My Road To Deep Learning

目录

Chapter 1

第一章 绪 论

1.1 一级标题

1.1.1 二级标题

1.1.2 三级标题

1.2 一级标题

TODO:
1.

this is test

Chapter 2

第二章 数学基础

2.1 高斯分布

一维的高斯分布 (Gaussian distribution) 函数表示为：

$$G(x) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(x-\mu)^2}{2\sigma^2}} \tag{2.1}$$

其中 μ 表示均值 (mean)，σ 表示标准差 (standard deviation)。一般用标准差的平方 σ^2 来表示方差 (variance)。均值表示所有数据的平均值大小，标准差则表示数据和均值之间的偏差程度。假设所有的 N 个数据为 $x_1, x_2, ..., x_n$，则均值计算方式为:

$$\mu = \frac{1}{N} \sum_{i=1}^{n} x_i \tag{2.2}$$

标准差的计算公式为:

$$\sigma = \sqrt{\frac{\sum |x - \mu|^2}{N}} \tag{2.3}$$

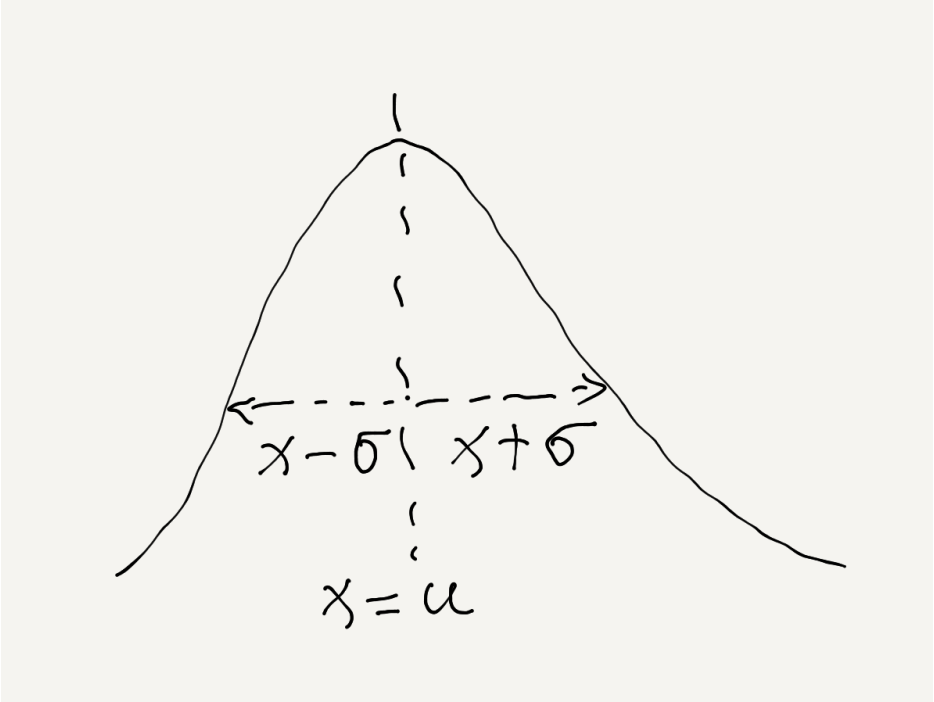

图 2.1: 高斯分布函数

二维高斯分布函数表示为:

$$G(x) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}} \tag{2.4}$$

这里假设均值是 0。

2.2 卷积

2.3 距离度量

距离度量用来表示两个点之间的距离信息，假设有两个坐标点 $(x_1, y_1), (x_2, y_2)$。则常用度量距离方式有:

Manhattan distance:

$$G = |x_1 - x_2| + |y_1 - y_2| \tag{2.5}$$

Euclidean distance:

$$G = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2} \tag{2.6}$$

2.4 角度

角度 (angle) 和弧度 (radian) 换算关系为:

$$angle = radian \times \frac{180}{\pi} \tag{2.7}$$

Chapter 3

第三章 计算机视觉

3.1 相机标定

相机标过程定主要是计算相机内参和外参的过程。一般是将现实的坐标点通过几个平面的映射关系映射到图像坐标上去 [4]。可以将转换过程分为三个部分：世界坐标系 (World Coordinate System) 到相机坐标系 (Camera Coordinate System)，相机坐标系到图像坐标系 (Image Coordinate System), 图像坐标系到像素坐标。整体映射过程如下:

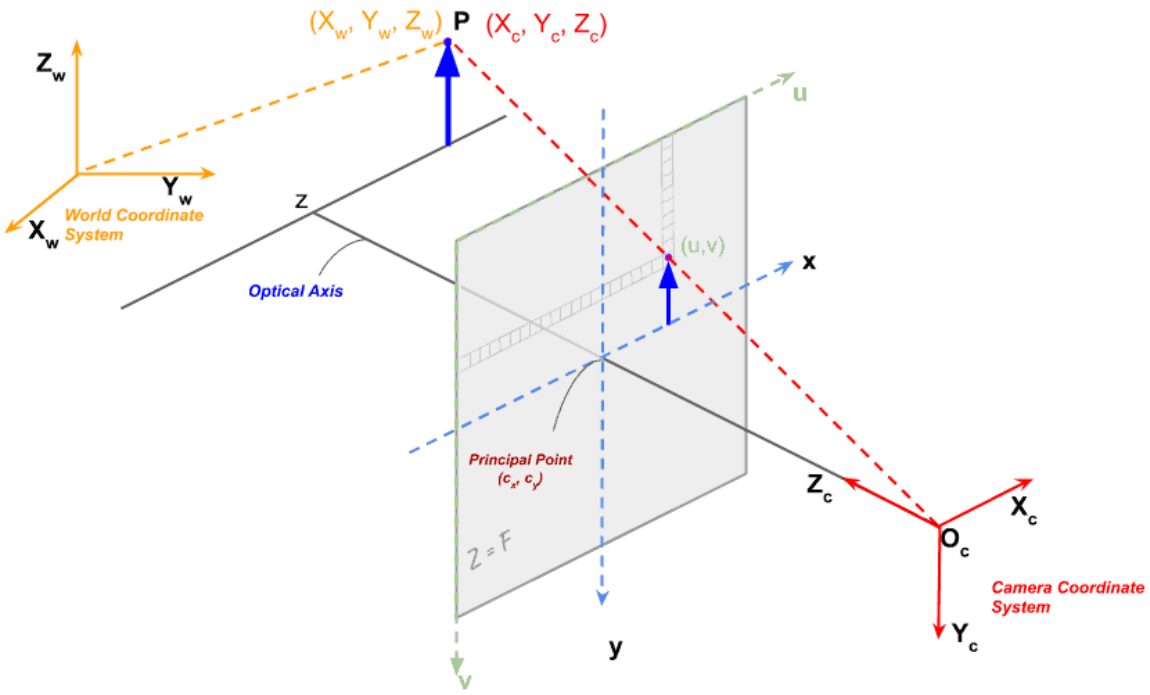

图 3.1: 世界坐标系到像素坐标系的转换过程，图像来源LearnOpenCV

。

3.1.1 世界坐标系到相机坐标系

如图3.1，假设有一个真实三维世界坐标系 (X_W, Y_W, Z_W) , 坐标系中的某个点坐标可表示为是 $P(X_W, Y_W, Z_W)$。一般情况世界坐标系的原点 O_W 可选择任意位置，比如墙角处等。同理，可以以相机处的物理位置为坐标原点 O_C 得到相机坐标系 (X_C, Y_C, Z_C)，则同一个世界坐标系下的 P 点相对于相机坐标系的坐标是 $P(X_C, Y_C, Z_C)$, 根据相机的小孔成像原理，世界坐标系和相机坐标系下的同一点会在各个坐标方向上存在旋转和平移。

根据 [4] 中公式 (2.24) 可以将两个坐标的旋转和平移表示为:

$$x' = Rx + t \tag{3.1}$$

其中 x 和 x' 是两个坐标系下的坐标，R 是一个 (3×3) 的标准正交旋转矩阵, t 是一个 (3×1) 的

平移向量。则从世界坐标系到相机坐标系的转换过程可以完整的表示为:

$$\begin{bmatrix} X_C \\ Y_C \\ Z_C \end{bmatrix} = R \cdot \begin{bmatrix} X_W \\ Y_W \\ Z_W \end{bmatrix} + t = [R|t] \begin{bmatrix} X_W \\ Y_W \\ Z_W \\ 1 \end{bmatrix} \tag{3.2}$$

一般情况称旋转矩阵和平移向量的组合 $[R \mid t]$ 为外参矩阵 (Extrinsic Matrix)。

3.1.2 相机坐标系到图像坐标系

图像坐标系图像所在平面上形成的坐标系，如图以图像中心点为坐标原点（记为 O），以 x 为横轴，y 为纵轴的坐标系即图像坐标系，OO_C 则是相机的焦距 f。O_CP 和图像坐标系的交点坐标为 (x, y)。则相机坐标系到图像坐标系可以用相似变换得到。如图:

