遥感数据驱动的地区经济发展评估与政策决策支持模型

[1] 龚舒凯　　[1] 郑楚睿　　[1] 谢航

[1] 中国人民大学

1 背景与项目动机

随着城市化和现代化的进程加快，准确评估一个地区的经济发展水平对于当地资源配置、政策制定和区域发展规划具有重要意义。传统的经济发展水平评估方法主要依赖于统计数据，如GDP、人均收入、就业率等，但在一些偏远地区和不发达地区，这些方法存在数据滞后性和数据不可获得性。

覆盖范围广、覆盖时间长的卫星遥感图像为地区经济发展水平的评估提供了一项重要的数据来源。近年来，随着深度学习的发展，产生了诸多使用卷积神经网络，通过卫星遥感图像评估地区发展水平的工作，如 [1] [2]。其中，[1] 提出了一种只基于大量卫星遥感图像和少量人工标注数据的方法，对像朝鲜、柬埔寨这样经济发展落后且披露数据较少的国家与地区的经济发展水平评估。受到这项工作的启发，我们希望**将这篇论文的方法迁移到中国某一地区加以应用**。

我们选择的地区是中国四川省。四川地域广袤、地形丰富，但在川西地区，受限于地理位置的偏远，难以获得大量的、直接统计的经济发展数据。因此，使用一小部分高质量标注的数据实现对四川全域的经济发展水平评估是非常具有意义的。

具体实现如图1所示。首先，我们需要通过卫星图像数据的爬虫、处理与少量标注，构建好四川卫星图像数据集。接着，我们与 [1] 的方法保持一致，训练了一个三阶段模型：第一阶段对卫星图像进行分类与聚类，第二阶段对第一阶段产生的卫星图像聚类的发展水平高低进行人工排序，得到一个偏序图 (POG, Partial Order Graph)，第三阶段根据 POG 训练经济发展水平打分模型。

在整个实现过程中，我们大量使用并行计算的方法加速整个工作流程，包括：(1) 使用 Python 多进程/多线程下载卫星图像数据，(2) 使用 MPI 并行加速图像的分割，(3) 异步并发的向大模型 API 发送请求进行数据标注，(4) 引入矩阵运算优化聚合 POG 的计算，(5) 使用 GPU 加速深度学习模型的训练，(6) 使用 Pycuda 编写核函数并行加速图像的拼接。完整的代码实现见https://github.com/GONGSHUKAI/dev_measure。

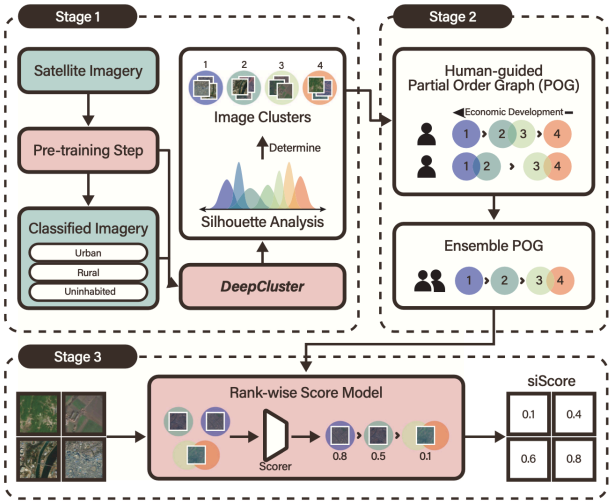

图 1: 三阶段模型训练流程

2 数据集的获取与构建

2.1 Sentinel-2 L2A 卫星图像的数据爬取：基于多种并行计算方法

2.1.1 数据来源与规模

在没有现成的开源数据集的情况下，**我们的数据采集工作极具挑战性，需要自行爬取、清洗、处理，构建起自己的卫星图像数据集**。经过对卫星图像数据平台的大量调研和尝试，本项目最终决定使用 Microsoft Planetary Computer 卫星图像数据库 [4]。该数据库包含了 Sentinel-2 L2A 级别的高分辨率卫星图像，非常适合我们基于高清卫星图像评估经济发展水平的项目方法。

如图2所示，得益于 Microsoft Planetary Computer 提供的细致区块划分索引系统，本项目得以实现对四川省全境的卫星图像数据进行精确的定位与索引。初始阶段，我们确定了覆盖四川省全部区域的 60 个关键区块，每个区块的边长均为 110 公里。接着，通过利用数据库所提供的 API 接口，我们对这 60 个区块在 2020 年至 2024 年间的卫星图像进行了详尽的检索。在图像检索过程中，我们观察到大量的卫星图像存在质量问题，例如包含较大面积的黑色无效区域，或是由于云层遮挡严重，使得地面特征难以辨识。这些因素均限制了卫星图像的直接应用。

针对这一问题，我们开发了专门的爬虫脚本，并设定了严格的筛选标准，以确保对于每个区块，仅下载完整性最高、云量最少的 GeoTiff 格式高清卫星图像，从而保证了数据集的质量与可用性，最终获得 60 张高清的卫星图像。在此基础上，我们对每个区块的图像进行了进一步的细分，每个区块被划分为 $32 \times 32 = 1024$ 张小区块图像，从而构建了一个包含 $60 \times 1024 = 61440$ 张图像的大型高质量四川省卫星图像数据集，其总大小约为 12G（数据集链接：https://pan.baidu.com/s/1xinCGtFcTcw1lm2Y95Tchw?pwd=mqwj，提取码：mqwj）。后续的图像聚类、发展水平评分环节中还会产生一些额外的图像数据 (如打分地图的生成)，使得我们的数据操作的总规模达到了 55G。

