CUDA 驱动的并行连通域标记

余笑轩
xiaoxuan_yu@pku.edu.cn
化学与分子工程学院

Jun 24, 2024

Abstract

连通域标记是计算机视觉中的基本操作，广泛应用于图像分割、目标检测和图像分析等领域。本次作业实现了对于二维二值图像的 8-连通域标记算法。我们使用 CUDA 实现了 KE 算法，并对其进行了性能测试。通过正确性验证和性能测试，我们验证了实现的正确性和性能，取得了相对于 CPU 串行实现的显著加速。我们使用 `nsys` 和 `nsight-compute` 工具对程序进行了性能分析，发现了程序的性能瓶颈，并提出了进一步的优化方向。

Keywords 连通域标记 · 并行计算 · CUDA

1 连通域标记问题

连通域标记（Connected Component Labeling, CCL）是计算机视觉中的一种基本操作，广泛应用于图像分割、目标检测和图像分析等领域。它的主要任务是识别和标记图像中相连的像素块，即连通域。连通域标记在图像处理、模式识别和计算机视觉的许多应用中起着关键作用。

在图像中，连通域是指所有像素值相同且通过某种连通性准则（如 4-连通或 8-连通，如 图 1 所示）相连的区域。连通域标记算法的目标是为每个连通域分配一个唯一的标签，以便后续的图像处理和分析工作。具体而言，本次作业将实现对于二维二值图像的 8-连通域标记算法。

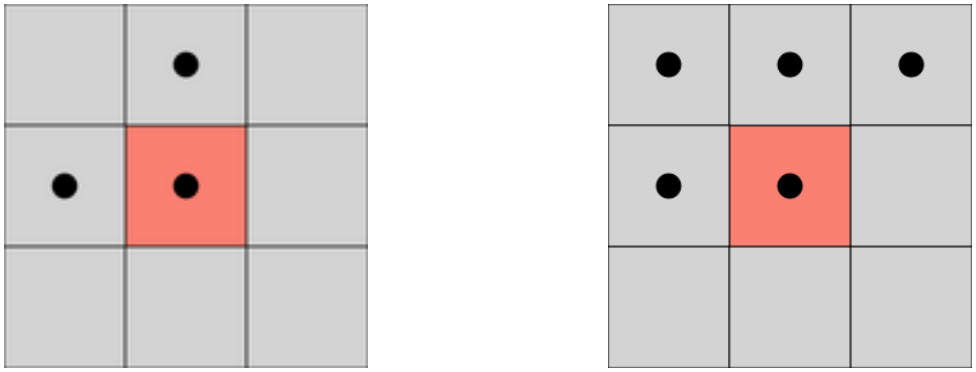

图 1. 4-连通性和 8-连通性的示意图 (Wikipedia contributors 2023)

2 算法

2.1 CCL 的串行算法: 并查集

基于并查集的串行算法是一种经典的连通域标记算法。并查集 (Union-Find) 是一种常用的数据结构, 能够高效地处理连通域标记问题。并查集主要包含两个操作: 查找 (Find) 和合并(Union), 如 图 2 所示。

