

C++ 教程 & Pytorch Lightning

菠萝芝士焗饭

2024 年 2 月 10 日

目录

1 前言	11
2 C++ 基础教程	11
2.1 C++ 如何工作	11
2.1.1 编译	12
2.1.2 链接	13
2.1.3 函数重载	13
2.1.4 Ambiguity	13
2.2 C++ 变量	14
2.3 函数	14
2.4 头文件	15
2.5 条件和分支	16
2.6 For&while	17
2.7 控制流	17
2.8 指针	18
2.9 引用	18
2.10 类	19
2.11 类和结构体的对比	20
2.12 static	20
2.13 类和结构体内部的 static	20
2.14 局部静态	21
2.15 枚举	22
2.16 构造函数	22

2.17 析构函数	22
2.18 继承	22
2.19 虚函数	24
2.20 接口	24
2.21 可见性	25
2.22 数组	25
2.23 字符串	26
2.24 const	26
2.25 成员初始化列表	28
2.26 三元操作符	29
2.27 创建以及初始化类	30
2.28 new 关键字	30
2.29 隐式转换和 explicit 关键字	30
2.30 运算符重载	31
2.31 对象生存期	32
2.32 智能指针	32
2.33 C++ 复制与拷贝函数	32
2.34 箭头运算符	34
2.35 动态数组	35
2.36 处理多返回值	35
2.37 templates	36
2.38 栈和堆的比较	37
2.39 宏	37
2.40 auto 关键字	38
2.41 静态数组 array	38
2.42 函数指针	38
2.43 命名空间	39
2.44 线程	40
2.45 多维数组	40
2.46 排序	41
2.47 Union	41
2.48 虚析构函数	42
2.49 类型转换	43
2.50 Safe	43

2.51 预编译头文件	43
2.52 dynamic_cast	43
2.53 结构化绑定	44
2.54 如何能让 C++ 字符串更快?	44
2.55 单例模式	45
2.56 小字符串优化	46
2.57 跟踪内存分配	46
2.58 左值和右值	46
2.59 参数计算顺序	47
2.60 移动语义	47
3 C++ 小游戏	49
3.1 俄罗斯方块	49
3.1.1 坐标	49
3.1.2 前期工作准备	50
3.1.3 碰撞检测	52
3.1.4 控制移动	52
3.1.5 下一块	53
3.1.6 删除	54
3.1.7 渲染	55
3.2 第一人称射击游戏	56
3.2.1 玩家控制和碰撞检测	57
4 Python 及其相关练习	57
4.1 Python 的一些 API	57
4.2 Dataset 与 DataLoader	59
4.2.1 Custom Dataset	59
4.2.2 DataLoader 的源码	60
4.3 构建网络	63
4.3.1 transform	63
4.3.2 Build Network	63
4.4 Module	64
4.4.1 Module 源码	65
4.4.2 Sequential	66

4.5	autograd	67
4.5.1	雅可比矩阵	68
4.6	Training model	68
4.7	自动微分 Forward 与 Reverse 模式	69
4.8	模型保存	70
4.9	Dropout	70
4.9.1	Dropout 源码	71
4.9.2	dropout 复现	73
4.9.3	R-Dropout	74
4.10	残差模块	74
4.10.1	CNN	74
4.10.2	残差模块复现	75
4.11	Transformer	77
4.11.1	Encoder	78
4.11.2	Transformer 复现	79
4.11.3	Attention	81
4.11.4	总结	84
4.12	CNN	84
5	CS106B C++ 抽象编程	84
5.1	Welcome	84
5.2	C++ 基础	87
5.2.1	Include	87
5.2.2	console	87
5.2.3	变量和类型	87
5.2.4	function and parameter	88
5.2.5	控制流	90
5.2.6	String	90
5.2.7	string utility functions	91
5.2.8	Two types of string	93
5.3	Assignment 0. Welcome to CS106B!	94
5.4	Assignment 1. Getting Your C++ Legs	94
5.5	Console and Vector	101
5.5.1	String	101

5.5.2	Interact with Users	102
5.5.3	structure data	103
5.5.4	Unordered Structure	106
5.6	Using Abstraction	109
5.6.1	word ladders	109
5.6.2	BFS	111
5.7	Big O notation and algorithm analysis	112
5.7.1	Nested DS	112
5.7.2	Big-O	112
5.8	Assignment 2. Fun with Collections	114
5.8.1	warmup	115
5.8.2	maze	116
5.8.3	search engine	119
5.9	递归	124
5.9.1	Factorials	124
5.9.2	Inverse String	124
5.9.3	Summary	125
5.10	Recursive Fractals	125
5.10.1	Fractal	126
5.10.2	Why Recursion?	127
5.10.3	回溯	127
5.11	Recursive Optimization	132
5.11.1	Knapsack Problems	132
5.12	Assignment 3	133
5.12.1	Recursion Warmup	134
5.12.2	Balanced Operators	136
5.12.3	Sierpinski	137
5.12.4	Binary merge	139
5.12.5	Backtracking Warmup	143
5.12.6	Boggle Score	144
5.12.7	Voting Power	146
5.13	Object-Oriented Programming	148
5.14	Dynamic Memory and Arrays	150
5.14.1	动态内存分配数组	150

5.15	Implementing an ADT	151
5.16	Priority Queues and Heaps	151
5.16.1	Binary Heap	152
5.17	Memory and Pointers	153
5.18	Assignment 4. Priority Queue	154
5.18.1	Warmup	154
5.18.2	PQArray and PQHeap	154
5.18.3	PQueue Client and Data Science Demos	156
5.19	Linked Lists	156
5.19.1	Linked List	156
5.19.2	Linked List Operation	157
5.19.3	LinkedList sorted	160
5.20	Assignment 5. Linked Lists	160
5.20.1	Memory Debugging Warmup	160
5.20.2	Labyrinth Escape	161
5.20.3	Sort linked list	162
5.21	Advanced Sorting	166
5.21.1	Merge Sort	166
5.21.2	QuickSort	167
5.22	Tree	167
5.22.1	Tree Properties	168
5.22.2	Tree Traversal	168
5.23	Binary Search Tree	169
5.24	Huffman Coding	170
5.25	Hashing	171
5.26	Assignment 6. Huffman Coding	172
5.26.1	Warmup	172
5.26.2	Huffman	177
5.27	Graph	184
5.27.1	Iterating over a Graph	185
5.27.2	Dijkstra 算法	186
5.27.3	A*	186

6 CUDA 官方文档	188
6.1 Introduction	188
6.2 Programming Model	189
6.2.1 kernel	189
6.2.2 Thread Hierarchy	190
6.2.3 Thread Block Clusters	192
6.2.4 Memory Hierarchy	194
6.2.5 Heterogeneous Programming	195
6.2.6 Asynchronous SIMD Programming Model	197
6.2.7 Compute Capability	197
6.3 Programming Interface	197
6.3.1 Compilation Workflow	197
6.4 Hardware Implementation	198
6.5 Performance Guidelines	198
6.5.1 Maximize Utilization	198
6.5.2 Maximize Memory Throughput	202
6.5.3 Maximize Instruction Throughput	204
6.6 C++ Language Extensions	205
6.6.1 Function Execution Space Specifiers	205
6.7 Variable Memory Space Specifiers	205
6.7.1 Built-in Vector Types	207
6.7.2 Memory Fence Functions	208
6.7.3 Synchronization Functions	209
6.7.4 Texture Object API	209
6.7.5 Surface Object API	209
6.7.6 Read-Only Data Cache Load Function	209
6.7.7 Load Functions Using Cache Hints	210
6.7.8 Store Functions Using Cache Hints	210
6.7.9 Time Function	210
6.7.10 Atomic Functions	210
6.7.11 Address Space Predicate Functions	211
6.7.12 Address Space Conversion Functions	211
6.7.13 Alloca Function	211
6.7.14 Compiler Optimization Hint Functions	211

6.7.15	Warp Vote Functions	212
6.7.16	Warp Match Functions	212
6.7.17	Warp Reduce Functions	212
6.7.18	Warp Shuffle Functions	212
6.7.19	Nanosleep Function	213
6.7.20	Warp Matrix Functions	213
6.7.21	Asynchronous Barrier	213
6.7.22	Dynamic Global Memory Allocation and Operations	213
6.7.23	Execution Configuration	214
6.8	Cooperative Groups	214
6.8.1	Partitioning Groups	216
6.8.2	Modularity	216
6.8.3	Optimizing for the GPU Warp Size	217
6.9	CUDA Dynamic Parallelism	217
6.9.1	Execution Environment and Memory Model	217
6.9.2	CUDA 实现 dropout	218
7	CUDA 总结	218
7.1	CUDA 基础	218
7.1.1	并行计算	219
7.1.2	并行性	220
7.1.3	计算机架构	220
7.2	异构计算与 CUDA	222
7.2.1	异构架构	222
7.2.2	CUDA	224
7.3	编程模型概述	226
7.3.1	CUDA 编程结构	227
7.3.2	内存管理	228
7.3.3	线程管理	229
7.4	组织并行线程	233
7.4.1	二维网格二维块	235
7.4.2	二维网格一维块	235
7.4.3	二维网格一维块	235
7.5	CUDA 执行模型	236

7.5.1	gpu 架构	236
7.5.2	CUDA 编程的组件与逻辑	238
7.5.3	Fermi 架构	239
7.5.4	Kepler 架构	241
7.6	Warp 执行的本质	242
7.6.1	线程块和线程束	242
7.6.2	warp 分化	242
7.6.3	资源分配	243
7.6.4	延迟隐藏	245
7.6.5	占用率	245
7.6.6	同步	245
7.7	并行性表现	246
7.7.1	并行规约问题	246
7.7.2	改进规约分化	250
7.7.3	交错配对	251
7.8	展开循环	252
7.8.1	展开规约	253
7.8.2	完全展开的归约	257
7.8.3	模板函数的归约	257
7.9	动态并行	258
7.9.1	嵌套执行	258
7.10	全局内存	259
7.11	内存层析结构的优点	259
7.11.1	CUDA 内存模型	260
7.11.2	寄存器	263
7.11.3	本地内存	263
7.11.4	共享内存	263
7.11.5	常量内存	264
7.11.6	纹理内存	264
7.11.7	全局内存	264
7.11.8	GPU 缓存	265
7.11.9	静态全局内存	265
7.12	内存管理	266
7.12.1	内存分配和释放	267

7.12.2 内存传输	267
7.12.3 固定内存	268
7.12.4 零拷贝内存	269
7.12.5 统一虚拟寻址	269
7.12.6 总结	271
7.13 内存的访问模式	271
7.13.1 对齐与合并访问	271
7.13.2 全局内存读取	273
7.13.3 缓存加载	273
7.13.4 没有缓存的加载	275
7.13.5 只读缓存	275
7.13.6 全局内存写入	275
7.13.7 性能调整	276
7.14 核函数可达到的带宽	277
7.14.1 内存带宽	277
7.14.2 矩阵转置	277
7.14.3 转置上下限	279
7.14.4 朴素转置：读取行与读取列	280
7.14.5 展开转置	281
7.15 使用统一内存的向量加法	283
7.16 共享内存和常量内存	283
7.17 CUDA 共享内存概述	283
7.17.1 共享内存	284
7.17.2 共享内存存储体和访问模式	285
7.17.3 同步	286
7.18 共享内存的数据布局	287
7.18.1 方形共享内存	287
7.19 合并的全局内存访问	288
8 AI 葵 cuda 课程笔记	288
8.1 pytorch, C++ 和 CUDA 的关系	289
8.1.1 简单的例子	289
8.1.2 核函数	290
8.1.3 Trilinear Interpolation	291

