTOM 为 74.4GiB。因此，Futtop 和 pyFANTOM 所需的设备内存分别约为 topax 的 2.28 倍和 2.22 倍。下文将进一步考察内存扩展情况及其与 Galerkin 层次结构不同表示形式之间的关系。
pyFANTOM 的 MGPCG 收敛差异尤为显著。其平均迭代次数从最小网格的 62.8 增至 $640 \times 320 \times 320$ mesh 的 435.75。在相同范围内，topax 的平均迭代次数则从 30.6 降至 20.8。在这方面，Futtop 的表现与 topax 更为接近。例如，在最大网格上，topax 每次设计迭代平均需要进行 18.9 次刚度 MGPCG 迭代，而 Futtop 为 22.85 次。因此，topax 与 Futtop 之间的运行时间差异，不能仅用 MGPCG 迭代次数始终较少来解释。相比之下，pyFANTOM 的迭代次数不断增加，是其运行时间增长的重要原因，尽管其开销更高的多重网格循环也造成了部分差异。

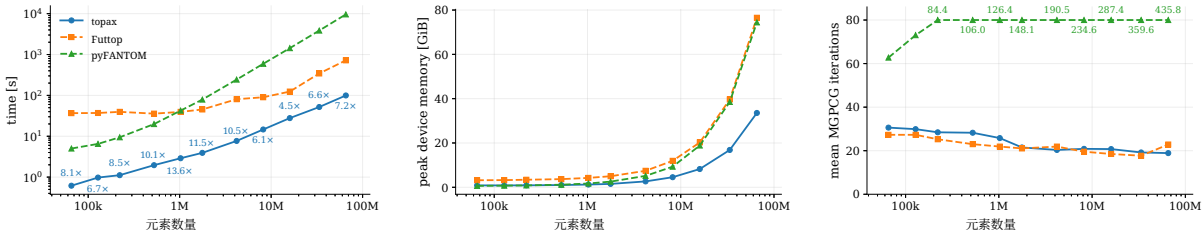

图 7：topax、Futtop 和 pyFANTOM 的比较。左：20 次设计迭代的运行时间（秒）；标注显示 topax 相较于最快的其他实现方案的加速比。中：设备峰值内存消耗（GiB）。右：20 次设计迭代中的平均刚度 MGPCG 迭代次数；超过 80 的数值会截断显示，并标注其实际值。

Galerkin 层次结构及预条件器的表示细节

这三种实现均采用 MGPCG 求解刚度系统，使用四种网格分辨率，相对容差为 10^{-5}。它们的粗算子通过 Galerkin 投影获得，参见公式 4。尽管三种实现采用相同的代数构造方式，但对所得算子的表示和应用方式各不相同。

在 topax 中，层次结构各层级的刚度算子采用不同的表示方式。在最细网格上，使用第 3.4 节所述的切片策略，以无矩阵方式计算刚度作用。在第一层粗化网格上，使用第 4.3 节所述的批处理公式，隐式应用 Galerkin 算子。在第二层粗化网格上，显式构造各个粗单元的刚度矩阵。在最粗网格上，将全局刚度矩阵组装为 CSR 格式，并通过 cuDSS 直接分解求解。每次粗网格校正前后均执行一次加权 Jacobi 步骤，阻尼参数为0.6 。

topax 和 Futtop 都能在不组装全局刚度矩阵的情况下应用细网格刚度算子。Futtop 通过显式的节点和单元索引实现这一操作：对于每个节点，它会识别相邻单元，收集其局部位移自由度，并将对应的刚度贡献相加。相比之下，topax 使用卷积原语和空间切片来表达相同的操作，无需显式构建或遍历单元到节点索引。两种方法对第一层粗化网格的处理也有所不同。topax 将该网格保留为多重网格循环中的活动层级，并隐式应用其 Galerkin 算子。Futtop 不会在这一层计算刚度算子或进行平滑，而是将残差直接限制到下一层更粗的网格。因此，Futtop 中的第一层粗化网格仅作为传递算子的中间阶段。在第二个粗化层级，Futtop 与 topax 一样，使用显式 Galerkin 算子

通过将粗单元矩阵显式化得到。该算子会再粗化一次，以形成最粗层系统。与 topax直接求解该系统不同，Futtop 使用 Jacobi预条件共轭梯度法，设置相对容差为 10^{-10}，最大迭代次数为 800。平滑时，Futtop 使用节点对称逐次超松弛，该方法改编自 Wu 等人 [2015], ，而非 topax使用的加权Jacobi方案。需要注意的是，Futtop 使用混合精度：其辅助MGPCG向量以单精度存储，而刚度算子和多重网格层次结构则以双精度计算。 topax 和 pyFANTOM 则始终使用双精度。

与 Futtop 类似，pyFANTOM 会构建单元到节点连接关系和节点到单元查找数组，并利用它们计算细网格刚度作用。与 topax 一样，pyFANTOM 会将第一层粗化网格保留为多重网格循环中的活动部分。但与 topax 不同，pyFANTOM 会显式构建并以 CSR格式存储该层级的 Galerkin 算子。后续所有粗网格上的算子也同样会显式构建并存储。
每次应用多重网格预条件器时，pyFANTOM 都会对最深的两个网格重复计算两次修正量，因此采用的是 W循环，而不是 topax 使用的 V 循环。在最粗网格上，组装系统会传输到 CPU，并通过直接 CHOLMOD 分解求解。在平滑过程中，pyFANTOM 会在每次粗网格校正前后各执行三次加权Jacobi步骤，并自适应调整阻尼参数。
这些不同的表示方式体现于设备内存扩展中。在最小分辨率下，pyFANTOM 使用的设备内存少于 topax；对于 topax，与运行时间和编译程序相关的固定内存占比相对较大。两条曲线在 $96 \times 48 \times 48$ 和 $128 \times 64 \times 64$ 网格之间交叉。在$400 \times 200 \times 200$，pyFANTOM 使用 18.9GiB，而 topax 使用 8.2 GiB，约为其 2.29 倍。差距不断扩大，这与 pyFANTOM 在每个粗网格上（包括第一层粗化网格）都显式存储 Galerkin 算子相符。