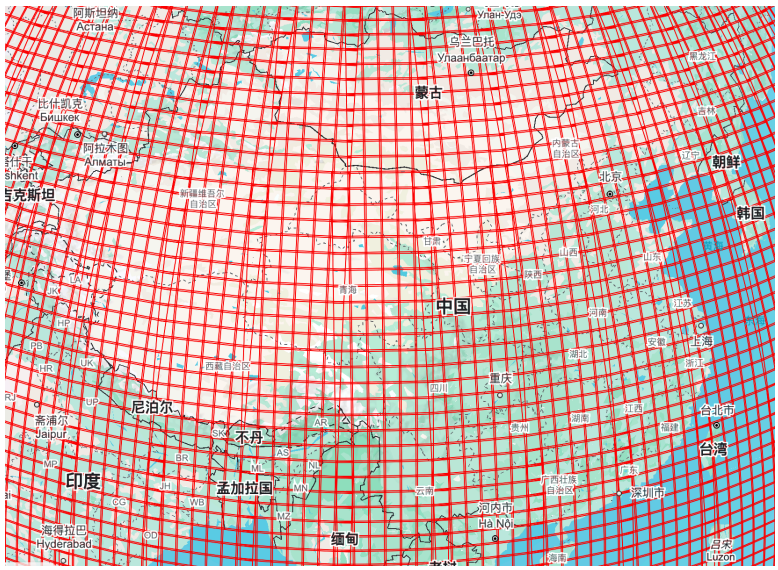

图 2: Sentinel-2 卫星的地理区块区域划分：每个区块可由唯一的编号索引

2.1.2 并行计算加速卫星图像数据爬取

在成功通过 API 检索到 Planetary Computer 数据库中符合要求的图像数据后，我们采用了**并行计算技术加速图像数据的下载过程**。为了测试并行下载脚本的效率，我们以 51RUQ 区块的 41 张图像数据下载为例，比较了在 Python 环境中使用串行、多线程和多进程三种不同方法的下载效率。如表1所示，多线程并行计算在数据下载任务中表现最为出色，其耗时相较于串行方法平均减少了 17.67 秒。多线程与多进程方法相较于串行方法均显著提升了数据下载的速度，这充分证明了并行计算在高效数据爬取任务中的重要性。并行加速下载数据的代码见附录A使用多线程、多进程、串行爬取卫星图像数据。

并行加速方法	均值 (秒)	标准差 (秒)
Python Threading	22.11	4.84
Python Multiprocessing	27.21	3.51
Python Serial	39.78	5.94

表 1: 数据爬取下 Python 并行计算方法效率对比

2.2 区块 (Granule) 图像的并行分割

如图3所示，为了捕捉卫星图像区块的局部信息，从而实现细粒度的经济发展水平打分，我们将每个区块的高清 GeoTiff 图像 (分辨率 10976×10976) 分割为 $32 \times 32 = 1024$ 个小区块图像 (分辨率 343×343)，每个小区块代表现实世界中边长为 3.4km 的正方形区域。

为了加速对大量图片分割的效率，我们**使用 MPI 对图像分割代码进行并行加速**，并与 Python 多进程进行运行速度的提升比较。MPI 的代码实现见附录B使用 MPI 加速的图像分割。具体而言，主进程 (rank 0) 将输入图像路径收集并广播给所有进程，然后根据进程总数 (world_size) 将图像处理任务均匀划分给每个进程，每个进程并行执行分配的图像分块任务，最后使用 MPI_Reduce 将所有进程的处理时间汇总到主进程，主进程输出总处理时间和每个进程的处理时间。

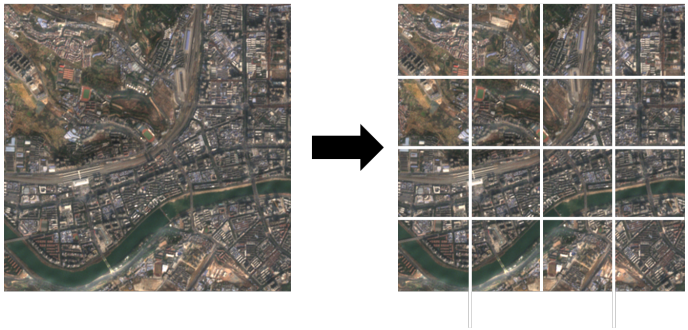

图 3: 卫星图像分割图例

我们以同时处理 41 张区块 GeoTiff 图像的分割为例对比 Python 多线程与 MPI 对图像分割的加速效果，其中设置 MPI 使用的进程数为 4。代码运行总时长比较如表2所示，MPI 处理速度快于多线程，**我们认为图像分割更适合使用多进程类型方法并行计算，而之前数据爬取则更适合使用多线程**。

并行加速方法	用时 (秒)
Python Threading	9.30
MPI (np = 4)	6.26

表 2: 图像分割下并行计算方法效率对比

2.3 图像的四分类数据标注：基于异步并发 API 请求

由于在模型训练的阶段一中，我们需要通过训练一个图像分类器将四川省的卫星图像先统一分为城市、农村、山地和高原这四类，因此需要对少部分数据进行标注用于该分类器的训练。我们从分割为小块后的四川卫星图像数据中人工抽取分布在四川各地的 88 张城市图像、408 张农村图像、172 张山区图像、232 张高原图像，并定义每张图片的标注为一个向量 $\mathbf{p} \in \mathbb{R}^4 = (p_{\text{Urban}}, p_{\text{Rural}}, p_{\text{Mountain}}, p_{\text{Highland}})$，表示这张图片属于城市、农村、山地、高原的概率。显然，人工对 900 张图片进行 4 分类标注是工作量巨大的，**为了降低时间成本，提升标注数据，我们采用视觉大模型进行标注**，具体流程如下：

1. 撰写合适的 Prompt 严格规范大模型的输出。(见附录C用于卫星图像四分类标注的 Prompt)
2. 通过 Qwen2.5-72B-Instruct 的 API 将选取的 900 张图片**异步并发**发送给视觉大模型，要求大模型对每张图片生成三次图片标注。
3. 从大模型的三次输出中提取三个图片标注 $\mathbf{p}_1, \mathbf{p}_2, \mathbf{p}_3$ 并求平均，得到每张图片的标注 $\bar{\mathbf{p}}$。

训练好的商用视觉大模型对图像的属性 (城市、农村、山地、高原) 有着充分的认知，可以近似为人类标注者，从而大大减少我们的标注工作量。在对大模型的基础标注进行审核和修正后，我们得到了一个高质量的有监督数据集 (链接: https://pan.baidu.com/s/1MOL7K-VhKYxiShNGNCsVVQ?pwd=kxx2，提取码: kxx2)。

3 三阶段模型的构建

3.1 阶段一：图像分类与聚类

这一阶段的目标是将四川卫星图像数据集进行聚类，使得每一个聚类内部的图像经济发展水平相似，而聚类与聚类之间的图像经济发展水平有高低之分。参照 [1] 中的设置，直接在完整的数据集上进行聚类容易导致聚类算法收敛到平凡解，因此，可以先将卫星图像按照地形进行大致的分类，再在分类内部做聚类。