8.1.4	cpp	293
8.1.5	cu-cpp	294
8.1.6	kernel	295
8.1.7	反向传播	296
9	ECE408 课程笔记	298
9.1	Heterogeneous Parallel Computing	298

1 前言

一些笔记，来自视频 https://www.bilibili.com/video/BV1uy4y167h2/?spm_id_from=333.788&vd_source=a059a118f33728f79abd79e02f8f72d4 只是一些简单记录，代码类的还是得上手实践，就算完完整整的全部记下来，最后也很容易忘记。就像学数学一样不做题不巩固，就不知道这些知识点要怎么用。所以简单过一遍，然后上项目，项目遇到了什么不懂的再回来补充。

2 C++ 基础教程

2.1 C++ 如何工作

首先看一段非常简单的程序：

```
#include <iostream>

// 如果找不到这个函数在其他文件的位置，就会出现链接报错
void Log(const char* message);
// 任何一个C++的程序都需要main函数，是程序的入口
// 注意到main函数返回了一个整数，当你什么都不写的时候默认返回0
// 但这只对main函数适用，对于其他函数是一定要有对应的返回值
int main()
{
    // <<是一个重载运算符
    // 将hello world推送到cout，然后打印到控制台上，endl是输出回车
    std::cout << "Hello, World!" << std::endl;
    // get函数是等待我们输入，也是一个暂停函数
    std::cin.get();
}
```

第一句 include 是预处理，因为他在编译之前就处理完了，所有的预处理语句都会用 # 开始。include 表示把后面的文件复制到当前 cpp 文件上。这个文件也叫头文件，因为他写在开头。

main 是这个整个程序的入口，默认返回 0，如果你不写她就默认返回 0，但对于其他函数是需要返回和类型相同的。«表示一个重载运算符，把 hello world 写到输出流，在控制台显示，endl 表示回车。get 表示等待用户输入，可以作为暂停的一种方式。

cpp 文件的处理可以分成几步：

- 预处理阶段，也就是执行头文件，在这里的主要操作就是把这些头文件复制到当前文件中
- 编译阶段，这个阶段会把 cpp 代码编译成机器执行的代码，这些代码会变成.o 的文件形式
- 然后链接，将这些 o 文件链接成一个 exe 可执行文件

通常，我们不会把一个函数写在一个代码里面，比如上面，把 log 函数单独拿了出来。需要在 test.cpp 文件上面单独声明这个函数，程序会自己去找这个函数在哪里。我把 Log 函数写在了 Log.cpp 文件上，那么在第二步编译阶段的时候，就会出现两个 o 文件，这个时候链接程序就会找到这个 Log 函数放进 test 程序里面。

首先编译：g++ -c test.cpp log.cpp，会生成.o 文件，然后链接，g++ test.o log.o -o main，就是 test.cpp 中找到这个 log 函数的过程包含在其中，最后生成 main.exe，执行即可。

2.1.1 编译

这个过程的具体表现就是生成.o 文件，主要做了两件事：

- 预处理代码
- 创建抽象语法树

首先是预处理的代码，预处理这部分就是复制，把头文件复制到当前文件中去。看一个例子

```
#include <iostream>

int Log(const char* message)
{
    std::cout << "Logging ..." << message << std::endl;
```

少了两个关键部分，`return` 和`}`，我们可以在头文件补上。新建一个 `my.h` 文件，里面补上缺失的内容：

```
    return 1;  
}
```

在代码后面补上

```
#include "my.h"
```

就可以运行了。然后就是生成 `obj` 文件，这个文件可以用 `visual studio` 可视化，但是我用的命令行直接 `g++` 编译看不到，里面都是一些汇编代码，也就是说编译器会把这些代码变成汇编指令，让机器执行。

2.1.2 链接

链接的主要的目的是寻找每个符号和函数是在哪里，链接本身还是比较好理解的，这一节主要解决的是一些链接的问题。

2.1.3 函数重载

按照视频的内容，返回值不同，那么函数就不同，代码应该是能识别出来的。但实际上，在我的测试代码里面，仅仅只有返回值不同的函数是无法进行函数重载的，这个标准应该是新的标准，我以前学的时候返回值不同还是可以的。所以仅仅返回值不同的函数是不能重载的。声明 `void Log(const char* message)`，如果你的引用代码里面有 `int Log(const char* message)`，她是会链接到的。

2.1.4 Ambiguity

这个问题是出现在存在多个可以链接到的函数，程序混乱了导致的。比如在 `log.cpp` 我们存在两个除了返回值其他都一样的函数，那么模型是不知道链接哪一个函数的。当然，这个错误只会出现在我们链接的时候，编译的时候是不会出现的。

以下有几种特殊的 ambiguity 情形。首先我在头文件 `my.h` 定义了一个 `test` 函数，然后在 `test.cpp` 和 `log.cpp` 引用这个函数，需要引用我们就需要声明，我们直接用

```
#include "my.h"
```

加在这两个 `cpp` 文件里面，这个时候编译就会出现

```
multiple definition of 'test()'
```

的问题，这个问题就要回到预处理语句的定义上，`include` 是把预处理头文件复制到当前文件中，所以相当于在两个 `cpp` 文件定义了两个相同的函数，这样就出现了定义的问题。如何解决？有两个方法：

- 把 `test` 函数隐藏起来，对其他文件不可见。使用 `static` 关键字。`static int test()` 就没
问题了，虽然他们会被复制到各自的 `cpp` 文件中，但是 `static` 关键字使得这些函数是
只能自己看到，对其他文件来说不存在。
 - 另一种方法是用 `Inline` 关键字，这样程序并不会把 `test` 函数当成是一个函数，她
会把引用这个函数直接替换成这个函数的内容。比如 `test.cpp` 引用了 `test()`，而
`test() return 1;`，那么她会把 `test.cpp` 的 `test()` 换成 `return 1;` 这样没有了函数调
用自然就没有了链接。
- 另一种方法是通过头文件引入。把方法写到另一个 `cpp` 文件中，然后把这个方法引入
到 `.h` 头文件。其实就是用函数声明，只不过把这个函数声明分到了另一个文件里面。

所以 `.h` 的文件里面很多都是用 `static` 关键字，就是为了反正多重定义。

2.2 C++ 变量

在程序运行的过程中，我们需要为存储在变量里面的的数据命名，这个写数据就存储在变量
里面。这一节比较简单

- 变量之间的区别就是大小，他们的大小不同导致他们在计算机里面的区别
- 程序中默认了一些约定俗成的规则，比如 `char` 一般用于存储字符。但实际上字符的
存储也是数字，和 `ascii` 码表对应

常见的几类 `char`, `short`, `int`, `long`, `longlong`, `bool` 等。他们占用的内存都是字节，因为内存
只能以字节为单位查询。虽然 `bool` 只需要一个比特就能完成查询，但是模型查询的最小单
位是字节，所以只能分配字节给 `bool` 变量。

2.3 函数

函数是用来执行特定任务的代码块。使用函数的目的是希望代码的可读性更高，希望代码
块的重用性更高。

- 每次调用函数的时候，编译器会生成 `call` 指令，也就是说编译器会为每一个函数创建
一个 `stack` 结构，把这些变量都推送到 `stack` 中。

- 所以在运行函数的时候程序会不断在内存中跳跃执行

所以不要创建过多的函数。另外，如果函数在定义的时候明确要求返回值，那么这个函数就一定要返回。

2.4 头文件

主要了解几个问题

- 为什么需要头文件
- 头文件是干什么的
- 什么情况下需要

头文件通常是用来声明某些函数，这样可以在程序中使用。当我们使用一个定义在其他文件的变量的时候，我们需要声明这个函数，而头文件就是用来存储这些函数声明的地方。当然你也可以直接在你需要用到这个函数的地方进行声明，但如果你有 100 个函数要引用，你就要写 100 次，而且如果你要在其他的 cpp 文件里面也引用这些函数，那么你又要写 100 次。如果使用头文件只需要把这 100 个函数声明写进去，再引用这个头文件就行，只需要写一次。

再头文件开始的时候还加上了一段代码：

```
#pragma once
```

意思就是头文件里面的内容只会被转换成一个编译单元，这是为了防止你再头文件里面定义了某个函数或者类，然后重复引入同一个头文件导致的定义重复。

另一种复制重复定义的方法是

```
#ifndef _LOG_H
#define _LOG_H

#endif
```

ifndef 表示如果后面这个变量没有被定义，那么就编译后面的代码。如果定义了就不编译。简单来说 头文件是用于声明代码，而 cpp 文件是实现这些声明。

最后还有两个补充的点：

- <> 和”” 的区别，<> 表示从系统目前搜索，然后再搜索环境变量列出的目录，但是她不会查找当前目录，差不多就是只会查找系统目录。”” 是会查找当前目录下的

头文件，然后再去搜索系统目录，所以””是会搜索几乎所有可能存在头文件的地方。所有 <> 一般用来引入系统文件，“”一般用来引入自己写的文件。

- 在写 C++ 的时候引入系统都文件 iostream，这个文件后面是没有扩展名的，这是为了和 C 语言区分开。

2.5 条件和分支

在写代码的时候，我们需要根据结果或者根据某些条件判断我们应该执行哪些语句。但实际上在代码中添加分支会减慢代码的速度。这部分比较简单，主要讨论一些 char * 和数组的内容。

首先有几个关键 Insight：

- 数组是多个元素的组合，他们的地址是连续的
- 指针的类型为整形 int，永远为四个字节，定义的时候 float * q 表示 q 指向一个 float 类型的数据
- 数组的本质就是首个元素的地址

在本章的示例代码中，定义字符串是通过：

```
const char * s = "Neural Radiance Field";
```

以前学的时候不需要 const 是没问题，可能是标准不同了，右边的 NeRF 字符串是一个常量，类型是 const char *，左边和右边的类型对不上，所以报错了。输出的时候直接针对这个地址输出即可：

```
std::cout << s << std::endl;
```

如果定义成

```
char arr[] = "Neural Radiance Field";
```

也是一样，arr 就是字符串的首地址，输出地址对比可以看到是一样的：

```
char * p = arr;
std::cout << static_cast<void *>(p) << static_cast<void *>(&arr[0]) << std::endl;
```

2.6 For&while

这两个都是用来表示循环，for 的格式

```
for(int i = 0; i < 5; i++)  
{  
}
```

定义变量，变量条件，变量自增。如果想到达无限循环，可以在条件部分补上 true，或者直接不写。如果想要达成等差数列，或者等比数列，你可以直接在变量自增部分改成 $i+=2$, $i+=3$, $i*=3$ 等。

while 也是类似：

```
while(i < 5)  
{  
    i++;  
}
```

每次都会判断是否小于 5，while 之后的括号一定要写上 i 自增或者其他改变 i 的语句，否则就会一直循环。但实际上，同样，要达成无限循环也只需要在条件写 true，或者你对 i 这个变量不处理就行；等差数列和等比数列也需要对 i 这个变量的自增处理一下。所以 while 和 for 其实是可以互换的。

除此之外还有一个 do while 的循环，do while 和 while 其实区别不大，主要在于 do while 是一定要走一次的，while 可以一次都不走。