因此，这三种实现对第一层粗化网格采用了不同的处理方式。pyFANTOM 将该网格保留为活动层级，但会以稀疏格式显式存储对应的 Galerkin 算子。Futtop 不存储该网格，而仅将其用作中间阶段，把残差传递到更粗层；既不在该网格上进行平滑，也不执行独立的粗网格校正。与 pyFANTOM 类似，topax 也将第一层粗化网格保留为活动层级，但会隐式计算其 Galerkin 算子。在 topax 的基准测试实现中，只有从第二层粗化网格开始才会生成粗单元刚度矩阵，而直接求解器所需的全局稀疏矩阵则仅在最粗网格上组装。测量结果表明，这种策略在保留第一层粗化网格在多重网格循环中的作用的同时，避免了显式存储该网格上的算子所导致的内存增长。

计算设置与测量结果

所有基准测试均使用一块 NVIDIA GH200 GPU，其设备内存约为 96 GiB，并分配了 72 个 Grace CPU 核心。三种实现均在同类型的计算节点上运行，但使用不同的节点，且软件环境各不相同。

topax 测量使用 Python 3.12.14 和 JAX 0.11.1，并采用 CUDA 后端。Futtop 基于已发布的 Futhark 实现构建，并采用 OpenCL 后端。pyFANTOM 的测量使用 0.1.0 版本、Python 3.9.25 和 CuPy 13.6.0。其 CPU 粗网格求解器使用 CHOLMOD，并获得 METIS [Karypis 和 Kumar， 1998] 的支持。各软件环境均根据每种实现的依赖项独立配置。
报告的运行时间仅包含 20 次设计迭代。对于 topax，不计入设置、即时编译和预热时间。对于 Futtop 和 pyFANTOM，运行时间是将报告的 20 次设计迭代时间相加得出，不包括首次迭代前的初始化时间。设计迭代中执行的所有工作均计入其中。每个完成的网格和每种实现仅进行一次测量；因此，报告值是单次观测结果，而非多次运行的平均值。
设备内存使用量每隔 0.2 秒采样一次。报告的数值为采样到的最大用量，且已扣除相应运行开始前立即测得的基线。topax 的内存测量是在禁用 JAX 设备内存预分配的情况下，分别进行十次设计迭代得到的。内存结果仅包括设备内存，不包括主机分配。对于 pyFANTOM 而言，这一区别尤为重要，因为其最粗层矩阵和 CHOLMOD 分解位于主机内存中。即使不计这部分主机内存，pyFANTOM 在最大规模问题上所需的设备内存仍约为其他方法的 2.22 倍。

测量值描述的是各进程预留的设备内存，而非同时存活的数组的解析求和。具体而言，Futhark OpenCL 运行时会保留设备分配以供重复使用。因此，即使算法不再主动使用缓存分配，Futtop 的测量值仍会将进程预留的这部分内存计入其中。虽然这可能使测量值高于对工作集进行解析核算所得的数值，但它反映了执行期间实际占用的设备内存量。

Galerkin 层次结构的多分辨率近似

为减少表示精确 Galerkin 粗网格算子所需的内存，Träff 等人 [2023] 还提出了一种多分辨率近似。该方法不通过 Galerkin 投影构造粗单元矩阵，而是根据每个粗单元所包含的空间变化细网格密度，采用 Nguyen 等人 [2010]提出的多分辨率单元公式计算其刚度矩阵。因此，所得算子保留了细网格材料分布的信息，但并非精确的 Galerkin 投影，因而预条件效果略差 [Träff 等人， 2023]。在所提供的实现中，最深的两个网格存储并在 CPU 上处理。在第二层粗化网格上，物化的粗单元矩阵会被存储并用于计算刚度作用，而无需组装全局稀疏矩阵。在最粗网格上，则会组装全局刚度矩阵，并通过 CHOLMOD 直接分解求解所得系统。这样便将多重网格层次结构所需的大量内存从 GPU 转移到了主机。

对于最大规模的问题，采用混合精度实现的多分辨率预条件子在稳态运行期间使用了 13.68 GiB 的 GPU 内存和 23.76 GiB 的 CPU内存，总内存占用为 37.45 GiB。topax 的相应数值为 31.60 GiB（ GPU）和 5.35 GiB（CPU），总计 36.96 GiB。因此，两种方法的总内存需求几乎相同，尽管内存在 CPU 和 GPU 之间的分配方式大不相同。这些测量结果表明，要实现相当的内存效率，并不一定要近似 Galerkin 层次结构，也可以通过隐式算子应用、延迟物化以及谨慎管理初始化数据来实现。

这些内存测量使用已发布的 OpenMP [Dagum 和 Menon， 1998] 多分辨率公式实现[4]获得。尽管该实现可在 GH200 上运行，但只有两个最小规模的问题完成了全部 20 次设计迭代。对于 $128 \times 64 \times 64$ 问题，MGPCG 迭代次数从设计迭代 8 时的 55 次增加到设计迭代 9 时的 105 次；在设计迭代 10 和 11 时，求解器则达到了其硬编码的 1000 次迭代上限。我们通过记录每次应用多重网格 V 循环后的$\mathbf{r}^{\mathsf{T}}\mathbf{z}$ 来研究收敛性下降的原因。这里，$\mathbf{r}$ 是当前的共轭梯度残差，而$\mathbf{z} = \mathcal{G}(\mathbf{r},0)$是算法 1 (第 8 行和第 12 行）中对应的预条件残差。对于非零残差，正定性要求$\mathbf{r}^{\mathsf{T}}\mathbf{z} > 0$。然而，在设计迭代 8 的第七次 MGPCG 迭代中，我们得到$\mathbf{r}^{\mathsf{T}}\mathbf{z} = -3.42 \times 10^{-4}$。因此，此时多重网格循环定义的预条件器并非正定。基于此，我们在运行时间比较中排除该实现。

迭代求解器失效并未影响内存测量结果。主数组、粗网格算子和 CHOLMOD 分解的维度取决于网格，并且在预条件器性能退化之前就已分配。测量期间，它们在主机和设备上的内存占用均达到稳定值。此外，两个最小规模的问题均正常完成，测得的 GPU 内存消耗与Träff 等人 [2023]报告的数据非常接近。因此，尽管未报告该实现的运行时间，我们仍保留这些内存结果，认为它们能够代表该实现的情况。