基于对四川省地形结构的考察，我们决定将爬取到并分割好的卫星图像数据分为城市 (U)、农村 (R)、山地 (M) 和高原 (H) 四类。我们在此前构建的有监督卫星图像数据集上使用 RTX 3090 训练了一个 ResNet-18 图像分类器，在验证集上达到 95% 的准确率。如图4所示，接下来我们同样使用 CUDA 训练一个 DeepCluster 聚类模型 [3] 对四个类别内部进行聚类：

- **训练阶段：** 通过预训练的图像编码器 M 将图像通过 DeepCluster 的 ResNet18 编码器以获取图像嵌入。使用 K-means 算法对图像嵌入进行聚类，并将伪标签确立为集群分配。使用伪标签更新图像编码器 M 的参数。
- **推理阶段：** 训练完模型后，对于每个类别内的图像，获取图像嵌入并将图像聚集成 K 个簇 ($K \in [3, 20]$)，检查轮廓分数 (Silhouette score) 以选择最优的 K 值。

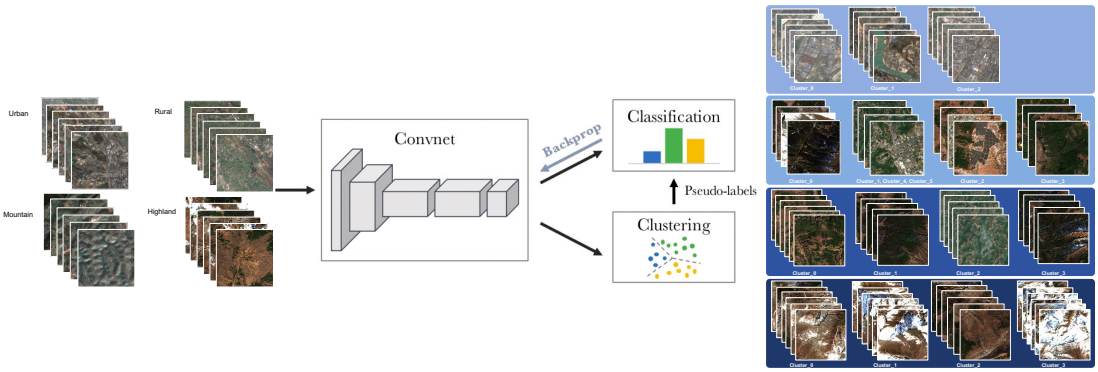

图 4: 图像四分类与类别内部的聚类

城市、农村、山地的轮廓分数如图5所示，城市被聚为 $|\mathcal{C}_U| = 3$ 类，农村被聚为 $|\mathcal{C}_R| = 11$ 类，山地被聚为 $|\mathcal{C}_M| = 8$ 类。我们默认高原地区所有图像为同一类，即 $|\mathcal{C}_H| = 1$。

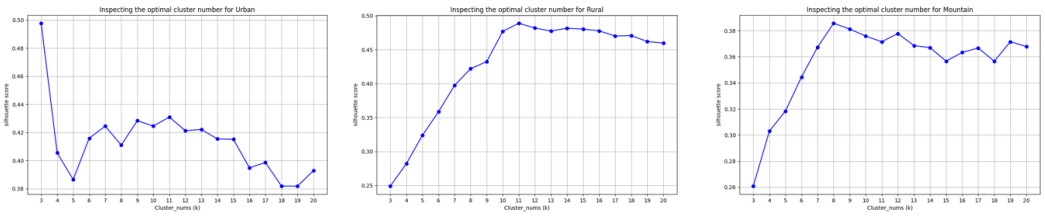

图 5: Silhoutte score of each category for $K \in [3, 20]$.

3.2 阶段二：POG 的生成与加聚

基于阶段一生成的 23 个分别对应不同的地理和经济特征的聚类，阶段二将通过视觉语言模型 (VLLM，包括通义千问 Qwen2-VL-72B-Instruct, DeepSeek-VL2) 辅助生成并聚合 POG，为下一阶段的经济发展水平打分模型提供基础。

3.2.1 第一部分：POG 生成

为了生成 POG，我们从每个聚类中随机采样 3 张图像，总计 3×23 张图像，并编写了 4 种不同的文本提示词，分别从卫星遥感图像专家、四川本地人、发展经济学教授和美国经济学大学生的角度 (完整 prompt 见附录D用于 POG 生成的 Prompt)，引导 VLLM 为每张图像的经济发展水平从 0 到 100 进行评分。将图像与提示词通过 API **异步并发**至 VLLM，为每张图像生成 3 次评分。

接着，我们对每张图像的三个评分取平均值，再对每个聚类的评分取平均值，从而得到簇的最终得分。根据得分进行从大到小的排序。如果分数差距小于等于 10 分，认为两个簇在排名上是相等的。即：

$$\text{Rank A} = \text{Rank B, when} \quad |\text{ScoreA} - \text{ScoreB}| \leq 10$$

$$\text{Rank A} > \text{Rank B, when} \quad \text{ScoreA} - \text{ScoreB} > 10$$

代码实现见附录E使用 API 进行异步并发请求标注数据。我们还对 API 的异步并发效率进行了测试。如表3所示，我们比较了一次运行生成 1 个、3 个、5 个和 10 个 POG 的平均用时，发现在有 API 访问频率的限制下，一次运行生成 3-5 个 POG 的效率更高。

轮数	总用时 (秒)	平均每轮用时 (秒)
1	9.98	9.98
3	27.84	**9.28**
5	57.24	11.45
10	353.81	35.38

表 3: POG 生成平均用时情况

3.2.2 第二部分：POG 加聚

我们按照 [1] 中的设置，用如下算法对生成的多个 POG 进行加聚：这种加权迭代方法相比简单平均更具鲁棒性，它能够减少离群值对最终结果的影响，同时保留数据的多样性。

1. **离散到连续的转换**：首先将多轮 POG 生成的离散排名结果转换为连续排名向量，以更好地反映簇间的细微差异。

$$r_i = \text{rank}(c_i) + \frac{1}{2} \sum_{c \in C \setminus \{c_i\}} 1(\text{if rank}(c) = \text{rank}(c_i), c_i \in C)$$