2.7 控制流

主要介绍三个控制流语句：

- continue
- break
- return

continue 用于跳过当前循环直接进入到下一个循环。Break 推出整个循环。return 直接结束。

2.8 指针

对于程序来说，最重要的部分就是内存，当写下程序运行的时候，程序会被写入到内存中，控制语句，循环语句本质上也是内存之间的不断跳跃。每一个变量第一次会存储在内存中，为了找到这些变量，我们需要为每一个内存编号，这个编号会由另外一个变量存储，这个存储编号的变量就是指针。

定义一个指针

```
void* ptr = nullptr;
```

指针无论前面定义什么类型，都是一个整形的指针，前面定义的类型只是表明了她需要指向什么样类型的数据。nullptr 表示目前这是一个空指针。现在定义一个变量，用这个指针指向它：

```
int var = 8;
ptr = &var;
```

可以通过指针访问指向的内存数据，`&` 可以获得当前变量的地址。

如果我们想要一些内存存储，但是又不知道内容的时候，就采用分配的方法。

```
char* buffer = new char[8];
memset(buffer, 0, 8);
```

`new char` 是分配一个空间为 8 字节的 `char` 类型内存空间，返回指向这个空间的指针。`memset` 是填充这个内存，这里表示用 8 个 0 填充。除此之外，还有指针的指针等。

2.9 引用

引用通常就是地址的伪装，引用和指针很大的区别在于指针可以先定义，然后设置他们称为一个空指针；但是引用不可能创建一个空的引用，必须要引用已经存在的变量。

假设一个已知的变量 `a`，可以通过

```
int& ref = a;
ref = 90;
```

创建一个 `a` 的引用 `ref`，`ref` 就是 `a` 的别名，对于 `ref` 的改动其实就是对 `a` 的改动，之所以说引用没有内存是因为她和 `a` 共用地址。其实通常并不是会创建一个新的变量指向 `a`，通常引用是在函数中使用，希望能直接在函数中修改变量的值。

定义一个自增函数

```
void Increment(int value)
{
    value++;
}
```

直接把 a 传入进去，是不会改变 a 的值。解决这个问题有两个方法，可以利用指针传递，参数改成输入指针：

```
void Increment(int* value)
{
    (*value)++;
}
```

需要加括号，不加括号相当于是取出下一个地址的内容，然后不做处理。这样相当于获得内存的地址，直接对地址进行修改。另一种方法就是再方法的参数加上 & 引用，传入一个引用。

2.10 类

这部分开始进入面向对象编程。假设一个场景，我们需要写游戏，编写一个玩家的类信息。当然我们可以直接把玩家的所有信息都写进 main 函数里面，位置 xy，速度 velocity 等等。如果定义另外一个人，又需要重新定义一次位置，速度等等。简单定义一个类：

```
class Player
{
public:
    // 公有变量，意味着可以在任何地方访问
    int x, y;
    int speed;

    void Move(int xa, int ya)
    {
        x += xa * speed;
        y += ya * speed;
    }
};
```

C++ 的类如果默认设置就是私有的，除了自己谁也访问不了，所有通常会写一些公有的访问函数。

2.11 类和结构体的对比

主要是对比结构体和类的区别，说实话学了类之后很少写结构体了，类的功能比结构体强大很大，完全可以替代结构体的功能。

事实上这两者的区别不大，从技术上讲，两者的区别在于结构体的内容都是公有的，但是类是私有的。当然了结构体也可以定义 private 使得变量函数私有。

2.12 static

static 这个关键字在之前就已经见过，用于限制方法的可见范围。通常写在头文件的都是 static。如果 static 写在类的内部，那么说明这个变量是共享的。

在类外的定义：

```
static int static_variable = 5;
```

这个变量只会在当前的编译单元见到。之前这个做法是用来解决链接冲突的问题。A, B 两个 cpp 文件都引用了头文件.h 的一个 log 函数，预处理代码会先把头文件的内容都拷贝到 A, B 文件中，所以这两个文件实际上是定义了两个相同函数，链接的时候编译器就会疑惑怎么会有两个相同的函数。为了解决这个问题，我们把头文件的函数写成 static，这样辅助到 cpp 文件的函数也是 static，他对其他 cpp 文件不可见，这样就只有一个函数可以链接。至于为什么需要 static，这和类中为什么需要 private 变量一样。

2.13 类和结构体内部的 static

上一节的 static 是定义在方法，变量前面，也就是类别外面的。这节的 static 是定义在类内部。

static 成员只能在类内声明不能初始化，要初始化 static 成员必须在类外初始化。比如创建 Player 类别，在类内定义两个变量 x, y。static 变量初始化一定要在所有代码之前，初始化是在编译器初始化。也就是放在所有程序的外面。

有两点需要注意：

- 静态变量的初始化要在所有代码之前，所有一般在头文件写好然后引用到 cpp。

- 类内静态函数只能访问静态变量。和 python, java 类似，每一个类内函数都会传入一个类的实例，C++ 里面的 this 指针，默认传入，python 传入 self 对象。静态方法没有实例也就是没有 this 指针。

2.14 局部静态

在定义变量的时候，我们需要考虑变量的两个属性，生命周期和作用域。通常限制变量生命周期是变量定义的位置，限制作用域通常由特定关键字，比如 static, inline 等，或者定义的位置决定。

假设函数 add，我们希望作用于一个全局变量上，可以如此定义：

```
int sum = 0;
void sum_f()
{
    sum++;
}
```

但如果我们不想要让其他函数对次变量修改编辑，我们就可以把这个变量编程静态局部变量：

```
void sum_f()
{
    static int sum = 0;
    sum++;
    LOG(sum);
}
```

此时 sum 变量就变成局部可见了，仅仅只有 sum 函数可见了。用处还是蛮多的，可以用来 debug，测试该方法跑了几回；也可以用这种方法来实现一个类的功能，因为他可以隐藏变量，相当于定义一个全局变量。

总结一下 static 的用法：

- static 在类外定义，在方法或变量前面，该方法和变量仅仅对于该文件可见
- static 在类内定义，该方法相当于类的单例变量，实例通过. 访问，类通过:: 访问
- static 在方法内定义，相当于把全局变量的可见范围缩减到该方法内部。

2.15 枚举

枚举实际上就是一个数值组合，实际上是一种清洁代码的方法。比如假设日志的级别，我们会给 0, 1, 2 这些数字赋予一些人为的约定，比如 0 是警告等，这个时候如果用枚举就可以简化我们的书写。

```
enum levels{
    A, B, C, D, E, F, G, H, I, J, K
};
```

这样就创建一个枚举类，这个类里面定义的 A,B,... 都是 static 变量，完全可以通过 levels::A 访问，或者直接就 A 也能访问，这样能让代码更直观。可读性更强。当然也可以直接创建一个实例，但是这个实例只能填 ABC... 那几个给好的数字。

2.16 构造函数

java 也有这东西。构造函数是用来初始化变量的，在定义一个类的时候，我们有时候需要一些特定的操作去初始化函数，比如 python 中的 init 函数。这个时候如果把他们全部写在变量一起，代码看起来就会很混乱，所以构造函数就可以增加模型的可读性。

以类名为函数名，不需要准备返回值类型，就可以定义一个构造函数。

```
Player(int x, int y)
{
```

在实例化一个类的时候就会自动调用。

2.17 析构函数

这个函数是在删除实例的时候调用，也就是在该类释放内存的时候调用。

2.18 继承

类之间的继承是 C++ 的最强大特性之一。我们可以通过继承定义一个互相关联的类的层次结构，这些不同的子类都包含了一个共同的祖先。先定义一个父类：

```
class Player
{
```

```

public:
    // 公有变量，意味着可以在任何地方访问
    static int x, y;
    int speed;

    Player(int x, int y)
    {

    }

    void Move(int xa, int ya)
    {
        x += xa * speed;
        y += ya * speed;
    }

    static void s_print()
    {
        LOG(x);
        LOG(y);
    }
};


```

同时定义一个子类继承他：

```

class Good : public Player{
public:
    using Player::Player;
};

```

Good 这个子类会拥有这个父类 Public 的所有变量和方法。using 这条语句是 C++11 的标准，可以用这种方法把父类的构造函数继承下来。所以在 Good 类中不用初始化了。注意：

- 如果一个 function 要求参数是 Entity 类，那么可以输入 Entity 及其子类。但是如果定义了子类，就不能传父类，因为父类有可能没有子类方法。比如定义了 Player 类作为参数类型，那么可以传 Good，但是用 Good 类作为参数类型就不能传 Player。这种其实就多态，自适应各种不同类型。

- 继承父类之后重载函数实际上是重新定义函数，而不是重载某一个父类函数。比如父类有 aabb(int i, int j) 和 aabb() 两个函数，子类继承了下来，子类希望重新覆盖 aabb 函数，于是子类自己定义了一个 aabb 函数，那么父类的 aabb(int i, int j) 也会被隐藏起来。

2.19 虚函数

视频中列举了一种情况，现在存在两个类，他们是继承关系：

```
class Entity{
public:
    void printName(){
        LOG("Entity");
    }
};

class player : public Entity{
public:
    void printName(){
        LOG("Player");
    }
};
```

编译运行以下语句： void testName(Entity* entity) entity->printName();
int main() Entity* entity = new Entity(); Entity* ply = new player(); testName(entity);
testName(ply); 会输出两个 Entity，我们原本是创建了一个 player 类但是却被识别成 Entity 的类，使用了父类的函数。因为通常在声明函数的时候，我们的方法通常是在类的内部起作用，这个也很容易理解，因为这两个指针实际上是经过了强转的，模型是不知道他们的来历的，只能知道他们的类型是什么，除此之外就一模一样了。所以这个时候我们就要想个办法让模型能意识到，当前的指针的来自 player 的。

虚函数就是解决这个问题，使用虚函数模型会创建一个 v 表，这个表会记录当前虚函数的映射，也就是被哪个函数复写了。每次运行都需要查表，当然这就出现了额外的性能损失。

2.20 接口

接口函数比虚函数更极端，他会强制子类去实现这个虚函数。他不需要实现任何东西，只需要给函数参数就行。

但是继承的子类需要强制实现，类似于给子类定义了一个标准，按照这个标准去声明。

2.21 可见性

这部分比较重要。这里的可见性一般是指类成员的可见性。

之前在提到类的时候，类成员定义的时候不加修饰默认是私有。

- `private`: 私有变量只能当前类，或者友元类能访问。
- `protected`: 相比于 `private`，他的可见性更强，因为他的子类是能访问到的。

所以为什么要设置可见性？

首先代码里面纯用 `public` 是一个比较糟糕的做法，因为对于开发者和程序来说，你需要保证你代码的安全性，比如在管理数据库的时候你需要写好 API 提供给你的程序员调用，如果全部开放很容易就出现删库跑路或者数据泄露的问题。

2.22 数组

数组和指针息息相关，数组就是指针的基础，可以通过对于指针的操作对数组直接操作。

创建数组有两种形式：

- 直接调用 `a[n]` 创建数组。这种方式是在栈上创建，到达第一个花括号就会自动释放。
- 调用 `new` 创建，那么会创建在堆上，不会自动释放，除非关闭了程序。

这两者最大的区别就是生存周期，其次还有一定性能的区别，`new` 创建的数组实际上是用指针去接受了这个数组的第一个元素的地址，然后再根据这个地址寻找其他变量。这是一种间接寻址，而在栈上创建是直接寻址。

视频后面讲了一堆，感觉没有什么记录的必要，主要需要记录的是栈数组初始化的时候：

```
class Entity{  
public:  
    const int size = 5;  
    int example[size];  
};
```