- 查找: 确定某个元素属于哪个连通域。
- 合并: 将两个连通域合并为一个。

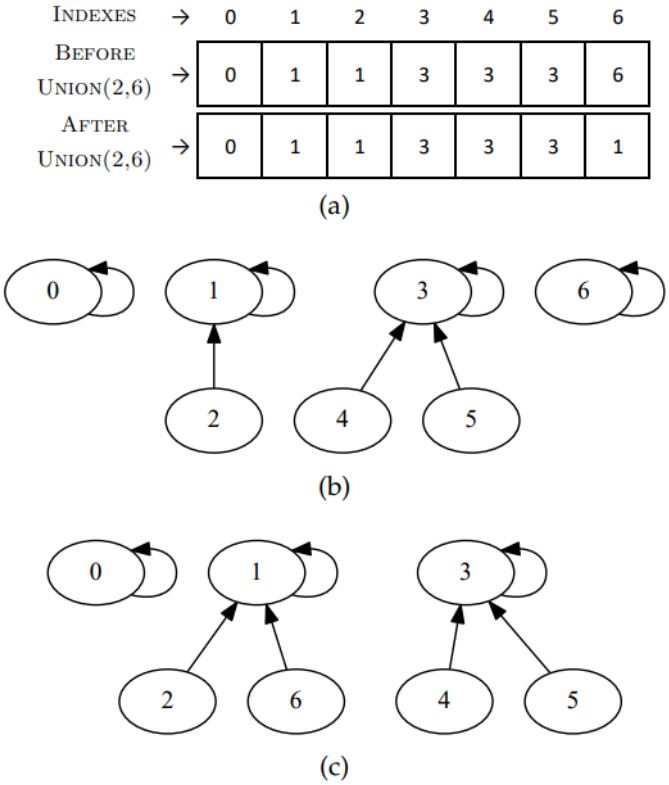

图 2. 并查集示意图 (Allegretti, Bolelli, 和 Grana 2020)

基于并查集的算法通过逐像素扫描图像, 使用并查集记录和合并相邻像素的连通信息, 从而实现连通域标记。具体步骤如下:

1. 初始化并查集，每个像素作为一个独立的集合。
2. 逐像素扫描图像，对每个像素检查其上方和左方像素的连通情况，进行合并操作。
3. 第二次扫描图像，对每个像素进行查找操作，确定其最终的连通域标签。

2.2 GPU 并行的 CCL: Komura Equivalence 算法

对于并查集的并行化并不是显而易见的。传统的并查集算法是基于串行处理的, 直接并行化会面临许多挑战, 尤其是在处理等价类合并时, 需要解决多个线程之间的同步和冲突问题。为了有效地在 GPU 上实现并行的连通域标记, 研究者们提出了多种改进方案, KE(Komura-

Equivalence) 算法就是其中之一。KE 算法(Komura 2015) 通过一系列步骤来实现高效的 GPU 并行连通域标记, 其过程如 图 3 所示:

- 初始化: 为每个像素分配一个唯一的初始标签，通常使用其线性索引值。
- 等价类更新: 在此步骤中, 多个 GPU 线程并行处理像素, 检查每个像素与其相邻像素的连通性, 并更新等价类信息。这一步骤通常需要多次迭代, 直到所有像素的标签稳定下来。
- 标签压缩: 使用路径压缩技术对等价类进行压缩，确保所有等价像素的标签一致。这一步骤通过在并查集的“查找”操作中进行路径压缩来实现。
- 标签传播: 将最终标签传播到所有连通像素，确保每个连通域的所有像素共享相同的标签。

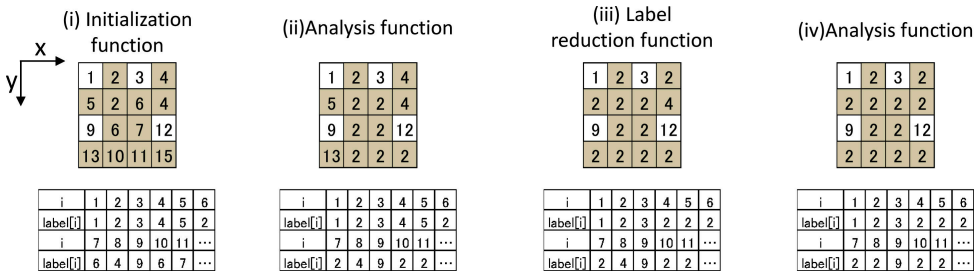

图 3. KE 算法示意图

KE 算法通过分阶段处理和并行化技术, 有效地克服了传统并查集在 GPU 上的并行化困难, 提高了连通域标记的效率。2018 年，Allegretti 等人给出了 8 连通的 KE 算法 (Allegretti 等 2018), 该算法主要对图 3 中的 reduction 部分进行了改进，如图 4 所示。