2. **生成初始聚合向量**：接着通过多轮生成的排名向量计算初始聚合向量 R^*，这个聚合向量

是后续迭代优化的起点。

$$R^* = \frac{1}{10}\sum_{i=1}^{10} R_i$$

3. **加权迭代优化**：在每轮迭代中，我们通过计算单轮向量与当前聚合向量的欧氏距离，得到一个权重因子：

$$\alpha_m = 1 - \exp\left(-\frac{\|R_m - R^*\|^2}{\sigma^2}\right), \quad m = 1, 2, \ldots, M$$

然后对权重进行归一化处理，得到归一化权重 $w_m = \frac{\alpha_i}{\sum_{i=1}^{M}\alpha_i}$。这一步确保了与当前聚合向量距离较近的单轮向量对优化贡献更大。根据权重计算新的聚合向量 $R^* = \sum_{m=1}^{M} w_m R_m$。优化过程持续进行，直至聚合向量的变化量小于一个非常小的收敛阈值。

$$\epsilon = \|R^* - \sum_{i=1}^{M} w_m R_m\|^2 < 10^{-6}$$

4. **一维 K-means 聚类**：我们获得的 R^* 是一个连续排名向量，包含所有聚类的排名。例如：

$$R^* = (3.8, 3.35, 3.25, 2.45, 3.05, 5.1, 2.75, 3.75, 5.25, 4.75, 5.75,$$
$$10.7, 10.2, 10.1, 7.6, 8.5, 7.7, 9.65, 10.15, 10.8, 8.7, 9.45)$$

这个向量中的每个值代表一个聚类的经济发展水平排名。值越小表示该聚类排名靠前，经济发展水平越高。最后我们采取一维 K-means 聚类方法，将向量内的数据点分为 K 个聚类类别。例如对上述 R^* 进行一维 K-means 聚类后的结果如下：

3 = 4 = 6 > 0 = 1 = 2 = 7 > 5 = 8 = 9 = 10 > 14 = 16 > 15 = 20 = 21 > 11 = 12 = 13 = 17 = 18 = 19

加聚过程中的一大挑战是如何处理大量数据以提高计算效率，因此我们**引入矩阵运算，将权重计算、聚合向量更新等操作并行化**，随着数据规模增大，计算效率获得有效提高。如表4所示，在要加聚的 POG 数量极大时，矩阵运算耗时仅为 0.146 秒，而串行循环聚合耗时 3.784 秒；在加聚 1 万个聚类时，矩阵运算的耗时明显小于串行循环聚合的耗时。

POG 数量	簇数	簇大小	矩阵加聚时间（秒）	串行循环加聚时间（秒）
10	22	7	0.038	**0.005**
10000	22	7	**0.048**	0.384
100000	22	7	**0.146**	3.784
10	22	7	0.038	**0.005**
10	1000	7	0.084	**0.069**
10	10000	7	**0.184**	1.914
10	10000	7	**0.084**	0.190
10	10000	1000	**0.188**	0.324
10	10000	5000	**0.348**	0.598

表 4: POG 加聚在不同数据规模下的效率对比。

通过上述算法，我们最终将四川省的 23 个聚类组织成如下的 POG。这将为经济发展水平打

分模型的训练提供清晰的数据输入。

$$\underbrace{0>1>2}_{\text{Urban}}>\underbrace{4=3=6>9=12>17=11>13=10}_{\text{Rural}}$$
$$\underbrace{>7=20=19=21=5=14=15=16=18=8=22}_{\text{Mountain and Highland}}$$

3.3 阶段三：发展水平打分模型训练

阶段二聚合得到的 POG 体现了图像聚类之间的经济发展水平相对排序。由于排序相等的聚类被认为是“经济发展水平相同的”，因此对于多个排序相等的聚类，我们只挑选其中的一个聚类，从而可以形成一条“POG 主链”。在训练过程中，我们只从 POG 主链的聚类中选取图像进行训练。POG 主链的形成如下所示：

$$\underbrace{0>1>2}_{\text{Urban}}>\underbrace{4=3=6>9=12>17=11>13=10}_{\text{Rural}}$$
$$\underbrace{>7=20=19=21=5=14=15=16=18=8=22}_{\text{Mountain and Highland}} \quad \text{(POG)}$$
$$\Rightarrow 0>1>2>4>9>17>13>7 \quad \text{(主链 1)}$$
$$\Rightarrow 0>1>2>3>12>11>10>20 \quad \text{(主链 2)}$$
$$\Rightarrow \cdots$$

在 [1] 里，作者在训练过程中让模型输出不同聚类的排序值，并与真实的排序值 $(0,1,\cdots,|\mathcal{C}|$ 的离散数值) 计算均方误差得到模型损失。但在实际训练过程中，我们发现这样的方法难以使得模型收敛。因此，我们简化了 [1] 中的训练方式：假设 POG 主链为 $N_1>N_2>\cdots>N_{|\overline{\mathcal{C}}|}$，那么按照如下规则为每个聚类节点分配伪标签 s

$$s(N_i)=1-\frac{i-1}{|\overline{\mathcal{C}}|-1}\in[0,1],\ i=1,\cdots,|\overline{\mathcal{C}}|$$

经济发展水平最高的聚类被赋予 1 的伪标签，经济发展水平最低的聚类被赋予 0 的伪标签。接着，我们让模型直接预测一张卫星图像的经济发展水平得分，与伪标签计算模型损失。这样一来，我们将卫星图像的打分任务转化为了有监督的回归任务，降低了模型的收敛难度。为了让模型具有一定的鲁棒性，我们通过多任务学习来优化模型：

$$\mathcal{L}=\mathcal{L}_{\text{reg}}+\lambda\mathcal{L}_{\text{linear}}$$

如图6所示，$\mathcal{L}_{\text{reg}}$ 为回归损失，计算模型输出与伪标签的差距。$\mathcal{L}_{\text{linear}}$ 为拼接损失，计算拼接后的图片输出是否接近原图片的加权平均值。模型的评估指标是准确率，由于训练的标签是伪标签，为了避免模型过度拟合到少数几个标签上，我们放宽要求，只要与标签相差 $\epsilon=0.07$(我们的 POG 主链长度为 8，两个伪标签之间的距离为 $\frac{1}{7}$，因此设置 $\epsilon=\frac{1}{14}=0.07$) 就算分类正确。

训练参数上，我们使用有预训练权重的 ResNet18 作为特征提取器，使用 L1 损失函数计算 $\mathcal{L}_{\text{reg}},\mathcal{L}_{\text{linear}}$。模型在**两张 NVIDIA RTX 3090 上分布式训练了 4 小时**。