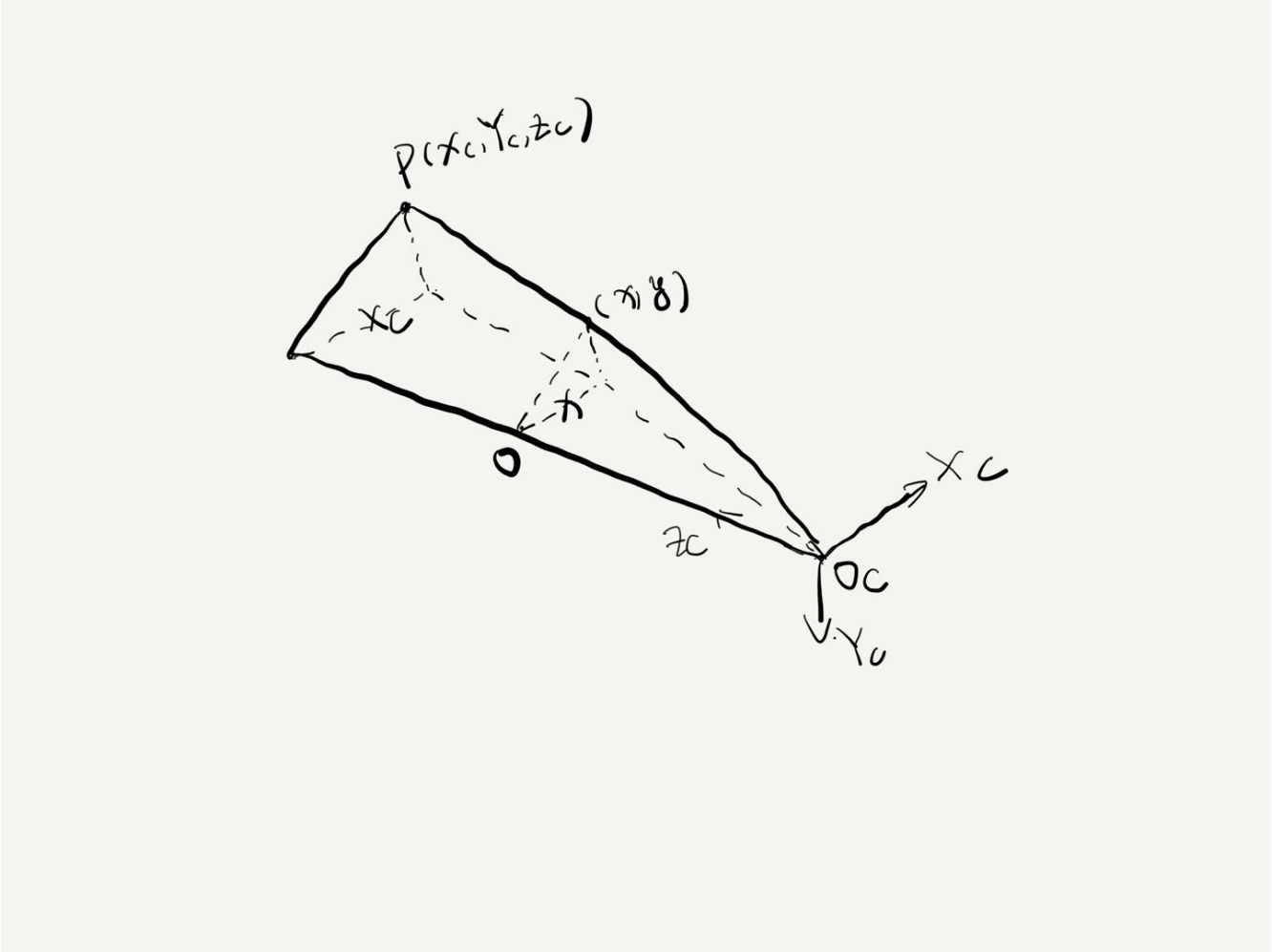

图 3.2: 相机坐标系到图像坐标系 x 轴变换过程

由于 Z_C 轴是和图像坐标系垂直的，则根据相似三角形的变换:

$$\frac{x}{X_C} = \frac{OO_C}{Z_C} = \frac{f_x}{Z_C} \tag{3.3}$$

同理，对 y 轴有 $y = Y_C f_y / Z_C$。则从坐标 $P(X_C, Y_C, Z_C)$ 到 (x, y) 的变换可以用矩阵表示为:

$$\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \begin{bmatrix} f_x & 0 & 0 \\ 0 & f_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X_C \\ Y_C \\ Z_C \end{bmatrix} \tag{3.4}$$

其中 $x = x'/z', y = y'/z'$。但是一般的图像平面和相机坐标的位置没有那么标准，可能会存在一定的偏差，偏差来源有两个，一个是 Z_C 坐标轴可能不是刚好通过图像坐标系的坐标原点 O，两者之间会有一定的偏移，记作 (c_x, c_y)。另一个是图像平面 x 轴和 y 轴可能存在一定的旋转，不一定是垂直的。对第一种情况，在计算的时候需要考虑偏差，则公式3.3变成 $x + c_x = X_C \frac{f_x}{Z_C}$。这里可能是加上偏差也可能是减去，$c_x$ 符号则有后续实际计算决定。第二种情况是模型可以简化:

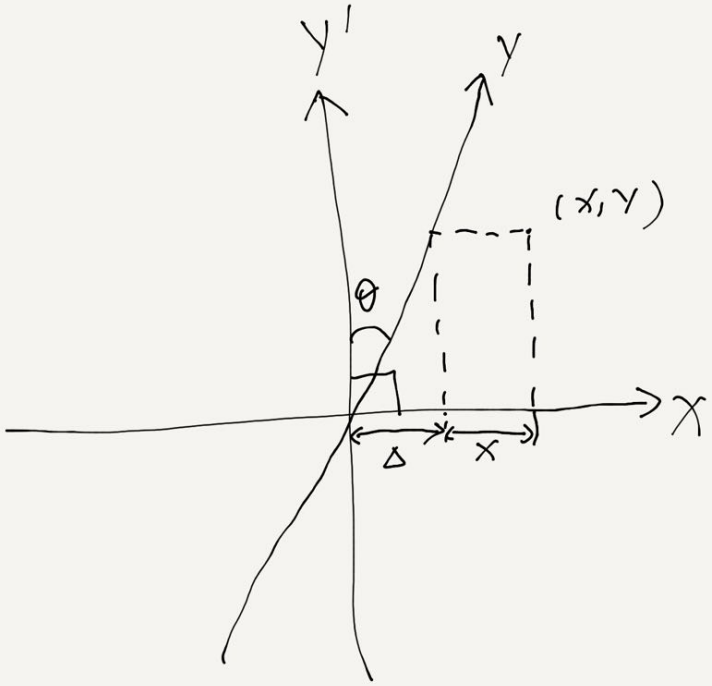

图 3.3: x 和 y 轴存在一定的旋转情况

假设旋转角度为 θ, 则公式3.3变为

$$\begin{aligned} x &= X_C \frac{f_x}{Z_C} + \Delta = X_C \frac{f_x}{Z_C} + \tan(\theta) y \\ &= X_C \frac{f_x}{Z_C} + \tan(\theta) Y_C \frac{f_y}{Z_C} \\ &= f_x \frac{X_C}{Z_C} + \gamma \frac{Y_C}{Z_C} \end{aligned} \tag{3.5}$$

这里因为 θ 和 y 轴的焦距 f_y 一般都是固定的，所以用 γ 来表示这个常数。结合以上变换最终得到相机坐标系到图像坐标系的变换矩阵为:

$$\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \begin{bmatrix} f_x & \gamma & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X_C \\ Y_C \\ Z_C \end{bmatrix} \tag{3.6}$$

定义这个变换的矩阵为内参矩阵 (Intrinsic Matrix) 也称为相机矩阵 (camera matrix)。表示为:

$$K = \begin{bmatrix} f_x & \gamma & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \tag{3.7}$$

其中 f_x 和 f_y 分别表示 x 轴和 y 轴上的摄像机焦距。γ 是 x 轴和 y 轴之间的倾斜偏移量，(c_x, c_y) 是像素坐标中心和相机光学中心轴 Z_C 之间的偏移值。

3.1.3 图像坐标到像素坐标

将图像坐标的坐标原点放在图像的左上角位置则变成了像素坐标，因为一般处理图像时的像素值都是以左上角为原点的。这样的话从 (x, y) 到 (u, v) 像素之间的变换就变成了线性变换:

$$\begin{aligned} u &= x + \frac{W}{2} \\ v &= y + \frac{H}{2} \end{aligned} \tag{3.8}$$