计算与内存的权衡

前面的比较采用了一种混合表示：第一个粗化算子以隐式方式应用，而更深层网格上的算子则显式构造。这一选择并非由该方法所限定。隐式应用算子可以避免为每个粗单元存储单独的刚度矩阵，但每次使用算子时都必须重新构建其贡献。显式构造会在设置阶段和整个优化过程中增加存储开销，但能减少每次多重网格循环所需的计算量。
为量化这种权衡，我们比较了上述基准测试中的两种 topax 配置。两者均采用四种网格分辨率，区别仅在于如何处理第二层粗化网格。我们分别将这两种配置记为 L4-C2 和 L4-C3。L4 表示总共有四个层级（包括位于 $\ell = 0$ 的细网格），而 C2/C3 表示 Galerkin 算子首次显式构造的层级。这意味着在 L4-C3 配置中，两个中间层的 Galerkin 算子都以隐式方式应用，直到为直接求解构建最粗层算子时才进行显式构造。前面的比较已使用 L4-C2 配置，其中第一个 Galerkin 算子以隐式方式应用，第二个则显式构造。

图8 展示了两种配置的运行时间、设备峰值内存消耗以及平均刚度 MGPCG 迭代次数。两者的 MGPCG 迭代次数基本相同，因为它们采用相同的四层 Galerkin 层次结构，区别仅在于其中一个算子的求值方式。因此，观察到的运行时间和内存差异可归因于算子表示，而非预条件器质量的变化。

在每个测试分辨率下，将第二个粗化算子实体化都能获得更短的运行时间。对于最大规模的问题，L4-C2 在 99.6 秒内完成 20 次设计迭代，而 L4-C3 则需要 104.4 秒。因此，在该分辨率下延迟实体化会使运行时间增加约 4.9%。然而，隐式表示无需存储空间变化的粗单元矩阵，从而将最大设备内存消耗从 33.5 GiB 降至 27.5 GiB。这相当于减少约 6.0 GiB，即 17.9%。

[4] https://github.com/topopt/TopOpt-in-OpenMP-GPU

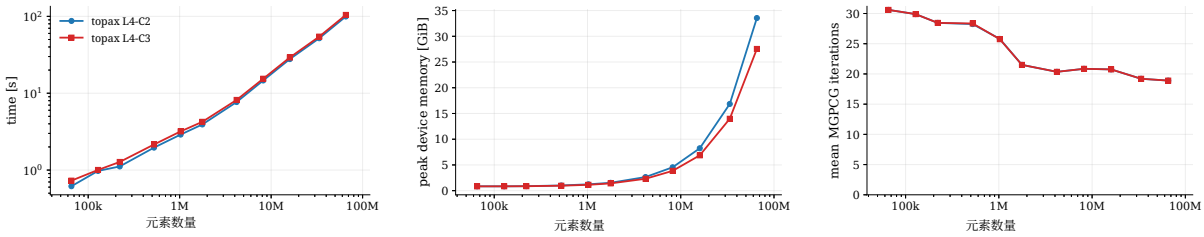

图 8：两种 topax Galerkin 层次结构的比较，适用于 Träff 等人 [2023] 基准测试。两种配置均使用四种网格分辨率；L4-C2 在第二次粗化后生成粗单元矩阵，而 L4-C3 则延迟至第三次也是最后一次粗化后才生成。左：20 次设计迭代的运行时间。中：设备内存峰值消耗。右：20 次设计迭代期间刚度 MGPCG 迭代次数的平均值。

该比较表明，可以根据可用设备内存选择交叉层级。当内存充足时，提前实体化算子更为理想，因为这样可以避免反复计算隐式伽勒金缩并。延迟实体化则提供了一种更低内存的替代方案，同时不会改变多重网格层次结构，也不会降低MGPCG收敛性。对于本文所考虑的最大规模问题，这种方法可以将峰值设备内存消耗减少近五分之一，而运行时间成本不到百分之五。

8.2 面向大规模拓扑优化的节省内存多重网格

前面的实验表明，Galerkin 层次结构占用的内存很大程度上取决于粗算子的表示方式。对于包含数亿个单元的三维问题，这一点尤为重要，因为多重网格层次结构、细网格场以及算子计算过程中使用的临时数组所需的内存总量可能会耗尽可用内存。

因此，我们将 topax 与 TOP3D_XL（由 Wang 等人 [2025] 开发的无矩阵 MATLAB 实现）进行比较。TOP3D_XL 专为限制此类规模下的内存消耗而设计。除了避免组装细网格刚度矩阵外，它还通过将前两次常规二倍粗化替换为一次四倍传递，缩减 Galerkin 层次结构的规模。这一做法移除了二倍粗化层次结构中原本会出现的第一个网格。相比之下，topax 保留该网格作为活动校正层，但隐式应用其 Galerkin 算子。因此，此比较旨在考察：省略一个网格所带来的内存节省，能否通过高效利用内存的算子表示方式来实现。

基准测试问题

基准测试采用Wang等人 [2025]所研究的矩形长方体悬臂梁。设计域的固定物理尺寸为 $1 \times 0.5 \times 0.5$。在 $x = 0$ 面上约束所有位移分量。总大小为 1 的分布载荷沿负 z方向作用于相对的面。载荷和边界条件沿用原始基准测试；其完整定义可参见引用论文以及本研究附带的源代码。

我们考虑包含 $96 \times 48 \times 48$ （R96）的网格、 $192 \times 96 \times 96$ （R192）的网格、 $384 \times 192 \times 192$ （R384）的网格，以及$800 \times 400 \times 400$（R800）的网格。最大网格包含 1.28 亿个单元，约有 386.4 百万个位移自由度。规定的体积分数为 0.12。SIMP 和材料参数与第 8.1 节相同： $p = 3$、 $E = 1$、 $\nu = 0.3$，且最小相对刚度为 10^{-6}。

遵循Sigmund [2001], 通过 Helmholtz 型偏微分方程滤波器实现的 [Lazarov 和 Sigmund的经典灵敏度滤波器公式， 2011], 未滤波的设计密度参与刚度插值，而密度加权的柔顺度灵敏度则在最优性准则更新之前进行正则化。通过使用 $\sqrt{3}$ 个单元宽度的滤波半径来控制网格依赖性。采用最优性准则法更新设计，移动限值为0.2，每个问题均运行 50 次设计迭代。两种实现的刚度求解相对残差容差均为 10^{-3}，Helmholtz 滤波器求解的相对残差容差均为 10^{-4} 。已发布的 TOP3D XL_实现最多允许进行 800 次 MGPCG 迭代，而topax 中的对应上限为 600；已完成的测量中均未达到这两个上限。