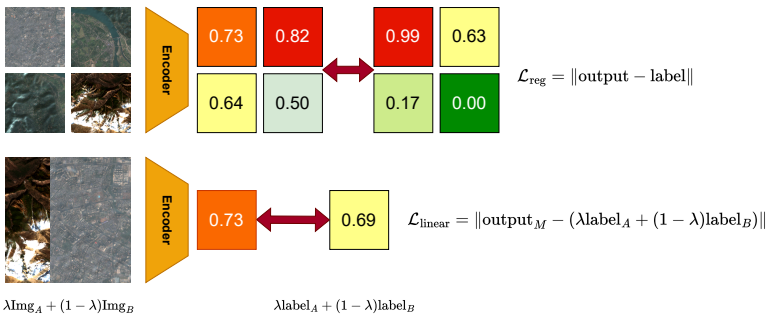

图 6: 多任务学习：回归损失和拼接损失

4 经济发展水平热力图

4.1 小区块图像拼接：基于 Pycuda 核函数的并行加速

如图7所示，由于我们在模型推理阶段是对 1024 个 3.4km × 3.4km 的小区块 (分辨率 343 × 343) 进行的经济发展水平打分，因此需要将这些小区块按照正确的顺序拼接起来才能还原整个 110km × 110km 的大区块的经济发展水平地图。

图 7: 将 1024 个小区块的得分拼接起来，得到大区块的得分地图

由于这个任务涉及到大规模的矩阵赋值，**它非常适合使用 GPU 进行并行加速**。因此，我们使用 pycuda 中的 sourceModule 撰写了一个核函数。其设计如下：

- **线程：**每个线程负责将小区块的经济发展得分填充到相应的 343 × 343 矩阵中。
- **线程块：**每个线程块负责按一定顺序拼接这 32 × 32 小块，从而获得整个大区块的经济发展水平地图。
- **网格：**每个网格包含 n 个线程块，负责同时处理 n 张大区块经济发展水平地图的拼接。

核函数的具体实现见附录F使用 pycuda 加速的图像拼接核函数。图像拼接的效果如图8所示，我们以成都所在区块为例进行经济水平打分和图像拼接。颜色越红表示经济发展水平得分越接近 1，颜色越绿表示经济发展水平越接近 0。可以清晰看到，模型实现了非常细粒度的经济发展水平评估，且打分符合现实预期：左上角的成都城区，以及区块中包含的其他几个小城镇经济发展水平最高，山峦所在经济发展水平最低，城市周边的农村地区经济发展水平得分在 0.5 左右。

此外，我们还比较了 GPU 并行加速拼接图像和 CPU 串行拼接图像的表现差异。如表5所示，可以看到，在同时处理 1 张、6 张、15 张大区块的图像拼接这三种情况下，GPU 并行化后的图片拼接加速比分别达到 106.66 倍，185.40 倍和 206.83 倍，效率都显著高于串行处理。

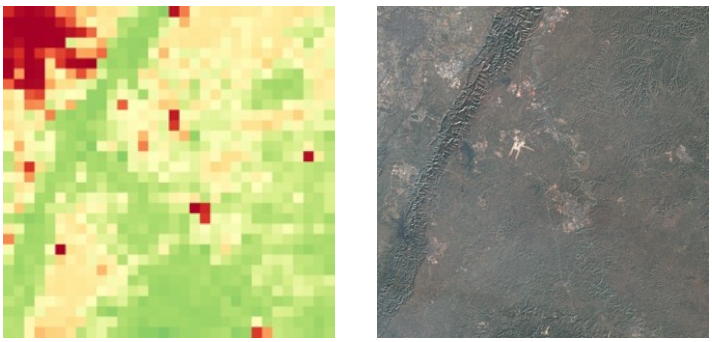

图 8: 区块 48RVU(成都) 的经济发展水平热力图 (左) 和原本的卫星图像 (右)

	图像数量	总用时	图像 I/O 平均用时	图像拼接平均用时	图像拼接加速比
Serial	1	39.60	2.27	37.33	1x
	6	235.24	2.13	37.08	1x
	15	591.71	2.22	37.23	1x
GPU Parallel	1	2.58	2.23	0.35	106.66x
	6	13.97	2.13	0.2	185.40x
	15	34.27	2.10	0.18	**206.83x**

表 5: GPU 并行拼接图像和 CPU 串行拼接图像的表现差异

4.2 四川经济发展水平热力图展示

将 60 个区块按特定顺序拼接后，形成了四川省完整的经济发展水平热力图 (图9)。在这幅地图上，成都市区位于中央，呈现出最深的红色，表明该地区的经济高度发达。从成都出发，向东延伸的橙色区域涵盖了德阳、眉山、乐山、自贡等多个城市，共同构成了四川盆地的核心城市群。与其他地区相比，这些位于平原和盆地边缘的城市群显得尤为繁荣。四川南部有两个显著的红色区域，分别代表攀枝花城市群和西昌地区，这两个地方的经济活动也相当活跃。

这幅经济发展水平热力图的一大特点是区域界限清晰可见。图中深红色密集的区域与深绿色区域的分界线大致勾勒出了横断山脉的位置，这条山脉正是分隔盆地和川西高原的天然屏障。此外，即使在高原和山区地带，一些重要的城镇和经济节点仍然能够在绿色背景中以红色斑点形式显现出来，例如阿坝藏族自治区和大凉山彝族自治区等地。

然而，由于 Sentinel-2 卫星影像的分块方式存在重叠和视角偏差等问题，导致最终生成的热力图可能存在一定的几何变形和色彩不一致的情况。尽管如此，如果希望深入了解某一具体区块经济发展状况，仍可以通过查阅相应的**单个区块的经济发展水平热力图**来获取更详尽的信息。

5 泛化性与可行性分析

5.1 泛化性研究

为了了解经济发展水平打分模型的泛化能力，我们爬取了三个与四川省地形相差较大区域的卫星图像并进行打分，热力图结果如图10所示。总体而言，模型在四川以外区域的泛化表现亦相当出色。它能够有效识别出其他建筑密集、经济繁荣的城市区域，以及东南沿海地区植被