其中 (H, W) 表示图像的长宽，也是定值。这样可以对其变换:

$$u = x + \frac{W}{2} = f_x \frac{X_C}{Z_C} + \gamma \frac{Y_C}{Z_C} + c_x + \frac{W}{2}$$

由于 c_x 和 $W/2$ 都是常数，可以用一下变化表示新的 c_x

$$c_x \triangleq c_x + \frac{W}{2}$$

这里如果图像坐标系的中心点和 Z_C 轴没有偏差，那么 $c_x = 0$, 则最终得到的内参矩阵中的值 $c_x = \frac{W}{2}$。所以一般可以用 $W/2$ 来表示内参值 c_x。

3.1.4 Homography 矩阵

基于以上变化，可以到得到从点 $P(X_W, Y_W, Z_W)$ 到像素 (u, v) 的变换过程：

$$\begin{aligned} \begin{bmatrix} u' \\ v' \\ w' \end{bmatrix} &= \begin{bmatrix} f_x & \gamma & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X_C \\ Y_C \\ Z_C \end{bmatrix} \\ &= \begin{bmatrix} f_x & \gamma & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} [R|t] \begin{bmatrix} X_W \\ Y_W \\ Z_W \end{bmatrix} \end{aligned} \tag{3.9}$$

一般为了计算会将世界坐标系的 Z_W 设为 0[5]。则变换矩阵为:

$$\begin{aligned} s \begin{bmatrix} u' \\ v' \\ w' \end{bmatrix} &= \begin{bmatrix} f_x & \gamma & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} r_1 & r_2 & r_3 & t \end{bmatrix} \begin{bmatrix} X_W \\ Y_W \\ 0 \\ 1 \end{bmatrix} \\ s \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} &= K \begin{bmatrix} r_1 & r_2 & t \end{bmatrix} \begin{bmatrix} X_W \\ Y_W \\ 1 \end{bmatrix} \end{aligned} \tag{3.10}$$

其中 r_i 是旋转矩阵的第 i 列，定义 homography 矩阵为:

$$H = K[R|t] \tag{3.11}$$

对于一般的相机标定，求解内参和外参就可以变换为求解 homography 矩阵的形式。这里如果可以得到真实世界和像素坐标的对应点作为数据，则可以根据矩阵求解得到 homography 矩阵的值。但是一般真实世界的坐标不容易测量，常用的方法是使用相同的一个相机对同一个物体拍摄来获取参数，即自我标定的方法 [1]。根据 [4] 可知三维坐标系点之间的变换有旋转，平移，仿射变换，投影变换等。其中投影变换 (Projective) 表示为:

$$x' = Hx$$

其中 x 和 x' 表示变换前后对应的坐标点。H 被称为透视变换矩阵或者单应性矩阵（perspective transform or homography）。

3.1.5 Homography 估计

如果知道坐标点 (x, y, z) 和其经过变换后对应的坐标点 (x', y', z')。则可以得到 homography 变换公式:

$$\begin{bmatrix} x' \\ y' \\ z'two_plane \end{bmatrix} = \begin{bmatrix} H_{11} & H_{12} & H_{13} \\ H_{21} & H_{22} & H_{23} \\ H_{31} & H_{32} & H_{33} \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} \tag{3.12}$$

展开矩阵：

$$\frac{x'}{z'} = \frac{H_{11}x + H_{12}y + H_{13}z}{H_{31}x + H_{32}y + H_{33}z}$$

一般为了方便计算可以令 $z = 1$，即图像变换的时候 z 轴可以保持一致。则:

$$x'(H_{31}x + H_{32}y + H_{33}) = H_{11}x + H_{12}y + H_{13}$$

$$-xH_{11} - yH_{12} - H_{13} + 0 + 0 + 0 + x'xH_{31} + x'yH_{32} + x'H_{33} = 0$$

同理得到 y'/z' 的变换展开:

$$0 + 0 + 0 - xH_{21} - yH_{22} - H_{23} + y'xH_{31} + y'yH_{32} + y'H_{33} = 0$$

用矩阵来表示上述变换为:

$$\begin{bmatrix} -x & -y & -1 & 0 & 0 & 0 & x'x & x'y & x' \\ 0 & 0 & 0 & -x & -y & -1 & y'x & y'y & y' \end{bmatrix} \begin{bmatrix} H_{11} \\ H_{12} \\ H_{13} \\ H_{21} \\ H_{22} \\ H_{23} \\ H_{31} \\ H_{32} \\ H_{33} \end{bmatrix} = 0 \tag{3.13}$$

即:

$$Ah = 0$$

根据线性代数 [3] 可知这是一个求解线性方程的特征向量问题。求解线性方程即可。常用的方法是将其变成对称矩阵:

$$A^T Ah = A^T 0 = 0$$

然后对其进行 SVD 分解:

$$A^T A = U\Sigma U^T$$

U 是正交矩阵，Σ 是奇异值组成的对角矩阵。

3.1.6 根据 Homography 求解旋转矩阵和平移向量

在相机标定中常用的一种方式是将标定板先选择一个标准位置拍照，然后进行一定的旋转平移拍照。通过几次的图像来计算 Homography 矩阵。在上一节中可得到计算 Homograohy 的方法，那如果图像只有旋转和平移，homography 的分解方法可参考 [2]。

3.2 图像处理

3.2.1 滤波

由于一般拍摄的图像中都含有比较多的噪声，噪声一般是图像上亮度或者颜色值随机变化的信息。比如一个小区域图像像素值都是 0，但是其中一个像素值是 255, 那么这个点就是噪声。平滑 (smooth) 处理一般就是为了尽可能消除这样的噪声区域，让噪声点部分的像素值不那么明显。一般的方法是: 对某一个像素点，求取以这个像素点为中心的局部区域的像素值的加权像素值，然后用新的像素值来代替之前的像素值。这种减小噪声的方法称为滤波 (filter)。常见的滤波方式有均值滤波和高斯滤波，均值滤波是在计算新的像素值的时候将区域内的所有像素值求平均，这种操作一般则可以通过卷积的方式来实现，比如:

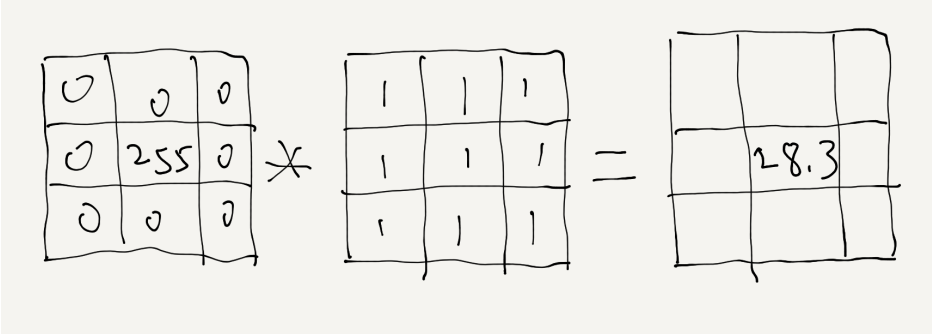

图 3.4: 均值滤波卷积

通过使用一个 $3x3$ 的卷积核对原始图像进行计算，之前的 255 的像素值就变成了 28.3。这样就使得像素值差别没有之前那么明显，整张图像看起开也变得更加平滑。另一种常用的滤波方式是高斯滤波，像均值滤波里使用了全是 1 的卷积核一样，高斯滤波的区别是使用一个具有高斯分布的卷积核。卷积核的目的主要是作为像素的权重值，那么对于某个点我们一般是希望这个点的权重要比周围像素点权重大的，这样可以尽可能的保留像素的原始信息。而高斯分布则正是中心点概率值大，周围概率值小，所以高斯滤波具有更好的效果。由于图像是二维的，所以一般需要二维高斯分布函数生产一个二维的卷积核。

Chapter 4

第四章 深度学习

本章主要叙述深度学习及机器学习基础及部分网络结构。

4.1 机器学习

4.1.1 K-MEAN

4.2 分类模型

4.3 检测模型

4.4 分割模型

Chapter 5

第五章 算法

5.1 复杂度计算

时间复杂度可表示为: $O(1), O(n), O(logn)$ 等，表示完成计算需要进行的次数。

```
std::unordered_map<int, int> map;
int value = map[0];    // O(1)-> hash map get outputs only once

for (int i = 0; i < n; ++i) {
 // ... for loop n -> O(n)
}

// O(logn)
std::vector<int> nums{1, 2, 3, 4, 5, 6, 7, 8};
// if search for 2
// first dichotomy: 1, 2, 3, 4, size = 8 * (1/2)  = 4
// second dichotomy: 1, 2, siz = 8 * (1/2) * (1/2) = 2
// third dichotomy: 2,  size = 8 * (1/2) * (1/2) * (1/2) = 1
```