这种初始化是错误的，需要在 `size` 前面加上 `static`，因为编译器不能确定 `example` 初始化前，你这个 `size` 和 `array` 哪一个先创建，很可能是同时创建的。如果你直接在 `main` 函数，或者其他函数这样创建是没有问题的，

```
int main()
{
    const int size = 5;
    int example[size];
}
```

最后提了一嘴 STL。

2.23 字符串

另一种数组，只不过用来表达字符串而已。

```
const char* name = "new array";
std::cout << name << std::endl;
```

定义了 const 之后就不能对模型进行改变，定义了 name 并且赋值之后，这个字符串的最后会补上一个 0，表示字符串的终结，这也是为什么直接输出 name 能完整的输出字符串。如果用数组定义，那就需要显示的补上 0，否则就会出现乱码：

```
const char name1[10] = {'n', 'e', 'w', ' ', 'a', 'r', 'r', 'a', 'y', 'a'};
std::cout << name1 << std::endl;
```

这样就会出现乱码。

另外 STL 也有 string，可以直接用，不够 C++ 对编译器优化了很多问题已经不存在了。

2.24 const

const 就像一种承诺，定义的时候就承诺定义的变量我们就不会改变，但是既然是承诺那也能违背，所以其实是可以通过某种方法去绕过这个变量的。当我们定义一个变量 $a = 5$ ，可以随意改变这个变量的值，但如果前面加了 const，那就不能修改这个变量的值了。

首先是 const 和指针搭配使用：

- const int 和 int const 是一个意思，可以改变指针的指向，但不能改变指针指向的内容，也就是能使得指针 $b = \&a$ 某个变量，但是不能 $b = a$ 。
- int const 能改变指针的内容，但是不能改变指针的指向。
- 放在方法的后面，这种做法只能在类中使用，`int getX() const return x;`，意味着你不能修改类的变量。通常在需要 print，查看类信息的时候使用。

极端点的情况：

```
const int* const getX() const
{
    return &x;
}
```

这个函数有三个 const，当然是在类里面定义的函数，返回的指针不能修改指向，不能修改内容，并且这个函数也不能修改类变量。

突然发现设计一个好的语言真的很烦，随便一个改动就要考虑各种各样的情况。有一个需要调用这个 getX 类内函数的普通函数

```
void print_Get(Entity& entity)
{
    entity.getX();
}
```

这样调用是没问题的，getX 的 const 是限制类内函数和外面的环境没有关系。但是如果我在参数加上 const，那么 getX 就不能把后面的 const 留下了，也就是说这样是错误的：

```
class Entity{
public:
    int x = 4;

    static const int size = 5;
    int example[size];

    const int* const getX()
    {
        return &x;
    }
};

void print_Get(const Entity& entity)
{
    entity.getX();
}
```

因为输入 print_Get 函数的时候你已经要求了 entity 不变，但是如果 getX 不加 const，是

可以改变的，那么这个 `const` 就失去了意义。所以这个 `getX` 就需要加 `const`。所以经常会看到两个版本的 `getX` 就是在这个原因。

但如果你确实是想做一些标记，希望能在 `const` 的情况下继续修改，那么就需要 `mutable` 关键字，这样你的变量就可以修改了。总之 `const` 就类似一些强行施加的规定，保证代码的安全性。

2.25 成员初始化列表

主要是对类成员初始化所用。最常见的一种初始化方式就是构造函数了，定义不同的构造函数可以使用不同的方式进行初始化。

另一种初始化就是成员列表初始化，在构造函数后面加上冒号按照顺序写好：

```
class Player{
public:
    int x, y;
    int velocity;

    // 列表初始化需要按照声明变量的顺序来写，顺序不能乱
    Player(): x(0), y(0), velocity(0)
    {

    }

    Player(int x, int y, int velocity): x(x), y(y), velocity(velocity)
    {

    }
};
```

所以为什么我们需要使用这个，成员列表初始化和一般的初始化方法其实结果都是一样的，完全可以互相替代，但是如果使用成员列表初始化，就可以对代码进行简化，当你有很多变量的时候，一般的初始化会使得构造函数很难看出是在做什么，
但如果你用成员列表初始化，构造函数就会很清晰。初次之外，使用成员初始化还能增加程序的性能。假设类内构造函数初始化如下图所示：

```
int x, y;
int velocity;
Entity* entity;
```

```
// 列表初始化需要按照声明变量的顺序来写，顺序不能乱
// Player(): x(0), y(0), velocity(0), entity()
// {

// }
Player()
{
    entity = new Entity(8);
}
```

那么 Entity 会创建两次，第一次是定义的时候，第二次是构造函数声明的时候，如果用构造成员列表，那么只会构建一次：

```
Player(): x(0), y(0), velocity(0), entity(Entity(7))
{

}
```

最后的结论就是尽量使用成员列表初始化。

2.26 三元操作符

三元操作符实际上是一种简化代码的形式，if 语句的一种语法糖。正常 if 的写法：

```
if (level < LEVEL){
    level = LEVEL;
}
```

用语法糖可以一行写完：

```
level = level < LEVEL ? LEVEL : level;
```

有两个优点：

- 代码更简洁
- 速度更快，因为他不会创建一个临时的字符串作为比较结果

2.27 创建以及初始化类

首先是栈和堆的区别，内存主要也是分成两部分，堆和栈，栈的生命周期是和他在哪里声明相关，如果在函数中声明，那么函数结束的时候就会把这些变量都弹出来删除。堆不一样，堆内的对象不会自动释放，而是会等待你做出释放的命令，否则他会一直停留到结束。对于类来说，默认初始化就是在栈上创建，new 关键字创建就是在堆上创建。

2.28 new 关键字

如果希望用到 C++，一般都要关注于内存，性能，并行计算和优化问题。Java，python 这类语言带有内存清理的机制，C++ 没有，需要自己控制内存。

使用 new 初始化类，是一种非常常见的操作了。new 实际上是一个操作符，她不仅仅分配了空间，她还调用了类的构造函数。**new 实际上是调用了 c 语言的 malloc 分配函数**

```
Entity* e = new Entity();
Entity* e = (Entity*)malloc(sizeof(Entity));
```

上面两行的区别在于第二行只是分配了空间，而第一行不仅分配了空间，还调用了构造函数。在调用 new 操作符的时候，需要手动清理内存空间，因为在堆上的变量是不会自动释放的。所以如果使用了 new，最后要接上 delete。

2.29 隐式转换和 explicit 关键字

首先作者举了一个类的例子，对于有构造函数

```
Entity(int a)
{
    cout << "create " << a << endl;
}
```

的类，我们可以直接对她进行赋值，Entity a = 23;。这里会自动做一次隐式变换，**隐式变换只能做一次，如果需要两次变换的会报错**。比如 Entity a = "67" 就有问题，她需要两次变换，首先变成 string，然后变成 int，再变成 entity。这种代码的书写方式可读性不高，所以如果想禁止这种方式可以使用 explicit，只需要再构造函数之前加上 explicit。

隐式变换在书写代码的过程中处处都有，比如对于一个函数参数为 double 类型的函数，如果调用时候输入 1，那么她会把 1 变成 double 类型，这就是一种隐式变换。

2.30 运算符重载

- 首先什么是运算符：运算符是一种符号，可以用来代替函数进行执行，常见的数字运算符加减乘除，new，delete 或者问号都是运算符。

看一个简单的重载例子，定义一个类

```
class Vector_two{
public:
    int x, y;
    Vector_two(int a, int b):x(a), y(b) {}
};

Vector_two speed(1, 3);
Vector_two position(2, 4);
```

如果想要把这两个类对应的元素相加，可以写一个类内的函数调用，另一种比较方便的写法是重载 + 运算符。

```
Vector_two operator+(const Vector_two& b) const
{
    return Vector_two(x + b.x, y + b.y);
}

Vector_two operator*(const Vector_two& b) const
{
    return Vector_two(x * b.x, y * b.y);
}
```

重载加号和乘号的运算符。

这是在类内的重载，我们也可以重载 «

```
ostream& operator<<(ostream& stream, const Vector_two& other)
{
    stream << other.x << ", " << other.y;
    return stream;
}
```

2.31 对象生存期

- 如何理解栈上的创建的对象？
- 如何写出高效的代码？

栈可以认为是一种数据结构，这种数据结构的特点就是，她会在作用域结束之后删除，如果你在一个函数里面不用 new 方法创建了一个类，然后返回指向这个类的指针，那么在函数结束之后这个类就会被释放，返回的指针指向了一个空区域。

2.32 智能指针

这算是一个新东西了，在使用关键字 new 创建一个新的变量的时候，我们需要配套使用 delete 删除这个内存，这样相对来说比较麻烦，而智能指针的对这一过程简化。引入 memory 头文件，

```
int main()
{
    {
        // std::unique_ptr<Entity> entity(new Entity());
        // 这种方式最安全
        std::unique_ptr<Entity> entity = std::make_unique<Entity>();
    }
}
```

这样就创建一个 entity 对象，跳出这个作用域那么智能指针会自动删除。但是智能指针是不能复制的，也就是说当智能指针指向了这个内存后不能有其他指针指向了，因为如果智能指针删除了当前变量，那么另一个指针就没有效果了。

shared_ptr 是另一种指针，她的释放内存方式和 python 的差不多，都是计算内存指向的指针数目，如果是 0 那么就释放。shared_ptr 解决了 unique_ptr 不能复制的问题。weak_ptr 是 shared_ptr 的一种，她不会增加 shared_ptr 的引用次数。

2.33 C++ 复制与拷贝函数

在 python 中复制还分深度拷贝还浅度拷贝，其实这里说的就是这两个概念。创建两个变量：

```
int a = 3;
int b = a;
```

这种就相当于深度复制，a 和 b 相当于两个不同的变量，只不过数值相同而已。如果我用指针复制：

```
int* a = new int();
int* b = a;
```

那 a 和 b 指针相当于共享内存了。

看一个简单的例子：

```
class String
{
private:
    char* m_Buffer;
    unsigned int m_Size;
public:
    String(const char* p)
    {
        this->m_Size = strlen(p);
        this->m_Buffer = new char[this->m_Size];
        memcpy(m_Buffer, p, this->m_Size);
    }

    // 声明为友元重载，可以访问私有变量
    friend std::ostream& operator<<(std::ostream& os, const String&);

};

std::ostream& operator<<(std::ostream& os, const String& string)
{
    os << string.m_Buffer;
    return os;
}
```

在这个类中，我们声明了一个字符串，一个友元重载，因为我们需要访问私有变量。在运行输出的时候由于没有终止符，会出现一些错乱的输出，我们只需要在 m_Size 改成 m_Size + 1 就行，memcpy 多余的字符会用 0 填充。

但是如果我们将创建另外一个变量，这个变量 string 赋值，那就出现问题：

```
String string = "new code";
String second = string;
```

```
    std::cout << string << std::endl;
```

因为这个时候复制的变量是指针，其中一个指针释放了内存另一个又要释放，完了找不到内存就崩溃了。这种只是浅拷贝，要解决这个问题，就需要构造一个拷贝构造函数。

```
String(const String& other): m_Size(other.m_Size)
{
    this->m_Size = strlen(other.m_Buffer);
    this->m_Buffer = new char[this->m_Size + 1];
    memcpy(m_Buffer, other.m_Buffer, this->m_Size + 1);
}
```

相当于重新分配了一个内存给这个变量，这个时候

```
String string = "new code";
String second = string;
```