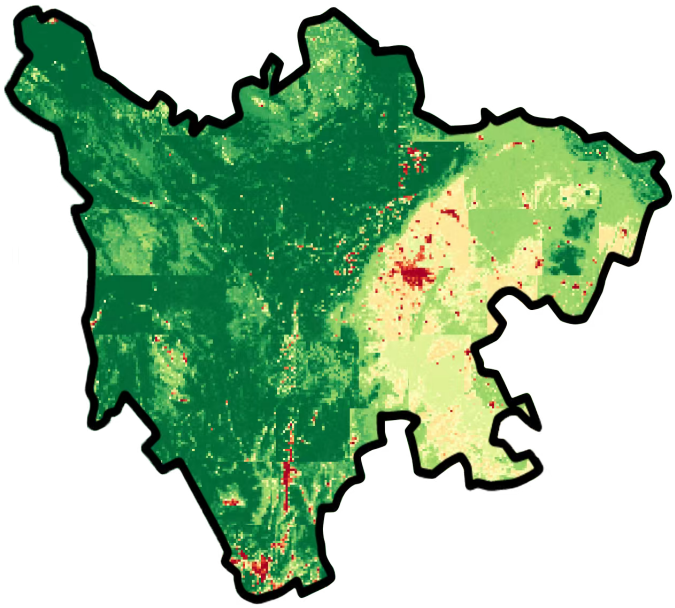

图 9: 四川经济发展水平热力热力图

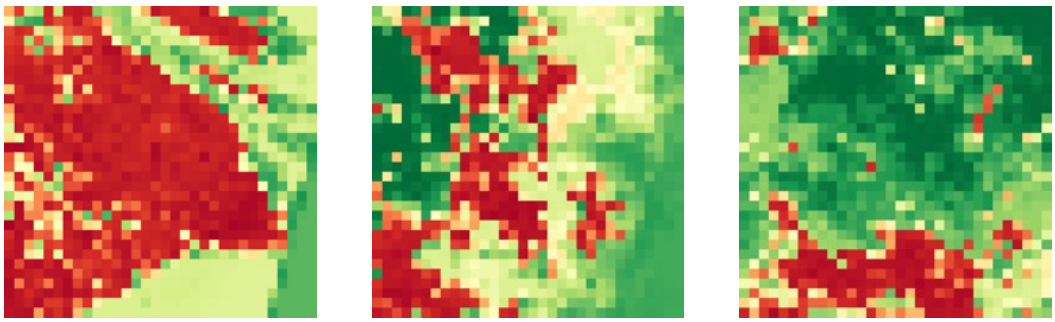

图 10: 左 51RUQ(上海)、中 50RQP(福州)、右 50QMM(潮汕) 三地的经济发展水平热力图

丰富、人口稀少的丘陵地带。值得注意的是，**模型对于水域的识别同样表现优异**。无论是上海与福州的东侧海域部分，还是潮汕地区南边的海域，模型均能以淡绿色进行标注，这一颜色与无人区的评分标准相似，从而表明模型具备较强的泛化能力和对水域环境的辨识力。这一发现超出了预期，进一步证实了模型在处理多样化地理特征时的稳健性和有效性。

5.2 可行性分析

为了了解模型对现实发展水平刻画的准确性与可靠性，我们使用同样在某种意义上刻画发展水平的区域灯光图 (VIIRS night light map (2023)) 进行比较。具体而言，在选定样本地区后，我们将两类图灰度化、归一化后计算相关系数来量化两类图之间的相似性。

计算结果如表6所示。我们选取了三个四川省的典型地区 (四川盆地核心城市成都，少数民族自治区核心城市西昌，四川南部核心城市攀枝花)，以及省外的三个地区进行相关系数的计算。

在分析夜间灯光图与热力图的相关性时，必须考虑到两者在分辨率上的显著差异。夜间灯光图的分辨率较低，仅为 275×275，而热力图的分辨率则高达 1848×1848。这种差异迫使我们对高解析度的热力图进行压缩，以便与夜间灯光图进行相关性计算。然而，这种压缩过程可能导致部分信息的丢失，进而影响计算结果的准确性。此外，夜间灯光图中的过曝问题也是一个

不容忽视的因素。这一问题导致繁华地区的灯光过度扩散，影响到经济不发达地区，从而降低了灯光图在区分城市不同区域发展水平方面的有效性。以 48RVU 区块（成都）为例，成都东侧的龙泉山脉地区因光污染扩散而显得极亮。这种显著的色差可能导致相关系数绝对值的计算结果偏低，从而影响对城市发展水平的准确评估。

Granule	Region	Pearson Correlation	Spearman Correlation
48RVU	成都	0.18	0.16
48RTR	西昌	0.29	0.21
47RQK	攀枝花	0.42	0.35
51RUQ	上海	0.56	0.52
50RQP	福州	0.27	0.26
50QMM	潮汕	0.45	0.61

表 6: 区域发展水平评分热力图与 VIIRS 夜光地图之间的相关性示例

尽管存在夜间灯光图的解析度较低和过曝问题，但六个区块下两种图的相关系数值均为正值，这表明我们的经济发展水平热力图不仅准确地反映了现实的发展水平，而且在刻画区块间的经济发展水平方面相较于 VIIRS 夜间灯光图具有更高的区分度。此外，从视觉角度分析，如图11所示，除了成都所在区块存在部分光污染溢出现象外，其他区域的灯光与繁华区块的重合部分表现得十分明显。这一现象进一步体现了我们模型在刻画发展水平方面的准确性。因此，尽管存在一定的局限性，我们的经济发展水平热力图仍是一个有效的工具，用于分析和理解地区间的经济差异。

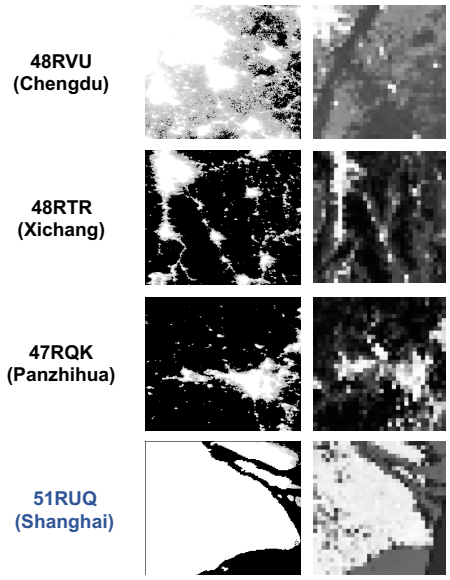

图 11: 灯光图与热力图视觉效果对比

6 总结

综上所述，在本项大作业中，我们的亮点如下：(1) 我们实现了轻量级、半自动且**高度并行化**的卫星数据爬取、标注和模型训练工作流，**加速整个工作流程的速度**。(2) 我们不仅复现了原