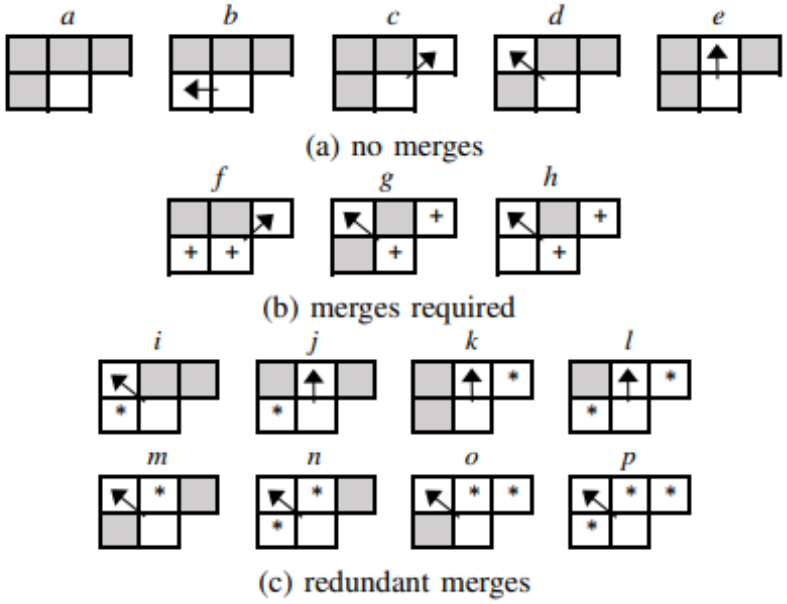

图 4. 8-连通的 KE 算法示意图

3 实现

3.1 概述

我们使用 CUDA 实现了 KE 算法，并对其进行了性能测试，参见 `algorithm/` 文件夹下的相关代码。

```
.
├── CMakeLists.txt
└── src
    ├── alg
    │   ├── KE.cu
    │   └── UFTree.cuh
    ├── CMakeLists.txt
    ├── include
    │   ├── alg_runner.cuh
    │   ├── Matrix.cuh
    │   └── timer.cxx
    ├── main.cu
    └── serial.cxx
```

`main.cu` 是并行版本的主程序入口，包含输入输出的处理，`Matrix` 对象的初始化以及 `KE` 算法的调用及性能评估。`include` 文件夹下包含了必要的功能函数和类，主要是用于包装矩阵的结构体 `Matrix` 以及其输出函数，计时器 `Timer` 以及用于性能评估的并行算法的运行器 `alg_runner`。`alg` 文件夹下包含了 `KE` 算法的实现以及并查集相关操作的实现。`serial.cxx` 是给定的串行版本的主程序，只是进行了简单的修改用于多次运行算法以进行性能评估。

在我们的实现中，每个 CUDA `thread` 负责处理一个像素。我们选择了 `dim3(16,16,1)` 作为 `block` 的大小，并计算出所需的 `grid` 的形状 `dim3(cols+BLOCK_COLS-1)/BLOCK_COLS, (rows+BLOCK_ROWS-1)/BLOCK_ROWS,1)`，以保证开启的总线程数至少多于图像的像素数。

3.2 Komura-Equivalence 算法的实现

如 小节 2.2 描述，KE 算法主要包含三个不同操作：初始化、路径压缩和归并。在此，分别给出相应的伪代码。在 InitKernel 中，我们为每个像素分配初始标签。

```
def InitKernel(images, labels, index):
    row, col = index // images.cols, index % images.cols
    # row and col from threadID, blockDim, blockIdx, thus need to check if it is valid
    if (row < images.rows and col < images.cols):
      if (row>0 and images[row][col] == images[row-1][col]):
        labels[row][col] = index - images.cols + 1 # labels[row-1][col]
      elif (row>0 and col>0 and images[row][col]s == images[row-1][col-1]):
        labels[row][col] = index - images.cols # labels[row-1][col-1]
      elif (row>0 and col<images.cols-1 and images[row][col] == images[row-1][col+1]):
        labels[row][col] = index - images.cols + 2 # labels[row-1][col+1]
      elif (col>0 and images[row][col] == images[row][col-1]):
        labels[row][col] = index # labels[row][col-1]
      else:
        labels[row][col] = index + 1
```

在 CompressionKernel 中，我们对并查集进行路径压缩。这个 Kernel 也在完成归并操作后调用，用于进行标签的传播

```
def CompressionKernel(labels, index):
    row, col = index // images.cols, index % images.cols
    if (row < images.rows and col < images.cols):
      label = labels[row][col]
      if (label){
        labels[row][col] = Find_label(labels, index, label) + 1
      }
```