就相当于是两个不同的变量了。

另外一个点就是，形参也是会重新复制一份，所以如果不需要复制，那就用引用就行，这样就不会复制，而是直接把指针传过去。**尽可能使用 const 去传递参数，没必要到处去复制，可以在程序内部决定要不要复制。**

2.34 箭头运算符

我感觉是不是顺序有问题，这个内容应该就在前面就讲。

总的来说就是指针用箭头。这节不打算记录了，efficients C++ 里面不太建议这样写。

最后作者还补了一个语法糖，简单看了一下：

```
class Vector3
{
public:
    float x, y, z;
};

int offset = (long long)(&((Vector3*)nullptr)->z);
```

offset 就是偏移量。首先 nullptr 是空指针，强转成 Vector* 就是初始化了一个指针。然后-> 得到类内元素，& 取得当前元素的地址，再强转成 long long。作者是转成 int，但是我这出问题，long long 是可以隐式转 int 的。nullptr 就是存储在 0 地址中，所以 xyz 是从 0 开始分配。

emmm 这种代码我也不不会写，就放着吧。

2.35 动态数组

也就是 vector。一般的数组在规定了她的大小之后就不能再增加元素个数了。但是 vector 是可以不断增加元素个数的。for 循环可以遍历：

```
for(Vertex& x : vertices)
{
}
```

注意还是尽量使用引用，避免出现复制。

在 vector 的使用中，有两个部分是可以进行优化的：

- vector.push_back 添加元素的时候，是现在 main 函数内创建，然后再移动进 vector 里面，在移动的过程中就相当于是复制了。这里复制了一次。
- vector 是动态数组，每一次增加都需要扩充容量，每一次扩充容量就相当于把原来的数组的复制一遍。

解决方法都很直接

```
std::vector<Vertex> vectices;
vectices.reserve(3);
```

直接将容量变成 3，这样就不用动态扩充了。

第一个问题用 emplace_back，她并不会创建变量然后推进 vector 中，而是会把创建的参数丢进 vector 里面，然后再 vector 里面创建。所以尽量还是用 emplace_back 创建吧。

2.36 处理多返回值

处理方式有很多种

- 可以用数组或者 string 来接收这些返回值然后返回，包括各种数据结构了。
- 也可以定义一个 static 函数，然后引用接收参数

2.37 templates

模板，就是设计一套规则，让机器帮你写代码。比如需要输出不同类型的变量，对于一个 print 函数可能要写很多次，对于不同的变量都需要进行重载。

```
template<typename T> void print(T value)
{
    std::cout << value << std::endl;
}
```

类型并没有指定，等待人为指定或者自动的隐式变换指定。一开始第一次编译碰见这个函数的时候并不会定义出来，知道编译到调用这个函数的位置才会创建，也就是只有调用这个函数的时候才会正式创建这个函数。

```
print(5);
print("cahjks");
```

当然也可以自己指定。

```
print<int>(5);
print<String>("cahjks");
```

如果在类上使用模板呢？

```
template<int N>
class Array
{
private:
    int m_Array[N];
public:
    int GetSize() const
    {
        return N;
    }
};
```

在声明的时候并不会报错，编译器会根据使用情况进行编译。类定义的变量类型可变可以通过模板定义。前面加两个模板就行：

```
template<typename T, int N>
```

2.38 栈和堆的比较

当我们请求内存分配的时候，栈和堆的内存分配方式是不一样的，

栈的分配速度很快，栈的分配只需要移动栈指针即可，分配四个字节的内存，只需要将栈指针移动四个单位即可。所以栈的内存都紧挨着的。

对于堆的分配一般是用 new 关键字，new 关键字会调用 malloc 分配内存。当启动这个程序的时候，os 会分配一些内存给你，程序会去维护一个空闲列表的数据结构，里面记录了哪些内存是空闲的，哪些不是空闲的。当使用 malloc 分配内存的时候，他会寻找这个空闲列表，找到一些空闲的块，分配给程序。当空闲列表中内存不够的时候，就需要向 os 请求，这个过程是很麻烦的。

这两种分配方法最大的区别就是在于分配方法的速度，在栈上分配的速度非常快，只需要一条 cpu 的指令，就可以快速分配。堆的分配会慢一些。所以尽量在栈上分配，除非不能或者需要更长的生命周期。

2.39 宏

一开始的时候就提到过宏，比如 define，if 都是，是在预处理阶段处理的语句。

```
#define WAIT std::cin.get()
```

注意到这里没有加;，那么你在调用的时候就需要写 WAIT;，因为他是替换这行。如果 define WAIT std::cin.get() 写了; 号，那么 WAIT 就不用写了。就像前面 include 把 { 复制进来一样。

define 就是把 WAIT 定义成后面那句话，在预处理的时候，他会把 WAIT 语句替换成后面那句话，和 inline 做一样的事情。这种方式比较蠢，不推荐这样做，因为这样会导致可读性不强。

在一些特定的情况，宏还是有用的，比如日志的输出，在 debug 阶段是需要输出各种日志，但是在运行阶段我们就不希望输出了，这个时候可以如下定义：

```
#ifdef PR_DEBUG
#define LOG(x) std::cout << x << std::endl;
#else
#define LOG(x)
#endif
```

如果是 debug 模型，把 PR_DEBUG 定义出来，这样就会输出日志，否则就不输出。宏必须是要在同一行，所以如果一行写不下，可以用

跳到下一行写。

2.40 auto 关键字

auto 让模型自动推断变量的类型。emmmmm 说实话我一般不会经常用 auto，除非是接收函数返回的时候，因为这会严重影响可读性。

2.41 静态数组 array

和动态数组 vector 相对应的，他一出生就决定了他的大小，不能改变 size 了。

```
std::array<int, 5> data_array;
```

这样定义的静态数组有个问题：不知道他的个数，没办法传递参数到函数里面，因为传递函数需要把这个静态数组的完整类型写出来。这个时候可以使用模板

```
template<typename T> void print_array(const T& value)
{
    for(int i = 0; i < value.size(); i++)
    {
        LOG(value[i]);
    }
}
```

我原本想着用 auto 让模型自行推断，但是不行。相比直接创建的数组，`std::array` 数据结构有边界检查。应该尽量用 `std::array` 替代 C 语言的数组：

- 首先他提供了一系列的安全检查
- 其次并没有声明性能损失，还能记录数组大小。

2.42 函数指针

获得函数指针之后可以通过指针调用这个函数

```
auto function_pointer = hello_world;
function_pointer();
```

`function_pointer` 的类型是 `void(*function_pointer)`，`function_pointer` 只是一个名字，可以随意取。

定义一个函数指针有两种方法

- 一种是直接定义，`void (*fun_name) (int);` 直接定义一个 `fun_name` 的函数指针，`fun_name = hello_world;` 将这个函数 `hello_world` 赋值给函数指针。
- 第二种是先把函数指针类型定义出来 `typedef void (*fun_name) (int);` 再根据这个类型把变量定义出来 `fun_name fun = hello_world;` 第二种方式对后续使用更友好。
- 第三种就是直接 `auto` 定义了。

函数指针可以让函数进行参数的传递

```
void print_array(const T& array, void (*prt_array)(int))
{
    for(int i = 0; i < array.size(); i++)
    {
        prt_array(array[i]);
    }
}
```

直接把函数名称传递进去就行。

2.43 命名空间

为什么不用 `using namespace std;` 这里说的是作者为什么不用，我经常用其实。

作者不爱用这玩意的原因：

- 显示的写出这些命名空间，可以知道是使用哪一个库的内容。比如使用 `std` 就能知道是再 C++ 标准库使用的。
- 可能出现命名空间的冲突，如果两个命名空间里面都有同名称的函数，很容易就冲突了

为什么要使用命名空间？这是下一节的内容。

- 简单来说，就是为了解决在大工程中容易出现命名重复的问题。
- 命名空间是在限定作用域下进行。

2.44 线程

目前我们所完成的代码都是单线程的，一次只能做一条指令。但实际上多线程在日常中很常见，比如在制作游戏的时候，我们通常需要等待用户输入指令，但是如果主线程等待用户输入指令，那么整个游戏就会停下来，所以我们需要其他线程来接受用户指令，主线程运行。

main 就是一个主线程，定义一个函数

```
void DoWork()
{
    while(!is_finishing)
    {
        std::cout << "working" << std::endl;
    }
}
```

在主函数中，我们定义一个线程去运行这个函数

```
std::thread worker(DoWork);
```

这样我们就定义一个线程去运行这个函数，但是只是定义还没开始

```
std::cin.get();
is_finishing = true;
```

表示输入任一按键推出子线程。

```
worker.join();
```

开始运行线程，并且主线程会等待子线程运行完毕。还有另一个 detach，不过这个是主线程不等待子线程运行完毕。

2.45 多维数组

多维数组是数组的数组，把一个数组里面的元素换成数组即可。其实就是在创建了很多很多个数组，然后把指向这些数组的指针收集再一起，就变成了二维指针。多维数组就相当于多层次指针的意思。

```
int** a2d = new int*[50];
```

a2d[0] 是一个指针，指向的是个数组。上面那行代码我们只是初始化了一系列指针，并没有实质分配空间，需要用 for 循环挨个挨个分配：

```
for(int i = 0; i < 50; i++)
{
    a2d[i] = new int[50];
}
```

删除的时候不能直接用 delete 删除，如果直接 delete[] a2d，你只是删除了这一堆指针，而指针指向的内存并没有删除，删除也是需要循环删除，否则会造成内存泄漏。

二维数组如果这样初始化是会存在内存不连续的问题，因为每一个指针之间的内存不是连续的，导致运行效率的问题。up 提到了个 cache miss，意思就是每一次系统会把当前访问内存的一些邻居内存的内容也收进 cache 里面，因为系统认为相邻的内存的访问时空频率会相近。如果内存不连续，那么这样的假设不成立，会造成频繁的 cache miss。

二维数组不是连续的，那么一维数组是连续的吧？用一维数组去模拟二维数组就行了。这个比较简单，常见操作了。

2.46 排序

用 STL 了基本上就是。最简单的用法

```
std::vector<int> values = {3, 5, 4, 2, 1};
std::sort(values.begin(), values.end(), std::greater<int>());
for(int value : values)
{
    std::cout << value << std::endl;
}
```

第二行排序，按照降序排列。当然也可以自定义，这比较容易。后面再看很多代码的时候大部分都是自己实现的，因为不同排序在不同情况下效率不同。

2.47 Union

定义多个变量占用同一个内存，大小为最大的那个变量。定义一个结构体

```
struct test
{
    union
    {
```

```
    struct
    {
        float a, b, c, d;
    };

    struct
    {
        Vector v1, v2;
    };

};

};


```

如果修改 c 和 d 的值，会发现 vector v2 的 xy 的值也会随之修改。这个 Union 想要达成的目的是想设置一个变量能够存储多个不同类型的变量，不用定义多个类型。

2.48 虚析构函数

这个问题和之前的虚函数一样，都是解决编译器无法识别内存实际存储变量类型的问题。

```
class Base
{
public:
    Base() {std::cout << "Base" << std::endl;}
    ~Base() {std::cout << "Delete Base" << std::endl;}
};

class Derived : public Base
{
public:
    Derived() {std::cout << "Derived" << std::endl;}
    ~Derived() {std::cout << "Delete Derived" << std::endl;}
};
```