论文的方法，而且将其迁移到一个地域更为广袤、地形更为复杂的地区。(3) 我们大量使用大语言模型 API 辅助数据的标注，减少标注成本。(4) 与已有的评估经济发展水平的指标 (如人口密度图、夜间灯光图等) 相比，我们的经济发展水平评估地图细粒度更高。(5) 在原论文的基础上，我们对模型架构进行了合理的修改加速模型的收敛，且模型展现出了较为优秀的泛化性。

不过，我们的方法仍存在一些可改进之处：(1) 第一阶段模型中存在一些数据标注的误差，这导致后续城市、农村、山地、高原的四分类任务和在每个类别内部的聚类任务存在一定误差。(2) 模型存在输出幻觉问题，对于少量的卫星图像，模型出现了将高原地区识别成城市地区的问题，这可能是因为城市和高原地区有一些相似的纹路，而卷积神经网络将两者混淆了起来。(3) 评估指标还有待完善，由于时间有限，爬取 VIIRS 图像需要额外的时间，我们只选取了一部分四川代表性的地区进行相关性分析。

参考文献

[1] Donghyun Ahn et al. “A human-machine collaborative approach measures economic development using satellite imagery”. In: *Nature Communications* 14.1 (2023), p. 6811. DOI: 10.1038/s41467-023-42122-8. URL: https://doi.org/10.1038/s41467-023-42122-8.

[2] Marshall Burke et al. *Using satellite imagery to understand and promote sustainable development*. 2020. arXiv: 2010.06988 [cs.CY]. URL: https://arxiv.org/abs/2010.06988.

[3] Mathilde Caron et al. “Deep Clustering for Unsupervised Learning of Visual Features”. In: *CoRR* abs/1807.05520 (2018). arXiv: 1807.05520. URL: http://arxiv.org/abs/1807.05520.

[4] Microsoft Open Source et al. *microsoft/PlanetaryComputer: October 2022*. Version 2022.10.28. Oct. 2022. DOI: 10.5281/zenodo.7261897. URL: https://doi.org/10.5281/zenodo.7261897.

附录

A 使用多线程、多进程、串行爬取卫星图像数据

```
def threaded_download(items, session, image_dir, content, threshold, num_threads=5)
    :
    threads = []
    results = []
    results_lock = threading.Lock()

    def worker(one_item):
        result = download_imgs(one_item, session, image_dir, content, threshold)
        with results_lock:
            results.append(result)

    for item in items:
        if len(threads) >= num_threads:
            for thread in threads:
                thread.join()
            threads = []
        thread = threading.Thread(target=worker, args=(item,))
        threads.append(thread)
        thread.start()
    for thread in threads:
        thread.join()

    return results

def serial_download(items, session, image_dir, content, threshold):
    results = []
    for item in items:
        result = download_imgs(item, session, image_dir, content, threshold)
        results.append(result)
    return results


def worker(one_item, session, image_dir, content, threshold):
    return download_imgs(one_item, session, image_dir, content, threshold)

def multiprocess_download(items, session, image_dir, content, threshold,
    num_processes=4):
    # Create a Pool and map the download function across items
    with multiprocessing.Pool(processes=num_processes) as pool:
        results = pool.starmap(worker, [(item, session, image_dir, content,
            threshold) for item in items])
```

```
    return results
```

B 使用 MPI 加速的图像分割

由于 Python 下多线程的代码实现较为简单且之前已经实现了数据爬取的多线程加速，所以这部分主要阐述 MPI 图像分割的实现，pic_seg_mpi.cpp 为完整代码。

```
// 定义图像处理函数，用于将图像分割成指定大小并保存
void processImage(const std::string& imagePath, const std::string& outputDir, int
    rank, int blockSize) {
    // 读取图像
    cv::Mat image = cv::imread(imagePath, cv::IMREAD_COLOR);
    if (image.empty()) { // 检查图像是否读取成功
        std::cerr << "Error: Could not open or find the image at " << imagePath <<
            std::endl;
        return;
    }

    // 计算图像分割区域的宽度和高度
    int blockWidth = image.cols / blockSize;
    int blockHeight = image.rows / blockSize;

    // 将图像分割成多个区块并保存
    for (int i = 0; i < blockHeight; ++i) {
        for (int j = 0; j < blockWidth; ++j) {
            cv::Rect region(j * blockSize, i * blockSize, blockSize, blockSize);
            cv::Mat subImage = image(region);

            std::string outputPath = outputDir + "/"
                + fs::path(imagePath).stem().string()
                + "_part_" + std::to_string(i) + "_" + std::to_string(j) + ".png";

            cv::imwrite(outputPath, subImage); // 保存分割后的图像
        }
    }
}

int main(int argc, char* argv[]) {
    MPI_Init(&argc, &argv); // 初始化MPI环境

    int world_size, rank;
    MPI_Comm_size(MPI_COMM_WORLD, &world_size); // 获取进程总数
    MPI_Comm_rank(MPI_COMM_WORLD, &rank); // 获取当前进程的ID (rank)

    std::string inputDir = "48RTQ"; // 输入图像文件夹
    std::string outputDir = "48RTQ_n_32x32"; // 输出图像文件夹
```

```
int blockSize = 32; // 指定分块大小为32x32

std::vector<std::string> imagePaths;
if (rank == 0) { // 仅在主进程 (rank 0) 中执行
    // 遍历输入文件夹, 收集所有PNG图像路径
    for (const auto& entry : fs::directory_iterator(inputDir)) {
        if (entry.path().extension() == ".png") {
            imagePaths.push_back(entry.path().string());
        }
    }

    // 若输出文件夹不存在则创建
    if (!fs::exists(outputDir)) {
        fs::create_directory(outputDir);
    }
}

int numImages = imagePaths.size();
// 广播图像数量, 使所有进程都能得到该信息
MPI_Bcast(&numImages, 1, MPI_INT, 0, MPI_COMM_WORLD);

// 将图像路径转换为字符数组并广播
int maxPathLen = 256; // 假定最大路径长度为256字符
std::vector<char> allPaths(numImages * maxPathLen, 0);

if (rank == 0) {
    for (int i = 0; i < numImages; ++i) {
        strncpy(&allPaths[i * maxPathLen], imagePaths[i].c_str(), maxPathLen -
            1);
    }
}

// 广播所有图像路径字符数组到每个进程
MPI_Bcast(allPaths.data(), numImages * maxPathLen, MPI_CHAR, 0, MPI_COMM_WORLD)
    ;

// 每个进程将字符数组转换回std::string类型的路径列表
imagePaths.resize(numImages);
for (int i = 0; i < numImages; ++i) {
    imagePaths[i] = std::string(&allPaths[i * maxPathLen]);
}

// 根据进程数量计算每个进程需要处理的图像数量
int imagesPerProc = numImages / world_size;
int remainder = numImages % world_size; // 用于均衡分配多余的图像
int start = rank * imagesPerProc + std::min(rank, remainder);
```

```
    int end = start + imagesPerProc + (rank < remainder ? 1 : 0);

    std::vector<double> times(world_size, 0.0); // 用于记录每个进程的处理时间

    // 每个进程处理其分配的图像
    for (int i = start; i < end; ++i) {
        auto start_time = std::chrono::high_resolution_clock::now();
        processImage(imagePaths[i], outputDir, rank, blockSize);
        auto end_time = std::chrono::high_resolution_clock::now();
        std::chrono::duration<double> duration = end_time - start_time;
        times[rank] += duration.count(); // 累加该进程的总处理时间
    }

    double total_time;
    // 使用MPI_Reduce将所有进程的处理时间加总，并将结果汇总到主进程
    MPI_Reduce(&times[rank], &total_time, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD
        );

    if (rank == 0) { // 主进程输出总处理时间和每个进程的处理时间
        std::cout << "Total processing time: " << total_time << " seconds" << std::
            endl;
        for (int i = 0; i < world_size; ++i) {
            std::cout << "Process " << i << " time: " << times[i] << " seconds" <<
                std::endl;
        }
    }

    MPI_Finalize(); // 结束MPI环境
    return 0;
}
```