$O(logn)$ 是比较不容易理解的，如代码一个数组使用二分法查找某一个数字，长度为 8 的数字需要进行三次二分才可以找到对应的值。那么假设有 n 个数字，需要经过 k 次二分可以找到值。则有:$n * (1/2)^k = 1$, 即 k 次查找之后数组长度是 1。log 变化之后:

$$k = log_2(n) \tag{5.1}$$

这称为对数时间，即为 $O(logn)$.(这里统一记作 logn 而不用管 log 的底是多少，因为根据换底公式，不同的底数只是一个常系数的差别)。比如长度为 8 带入公式得到 k 是 3，即最多进行三次运算就可以找到对应的值了。

5.2 排序算法

5.2.1 拓扑排序

对一个有向无环图，可以通过拓扑排序获取图上每个节点的执行顺序。假设对一个节点定义如下:

```
struct Node {
 public:
  Node(std::string name) : name_(name) {}
  void add(Node* node) {
        ++(node->in_degree);
        next_nodes.emplace_back(node);
  }
  int in_degree = 0;
  std::vector<Node*> next_nodes;

  std::string name_;
};
```

节点包含一下信息: 入度 (in_degree) 表示当前节点的输入节点个数。next_nodes 表示当前节点的所有输出节点。拓扑排序的思想为：当一个节点的入度是 0 的时候则可以执行当前节点（将节点放入 queue 中），同时可以从图上删除当前节点，并且将节点的所有 next_nodes 的入度都减去 1。同样的方法遍历完所有的节点即可得到一个可执行队列。

5.2.2 快速排序

快速排序方法: 先选一个基准数，然后从数组尾地址向前，如果值小于基准值，则将值移动到基准前面。同时从开始位置向后移动，如果值大于基准值，则将值移动到基准值后面。这样一

次排序得到了基准值在数组中的位置，然后使用分治加上递归思想。分别对基准位置前面的数组和后面的数组再次进行排序。调用递归函数直到起始位置等于终止位置。

```
void quick_sort(std::vector<int>& nums, int start, int end) {
  if (end < start) return;
        int base = nums[start];
        int index = start;

        int i = start;
        int j = end;
        while (i < j) {
                while (i < j && nums[j] >= base) {
                        j--;
                }
                nums[i] = nums[j];    // set nums[j] before base, set it to index i
                while (i < j && nums[i] <= base) {
                        i++;
                }
                nums[j] = nums[i];  // set nums[i] after base, set it to index j
                nums[i] = base;    // i == j, set nums[i] == base, set base in middle
        }
        quick_sort(nums, start, i - 1);
        quick_sort(nums, i + 1, end);
}
```

5.3 Graph

5.3.1 DFS and BFS

Depth First Search Algorithm(DFS)，深度优先搜索算法: 从某一个节点开始，沿着一条搜索路径一直走到底，如果不满足条件然后从这条路尽头的节点回退到上一个节点，再从另一条路开始走到底。依次按照这种规律递归所有的节点。Breadth-First Search (BFS) 广度优先搜索，从根节点开始，分别搜索左右节点，左右节点不满足条件在分别搜索左节点的子节点和右节点的子节点，依次遍历所有节点。对输入节点:

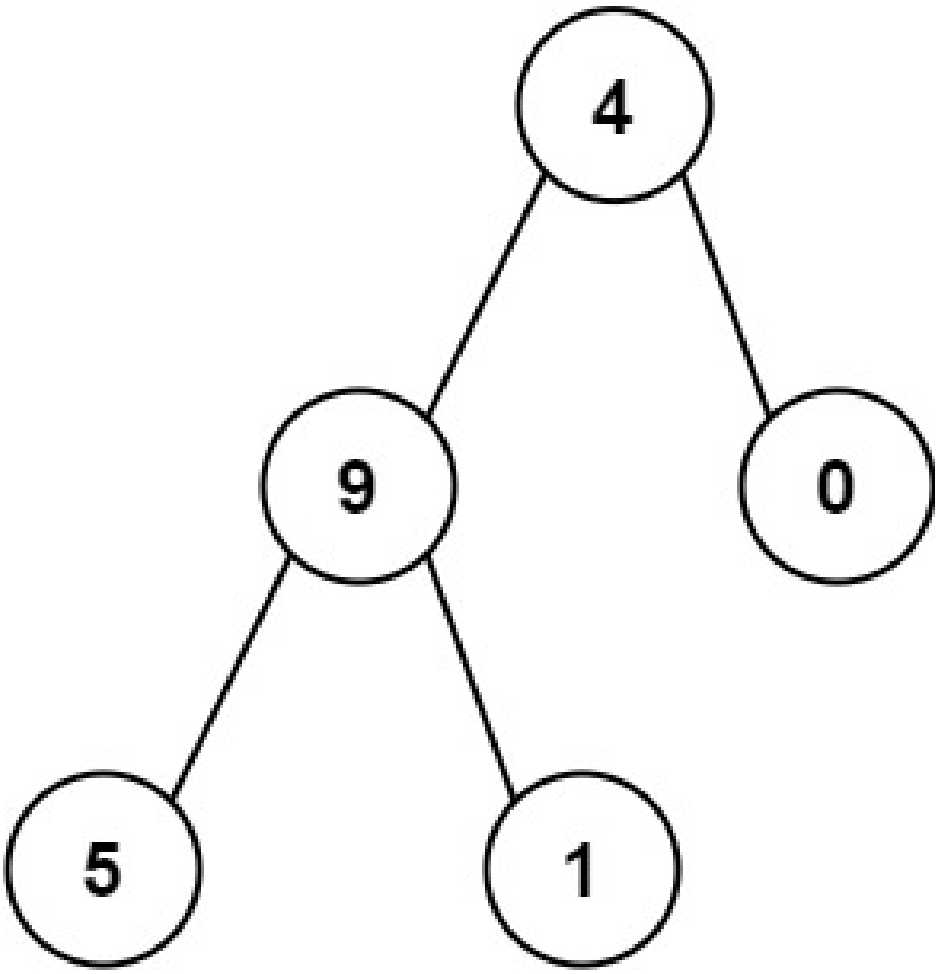

图 5.1: node-tree

dfs 和 bfs 算法如下:

```
#include <iostream>
#include <vector>
#include <queue>

struct TreeNode {
        int val;
        TreeNode *left;
        TreeNode *right;
```

```
        TreeNode() : val(0), left(nullptr), right(nullptr) {}
        TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
        TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left), right(right) {
};

void dfs(TreeNode* root) {
        if(nullptr == root) return;
        dfs(root->left);
        std::cout << root->val << " -> ";
        dfs(root->right);
}

void bfs(TreeNode* root) {
        if (root == NULL)
        return;
        std::queue<TreeNode*> q;
        q.push(root);
        while (!q.empty()) {
                TreeNode* node = q.front();
                std::cout << node->val << " -> ";
                q.pop();

                if (nullptr != node->left) q.push(node->left);
                if (nullptr != node->right) q.push(node->right);
        }
}


int print(TreeNode* root) {
        dfs(root);
        std::cout << "\n";

        bfs(root);
        std::cout << "\n";
}
```

```
int main() {
        TreeNode n0(0);
        TreeNode n1(1);
        TreeNode n5(5);
        TreeNode n9(9, &n5, &n1);
        TreeNode n4(4, &n9, &n0);
        int sum = print(&n4);
}
```

dfs 输出节点: $5 \to 9 \to 1 \to 4 \to 0$

bfs 输出节点: $4 \to 9 \to 0 \to 5 \to 1$

Chapter 6

第六章 计算机基础

6.1 基础概念

6.1.1 计算机体系结构

查看系统版本:

```
root@docker-desktop:/workspace/cleetcode# lscpu
Architecture:                    x86_64
CPU op-mode(s):                  32-bit, 64-bit
Byte Order:                      Little Endian
Address sizes:                   39 bits physical, 48 bits virtual
CPU(s):                          16
On-line CPU(s) list:             0-15
Thread(s) per core:              2
Core(s) per socket:              8
Socket(s):                       1
Vendor ID:                       GenuineIntel
CPU family:                      6
Model:                           165
Model name:                      Intel(R) Core(TM) i7-10870H CPU @ 2.20GHz
Stepping:                        2
CPU MHz:                         2208.008
BogoMIPS:                        4416.01
Virtualization:                  VT-x
Hypervisor vendor:               Microsoft
Virtualization type:             full
L1d cache:                       256 KiB
L1i cache:                       256 KiB
L2 cache:                        2 MiB
L3 cache:                        16 MiB
```

图 6.1: 计算机 Architecture

参数解析:

- Architecture: x86_64 表示是 64 位系统，x86_32 表示 32 位系统。
- CPU op-mode: 表示 Linux 在运行什么系统的版本,64 位系统支持 32 以及 64 位的系统程序。
- CPU: 一共有多少个 CPU

6.1.2 程序编译原理

程序编译步骤包括:

$$\text{源文件} \rightarrow \text{预处理器} \rightarrow \text{编译器} \rightarrow \text{连接器} \rightarrow \text{目标机器代码}$$