在 ReduceKernel 中，我们进行并查集的归并操作。由于实现的是 8-连通的 KE 算法，因此我们除了检查当前像素的左侧外，还需要检查右上方的像素。

```
def ReduceKernel(images, labels, index):
    row, col = index // images.cols, index % images.cols
    if (row < images.rows and col < images.cols):
      if (col>0 and images[row][col] == images[row][col-1]):
        Union(labels, index, index-1)
      if (row>0 and col<images.cols-1 and images[row][col] == images[row-1][col+1]):
        Union(labels, index, index-images.cols+1)
```

顺次调用这三个 Kernel 函数并最后调用一次 `CompressionKernel` 用于标签传播，即可完成整个 KE 算法的实现。

```
def KE(images, labels):
    InitKernel<<<grid, block>>>(images, labels)
    CompressionKernel<<<grid, block>>>(labels)
    ReduceKernel<<<grid, block>>>(images, labels)
    CompressionKernel<<<grid, block>>>(labels)
```

具体的 Kernel 函数及 KE 算法流程的实现参见 `algorithm/src/alg/KE.cu`，在此不再赘述。

4 结果与讨论

4.1 正确性验证

以作业中给定的串行程序作为参考，我们对并行程序的结果进行了验证。在正确性验证中，一个难以处理的问题是即使结果正确，标签的顺序和值也都可能不同。因而，我们实现了一个映射算法，将并行程序的标签映射到串行程序的标签，从而验证两者的结果是否一致。映射算法的实现非常平凡，我们将每个标签对应的像素坐标全部记录并进行匹配，从而得到一个标签之间的映射表。根据这个映射表执行映射后，我们可以直接比较两个 `label` 数组是否一致从而验证正确性。正确性验证的相关代码参见 `validation/val.py` 和 `validation/validation.ipynb`。此处给出核心功能函数的实现。

```
def get_match_dict(labels_a, labels_b):
    uni_labels_a = np.unique(labels_a)
    uni_labels_b = np.unique(labels_b)
    labels_a_dict = {label: [] for label in uni_labels_a}
    labels_b_dict = {label: [] for label in uni_labels_b}
    for i in range(labels_a.shape[0]):
        for j in range(labels_a.shape[1]):
            labels_a_dict[labels_a[i][j]].append(i * labels_a.shape[1] + j)
    for i in range(labels_b.shape[0]):
        for j in range(labels_b.shape[1]):
            labels_b_dict[labels_b[i][j]].append(i * labels_b.shape[1] + j)
    match_dict = {}
    for label in uni_labels_a:
        label_index_list = labels_a_dict[label]
        for b_label in uni_labels_b:
            b_label_index_list = labels_b_dict[b_label]
            if len(label_index_list) != len(b_label_index_list):
                continue
            if all(
                [
```

```
                    label_index_list[i] == b_label_index_list[i]
                    for i in range(len(label_index_list))
                ]
            ):
                match_dict[label] = b_label
                break
    return match_dict
def map_with_dict(labels, match_dict):
    labels = labels.copy()
    for i in range(labels.shape[0]):
        for j in range(labels.shape[1]):
            labels[i][j] = match_dict[labels[i][j]]
    return labels
```

在验证过程中，我们发现并行程序的结果与串行程序的结果一致，验证通过。对于每一个标签，我们将其重新映射到一个随机的颜色，用于可视化连通域标记问题的结果。以下给出对于作业中要求的四个样例的可视化结果。