定义一个类，另一个继承。在初始化的时候，显示调用的构造函数，模型自然是知道这个类就是 Derived，但是删除的时候模型并不知道这个内存里面的内容是哪一个类，只能通过类型判断，这个时候就需要使用虚函数了。如果析构函数不对，会出现内存泄漏。

只需要在析构函数上上 virtual 即可，他会创建一个新表记录。

2.49 类型转换

主要还是显示类型转换，定义 double 变量，如果强行赋值给 int 变量，编译器会默认使用隐式转换，因为这样会有精度丢失，当然可以指定显示转换 static_cast 等。这些 cast 会自带编译器的一些安全检查，会有一定的速度损失。

作者后面给了一堆例子，没看太懂，暂不记录。cast 的类别有四类

- static_cast: 针对的是精度算是较小的转换，比如浮点和整数，不能进行指针的转换
- reinterpret_cast: 用于不同类型指针的强转，常用于继承类之间的转换
- const_cast: 去除 const 属性
- dynamic_cast: 不带安全检查，仅仅只在运行的时候进行检查，如果失败返回 null。这个可以用来判断一个类是否是其子类，当然也有 instance 可以用。

2.50 Safe

C++ 编程中的安全是指降低崩溃，内存泄漏，非法访问等问题。随着 C++11 的到来，更应该转向智能指针而不是原始指针。这是作者说的。

- 内存泄漏问题：在一个堆上分配内存，如果不人为进行删除，那么这个内存将一直存在。如果存在循环的话这个内存就会一直分配。
- 内存清理的所有权问题：由谁来清理管理内存。

这集 up 估计是被评论喷了回一波，感觉没什么内容在里面，过。

2.51 预编译头文件

主要解决的问题就是编译速度过慢。每一次在头文件引入 vector, string 这些头文件的时候，都需要重新对 vector, string 这些头文件代入的代码进行重新编译，速度会慢很多。

预编译头文件就是引入他们的二进制格式的文件，这样会加快编译。

2.52 dynamic_cast

类型转换四种之一。C 语言的类型转换直接在变量前加上类型即可，C++ 的类型转换有四种 static_cast, dynamic_cast, reinterpret_cast, const_cast。

`dynamic_cast` 只适合多态类之间的转换，可以通过这种方式判断一个当前的实体是否是可以转换的，或者确认当前实体是否是某个类的子类。

编译器是如何知道不能转换的呢？因为在运行的时候编译器存储了类运行时的信息，RTTI。这是会增加开销，

2.53 结构化绑定

作者提到了常用的接受多个返回值的方法。用 `tuple` 接收：

```
std::tuple<std::string, int> CreatePerson()
{
    return {"LY", 24};
}
```

这个 `tuple` 接收两个返回值，一个 `string`，一个 `int`。一种是用 `get` 取出值：

```
auto person = CreatePerson();
std::string& name = std::get<0>(person);
int age = std::get<1>(person);
```

另一种是 `tie`，相比来说 `tie` 更整洁

```
std::string name;
int age;
std::tie(name, age) = CreatePerson();
```

如果使用结构化绑定，不需要先定义两个接收变量的类型，直接使用即可：

```
auto [name, age] = CreatePerson();
```

不过这个特性是 17 后生效。现在 22 年了大家基本都是大于 17 版本肯定生效的。

后面两章介绍 17 新特性的，先跳过。

2.54 如何能让 C++ 字符串更快？

前面有提到过，尽量在栈内分配，避免在堆上分配。`string` 就是在堆上分配的，但是我阅读了一下 `string` 的源码，如果是在 16 字节内，他就是在换存池里面分配，也就是栈，如果是大于 16 字节那就是在堆上分配。事实上这也是很自然的，毕竟栈内存在编译后就确定了，如果是动态变量需要很大的内存空间，栈是满足不了的。

```
std::string name = "hajksdhjkkqwey";
std::string first = name.substr(0, 3);
std::string second = name.substr(4, 9);
```

这三步每一步都会分配内存，一共在堆上分配了三个内存。我们并不希望他分配内存，只需要能 access 到这个 name 就可以。

string_view 可以达成，string_view 只是记录了当前 string 的指针和偏移量，也就是说他只是 string 的一个指针类型，相比 string 需要分配更大的空间，string_view 明显效率更高。

```
std::string_view first(name.c_str(), 3);
std::string_view second(name.c_str(), 4);
```

c_str 返回一个指向当前字符串的指针，比如 char* c = a.c_str()，本身就是用来兼容 c 的。这样就只需要分配一次了。如果要把剩下的这个去除，只需要把 string 修改即可：

```
const char* name = "hajksdhjkkqwey";
// std::string first = name.substr(0, 3);
// std::string second = name.substr(4, 9);

std::string_view first(name, 3);
std::string_view second(name, 4);
```

这样一次分配都没有。

2.55 单例模式

这应该是常见的一种设计模式。当我们想让当前数据共享于所有的类或者结构体的时候，并且我们希望保证数据的一致性，单例模式就非常有用。比如渲染器就是一种全局通用的单例，不会因为不同物体就采用不同的渲染器。当然渲染也分很多种，有体渲染，也有面渲染，但他们都只需要创建一个就行。

这个比较简单，基本上面试笔试都要写。主要就是满足对于每一次调用都只能共用一个实例就行。创建

```
class Singleton
{
public:
    Singleton(const Singleton&) = delete;
```

```
    static Singleton& Get()
    {
        return s_Instance;
    }

private:
    Singleton() {}
    static Singleton s_Instance;
};


```

构造函数放在 private 是不允许显示调用，public 中还需要静止复制，在 private 初始化然后 public 写个窗口供调用就行。

2.56 小字符优化

小字符优化也叫 SSO，字符占用字节如果小于 16 字节，那么会在栈的缓冲区分配内存，如果超过了，就会堆上返回。这里的源码不一样，要找 if CXX14 的源码看，他看的是旧版源码，新的用的 construct 分配。

在 xstring 源码可以看到

```
if (_Count < _BUF_SIZE) {
    _My_data._Mysize = _Count;
    _My_data._Myres = _BUF_SIZE - 1;
    if constexpr (_Strat == _Construct_strategy::_From_char) {
        _Traits::assign(_My_data._Bx._Buf, _Count, _Arg);
    } else if constexpr (_Strat == _Construct_strategy::_From_ptr) {
        _Traits::move(_My_data._Bx._Buf, _Arg, _Count);
    } else { // _Strat == _Construct_strategy::_From_string

```

小于特定的 buf_size 的时候不会分配堆内存。

2.57 跟踪内存分配

主要还是为了性能服务，在写安全协议的时候也用到，不过是借助了其他语言工具。就是自己写一个能记录内存分配的类或者工具，感觉还是用给定好的包或者库好用。

2.58 左值和右值

最简单的一个例子

```
int i = 10;
```

左值一般在左边，右值一般在右边。但这种规则并不总是适用，比如 `int a = b` 这就不适合，两个都是左值。

- 左值一般来说是可寻址，有一定内存指向的变量。右值一般是临时变量。
- 赋值给左值是可以的，给右值不行，而左值是要非 `const` 可修改的。
- 左值引用，只能是给左值。
- `const` 左值引用可以给右值，`const int& a = 10`。实际上是创建了一个临时变量。所以 `const` 左值引用是兼容左值和右值的。这也是为什么 `string` 的调用通常使用 `const` 的原因，因为兼容左值和右值。

```
std::string a = "sjkdf";
std::string b = "aksla";
std::string c = a + b;
a + b;
```

`c` 是左值，`a+b` 是右值。因为在 `c = a + b` 的过程中调用了 `string` 的构造函数，而 `a + b` 没有，就是一个右值。

2.59 参数计算顺序

跳过了，更考试题一样，这节的内容谁写进工程代码里面我砍谁。

2.60 移动语义

为什么需要移动语义？

- 通常我们只是希望创建出一个临时变量给构造函数，如果在主进程创建一个变量然后传到构造函数，然后构造函数会复制这个变量。

一个简单的例子

```
class String {
public:
    String() = default;
    String(const char* string) {
```

```

        printf("Created!\n");
        m_Size = strlen(string);
        m_Data = new char[m_Size];
        memcpy(m_Data, string, m_Size);
    }

String(const String& other) {
    printf("Copied!\n");
    m_Size = other.m_Size;
    m_Data = new char[m_Size];
    memcpy(m_Data, other.m_Data, m_Size);
}

~String() {
    delete[] m_Data;
}

void Print() {
    for (uint32_t i = 0; i < m_Size; ++i)
        printf("%c", m_Data[i]);

    printf("\n");
}

private:
    char* m_Data;
    uint32_t m_Size;
};

class Entity {
public:
    Entity(const String& name)
        : m_Name(name) {}
    void PrintName() {
        m_Name.Print();
    }
private:
    String m_Name;
};

```

创建变量 entity 的时候，传入 entity(string("a")), 首先创建 string，然后在构造函数又复制一次。我们可以强制他传右值，把 string 的构造函数去掉。增加

```
String(String&& other) {
    printf("Moved!\n");
    m_Size = other.m_Size;
    m_Data = other.m_Data;
    other.m_Data = nullptr;
    other.m_Size = 0;
}
```

entity 构造函数也改成只接收右值

```
Entity(String&& name)
: m_Name(name) {}
```

但实际上还是没能解决问题，因为右值传进来的时候就已经退化了，在 entity 的构造函数传递 name 的时候就变成了左值，这个时候用 std::move 即可。**move** 实际上是移动资源，有时候需要额外操作防止内存泄漏。所以通常需要首先删除当前变量资源，然后赋值，然后删除赋值变量的资源。

3 C++ 小游戏

https://www.bilibili.com/video/BV14z4y1r7wX/?spm_id_from=333.999.0.0&vd_source=a059a118f33728f79abd79e02f8f72d4 的教程，自己的理解以及一些改进。

3.1 俄罗斯方块

创建方块，这里一共 7 个，作者用的 wstring，wstring 每一个字符占用 2 个字节，我看他做的都是字母，感觉 string 应该也行。

```
std::wstring tetromino[7];
```

然后挨个挨个赋值就行。

3.1.1 坐标

坐标这个问题比较重要，他涉及到碰撞检测，移动，显示等的问题。正常的坐标在左上角开始，xy 两个方向 0 开始增加：

```
/*
0 1 2 3 4 5
0
1
2
3
4
5
*/
```

作者用一维数组表示，每一个小格子用一个 index 表示，从左上角开始 0 一直编号下去

```
/*
0 1 2 3 4 5
6 7 8 9 10 11
*/
```

对于在 (x, y) 的点，用一维数组表示就是 $\text{index} = y * w + x$ 。当他顺时针旋转 90 度 $\text{index} = 12 + y - (x * h)$ ，当顺时针旋转 180 度 $\text{index} = 15 - (y * w) - x$ ，当顺时针旋转 270 度 $\text{index} = (y * w) + (h - x - 1)$

3.1.2 前期工作准备

没有写过 win，这些 API 看的头疼，只能一个个查了。首先区域分为三个区域

- 旋转区域：这个区域是方块按照玩家指示旋转的，上面提的顺时针旋转这些就是。
这个区域作者没有写出来，只是约定俗成了就是 4×4 我还以为是整个屏幕坐标
的，所以那四条公式我都写成 w 和 h 。但实际上就是再 4×4 的格子进行旋转。
- 玩家区域：也就是游戏区域，一整个窗口不会都是游戏。
 $pField$ 就是玩家区域，注意这个 $pField$ 只是会记录固定的方块，也就是说你这
个方块如果是在下落过程中，是不会记录的。只有落下了才会记录。因为 $pField$ 实
际上是就是游戏环境，而下落的方块不是游戏环境，是玩家。就好像迷宫一样，人是
玩家，迷宫就是这个 $pField$ 。
- 窗口区域：显示给用户看的整个区域。
 $screen$ ，需要把 $pField$ 移动到 $screen$ 才能显示。