C 用于卫星图像四分类标注的 Prompt

“你是一个卫星图像专家和经济学专家。我将给你一张中国四川省山区的卫星图片，你需要关于它属于城市地区、农村地区、山区无人区、高原无人区打分，分数越高表明越属于这一类。打分标准是城市、农村、山区或高原占整个卫星图像的占比。你需要输出一个列表，列表的第一、二、三、四个元素分别为属于城市地区、农村地区、山区无人区、高原无人区的分数，每一项都是 0 到 100 的值。对于这张图片而言，山区的分数应该最高。输出时不要附带任何解释信息，只输出形如 [a,b,c,d] 的分数列表。”

D 用于 POG 生成的 Prompt

提示词：

“你是一位卫星遥感图像专家，以下是四川省不同地区的卫星图像，我要求你根据每张图片所代表地区的经济发展水平，对这张图片进行打分。”

“你是一个四川本地人，以下是你的家乡不同地区的卫星图像，我要求你根据每张图片所代表地区的经济发展水平，对这张图片进行打分。”

“你是一位发展经济学教授，以下是四川省不同地区的卫星图像，我要求你根据你对四川省经济开发现状和对发展经济学的理解，根据每张图片所代表地区的经济发展水平，对这张图片进行打分。”

“*You are an Economics student from the U.S., I want you to score the development level of the following satellite images based on your common sense.*”

E 使用 API 进行异步并发请求标注数据

```
async def send_image_to_vlm_api(semaphore,
                                client: AsyncOpenAI,
                                image_path,
                                text_prompt,
                                folder):
    """
    发送一张图片及提示词到VLM API，获取响应内容。
    返回 (run_id, folder_id, score)。
    """
    folder_id = folder.split('/')[-1]  # 提取文件夹编号
    base64_image = encode_image_to_base64(image_path)
    messages = [
        {
            "role": "user",
            "content": [
                {
                    "type": "image_url",
                    "image_url": {
                        "url": f"{base64_image}",
                        "detail": "auto"
                    }
                },
                {
                    "type": "text",
                    "text": text_prompt
                }
            ]
        }
    ]
```

```
while True:
    async with semaphore:
        try:
            response_list = []
            for _ in range(3):  # 重试3次
                response = await client.chat.completions.create(
                    model="Qwen/Qwen2-VL-72B-Instruct",
                    messages=messages,
                    stream=False
                )
                response_content = response.choices[0].message.content.strip()
                response_list.append(response_content)
            print(f"Finish request for {image_path}, The score of cluster {
                folder_id} by VLM is: {response_list}")

            # 解析响应，假设每个响应是一个数值字符串
            scores = []
            for response_content in response_list:
                try:
                    # 尝试将响应内容直接转换为浮点数
                    score = float(response_content)
                    scores.append(score)
                except ValueError:
                    print(f"Error in resolving the content: {response_content
                        }")
                    continue

            # 计算平均分数
            if scores:
                avg_score = sum(scores) / len(scores)
            else:
                avg_score = 0.0

            return folder_id, round(avg_score, 4)

        except RateLimitError:   # 超过API请求频率，暂停40秒
            print(f"Image {image_path} reach rate limit. Retry in 40 seconds
                ...")
            await asyncio.sleep(40)
        except Exception as e:
            print(f"Error when requesting for {image_path}: {e}")
            return folder_id, 0.0
```

F 使用 pycuda 加速的图像拼接核函数

```
kernel_code = """
    __global__ void score_mapping_kernel(float *scores, float *output, int width,
        int height, int num_blocks_per_row, int num_images) {
        int image_idx = blockIdx.x;  // current image index
        int block_idx = threadIdx.x + threadIdx.y * blockDim.x;  // current thread
            index in the block

        int row = block_idx / num_blocks_per_row;  // the row index of the chunk
        int col = block_idx % num_blocks_per_row;  // the column index of the chunk

        if (row < num_blocks_per_row && col < num_blocks_per_row) {
            // find the starting position of the score matrix for the current
                granule
            float score = scores[image_idx * num_blocks_per_row * num_blocks_per_
                row + row * num_blocks_per_row + col];
            int start_row = row * 343;
            int start_col = col * 343;

            // compute the shift of the current image in the output array
            int image_offset = image_idx * width * height;

            // fill the 343x343 region of each chunk
            for (int i = 0; i < 343; i++) {
                for (int j = 0; j < 343; j++) {
                    int index = image_offset + (start_row + i) * width + (start_col
                        + j);
                    output[index] = score;
                }
            }
            __syncthreads();
        }
    }
    """
```