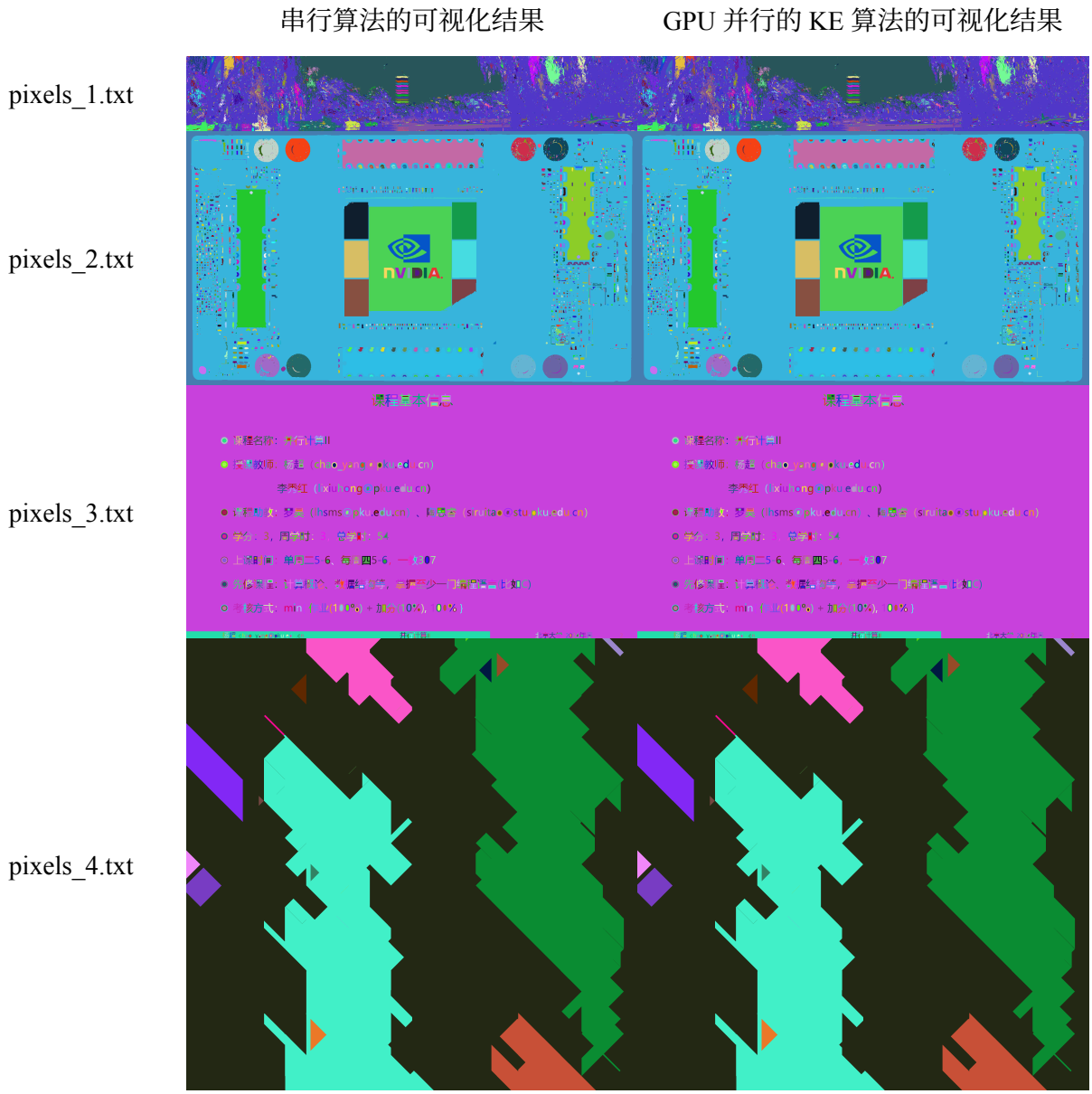

图 5. 连通域标记结果的可视化

上述结果直观地证明，我们的并行程序实现了正确的连通域标记。

4.2 性能测试

使用数院集群，我们完成了串行和 GPU 并行算法的性能测试。CPU 程序在单颗 E5-2650 v4 处理器上运行，计算节点内存为 128 GB；GPU 程序在一块 NVIDIA Titan XP GPU 上运行，其具有 3840 个 CUDA core, 12 GB GDDR5X 现存和 1582 MHz 的主频，计算节点内存为 256 GB。所有程序使用 `GCC Compiler 9.3.0` 和 `nvcc 11.3` 进行编译。

以下给出串行和并行算法的性能测试结果，如表 1 所示。受计算资源限制，串行算法运行 20 次取平均运行时间，而 GPU 并行算法运行 100 次取平均时间作为测试结果。性能测试结果的原始文件见 `perf/CPU_CCL.out` 和 `perf/GPU_CCL.out`。

表 1. 串行算法和并行算法的运行时间对比

算法	**运行时间** (μs)				
	Debug	Sample 1	Sample 2	Sample 3	Sample 4
Serial UF	67.3341	293872	727506	4654530	8283640
GPU KE	64.2954	2058.72	4092.8	22955.9	40088.3