CPU 评估

TOP3D_XL 无需组装全局刚度矩阵即可计算细网格刚度作用。其改进的多重网格循环通过省略常规二倍粗化后生成的网格，缩短了层次结构。取而代之的是，传递算子在细网格与每个空间方向上的单元宽度均为其四倍的网格之间直接进行映射。因此，所得粗网格上的每个单元都包含 $4^3 = 64$个细单元，其刚度矩阵则通过相应的四倍插值算子进行 Galerkin 投影获得。后续网格通过常规二倍粗化生成。这与 Futtop 中对第一层

粗化网格的处理方式相似，但 TOP3D_XL 在细网格与四倍粗化网格之间直接构建传递，而不经过中间的二倍粗化网格。

TOP3D_XL 会根据问题规模自适应地选择网格分辨率的数量。对于 R96、R192 和 R384，它使用三个网格层级，因此执行两次粗化。对于 R800，它增加了一个粗网格，并使用四个网格分辨率。得到的 R800 层次结构如下

$$800 \times 400 \times 400 \longrightarrow 200 \times 100 \times 100 \longrightarrow 100 \times 50 \times 50 \longrightarrow 50 \times 25 \times 25.$$

在中间网格上，逐单元刚度矩阵会被具体化；而在最粗网格上，则组装全局刚度矩阵，并使用 CHOLMOD 对其进行分解。

对于 topax，我们考虑 R800 的以下两种二进层级：

$$800 \times 400 \times 400 \longrightarrow 400 \times 200 \times 200 \longrightarrow 200 \times 100 \times 100 \longrightarrow 100 \times 50 \times 50,$$
$$800 \times 400 \times 400 \longrightarrow 400 \times 200 \times 200 \longrightarrow 200 \times 100 \times 100 \longrightarrow 100 \times 50 \times 50 \longrightarrow 50 \times 25 \times 25.$$

我们分别将这些配置记为 L4-C3 和 L5-C4。在这两种情况下，所有中间 Galerkin 算子都采用隐式方式应用，而全局矩阵仅在最粗层级上组装，用于 CHOLMOD 求解。对于 R800，L4-C3 与 TOP3D_XL 包含相同数量的层级，但其最粗网格在每个空间方向上的单元数量是后者的两倍。L5-C4 增加了一个层级，并达到与 TOP3D_XL 相同的最粗网格分辨率。

图9 比较了运行时间、CPU 峰值内存和平均刚度 MGPCG 迭代次数。所有测量均在 CPU 节点上进行，每个节点配备两颗 128 核 AMD EPYC 9755 处理器，运行频率为 2。7GHz，并配有768GB DDR5 内存。已禁用同时多线程。每次基准测试运行分配了 32 个物理 CPU 核心和约 86。4GiB 主机内存，这与 Wang 等人 [2025]所采用的配置相同。各实现均在相同类型的节点上运行，但各项测量是在不同节点上完成的。TOP3D_XL 使用 MATLAB R2025b 执行，OpenMP 和 MKL 的线程数均设为 32。topax 的测量使用 Python 3.12.14 和 JAX 0.11.1 的 CPU 后端完成。

L4-C3 层次结构的速度快于 L5-C4，而 L5-C4 中额外的粗化则降低了内存消耗。在所示的各个分辨率下，相较于 TOP3D_XL，L4-C3 的加速比为 1<style id="2">。8 至 4<style id="4">。4，L5-C4 的加速比为 1<style id="6">。5 至 3<style id="8">。4。对于最大的算例（包含 1.28 亿个单元），TOP3D_XL 完成 50 次设计迭代需要 118 149<style id="11">。0 秒（ 32<style id="13">。8 小时）。相应的

topax L5-C4 的运行时间为 27 058.8 秒（7.8 小时）（L4-C3）和 34 561.8 秒（9.6 小时）（L5-C4），加速比分别为 4.4 和 3.4。两者的差异还体现在刚度求解器的收敛表现上。TOP3D_XL 每次设计迭代平均需要 81.48 次 MGPCG 迭代，而 L4-C3 和 L5-C4 分别需要 16.30 次和 25.12 次。因此，将中间的二倍粗化网格保留为活动校正层，所产生的预条件器明显强于 TOP3D XL 使用的非二进层次结构。_

对于 R800，TOP3D_XL、L4-C3 和 L5-C4 的 CPU 峰值内存分别为 68.20 GiB、68.56 GiB 和 48.88 GiB。因此，尽管 L4-C3 直接分解的最粗层系统包含的单元数量是 TOP3D_XL 的八倍，其内存需求仍基本相同；而 L5-C4 所需的主机内存则少约 28%。在所有较小的已展示分辨率下，两个 topax 配置的内存需求也都低于 TOP3D_XL。R800 时差距缩小，与 TOP3D_XL 自适应层次结构的变化相吻合。对于 R96、R192 和 R384，TOP3D_XL 使用三种网格分辨率，并达到与 L4-C3 相同的最粗分辨率。在 R800 时，它引入第四种网格分辨率，将直接求解转移到一个 $50 \times 25 \times 25$ 网格上，而 L4-C3 则终止于 $100 \times 50 \times 50$。直接分解系统规模因此减小，这在很大程度上解释了为何 TOP3D_XL 的内存消耗接近 L4-C3。L5-C4 同样终止于 $50 \times 25 \times 25$，但保留了所有中间的二倍粗化修正量，同时比 TOP3D_XL 使用少得多的内存。

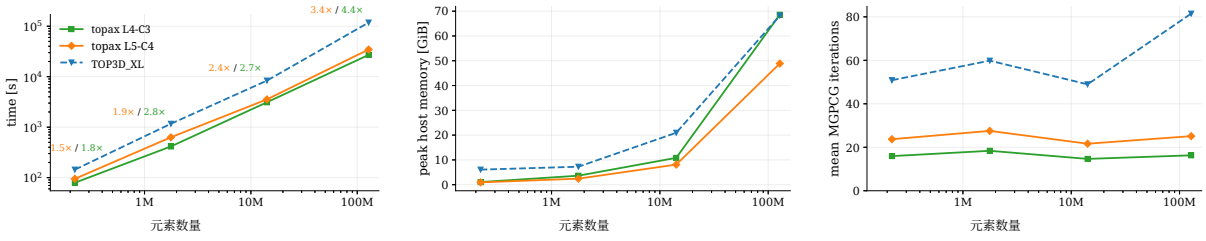

图9：TOP3D_XL 与 L4-C3 和 L5-C4 topax 层次结构的 CPU 对比。左：运行 50 次设计迭代的运行时间；注释分别给出了 L5-C4 和 L4-C3 相对于 TOP3D_XL 的加速比。中：CPU 峰值内存。右：50 次设计迭代期间的平均刚度 MGPCG 迭代次数。