玩家控制区域的建立

```
pField = new unsigned char[nFieldWidth * nFieldWidth];  
for (int x = 0; x < nFieldWidth; x++)  
    for (int y = 0; y < nFieldHeight; y++)  
        // 边界设置为9否则都设置为0  
        pField[y * nFieldWidth + x] = (x == 0 || x == nFieldWidth - 1 || y ==  
            nFieldHeight - 1) ? 9 : 0;
```

0 的地方是可以进行移动，9 的地方是边界。窗口的创建需要了解几个 API:

- CreateConsoleScreenBuffer: 文档地址 <https://learn.microsoft.com/zh-cn/windows/console/createconsolescreenbuffer>。
 - 作用的创建控制台缓冲区。第一个参数是访问权限，代码里面是 read 和 write 都可以。第二个参数表示缓冲区是否共享，意思就是这个缓冲区能不能被其他线程 access，肯定是不能，所以设置为 0。第三个参数表示句柄能不能让子线程继承，NULL 就是不能了。第四个参数是类型，这个类型没得选。最后一个参数是否包含这个数据，NULL 表示保护。
 - 返回句柄，表示创建了一个能被主进程读写，不能被其他进程共享，不被继承并且数据被保护类型指定的缓冲区。
- SetConsoleActiveScreenBuffer: 将指定缓冲区设置为屏幕控制缓冲区。emmm 人家文档好像不建议用这个，写完再看看。
- WriteConsoleOutputCharacter: 将字符写入到缓冲区。
 - 第一个参数表示写入的缓冲区，要求 access 必须是可写的。第二个参数要写入的字符。第三个参数是写入的字符数，第四个参数写入的字符坐标，默认 00 就是在左上角，如果想要在中间那可以在这调整，第五个参数表示一个指针，表示实际写入的字符数，作者用 DWORD 接收，这是 MFC 的数据类型，32 位无符号数。

所以他是先创建了一个缓冲区，然后把这个缓冲区设置成屏幕的缓冲区，让他显示在屏幕上，每一次游戏更新，把更新的数据写进缓冲区就行。作者的缓冲区用 wchar 表示，看代码实例里面大多数都是用一个共用缓冲区的联合体表示。

在这里运行的时候出现一个问题，在对 wchar_t 进行内存分配的时候，居然出现分配错误，一开始以为是 os 内存不够，但是后面看了一下是完全够的。后面看了一下是屏幕缓冲区的问题，调大点就没事了。

然后在测试的时候发现，屏幕显示有问题，这个窗口大小太大了。MoveWindow, SetWindow, Setbuffer 这些都没有效果，返回值显示都成功了，但就是没效果。后面上外网看了一下，他这个是因为 visual studio 和 clion 都强制设置了窗口大小，对没错，为了解决这个问题我还下了 clion。后面我在 dev c++ 上用 MoveWindow, SetWindow, Setbuffer 方法是可以的。说明这个问题是和编译器相关。反正只需要手动调整这个窗口大小就行了。

3.1.3 碰撞检测

用 0 表示空区域。方块的旋转是在一个局部空间内进行，先在局部空间内进行旋转，然后再转移到全局坐标上去。

```
if (nPosX + px >= 0 && nPosX + px < nFieldWidth)
    if (nPosY + py >= 0 && nPosY + py < nFieldHeight)
    {
        if (tetromino[nTetromino][pi] == L'X' && pField[fi] != 0)
        {
            return false;
        }
    }
}
```

碰撞检测主要就是判断当前点移动之后，会不会碰到其他的点。如果是空白区域那就是 0，否则就是不是。

3.1.4 控制移动

检测按键

```
for (int i = 0; i < 4; i++)
{
    // 检查最高位是不是1
    // 虚拟键码里面有，左，右分别对应38, 37, 39，十六进制对应27 25 28，还有一个Z用来变换
    bKey[i] = (0x8000 & GetAsyncKeyState(static_cast<unsigned
        char>("\x27\x25\x28Z"[i]))) != 0;
}
```

GetAsyncKeyState 主要是检测按键，x27 表示 27 是一个十六进制的。用虚拟键码表示按键。0x8000 做与或运算，和最高位对比，如果最高位是 1 说明就按下了，接下来就是移动了。

```
nCurrentX += (bKey[1] && DoesPieceFit(nCurrentPiece, nCurrentRotation, nCurrentX -  
    1, nCurrentY)) ? 1 : 0;  
nCurrentX += (bKey[0] && DoesPieceFit(nCurrentPiece, nCurrentRotation, nCurrentX +  
    1, nCurrentY)) ? 1 : 0;  
nCurrentY += (bKey[2] && DoesPieceFit(nCurrentPiece, nCurrentRotation, nCurrentX,  
    nCurrentY + 1)) ? 1 : 0;
```

旋转需要注意的是如果不加限制，会一直旋转的，因为 nCurrentRotation 一直都不会是 0，所以需要一个 flag 来确定是否需要旋转。

```
if (bKey[3])  
{  
    nCurrentRotation += (!bRotateHold && DoesPieceFit(nCurrentPiece,  
        nCurrentRotation + 1, nCurrentX, nCurrentY)) ? 1 : 0;  
    bRotateHold = true;  
}  
else  
    bRotateHold = false;
```

bRotateHold 来确定当前是否是继续旋转。不能直接把 nCurrentRotation 设置成 0，因为每一次旋转都是接着上一次。

3.1.5 下一块

首先要添加一个自动下落，这个比较容易，bForceDown 表示就行。

```
if (DoesPieceFit(nCurrentPiece, nCurrentRotation, nCurrentX, nCurrentY + 1))  
// 到底了  
{  
    ++nCurrentY;  
}
```

如果能下落，就继续下落，如果下一个位置是不能下落的，那么就不下落了。既然不下落了，那就要把这个物体固定在环境中，就是固定在 pField 中，前面提到过了这个 pField 实际上是游戏环境，方块是玩家，方块到底就变成一个环境了。就好像 CS 人在移动的时候他就是玩家，死了就变成环境的一部分。

```
for (int px = 0; px < 4; px++)  
    for (int py = 0; py < 4; py++)  
    {
```

```
if (tetromino[nCurrentPiece][Rotate(px, py, nCurrentRotation, 4, 4)] == L'X')
{
    pField[(nCurrentY + py) * nFieldWidth + (nCurrentX + px)] = nCurrentPiece + 1;
}
}
```

这里要注意的是 `tetromino[nCurrentPiece][Rotate(px, py, nCurrentRotation, 4, 4)]` 表示当前物体的状态，这里为什么要有一个 Rotate 呢？因为这个控制旋转的逻辑不是 subsequent 的，并不是说点击一次旋转，0 到 90 度，再点击一次 90 度到 180 度。而是点击一次 0 到 90 度，再点击一次 0 到 180 度。

这段代码相当于是把当前方块给画出来了。其实这个代码重复了很多遍，可以写成一个函数。设置下一块的话重新设置就好了

```
nCurrentX = nFieldWidth / 2;
nCurrentY = 0;
nCurrentPiece = rand() % 7;
nCurrentRotation = 0;
```

3.1.6 删除

主要是指删除某一行。这个检测是否一行都有方块并不是什么时候都要检查，肯定是等下落到底了之后再检查。只需要检查四行就行，因为当前这个方块就占 4 行，如果超过这个范围就不关你的事了，是之前或者之后下落的物体要检查的。

```
for (int py = 0; py < 4; py++)
{
    if (nCurrentY + py < nFieldHeight - 1) {
        bool bLine = true;
        for (int px = 1; px < nFieldWidth - 1; px++)
        {
            // 当前这个高度，全部的px都不是0，也就都不是空
            bLine &= (pField[(nCurrentY + py) * nFieldWidth + px]) != 0;
        }
        if (bLine)
        {
            // Remove Line
            for (int px = 1; px < nFieldWidth - 1; px++)
            {
```

```

        pField[(nCurrentY + py) * nFieldWidth + px] = 8;
    }
    // 要删除的行
    vLines.push_back(nCurrentY + py);
}
}
}

```

当前整行都不是空，那么就把这行放进 vLines 里面，等会删掉他。

```

for (auto& v : vLines)
{
    for (int px = 1; px < nFieldWidth - 1; px++)
    {
        for (int py = v; py > 0; py--)
        {
            pField[py * nFieldWidth + px] = pField[(py - 1) * nFieldWidth + px];
        }
        pField[px] = 0;
    }
}
vLines.clear();

```

`pField[px] = 0;` 这一句我一开始不知道是干什么，后面想了一下，我觉得应该是处理特殊情况的。当你所以方块都填满了整个 `pField`，你这个下移动只会影响 1 到 `nFieldHeight-1` 行，第 0 行是没有东西可以给他降下来，没有-1 行，所以这里额外需要处理一下。

3.1.7 渲染

`pField` 和移动方块是两种不同的物体，一个是环境一个玩家，自然渲染不一样。

```

for (int x = 0; x < nFieldWidth; x++)
for (int y = 0; y < nFieldHeight; y++)
// 乘上nScreenWidth是因为整个窗口长度就是nScreenWidth, 所以需要乘上这一行
// 边界是#号, 其他地方为空格
screen[(y + 2) * nScreenWidth + (x + 2)] = L" ABCDEFG=#"[pField[y * nFieldWidth
+ x]];

for (int px = 0; px < 4; px++)
for (int py = 0; py < 4; py++)

```

```

{
    if (tetromino[nCurrentPiece] [Rotate(px, py, nCurrentRotation, 4, 4)] == L'X')
    {
        screen[(nCurrentY + py + 2) * nScreenWidth + nCurrentX + px + 2] =
            nCurrentPiece + 65;
    }
}

```

如果想显示什么分数啊什么的，再 pField 写就行了，如果想增加新的方块，那就要额外渲染。

3.2 第一人称射击游戏

游戏完成如图 1 所示。

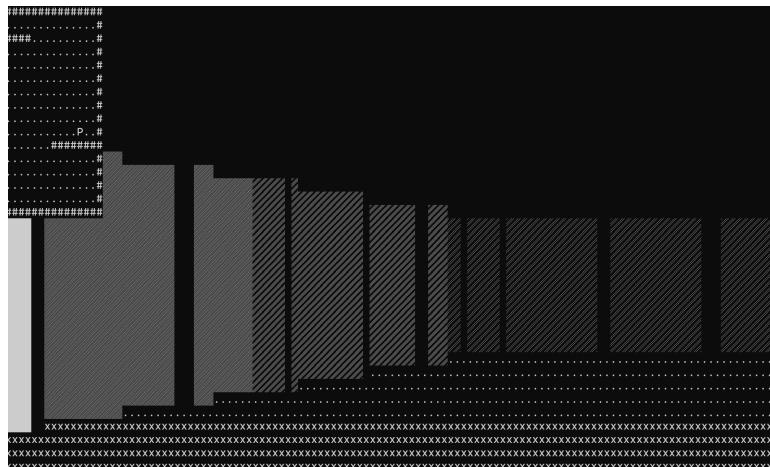


图 1: fps 游戏示意图

场景中主要有两样内容：墙壁，玩家。最主要的难点在于三维视觉的呈现，比如近大远小这些特征都需要数学几何的计算。

```

wchar_t* screen = new wchar_t[nScreenWidth * nScreenHeight];
HANDLE hConsole = CreateConsoleScreenBuffer(GENERIC_READ | GENERIC_WRITE, 0, NULL,
    CONSOLE_TEXTMODE_BUFFER, NULL);
SetConsoleActiveScreenBuffer(hConsole);
DWORD dwByteWritten = 0;