从而可以计算 $S = T_{\text{serial}}/T_{\text{parallel}}$，如 表 2 所示。

表 2. GPU 算法的加速倍数

Debug	Sample 1	Sample 2	Sample 3	Sample 4
1.047	142.75	177.75	202.76	206.63

可以看到，除了非常小的测试案例 Debug，GPU 算法在其他测试案例中都取得了显著的加速效果。在最大的测试案例中，GPU 算法的加速效果达到了 207 倍。

4.3 性能分析

4.3.1 总览

我们使用 `nsys` 工具对程序在 GPU 上的运行时间进行了详细的分析。我们注意到，在 `CUDA` 程序第一次启动时，需要执行初始化相关的 kernel 函数，同时完成 PTX 编译，因此第一次运行算法的时间会显著高于后续的运行时间。这也是我们在程序进行性能测试时从第二次运行算法开始计时的原因。对于一次典型的 KE 算法运行，给出 `nsys` 绘制的 timeline 图如 图 6 所示。

图 6. KE 算法的时间线图

在 `nsys` 的分析结果中，我们可以看到，GPU 程序的主要时间消耗在 `cudaMemcpy`。尽管仅仅在输入图像和输出标签时分别进行了一次 host2device 和 device2host 的数据传输，但它们的合计用时接近算法运行总时长的 2/3，与整个连通域标记算法的执行用时相当。对于我们自行编写的算法部分，其主要时间消耗在 `ReduceKernel` 和 `CompressionKernel` 两个核函数上，尤其

是 ReduceKernel 消耗了约一半的运行时长。这的确是符合逻辑的，因为在 KE 算法中，ReduceKernel 是用于进行并查集的归并操作，是整个算法的核心部分；但这同样提醒我们，进一步优化 ReduceKernel 的性能是提高整个算法效率的关键。另一个值得关注的点是对于 Init_L 这个 kernel，调用时存在 CPU 和 GPU 均处于空闲状态的情况，也是进一步优化的潜在方向。

4.3.2 性能热点的进一步分析

为了进一步的明确性能瓶颈，我们使用 nsight-compute 对核函数的性能进行了分析。主要关注 hotspot 也就是 ReduceKernel 和 CompressionKernel 两个核函数。首先，我们需要确定这两个核函数是计算密集型的还是内存密集型的。给出两个 kernel 分别的 GPU Throughput 结果如 所示。

表 3. 核函数的 GPU Throughput

Kernel	Compute Throughput / %	Memory Throughput / %
ReduceKernel	66.04 %	79.80 %
CompressionKernel	77.79 %	26.31 %

结合 nsight-compute 进行 roofline 分析 (Williams, Waterman, 和 Patterson 2009) 的结果，我们可以看到 CompressionKernel 是计算密集型的，而 ReduceKernel 是内存密集型的。从而，可以进一步分析相应的工作负载。