预处理器: 完成将多个源文件汇聚在一起的任务，主要包括将头文件代码 copy 到源文件中，将对应的宏进去扩展开。

编译器: 完成预处理后文件到汇编语言的转换。(编译器包括语法分析、代码生成等步骤)
汇编器: 将汇编程序生成可重定位的机器代码。
链接器: 将多个编译文件链接在一起。

6.1.3 BandWidth

6.1.4 内存

计算机中存储速度从快到慢依次是: 寄存器-> 缓存-> 主存-> 磁盘存储器。CPU 和 GPU 的主存都采用的是 DRAM(动态随机存取存储器)。

6.2 常用命令工具

6.2.1 Windows

6.2.2 Ubuntu

创建虚拟环境

```
# virtualenv
which python3.7
virtualenv venv -p /usr/bin/python3.7
```

Chapter 7

第七章 c++

7.1 基础概念

7.1.1 const

const 用来修饰变量或者函数为不可变的，但在不同地方意思不同:

```
int* const ptr = new int[4];  //  pointer const
const int* ptr;  //  const pointer
```

指针常量: 指针指向的地址不能被改变，即指针对象不能被修改。但是指针指向的内容是可以修改的。并且可以 cast 成 void*, 注意这里 cast 之后指针地址还是没有被改变的，只是类型变成 void* 了。指针变量中 const 修饰的是变量值，即这个变量值不能被修改，但是前面的指针是可以被操作的。指针常量在定义时必须被初始化，所以一般在 class 中如果有指针常量则必须在构造的时候使用初始化列表来初始化指针常量:

```
int* const ptr = new int[4];
// valid:
ptr[0] = 1;
void* void_ptr = static_cast<void*>(ptr);

// invalid
int* new_data = new int[4]; // invalid if set new data to ptr, ptr = new_data;
```

```
// in class
class A {
 public:
  A(int* data) : data_(data) {}
  //A() {
  //  data_ = new int[1];  // invalid init in here
  // }

 private:
  int* const data_;  // must init in initialize list
};
```

常量指针: 返回的指针不能被修改，比如 cast 成 void* 等。并且指针指向的值也是不可以修改的。常量指针定义时可以不用初始化。但是同一个指针变量可以指向新的指针位置，因为常量指针修饰的时指针地址 (int*)，即不允许对这个指针地址操作。但是指针变量 (ptr) 是可以被改变的。

```
// valid
const int* ptr;
ptr = new int[4];
ptr = new int[10];  // valid to change to pointr address for ptr;

// invalid
// ptr[0] = 1;  // invalid
// void* void_ptr = static_cast<void*>(ptr);  // invalid to cast it
```

7.1.2 字符

c++ 中单引号表示字符，双引号表示字符串。

```
std::string a = "a";
char b = 'b';
// if ('a' == a) // invalid, for a is tring, but 'a' is const char
std::cout << typeid(a).name() << "␣" << typeid(b).name() << "␣" << typeid(a[0]).name() << "\n
// output is: OKNSt7__cxx1112basic_stringIcSt11char_traitsIcESaIcEEE c c
```

```
// first a is string type, b and a[0] is c(char).
```

7.1.3 类型别名

类型别名 (type alias) 可以使用关键字 typedef 或者 using，用来表示将某个类型重命名。

```
typedef type new_name;
using new_name = type;
```

7.1.4 typename

在模板声明的模板参数列表中，typename 可以替代 class 来声明类型模板参数和模板模板参数。

```
template <typename T>
```

在模板内，如果要使用一个依赖于模板参数的类型，则需要用 typename 来说明，参考 c++ 标准:A name used in a template declaration or definition and that is dependent on a template-parameter is assumed not to name a type unless the applicable name lookup finds a type name or the name is qualified by the keyword typename。

```
template <typename T>
struct A {
  typedef typename T::Type type;
};
```

这里要使用模板参数 T 的一个类型 Type 并用 typedef 将其命名为 type。但是由于是一个依赖类型 (Dependent names)，即 Type 依赖传入的模板参数 T。但是 T 是一个未知的类型，所以是不清楚 Type 的具体含义的，比如是类型还是静态成员变量什么的。所以需要加上 typename 来修饰让编译清楚这是一个数据类型，从而编译时才不会出错。

7.1.5 函数原型

函数原型 (function prototype) 是函数的声明，它告诉程序函数返回值的类型以及参数的数量和类型。函数类型需要在头文件中进行定义然后在 cpp 文件中实现函数定义。函数原型中的名字是可选的可以省略，即参数的名字是可以省略的。

```
void fun(int a);
void fun(int);
void fun(int a = 1);
```

7.1.6 namespace

```
namespace {}
```

7.1.7 decltype

7.1.8 extern

7.1.9 volatile

volatile 关键字表示修饰的对象可能会被改变，不需要编译器进行优化。比如:

```
int value = 12581;

while(value == 12581)
{
  // ...
}
```

这种情况编译器会将其优化成 while(true) 的形式，但是实际上 value 可能会被其它编译器感知不到的地方所改变其值。这种情况就不希望编译器优化 value。则可以通过添加 volatile 关键字来告诉编译器不要优化修饰的对象。

7.1.10 函数指针

7.1.11 noexcept

函数

7.1.12 restrict

7.1.13 value initialization

7.1.14 预处理命令

#pragma

7.2 内存管理

7.2.1 基本内存概念

计算机以 bit 序列来存储数据，每个 bit(比特) 表示非 0 即 1。大多数计算机以 2 的整数次幂个 bit 作为块来处理内存，可寻址的最小内存块称为” 字节”(byte), 一个字节含有 8bit 的内存大小，计算机中将内存中每个字节和一个数字 (称为地址，address) 进行关联。1 byte = 8 bit, 1 K = 1024 byte, 1 M = 1024 K, 1 G = 1024 M, 1 KB = 1000 byte。c++ 中所有的数据类型都是以字节为基础单位的，常用数据类型字节大小: bool(1 字节), char(1 字节), int(4 字节), double(8 字节)。

7.2.2 const

7.2.3 heap

7.2.4 stack

7.2.5 Free Store

7.2.6 Global/Static

7.3 class

```
class derived_class : final base_class {
 public:
```

```
  derived_class() = default;
  derived_class & operator =( const derived_class & ) = delete;

  virtual void methods() final;  //
};
```

7.3.1 default

类中的 default 关键字表示应该默认使用 default 部分定义的函数方法。当在 class 中定义某个方法为 default 属性时，编译器会自动为该方法创建一个对应的函数体实例化，不需要手动的来实现对应的函数内容，这在部分构造函数定义中十分有用，比如需要定义一套拷贝构造和赋值构造函数，一般的拷贝构造时比较麻烦的需要处理类里所有的成员变量等，但是使用 default 的话就可以让编译器默认来实现。

```
class A {
 public:
  A(const A& rhs) = default;  // copy construct
  bool operator ==( const & A ) = default;
  bool operator !=( const & A ) = default;

 private:
  int data_;
};

// default same as:
A(const A& rhs) : data_(rhs.data_) {}

A aa;
auto a = aa;  // copy construct and initialize class a
```

拷贝构造是使用其他对象的值来初始化当前对象。注意是初始化类的时候对应等号左边的类是在拷贝构造的时候完成初始化的。赋值构造则是直接用一个新的对象来替换当前对象, 赋值构造在调用的时候等号前后两个类都已经初始化完了，赋值构造完成赋值替换。

```
class A {
```

```
 public:
  A& operator=(const A& rhs) = default;  // assignment construct

 private:
  int data_;
};

// default same as:
A& operator=(const A& rhs) {data_ = rhs.data_; return *this;}

A aa, a;  // a and aa already initialized.
a = aa;  // assignment construct
```

7.3.2 delete

类中使用 delete 修饰表示禁用对应的方法，编译器也不会自动实现对应的方法。使用 delete 的好处是在编译期就可以对代码进行规范性检查，如果有其他地方调用了 delete 的方法则会在编译器出错而不是到运行时出错。delete 常用的是禁用拷贝构造:

```
struct type {
 type & operator =( const type & ) = delete;
 type( const type & ) = delete;
};
```

使用 delete 完成模板的特化:

```
// Delete primary class template, allow specializations
template<typename>
struct s = delete;
template<>
struct s<int> {
};

// Delete class specialization
template<typename>
struct t {
```

```
};

template<>
struct t<int> = delete;
```

使用 delete 来修饰析构函数，这种情况其实有点怪，因为一单析构函数被 delete 关键字修饰则对应的类不能够被析构掉。修饰析构函数一般是为了防止类中有申请的自由存储的内存等没有被释放的情况。比如:

```
struct data {
  //...
};

struct data_protected {
 ~data_protected() = delete;
 data d;
};
struct data_factory {
  ~data_factory() {
  for (data* d : data_container) {
        // this is safe, because no one can call 'delete' on d
        delete d;
        }
  }

  data_protected* createData() {
        data* d = new data();
        data_container.push_back(d);
        return (data_protected*)d;
  }

  std::vector<data*> data_container;
};
```