这些结果表明，无需省略第一个二倍粗化网格，也能让精确的 Galerkin 层次结构保持可处理。隐式应用中间 Galerkin 算子，可以在不显式存储它们的情况下，将其贡献保留在多重网格循环中。两种 topax 配置展示了由此产生的权衡：L4-C3 中较大的直接分解粗系统改善了收敛性并缩短了运行时间，而 L5-C4 中增加的隐式层则大幅降低了内存需求。

所报告的运行时间仅包括 50 次设计迭代。对于 topax，初始化、JIT 编译和预热时间均不计入其中。TOP3D_XL 的运行时间通过累加其报告的设计迭代时间得出。峰值内存为整个进程的最大常驻集大小。每种网格和实现均采用一次完整测量结果。

Wang 等人 [2025] 报告称，1.28亿单元问题可以在配备 64GB 随机存取存储器的计算机上执行。我们无法使用已发布实现复现这一上限：相应运行的常驻集大小峰值达到 68.20GiB。我们的测量是在分配了更多内存且未发生交换的情况下进行的。差异可能源于 MATLAB 版本、软件环境，或内存消耗的不同统计方式。

GPU 扩展性

此外，我们使用 topax 的 GPU 后端评估 L4-C3 和 L5-C4 配置。数值公式和多重网格层次结构与 CPU 测量所用的完全相同；不同之处仅在于执行后端和最粗层直接求解器。在 GPU 上，最粗层矩阵使用 cuDSS 而非 CHOLMOD 进行分解。测量使用与第 8.1 节相同的 NVIDIA GH200 系统和设备内存测量流程。运行时间测量涵盖全部 50 次设计迭代；设备内存消耗则在禁用 JAX 内存预分配的情况下，单独测量 10 次设计迭代。

图 10 比较了运行时间、GPU 峰值内存消耗以及平均刚度 MGPCG 迭代次数。在两种配置均支持的分辨率下，L4-C3 的速度始终更快，且所需的 MGPCG 迭代次数更少。不过，在较高分辨率下，L5-C4 将直接求解转移到更小的最粗网格，因此显著降低了内存消耗。

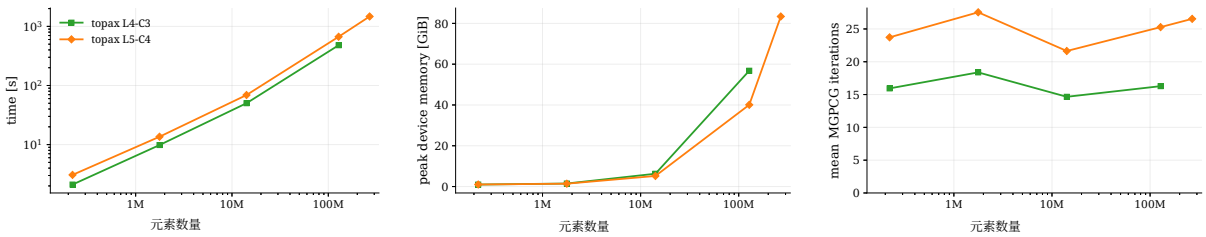

图 10：L4-C3 和 L5-C4 topax 层级结构的 GPU 扩展性。左：50 次设计迭代的运行时间。中：GPU 内存消耗峰值。右：50 次设计迭代期间刚度 MGPCG 迭代次数的平均值。

对于 R800，L4-C3 完成 50 次设计迭代需要 479.8 s，而 L5-C4 需要 667.7 s。相应的 MGPCG 平均迭代次数分别为 16.28 和 25.28。因此，在此分辨率下，L5-C4 慢约 39%，但可将设备内存消耗峰值从 56.79 GiB 降至 40.07 GiB，减少约 29%。L5-C4 较低的内存需求使 R1024 ($1024 \times 512 \times 512$) 问题能够在单个 GH200 上运行；该问题包含 268.4 million 个单元，约有 809 million 个位移自由度。完成 50 次设计迭代需要 1 471.3 s，即约 24.5 min；每次设计迭代平均需要 26.54 次 MGPCG 迭代。其 GPU 内存消耗峰值为 83.39 GiB。其单元数量是 R800 的 2.10 倍，而实测内存消耗是其 2.08 倍，表明内存扩展近似呈线性。

对于不同的问题规模，L4-C3 的平均 MGPCG 迭代次数介于 14。66 和 18。38 之间，L5-C4 则介于 21。64 和 27。54 之间。因此，迭代次数基本不受网格分辨率影响。这一对比揭示了与 CPU 上相同的权衡：L4-C3 提供更强的预条件器，运行时间也更短；而 L5-C4 增加的隐式层可减小直接分解系统的规模，从而在可用设备内存范围内处理大得多的问题。

8.3 大规模三维设计

图11 展示了对第 8.1 节和 8.2 节中介绍的两个悬臂梁问题进行重复计算后得到的优化设计，分辨率分别为 $640 \times 320 \times 320$ 和 $800 \times 400 \times 400$；所用滤波半径与性能基准测试中的不同。在性能基准测试中，滤波半径以单元单位固定。对于此处展示的设计，则调整半径，使其物理尺寸与最粗网格上相应基准测试所用的物理尺寸一致。对于第 8.1 节研究的悬臂梁，分辨率为 1。5 个单元宽度的滤波半径，在分辨率为 $64 \times 32 \times 32$ 时变为 15 个单元宽度，在 $640 \times 320 \times 320$下则为。对于第 8.2 节研究的悬臂梁，Helmholtz 型偏微分方程滤波器的半径为 $\sqrt{3}$ 个单元宽度（分辨率为 $48 \times 24 \times 24$），变为 $17\sqrt{3}$ 个单元宽度（分辨率为 $800 \times 400 \times 400$）。