对于内存密集型的 ReduceKernel, 给出其显存吞吐量如图 7 所示。

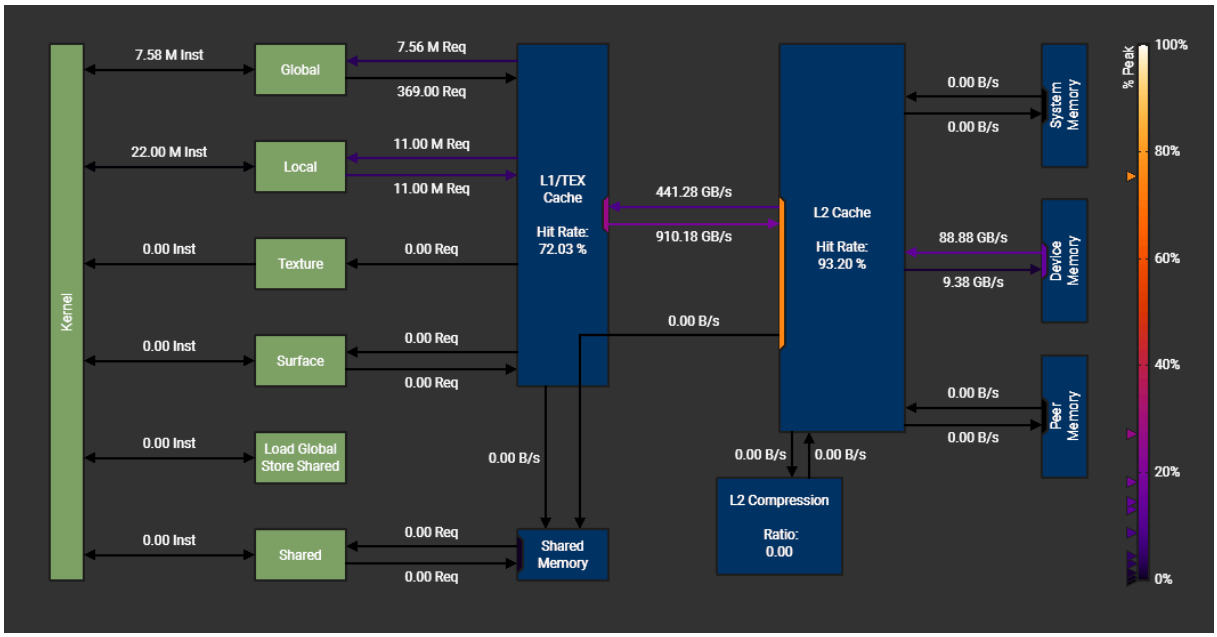

图 7. ReduceKernel 的显存工作负载

可以看到，大部分的数据传输发生在 L1 cache 和 L2 Cache 之间，这是相对较快的。但是，也有一部分数据传输发生在 L2 Cache 和 Device Memory 之间，相对较慢。查看 nsight-compute 的统计数据，发现这一部分访问中执行了比较多的越界访问，这些访问基本都出现在 if 语句的判断条件中，作为用于判断的索引值可能超出了数组的范围。尽管由于其他判断条件的作用，这并不会产生结果上的错误，但是会导致额外的访存，从而影响性能。nsight-compute 预计这个问题的修复可能带来约 15% 的性能提升。除此之外，L1 cache 的命中率约 72%，仍有提升空间。