这里对结构体 data_protected，里面的 data 属性由于是 new 出来的，我们希望在特定的情况外部控制其释放。这样的话就最好禁用 data_protedted 的析构函数使其一致存成，否则可

能无法访问到对应的 data 成员。所以一般如果希望外部用户来控制对象的生命周期的话可以用 delete。但是坏处是对应的类会一直存在。

7.3.3 final

7.3.4 explicit

7.3.5 virtual

7.3.6 虚函数表

7.3.7 虚析构函数

虚析构函数是为了解决基类的指针指向派生类对象，并用基类的指针删除派生类对象。如果基类析构函数不是 virtual 的，则无法通过基类指针删除派生类对象。

```
class Base {
        public:
        Base() { std::cout << "construct␣base␣class.\n"; };
        virtual ~Base() { std::cout << "destroy␣base␣class\n"; }
};
class Derived : public Base {
        public:
        Derived() { std::cout << "construct␣derived␣class.\n"; }
        ~Derived() { std::cout << "destroy␣derived␣class\n"; }
};

int main() {
        Base* method = new Derived();
        delete method;
        return 0;
}
// output:
// construct base class.
// construct derived class.
```

```
// destroy derived class
// destroy base class

// if Base destroy is not virtual, output is:
// construct base class.
// construct derived class.
// destroy base class
```

可以看出构造是先构造基类在构造派生类，析构是先析构派生类在析构基类。但是如果将基类析构函数设置为不是虚函数, 则派生类析构函数不会被调用，即派生类不会进行析构，从而导致内存泄露等问题。

7.4 模板

template 代码的位置不能在.cpp/.cc 之中,Templated code implementation should never be in a .cpp file: your compiler has to see them at the same time as it sees the code that calls them. 一般都是放在.h 文件中. 如果希望放在.cpp 文件中要对 class 进行 explicit instantiation, 对类模板 A 在.cpp 最后加上:template class A; 来实现显示生成 object. 不能在 class 中特化模板；模板的定义一般是放在.h 中, 但是对该模板的特化定义要放在.cpp 函数中. 否则会有重定义的问题。模板只是代码的简化和宏展开一个效果，编译器会对每个类型都增加一个实现。不要在 template 中对某个特殊类型进行不同逻辑的处理，如果需要这样请特化。

7.4.1 可选模板参数

分析:

```
template <typename T>
struct FUN {
        typedef int Type;
        ...
}

template <typename T, typename = typename FUN<T>::Type>
void fun();
```

模板 fun 有两个模板参数一个是 T 另一个是 typename = typename FUN<T>::Type, 根据 typename 的 Dependent names 含义可以将 fun 简化为:

```
template <typename T, typename = int>
void fun();
```

这里 typename = 表示 declaring template type parameters, 即需要定义一个模板变量。类模板定义的时候是可以指定一个默认值的。同时由于函数声明的时候函数名字可以省略，所以模板的第二个参数即定义了一个参数名字省略的 int 类型值。可以等价于:

```
template <typename T, typename Type = int>
void fun();

template <typename T, int>
void fun();
```

这里用 Type 表示省略的名字，实际上相当于实现了一个偏特化的类，第二个模板参数是 int。

7.4.2 可变参数模板

7.4.3 Traits

7.5 多线程并发

7.6 std

7.6.1 std::vector

```
std::vector<int> nums1{};
std::vector<int> nums2{0, 1, 2, 3};
std::vector<int> nums3{0, 1, 2, 3, 4, 5};

size_t nums1_size = sizeof(nums1);   // 24
size_t nums2_size = sizeof(nums2);   // 24
size_t nums3_size = sizeof(nums3);   // 24
```

这里对 std::vector 求 sizeof 大小都是 24 个字节，这个大小不是 vector 存储的内容的大小，二是 vector 这个结构体存储的大小，24 字节分别存储了三个指针: begin of vector, end if vector and end of reserved memory for vector。在 64 位系统上用 8 字节大小存储指针，比如:

```
sizeof(int*) == sizeof(float*) == 8;
```

所以 vector 大小就是 3*8=24 字节。std::vector 多了一个指向预分配内存的指针，如下:

```
std::vector<int> nums;
nums.reserve(10);
nums[0] = 0;
std::cout << nums.size() << "␣" << nums.capacity() << "\n";    // 0, 10
nums.emplace_back(1);
std::cout << nums.size() << "␣" << nums.capacity() << "\n";    // 1, 10
nums.reserve(20);
std::cout << nums.size() << "␣" << nums.capacity() << "\n";    // 1, 20

int* data = nums.data();  // vector to data pointer
```

reserve 会预分配给 vector 一个内存空间，会影响 vector 的 capacity(最大可容纳元素的个数)，但是不会影响 size, 只有 push_back, emplace_back 会影响 size 大小。所以除了存储首地址和尾地址之外还需要一个指向其最大 capacity 处的地址指针。比如上面的 nums，vector.begin() 指向下标为 0 的元素，vector.end() 是 vector.begin() + vector.size(), 指向末尾元素 +1 的位置，这里就是指向下标为 1 的元素 (只有一个元素)。而剩下 nums 其实还有 19 个存放元素的内存大小。所以如果没有使用 reserve 预先分配内存，则 vector 每次 push_back 元素的时候都会改变 vector 的申请内存的大小，会重复的释放和重新申请内存导致效率降低。如果 reserve 了一个 size，但是实际使用的时候 push 的个数大于预先分配的尺寸，则 vector 会重新分配一个 2 倍 reserve 大小的内存大小。

```
std::vector<int> nums;
nums.reserve(10);
nums.emplace_back(1);
std::cout << nums.size() << "␣" << nums.capacity() << "\n";    // 1, 10
for (int i = 0; i < 15; ++i) {
  nums.emplace_back(i);
}
std::cout << nums.size() << "␣" << nums.capacity() << "\n";    // 16, 20
```

如上,vector 的 capacity 在 push 个数大于 10 个之后 capacity 的值从 10 变成了 20。

7.6.2 std::forward and std::move

std::forward

当输入是一个 lvalue 时, 匹配到如下模板。remove_reference_t 会删除所有的 reference，则输入相当于是 _Ty& _Arg。然后经过 Reference collapsing(引用折叠)_Ty&& &_Arg 之后返回还是一个 lvalue 形式。

```
template<class _Ty>
_NODISCARD constexpr _Ty&& forward(remove_reference_t<_Ty>& _Arg) noexcept
{       // forward an lvalue as either an lvalue or an rvalue
        return (static_cast<_Ty&&>(_Arg));
}
```

Reference collapsing 规则是偶数引用可删除:

- T&& && → T&&
- T& && → T&
- T& & → T&
- T&& & → T&

当输入是一个右值时，

```
template<class _Ty>
_NODISCARD constexpr _Ty&& forward(remove_reference_t<_Ty>&& _Arg) noexcept
{       // forward an rvalue as an rvalue
        static_assert(!is_lvalue_reference_v<_Ty>, "bad forward call");
        return (static_cast<_Ty&&>(_Arg));
}
```

7.6.3 std::enable_if

7.6.4 std::find_if

7.6.5 std::map 和 std::unordered_map

std::unordered_map 无序数组主要是通过 hashmap 来实现的，通过数组和链表 hashmap 定义可表示为:

```
template <typename _Key, typename _Value>
struct HashNode {
        _Key key_;
        _Value value_;
        HashNode* next_;
};

template <typename _Key, typename _Value, class HashFun, class EqualFun>
struct HashMap {
 private:
  HashNode<_K, _Value>** hash_table;
  unsigned int size_;   // total hash map(hash_table) size
};
```

HashMap 模板函数包括: _Key 一般是一个 hashcode 用来表示唯一的 id 信息，也可以是整数,string 等。_Value 则表示需要存储的数据结构。hashmap 的数据都存储在一个 hash_table 中，hash_table 可以看出一个 array 或者 vector。具有一定的大小 (size_)，每个元素存储了一个链表的首节点地址指针。则对于任意一个 map 中的元素则可以通过整数索引来获取。那么如何获取这个整数索引呢? 这就需要输入一个 HashFun 将 _Key 转换成一个整数 id，比如用下面函数:

```
static int indexFor(int key, int max_size) {
  return key & (max_size - 1);
}
```