第一个设 计包含 65.5 百万个单元和 198.1 百万个位移自由度，而第 e
二种情况则包含 包含 1.28 亿个单元和 3.864. 亿个位移自由度。两者均针对.....进行了优化 or

在单台 NVIDIA GH200 上进行了 50 次设计迭代。完整计算包括设置、编译、预热和文件处理，分别耗时约四分钟和五分钟。这些时间短于基准测试中观察到的时间，因为较大的滤波半径会抑制小尺度特征并生成更平滑的密度分布，从而大幅减少 MGPCG 迭代次数。

在 $\rho = 0.5$ 处的连续密度场中提取表面，采用飞行边算法 [Schroeder 等人提出的方法， 2015] 该算法在 VTK 中实现 [Schroeder 等人提出的方法， 2006] 并通过 PyVista 调用 [Sullivan 和 Kaszynski提出的方法 2019]。未进行二值化、连通性滤波或几何平滑。这些示例表明，具有数亿个位移自由度的三维拓扑优化可以在单个 GPU 上完成，并实现近乎实时的周转。

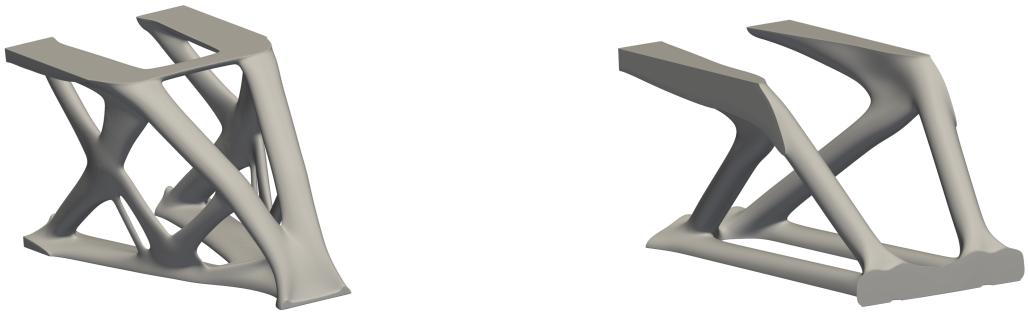

图11：经过 50 次设计迭代后的大规模三维悬臂梁设计。左：研究对象为 Träff 等人 [2023], 研究的悬臂梁，离散为 $640 \times 320 \times 320$ 个单元，规定体积分数为 0.1。右：研究对象为 Wang 等人 [2025], 研究的悬臂梁，离散为 $800 \times 400 \times 400$ 个单元，规定体积分数为0.12。

9 结论

本文提出了适用于结构化笛卡尔网格上的无矩阵有限元分析和 Galerkin 多重网格的内存受限张量公式。细网格刚度作用通过卷积原语表示，并在重叠的空间切片中进行计算，从而在不改变底层有限元算子的情况下限制中间内存。对于粗网格，Galerkin 算子通过细网格刚度插值系数和预计算局部算子表示，使其作用能够隐式计算，而无需为每个粗单元物化单独的刚度矩阵。将隐式计算与延迟物化相结合，在保持精确 Galerkin 层次结构的同时，实现了计算成本与内存消耗之间可配置的权衡。这些方法已在 topax 中实现。topax 是一个基于 JAX 的模块化库，用于基于密度的拓扑优化，结合了硬件加速的张量运算、即时编译和自动微分。

数值实验表明，这些表示方法能够直接带来运行时间和内存消耗方面的显著改进。在 GPU 对比中，topax 的速度是最快竞争实现的 6.1 至 13.6 倍，并且在规模最大的算例中所需的设备内存不到其一半。在 CPU 上，topax 的速度最高可达到 TOP3D_XL 的 4.4 倍，同时还降低了内存需求。规模最大的实验包含超过 8.09 亿个位移自由度，并在单个 GPU 上约 24.5 分钟内完成 50 次设计迭代，设备内存峰值消耗为 83.4 GiB。

本实现聚焦于均匀笛卡尔网格上的线弹性问题，但由此得到的求解器为更广泛的可微仿真和优化工作流奠定了基础。一个直接的方向是将多重网格实现开放为可复用的无矩阵线性求解器或预条件器，用于其他 PDE 约束问题，包括非线性求解器中产生的线性系统。最大规模的实验也已接近单块 96GiB GPU 的容量。在这一分辨率下，一个包含所有位移自由度的双精度向量约占用 6GiB。MGPCG 除了设计、滤波、灵敏度和优化器场之外，还需要同时使用多个这样的向量。这些持久化的全分辨率数组带来了无法通过进一步减少临时存储来消除的内存需求，因此更大规模的问题将需要分布式执行。采用分布式代数多重网格的多 GPU 拓扑优化已得到验证 [Herrero-Pérez 和 Castejón， 2021], 而基于 CPU 的分布式内存实现已达到十亿体素规模 [Aage 等人，2017，Baandrup 等人， 2020]。由于所提出的算子具有局部性、采用空间切片，并以支持多设备数组分区的 JAX 表达，因此扩展这套方法主要是一项工程任务，而无需重新表述方法。为促进结果复现和后续开发，我们随本文一并发布完整实现和基准测试脚本。

致谢

Pedro Zuidberg Dos Martires 获得了由克努特与爱丽丝·瓦伦堡基金会资助的瓦伦堡人工智能、自主系统与软件计划（WASP）以及瑞典研究理事会（Vetenskapsrådet）启动基金的支持（基金编号 2025-06142）。计算资源由瑞典国家学术超级计算基础设施（NAISS）提供，该基础设施由瑞典研究理事会资助。

生成式人工智能使用声明

在准备这项工作期间，作者使用 OpenAI Codex 协助起草和修订稿件文本，并提升其清晰度与可读性。技术内容、分析和结论均由作者确定。作者审阅、核实并编辑了所有人工智能生成的建议，并对已发表文章的内容承担全部责任。OpenAI Codex 还作为编程助手，用于重构、代码审查、调试，以及准备基准测试和评估脚本。作者检查了所有拟议的更改，并通过自动化测试、数值回归测试和基准测试复现对其进行了验证。