对于计算密集型的 CompressionKernel, 给出其计算的管线利用率如图 8 所示。

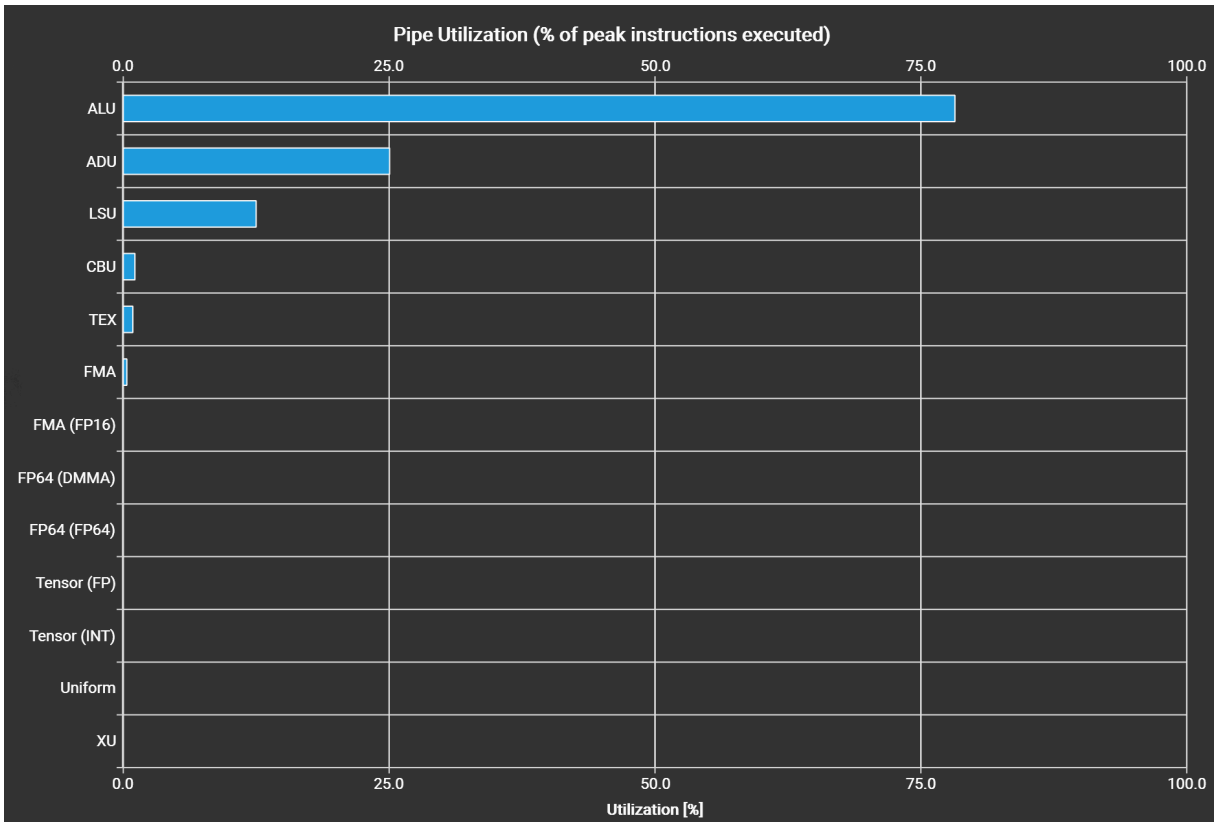

图 8. CompressionKernel 的计算工作负载

可以看到绝大多数操作是用于逻辑运算和位运算的 ALU 操作，同时涉及较高比例的条件判断、循环体等带来的 ADU 操作。这意味着在当前算法的框架下，此处的性能瓶颈优化潜力相对有限。在总调用次数难以减少的情况下，进一步的优化可能需要考虑对于逻辑运算和位运算的优化，例如使用高效的掩码位运算代替条件判断等。

除此之外，由于在整个运算过程中，图形的原始数组 `img` 实际上是只读的，同时也是与 Device Memory 交互的重要来源，因而，通过使用 CUDA 的纹理内存以加速访问，也是一个潜在的优化方向。此外，由于纹理内存对二维数据的访问有着天然的优势，因此，可以考虑将 `img` 和 `labels` 从一维数组转换为二维数组，以进一步提高访存效率和缓存的命中率。

5 总结

本次作业中，我们实现了基于 CUDA 的并行连通域标记算法。我们首先介绍了连通域标记问题的背景和串行算法，并详细介绍了 KE 算法的并行实现。我们通过正确性验证和性能测试验证了实现的正确性和性能，取得了相对于 CPU 串行代码百倍以上的加速效果。我们使用 `nsys` 和 `nsight-compute` 工具对程序进行了性能分析，发现了程序的性能瓶颈，并提出了进一步的优化方向。

代码可用性

本次作业的代码已经开源在 GitHub 上，地址为 https://github.com/xiaoxuan-yu/CUDA-CCL.

参考文献

[1] Wikipedia contributors, “Connected-component labeling — Wikipedia, The Free Encyclopedia.” [Online]. Available: https://en.wikipedia.org/w/index.php?title=Connected-component_labeling&oldid=1192036140

[2] S. Allegretti, F. Bolelli, and C. Grana, “Optimized Block-Based Algorithms to Label Connected Components on GPUs,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, no. 2, pp. 423–438, 2020, doi: 10.1109/TPDS.2019.2934683.

[3] Y. Komura, “GPU-based cluster-labeling algorithm without the use of conventional iteration: Application to the Swendsen–Wang multi-cluster spin flip algorithm,” *Computer Physics Communications*, vol. 194, pp. 54–58, 2015, doi: https://doi.org/10.1016/j.cpc.2015.04.015.

[4] S. Allegretti, F. Bolelli, M. Cancilla, and C. Grana, “Optimizing GPU-Based Connected Components Labeling Algorithms,” in *2018 IEEE International Conference on Image Processing, Applications and Systems (IPAS)*, 2018, pp. 175–180. doi: 10.1109/IPAS.2018.8708900.

[5] S. Williams, A. Waterman, and D. Patterson, “Roofline: an insightful visual performance model for multicore architectures,” *Commun. ACM*, vol. 52, no. 4, pp. 65–76, Apr. 2009, doi: 10.1145/1498765.1498785.