这里如果 key 是一个 int 值，max_size 表示 hash_table 的长度 size_, 则通过求余数将一个很大的整形数或者 hash 值转换成到 0 到 size_ 之间的一个整数 id。然后用 id 来从 hash_table 中索引具体的节点 HashNode。HashNode 主要包含三个成员变量，key 是 hashcode 值，表示当前

node 节点的唯一 id 值。value 表示存储的整数数据结构体，next 指针指向下一个 HashNode 节点。如果 hash 值映射的 id 都是唯一的，那么 hash_table 的 value 就是每一个 node 节点的地址指针。但是这里就会出现可能会出现多个不同的 hash 值映射成了一个索引 id（hash 冲突），因为没有办法将 size_ 设置的很大来容纳所有的 hash 值，所以如果对于同一个 id, 通过 next 指针指向下一个 HashNode 的形式多个 node 之间形成一个单链表，这些 node 的共同特点是其在 hash_table 中的整数 id 都是同一个值，然后在基于链表逐个 node 去比较 node 中的 key_ 值和 hashcode 是否一样，一样则说明查找到了对应的数值，如果找不到说明是一个新的值，则基于输入的 key 和 value 新建一个 node 并插入到链表头中（因为 hashtable 中存储的是表头的地址指针）。HashMap 通过数组和单链表的形式，获取到 id 之后只要最多遍历一遍链表就可以查找到对应的 value，如果没有 hash 冲突，算法复杂度则为 O(1)，如果有则算法复杂度 O(n)。另外当元素过少可以用单链表形式表示，但是元素过多的话则需要用红黑树来表示存储形式。

7.6.6 std::queue and std::stack

queue(队列) 原则是元素先进先出，stack(栈) 元素是后进先出。queue.pop() 移除的是第一个进 queue 的元素，stack.pop() 移除的是最后一个进 stack 的元素。

```
std::queue<int> queue;
queue.push(0);
queue.push(1);
int first = queue.front();  // 0
queue.pop();  // remove first element

std::stack<int> stack;
stack.push(0);
stack.push(1);
int top = stack.top();   // 1
stack.pop();  // remove the top element
```

7.6.7 std::sort

std::sort 可实现迭代排序功能，默认升序排序，也可以传入 lambda 函数来指定排序规则。sort 会直接改变输入的迭代器里的值。当传入一个 lambda 排序函数时，lambda 的两个输入表

示比较迭代项的值，比如 begin() 指向的值。lambda 必须返回一个 bool 值表示比较条件，如下 a[1] < b[1] 表示按照 vector 第 2 个元素升序排序，如果 a 的第二个值小 a 排在前面。

```
#include <algorithm>  // std::sort

std::vector<int> vec1{2, 1, 3};
std::vector<std::vector<int>> vec2{{2, 1}, {1, 3}, {3, 0}};
std::sort(vec1.begin(), vec1.end());   // vec1: {1, 2, 3}
std::sort(vec2.begin(), vec2.end());   // vec2: {{1, 3}, {2, 1}, {3, 0}}, sort for first elem
std::sort(vec2.begin(), vec2.end(), [&](std::vector<int>& a, std::vector<int>& b) { return a[
// {{3, 0}, {2, 1}, {1, 3}}
```

7.7 cmake

7.7.1 cmake demo

7.8 设计模式

Chapter 8

第八章 CUDA

8.1 基础概念

8.1.1 Hello Add

cuda 编程的思想是并行多线程同时在 cuda 核上运行，一个 cuda 的加法 demo 如下:

```
// cuda_add.cu
#include <cuda_runtime.h>

__global__ void cuda_add(float* A, float* B, float* C)
{
  int block_idx = blockIdx.x;
  int thread_idx = threadIdx.x;
  int blockdim = blockDim.x;
  int idx = block_idx * blockdim + thread_idx;
  C[idx] = A[idx] + B[idx];
}

int main()
{
  #define BLOCK_SIZE 64
```

```
  int N = 1000;
  size_t nBytes = N * sizeof(float);

  float *h_A, *h_B, *h_C;
  h_A = (float*)malloc(nBytes);
  h_B = (float*)malloc(nBytes);
  h_C = (float*)malloc(nBytes);
  // ...

  float *d_A, *d_B, *d_C;
  cudaMalloc((float**)&d_A, nBytes);
  cudaMalloc((float**)&d_B, nBytes);
  cudaMalloc((float**)&d_C, nBytes);

  cudaMemcpy(d_A, h_A, nBytes, cudaMemcpyHostToDevice);
  cudaMemcpy(d_B, h_B, nBytes, cudaMemcpyHostToDevice);

  cudaSetDevice(0);
  dim3 dimBlock(BLOCK_SIZE);
  dim3 dimGrid((N + BLOCK_SIZE - 1)/ BLOCK_SIZE);
  cuda_add<<<dimGrid, dimBlock>>>(d_A, d_B, d_C);
  cudaDeviceSynchronize();  // wait gpu compute end
  cudaMemcpy(h_C, d_C, nBytes, cudaMemcpyDeviceToHost);

  free(h_A);
  free(h_B);
  free(h_C);

  cudaFree(d_A);
  cudaFree(d_B);
  cudaFree(d_C);
}
// compiler:
// nvcc cuda_add.cu -o cuda_add
```

一个 cuda 核用 ___global___ 关键字来定义，cuda_add 执行通过 <<< ... >>> 操作符来定义，

dimGrid 和 dimBlock 定义了具体的线程数配置。一组的线程组成了 cuda block, cuda block 组成了 cuda grid。一个二位 cuda grid 可视化:

- tx = threadIdx.x
- ty = threadIdx.y
- bx = blockIdx.x
- by = blockIdx.y
- bw = blockDim.x
- bh = blockDim.y
- id_x = tx + bx * bw
- id_y = ty + by * bh

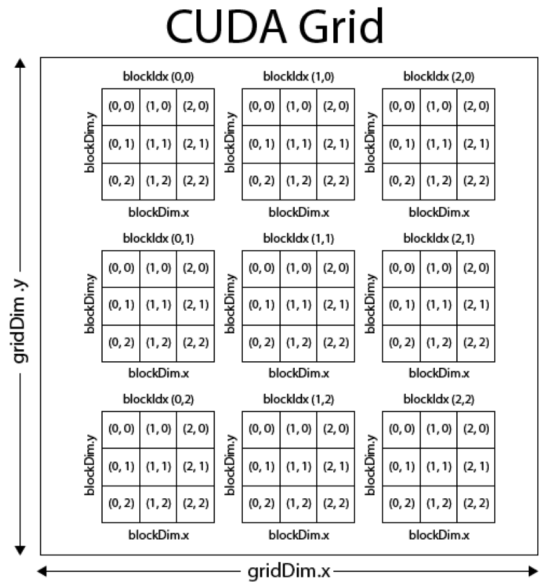

图 8.1: cuda-grid

如 cuda_add 中定义 BLOCK_SIZE 表示一个 cuda block 的大小是多少。在定义 dimGrid 和 dimBlock 的时候其实都是 dim3 是 3 个维度的，即 dimx, dimy, dimz。当不指定其他两个维度，大小默认是 1。所以 $dimBlock(BLOCK\_SIZE)$; 相当于 $dimBlock(BLOCK\_SIZE, 1, 1)$;, 即 blockDim.x = BLOCK_SIZE, blockDim.y == blockDim.z == 1。同理 dimGrid 为:

$$dimGrid = (N + BLOCK_SIZE - 1)/BLOCK_SIZE; \tag{8.1}$$

这种就发可以保证 dimGrid * dimBlock 可以将所有的输入数据 N 都包含在内，比如 N 是 1000, dimBlock 是 64, 则 dimGrid 是 16。这里只考虑 x 一个维度，即使用 16 个 cuda block(gridDim.x = 16), 每个 cuda block 的大小是 64(blockDim.x = 64)。所以一共有 1024 个线程在同时计算，每个线程的索引可以通过:

$$int\ idx = blockIdx.x * blockDim.x + threadIdx.x; \tag{8.2}$$

其中 blockIdx.x,blockDim.x,threadIdx.x 都是 cuda 函数内置的一些变量，表示当前线程核函数的信息: blockIdx.x 表示当前线程属于哪一个 block，范围在 0 到 gridDim.x-1 之间。blockDim.x：

一个常数通过表示 x 维度一个 block 包含了多少个线程。threadIdx.x: 当前 block 内这个 cuda kernel 的线程索引是多少，范围属于 0 到 blockDim.x-1。然后这里每个线程进行两个数的相加操作。那么 cuda kernel 和设备对应关系是什么呢？如下图对于 GPU 设备是由很多个 Multithreaded Streaming Multiprocessors (SMs) 组成的，图中一个绿色格表示一个 core core, 多个 cuda core 组成了一个 SM。这里一个 SM 包含 32 个 cuda core，一个 GPU 设备含有 8 个 SMs。