参考文献

Niels Aage、Erik Andreassen 和 Boyan S Lazarov。使用 PETSc 进行拓扑优化：一个易于使用、完全并行的开源拓扑优化框架。《结构与多学科优化》，51(3)：565–572，2015。

Niels Aage、Erik Andreassen、Boyan S Lazarov 和 Ole Sigmund。用于结构设计的千兆体素计算形态发生。《自然》，550(7674)：84–86，2017。

Oded Amir、Niels Aage 和 Boyan S Lazarov。关于用于高效拓扑优化的多重网格共轭梯度法。《结构与多学科优化》，49(5):815–829，2014。

Mads Jacob Baandrup、Ole Sigmund、Henrik Polk 和 Niels Aage。利用计算形态发生，缩小迈向超长悬索桥的差距。《自然·通讯》，11(1):2735，2020。doi: 10.1038/s41467-020-16599-6。

Satish Balay、William D Gropp、Lois Curfman McInnes 和 Barry F Smith。高效管理面向对象数值软件库中的并行性。载于《科学计算现代软件工具》，第 163–202 页。Springer，1997。

Igor A. Baratta、Joseph P. Dean、Jørgen S. Dokken、Michal Habera、Jack S. Hale、Chris N. Richardson、Marie E. Rognes、Matthew W. Scroggs、Nathan Sime 和 Garth N. Wells。DOLFINx：新一代 FEniCS 问题求解环境。预印本，2023。

Klaus-Jürgen Bathe。《有限元方法》。K. J. Bathe，马萨诸塞州沃特敦，第 2 版，2014。ISBN 978-0- 9790049-5-7。

Martin P. Bendsøe 和 Ole Sigmund。拓扑优化中的材料插值方案。《应用力学档案》，69(9):635–654，1999。doi: 10.1007/s004190050248。

Martin P. Bendsøe 和 Ole Sigmund。《拓扑优化：理论、方法与应用》。Springer，柏林、海德堡，2004。doi: 10.1007/978-3-662-05086-6。

Blaise Bourdin。拓扑优化中的滤波器。《国际工程数值方法杂志》，50 (9):2143–2158，2001。doi: 10.1002/nme.116。

James Bradbury、Roy Frostig、Peter Hawkins、Matthew James Johnson、Yash Katariya、Chris Leary、Dougal Maclaurin、George Necula、Adam Paszke、Jake VanderPlas、Skye Wanderman-Milne 和 Qiao Zhang。JAX：Python 的可组合变换+NumPy 程序，2018。网址 http://github.com/jax-ml/jax。

Tyler E. Bruns 和 Daniel A. Tortorelli。非线性弹性结构与柔顺机构的拓扑优化。《计算机在应用力学与工程中的方法》，190(26–27):3443–3459，2001年。doi：10.1016/S0045-7825(00)00278-4。

Cris Cecka、Adrian J Lew 和 Eric Darve。图形处理器上的有限元方法组装。《国际工程数值方法杂志》，85(5):640–669，2011年。

Aaditya Chandrasekhar、Saketh Sridhara 和 Krishnan Suresh。Auto：用于拓扑优化中自动微分的框架。《结构与多学科优化》，64(6):4355–4365，2021年。

Yanqing Chen、Timothy A Davis、William W Hager 和 Sivasankaran Rajamanickam。算法 887：CHOLMOD，超节点稀疏 Cholesky 分解及更新/降阶。*ACM* 数学软件汇刊 *(TOMS)*，35(3):1–14，2008年。Leonardo Dagum 和 Ramesh Menon。OpenMP：共享内存编程的行业标准 API。*IEEE* 计算科学与工程，5(1):46–55，1998年。Christian Dick、Joachim Georgii 和 Rüdiger Westermann。使用 CUDA 进行弹性模拟的实时多重网格有限六面体方法。模拟建模实践与理论，19(2):801–816，2011年。Juncai He 和 Jinchao Xu。MGNet：多重网格与卷积神经网络的统一框架。中国科学：数学，62(7):1331–1354，2019年。Troels Henriksen、Niels GW Serup、Martin Elsman、Fritz Henglein 和 Cosmin E Oancea。Futhark：采用嵌套并行和原地数组更新的纯函数式 GPU 编程。载于*ACM SIGPLAN* 编程语言设计与实现大会，第 556–571 页，2017年。David Herrero-Pérez 和 Pedro J Martínez Castejón。大规模基于密度的拓扑优化的多 GPU 加速。工程软件进展，157:103006，2021年。Cupjin Huang、Fang Zhang、Michael Newman、Xiaotong Ni、Dawei Ding、Junjie Cai、Xun Gao、Tenghui Wang、Feng Wu、Gengyan Zhang 等。用于模拟量子计算的张量网络收缩高效并行化。自然·计算科学，1(9):578–587，2021年。Yingqi Jia、Chao Wang 和 Xiaojia Shelly Zhang。FEniTop：一个简单的 FEniCSx 实现，支持并行计算的二维和三维拓扑优化。《结构与多学科优化》，67(140)，2024年。Jim E Jones 和 Panayot S Vassilevski。基于单元聚合的 AMGe。*SIAM* 科学计算杂志，23(1):109–133，2001年。George Karypis 和 Vipin Kumar。一种快速且高质量的非规则图划分多层方案。*SIAM* 科学计算杂志，20(1):359–392，1998年。Martin Kronbichler 和 Katharina Kormann。用于并行基于单元的有限元算子应用的通用接口。计算机与流体，63:135–147，2012年。doi: 10.1016/j.compfluid.2012.04.012。Boyan Stefanov Lazarov 和 Ole Sigmund。基于亥姆霍兹型微分方程的拓扑优化滤波器。《国际工程数值方法杂志》，86(6):765–781，2011年。Jesús Martínez-Frutos 和 David Herrero-Pérez。固定网格有限元分析的高效无矩阵 GPU 实现。分析与设计中的有限元，104:61–71，2015年。Jesús Martínez-Frutos、Pedro J Martínez-Castejón 和 David Herrero-Pérez。利用多层次粒度的 GPU 计算实现高效拓扑优化。工程软件进展，106:47–62，2017年。Andreas Neofytou、Thiago Rios、Mariusz Bujny、Stefan Menzel 和 H Alicia Kim。基于稀疏自动微分的水平集拓扑优化。《结构与多学科优化》，67(10):178，2024年。Tam H. Nguyen、Glaucio H. Paulino、Junho Song 和 Chau H. Le。多分辨率拓扑优化（MTOP）的计算范式。《结构与多学科优化》，41(4):525–539，2010年。doi: 10.1007/s00158-009-0443-8。Amin Heyrani Nobari 和 Faez Ahmed。PGD-TO：使用投影梯度下降实现多约束拓扑优化的一种可扩展替代方案，可替代 MMA。《结构与多学科优化》，69(9):209，2026年。doi: 10.1007/s00158-026-04376-w。Sebastian A Nørgaard、Max Sagebaum、Nicolas R Gauger 和 Boyan S Lazarov。自动微分在拓扑优化中的应用。《结构与多学科优化》，56(5):1135–1146，2017年。Ryosuke Okuta、Yuya Unno、Daisuke Nishino、Shohei Hido 和 Crissman Loomis。CuPy：兼容 NumPy 的 NVIDIA GPU 计算库。载于机器学习系统研讨会论文集，2017年。Rahul Kumar Padhy、Krishnan Suresh 和 Aaditya Chandrasekhar。ToFlux：用于多物理场流体问题的可微拓扑优化框架。《结构与多学科优化》，69(3):56，2026年。Adam Paszke、Sam Gross、Francisco Massa、Adam Lerer、James Bradbury、Gregory Chanan、Trevor Killeen、Ze-Ming Lin、Natalia Gimelshein、Luca Antiga、Alban Desmaison、Andreas Köpf、Edward Yang、Zachary DeVito、Martin Raison、Alykhan Tejani、Sasank Chilamkurthy、Benoit Steiner、Lu Fang、Jun-Jie Bai 和 Soumith Chintala。PyTorch：一种命令式风格的高性能深度学习库。载于神经信息处理系统进展，2019年。

Shashi Kant Ratnakar、Subhajit Sanfui 和 Deepak Sharma。采用定制节点连接关系存储的无矩阵共轭梯度有限元求解器实现基于 GPU 的拓扑优化。载于《跨学科工程进展：FLAME 2020 会议精选论文集》，第 87–97 页。Springer，2021年。Alex Rogozhnikov。Einops：采用爱因斯坦式记号进行清晰可靠的张量操作。载于国际学习表征会议，2022年。Suryanarayanan Manoj Sanu、Miguel A Bessa 和 Alejandro M Aragón。在现代机器学习框架中利用自动微分进行（神经）拓扑优化。《结构与多学科优化》，69(5):126，2026年。Stephan Schmidt 和 Volker Schulz。一个为显卡编写的 2589 行拓扑优化代码。《科学计算与可视化》，14(6):249–256，2011年。Will Schroeder、Ken Martin 和 Bill Lorensen。《可视化工具包》（第 4 版）。Kitware，2006年。ISBN 978-1-930934-19-1。William Schroeder、Rob Maynard 和 Berk Geveci。飞行边算法：一种高性能、可扩展的等值面算法。载于 2015 年 IEEE 第五届大型数据分析与可视化研讨会，第 33–40 页。IEEE，2015年。Ole Sigmund。一个用 MATLAB 编写的 99 行拓扑优化代码。《结构与多学科优化》，21(2):120–127，2001年。Ole Sigmund。用于拓扑优化的基于形态学的黑白滤波器。《结构与多学科优化》，33(4):401–424，2007年。Ole Sigmund 和 Kurt Maute。拓扑优化方法。《结构与多学科优化》，48(6):1031–1055，2013年。Bane Sullivan 和 Alexander Kaszynski。PyVista：通过精简的界面使用可视化工具包（VTK）进行三维绘图和网格分析。《开源软件杂志》，4(37):1450，2019年。Krister Svanberg。移动渐近线法——一种新的结构优化方法。《国际工程数值方法杂志》，24(2):359–373，1987年。Erik A Träff、Anton Rydahl、Sven Karlsson、Ole Sigmund 和 Niels Aage。简单高效的 GPU 加速拓扑优化：代码与应用。《计算机在应用力学与工程中的方法》，410:116043，2023年。Ulrich Trottenberg、Cornelius W. Oosterlee 和 Anton Schüller。《多重网格》。Academic Press，圣迭戈，加利福尼亚州，2001年。ISBN 978-0-12-701070-0。Fengwen Wang、Boyan Stefanov Lazarov 和 Ole Sigmund。关于拓扑优化中的投影方法、收敛性和鲁棒公式。《结构与多学科优化》，43(6):767–784，2011年。doi: 10.1007/s00158-010-0602-y。Junpeng Wang、Niels Aage、Jun Wu、Ole Sigmund 和 Rüdiger Westermann。利用无矩阵 MATLAB 代码进行高效的大规模三维拓扑优化。《结构与多学科优化》，68(9):174，2025年。Gaoyuan Wu。一种基于自动微分、伴随方法和加速线性代数的结构形状优化框架。《结构与多学科优化》，66:151，2023年。doi: 10.1007/s00158-023-03601-0。URL https://doi.org/10.1007/s00158-023-03601-0。Jun Wu、Christian Dick 和 Rüdiger Westermann。一个用于高分辨率拓扑优化的系统。《IEEE 可视化与计算机图形学汇刊》，22(3):1195–1208，2015年。Tianju Xue、Shuheng Liao、Zhengtao Gan、Chanwook Park、Xiaoyu Xie、Wing Kam Liu 和 Jian Cao。JAX-FEM：一种可微分 GPU 加速三维有限元求解器，用于自动逆向设计和机制数据科学。《计算物理通讯》，291:108802，2023年。Shaoliang Yang、Jun Wang 和 Yunsheng Wang。一种用于单 GPU 三维 SIMP 线性系统的无矩阵 Galerkin 多重网格求解器和失效模式筛选。arXiv 预印本 arXiv:2604.26441，2026a。Shaoliang Yang、Jun Wang 和 Yunsheng Wang。采用融合 gather-gemm-scatter 核的无矩阵三维 SIMP 拓扑优化。arXiv 预印本 arXiv:2604.18020，2026b。Alexandros Nikolaos Ziogas、Grzegorz Kwasniewski、Tal Ben-Nun、Timo Schneider 和 Torsten Hoefler。Deinsum：实用上 I/O 最优的多线性代数。载于 SC22：高性能计算、网络、存储与分析国际会议，第 1–15 页。IEEE，2022年。