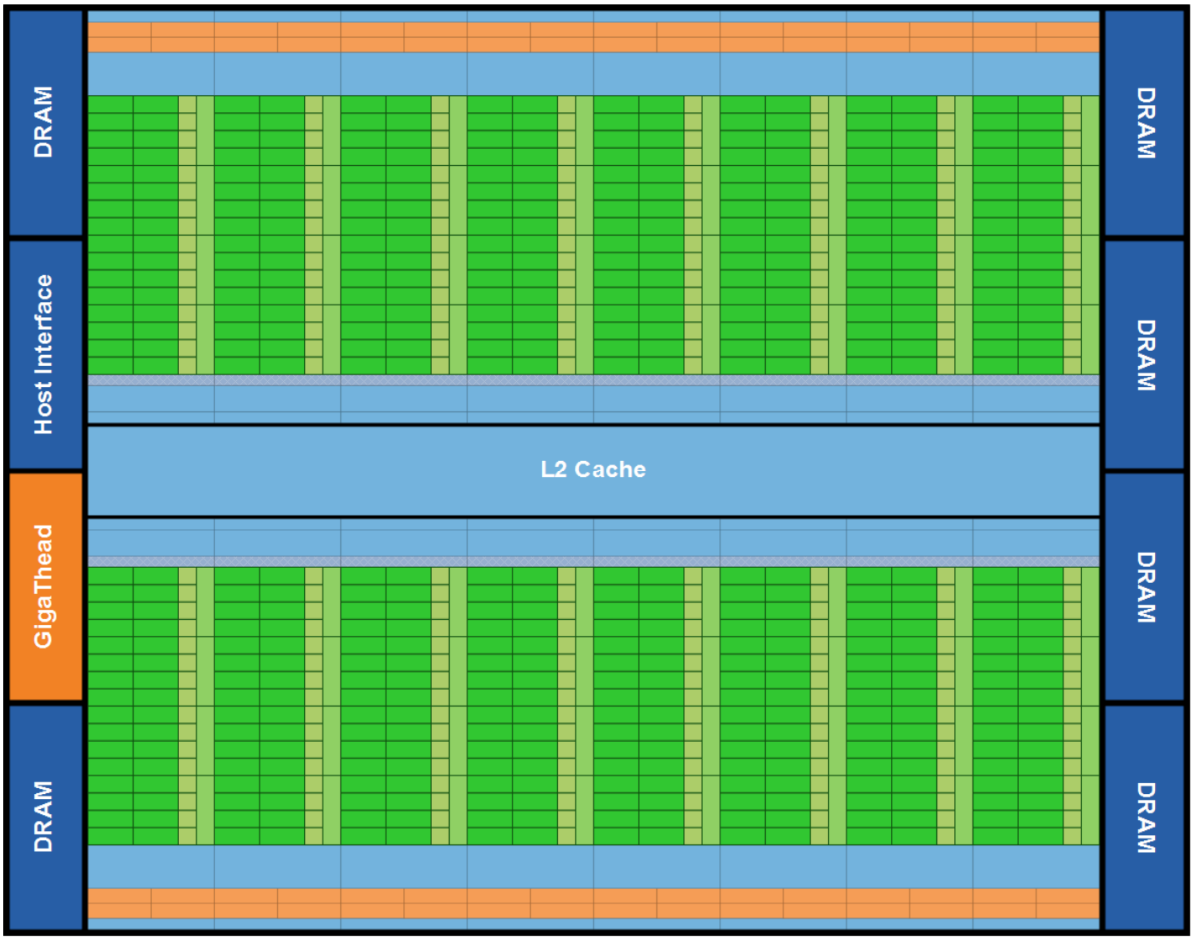

图 8.2: GPU-Architecture

一个 SM 的结构如下,cuda core (也称为 Stream Processors (SP)) 最基本的计算单元，一个 core core 具有浮点计算单元和整形计算单元。当一个 cuda kernel 被执行时，会根据线程数目将 cuda grid 分配到不同的 SM 上去执行，但是对于一个 cuda block 来说只能在一个 SM 上执行。

一个 SM 的 CUDA core 会分成几个 warp（线程束），即 CUDA core 在 SM 中分组，一个 warp 中的线程必然在同一个 block 中，由于 warp 的大小一般为 32，所以 block 所含的 thread 的大小一般要设置为 32 的倍数。一个 cuda core 可以运行 16 个线程，两个 cuda core 就可以组成一个线程束。

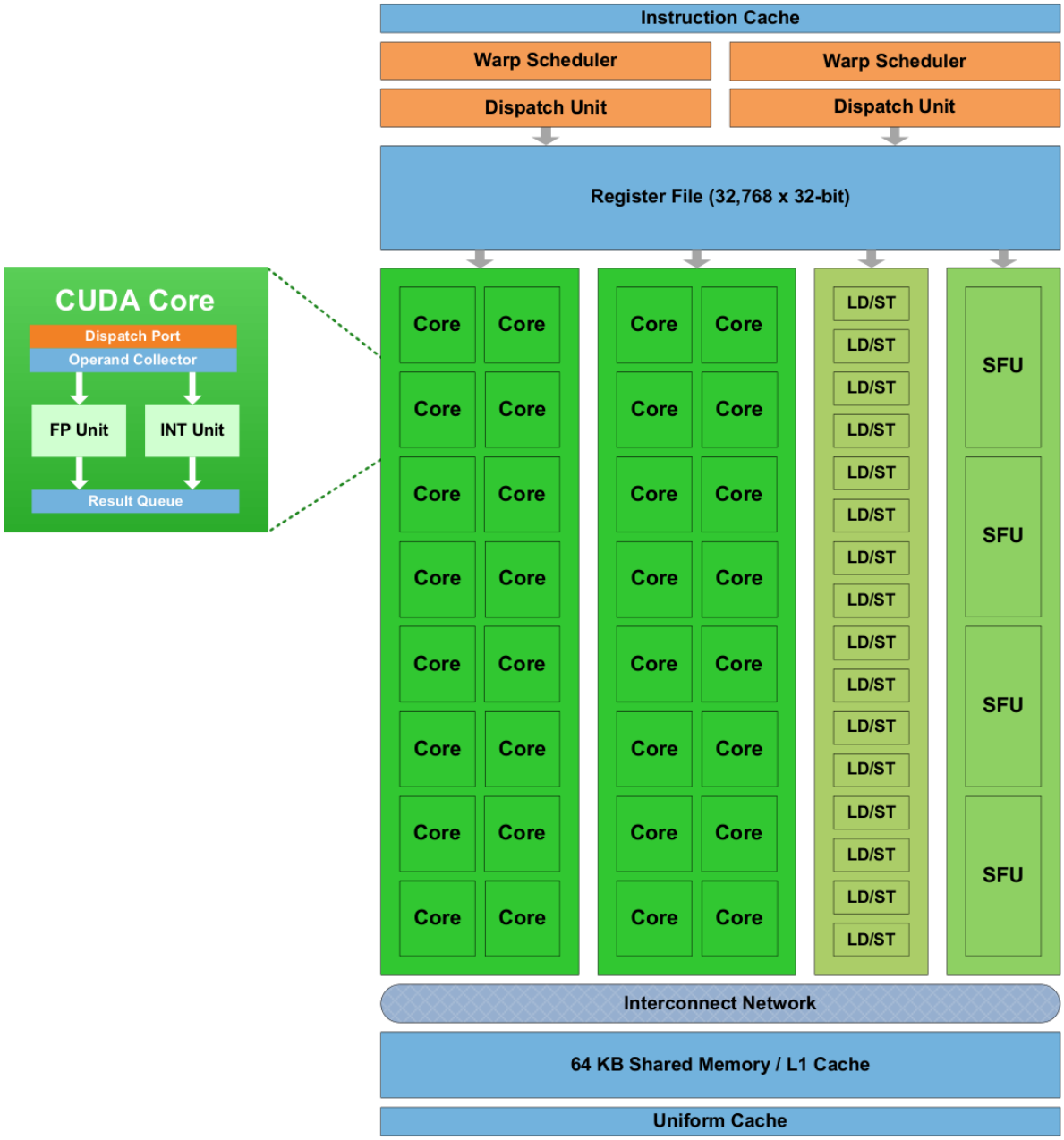
Instruction Cache
Warp Scheduler
Warp Scheduler
Dispatch Unit
Dispatch Unit
Register File (32,768 x 32-bit)
CUDA Core
Dispatch Port
Operand Collector
FP Unit
INT Unit
Result Queue
Core
LD/ST
SFU
Interconnect Network
64 KB Shared Memory / L1 Cache
Uniform Cache

图 8.3: SM

8.2 keyword

Chapter 9

参考文献

参考文献

[1] Richard I. Hartley. Self-calibration from multiple views with a rotating camera. In *ECCV*, 1994.

[2] Ezio Malis and Manuel Vargas. *Deeper understanding of the homography decomposition for vision-based control*. PhD thesis, INRIA, 2007.

[3] Gilbert Strang, Gilbert Strang, Gilbert Strang, and Gilbert Strang. *Introduction to linear algebra*, volume 3. Wellesley-Cambridge Press Wellesley, MA, 1993.

[4] Richard Szeliski. *Computer vision: algorithms and applications*. Springer Science & Business Media, 2010.

[5] Zhengyou Zhang. A flexible new technique for camera calibration. *IEEE Transactions on pattern analysis and machine intelligence*, 22(11):1330–1334, 2000.