```

前面的俄罗斯方块展示过了，常规操作，显示在屏幕上。用 cout 太慢了。

3.2.1 玩家控制和碰撞检测

```
if (GetAsyncKeyState(static_cast<unsigned short>('A')) & 0x8000)
{
    // 一帧的时间很快，需要加以限制
    fPlayerA -= ((0.8f) * fElapsedTime);
}

if (GetAsyncKeyState(static_cast<unsigned short>('D')) & 0x8000)
{
    fPlayerA += ((0.8f) * fElapsedTime);
}
```

按下 A 和 D 玩家的朝向会像左像右移动。fPlayerA 表示玩家的角度。初始化角度设置成 0，那么他的朝向就是 $\sin(A)$, $\cos(A)$ ，也就是向下。fps 游戏是一个第一人称的游戏，所以我们以玩家为局部坐标的中心，朝向为 0 度，左边角度为负，右边角度为正。按下 A 和 D 未变角度减小和增加即可。

乘上 fElapsedTime 是为了移动的更丝滑。计算方法

```
tp2 = std::chrono::system_clock::now();
std::chrono::duration<float> elapsedTime = tp2 - tp1;
tp1 = tp2;
float fElapsedTime = elapsedTime.count();
```

4 Python 及其相关练习

课程链接

- https://www.bilibili.com/video/BV1wQ4y1q7Bm/?spm_id_from=333.788&vd_source=a059a118f33728f79abd79e02f8f72d4

4.1 Python 的一些 API

张量的核心操作

- chunk：把张量分割成指定数量的 tensor。如果维度是不能被整除的，那么最后一个张量就只能留下来了。

- 如果张量的 shape 是 [3, 2], chunk(a, chunks=2), 那么就会分成 [2, 2] 和 [1, 2] 这两个张量。所以 chunK 的三个参数分别是，需要 chunk 的张量，需要拆分 tensor 的数量，拆分的维度默认是 0.
- gather: 这个比较重要。沿着不同的维度取变量。参数分别有三: input, 输入的张量; dim, index 的维度; index, 需要 gather 的维度。
 - gather 和一般检索不同的地方在于，检索的维度是可以变化的。比如 tensor([[1, 2], [3, 4]]), 使用 gather(t, 1, tensor([[0, 0], [1, 0]])), 按照维度 1 进行检索, tensor([[0, 0], [1, 0]]) 的四个索引分别是 (0,0), (0, 1), (1, 0), (1, 1), 维度 1 上索引，所以把维度为 1 的那块替换了，变成 (0,0), (0,0), (1, 1), (1, 0)。取出来就是 1, 1, 4, 3。这个比较适合那种 NLP 任务 seqence 序列取值的时候使用。
- reshape: 这个函数就是把 shape 重排，这个就重要的多了。这个 reshape 其实没有改变内存的存储顺序。
- scatter: 参数包含 input index src reduce, 将 src 的张量写入当前调用了这个变量的 tensor 中，通过 index 取指定。这个写入和 gather 一样，按照 index 这个维度写入。
- scatter_add: 这个就是不是替换了，找到相应的维度相加
- split: 和 chunk 类似，如果划分大小是正整数，和 chunk 一样，如果划分是 list，会按照列表里面的数字划分，这也是他和 chunk 的区别，可以指定划分，chunk 是不可指定划分。比如 [6, 3], chunk 只能均分，但是 split 可以输入 [1, 2, 3], 划分成 [1, 3], [2, 3], [3, 3]
- squeeze: 压缩，如果一个维度是 1，那么就可以用 squeeze 对其进行压缩
- unsqueeze: 升维，和 squeeze 反过来
- stack: 把不同的 tensor 进行拼接，他和 cat 的区别在于，cat 是不会创造新的维度，stack 是会创建新的维度。
- dtype: 数据类型。一般就是 32, 64, 16 位。float64 和 double 是相同的，都表示双精度。浮点数由三部分组成
 - Sign bit: 符号位，表示正负，如果是 32，那么符号位是 1
 - Exponent: 指数位，如果是 32 位，那么指数位是 8

- Significand Precision: 精度位, 如果是 32, 24 位
- take: 按照索引取值, 把这个张量铺平, 按照 1D 去索引取值。
- tile: 复制, 对 tensor 不同位置进行复制就需要用到。tile 都有一个 dim 的参数, 表示对哪一个维度复制几份。如果维度没有给出, 那么默认是 1.
- unbind: 把某个维度展开成 tuple。比如一个 3×3 的 tensor, 指定 dim=0, 表示沿着 0 维度进行展开, 于是返回三个张量。
- unsqueeze: 增加维度, 在广播时候可能会用到。
- where: 判断, 根据条件觉得返回是 x 还是 y, where(condition, x, y), 如果满足 condition, 那么保留 x, 否则就把 y 替换过来。

4.2 Dataset 与 DataLoader

加载数据集的包, 这算是 pytorch 的标准加载方式了, 其实很多代码里面都不用这个, 比如 nerf 里面除了一些 pl 的代码, 其他的就很少会用 dataset 这些包。主要涉及两个

- Dataset: 用来处理单个训练样本, 如何读取数据, 变成单个训练样本
- DataLoader: 对于多样本, 用 Dataset 得到当个样本后, 在用 loader 变成 batch 的形式

4.2.1 Custom Dataset

继承 Dataset 类, 然后需要自定义的 Dataset 必须要实现三个函数

- `__init__`: 通常需要传入需要传入数据集所在目录, 主要就是知道数据保存在哪里, 包括 label 的路径, 数据的路径
- `__len__`: 返回数据的大小
- `__getitem__`: 通过 index 返回样本, 如何实现可以自己定义, 输入就是 idx, 输出就是数据和 Label

Dataset 实现好之后, 就是装载到 DataLoader, 参数

- `dataset`: 上面实现好的 Dataset 类
- `batch_size`: 一个 batch 的大小

- shuffle: 是否需要打乱
- sampler: 如何采样
- num_worker: 就是开启多进程, 默认使用主进程进行加载
- pin_memory: 把数据都保存在 GPU 中, 这个不一定能提高效率
- drop_last: 如果当前的数据凑不齐一个 batch, 那就丢弃最后一批
- collate_fn: 对 sample 采样的批次做一个处理, 比如 NLP 里面可能会需要对一个 batch 做一个预处理

4.2.2 DataLoader 的源码

首先是对一些参数复制

```
self.dataset = dataset
self.num_workers = num_workers
self.prefetch_factor = prefetch_factor
self.pin_memory = pin_memory
self.pin_memory_device = pin_memory_device
self.timeout = timeout
self.worker_init_fn = worker_init_fn
self.multiprocessing_context = multiprocessing_context
```

如果不是 IterableDataset 数据类型, 直接跳过

```
if isinstance(dataset, IterableDataset):
    self._dataset_kind = _DatasetKind.Iterable
```

一般的数据都是 mapDataset 类型, 很少有 IterableDataset 类型的。

判断是否是需要 shuffle

```
if sampler is not None and shuffle:
    raise ValueError('sampler option is mutually exclusive with '
                     'shuffle')
```

如果你又指定了采样方式, 又指定了需要随机, 那就出问题了。反正只要你设置了 sampler, 你就不能设置其他的采样方式, 比如 shuffle, batch_size 等。如果没有设置 sampler, 那就根据 shuffle 选择

```
if sampler is None: # give default samplers
    if self._dataset_kind == _DatasetKind.Iterable:
        # See NOTE [ Custom Samplers and IterableDataset ]
        sampler = _InfiniteConstantSampler()
    else: # map-style
        if shuffle:
            sampler = RandomSampler(dataset, generator=generator) # type:
                ignore[arg-type]
        else:
            sampler = SequentialSampler(dataset) # type: ignore[arg-type]
```

如果是 shuffle, 选择随机采样器, 否则就是序列采样器。随机采用器也就是 RandomSampler, 主要用的就是 torch.randperm, 随机返回 0-n-1 的数值。序列采样直接就是一个迭代器 iter(range(len(self.data_source)))。

然后就是 batch 采样

```
if batch_size is not None and batch_sampler is None:
    # auto_collation without custom batch_sampler
    batch_sampler = BatchSampler(sampler, batch_size, drop_last)
```

按照 batch 来采样, 就是对上面生成的 sampler 进行一个 for 循环的 batch 的采样即可。核心在于这一段

```
batch = [0] * self.batch_size
idx_in_batch = 0
for idx in self.sampler:
    batch[idx_in_batch] = idx
    idx_in_batch += 1
    if idx_in_batch == self.batch_size:
        yield batch
        idx_in_batch = 0
        batch = [0] * self.batch_size
if idx_in_batch > 0:
    yield batch[:idx_in_batch]
```

sampler 就是把 Dataset 的数据一个个拼接成 batch 的形式返回。这只是返回了索引。初始化差不多结束, 大概就是三件事

- 构建 sampler, 主要就是随机或者顺序采样, 如果有自定义的也行

- 构建 batch Samplers: 把每一个采样出的样本集合成一个 batch
- 构建 collate_fn: 主要是后处理, 默认是 default collate 哪也没干

然后就是读取了, 官方案例说可以用

```
next(iter(train_dataloader))
```

所以只需要看

```
def __iter__(self) -> '_BaseDataLoaderIter':
    # When using a single worker the returned iterator should be
    # created everytime to avoid resetting its state
    # However, in the case of a multiple workers iterator
    # the iterator is only created once in the lifetime of the
    # DataLoader object so that workers can be reused
    if self.persistent_workers and self.num_workers > 0:
        if self._iterator is None:
            self._iterator = self._get_iterator()
        else:
            self._iterator._reset(self)
        return self._iterator
    else:
        return self._get_iterator()
```

主要就是 get iterator 这个方法。

```
def _get_iterator(self) -> '_BaseDataLoaderIter':
    if self.num_workers == 0:
        return _SingleProcessDataLoaderIter(self)
    else:
        self.check_worker_number_rationality()
        return _MultiProcessingDataLoaderIter(self)
```

如果是多进程就是 multi, 单进程就是 single。这个函数也就两步, 得到 batch 的 index, 然后通过 index 得到 data

```
def _next_data(self):
    index = self._next_index() # may raise StopIteration
    data = self._dataset_fetcher.fetch(index) # may raise StopIteration
    if self._pin_memory:
```

```
    data = _utils.pin_memory.pin_memory(data, self._pin_memory_device)
    return data
```

next data 虽然是能获得数据，但是并没有显示是被哪里调用了，而这个调用其实是写在了父类里面 `_BaseDataLoaderIter`。在这个类里面他写明了需要补充实现 next data 方法

```
def __next__(self):
    raise NotImplementedError
```

在下面的 `next` 函数中就调用了这个 `next data`。所以在

```
next(iter(train_dataloader))
```

就调用了这个 `next` 方法。

4.3 构建网络

4.3.1 transform

这个方法是对数据做预处理的，比如我们的模型是处理一个灰度图，那么这个 `rgb` 到灰度图的转换就可以写到这个 `transform` 里面，或者一些自定义的数据增强方法也可以用到这上面来。

4.3.2 Build Network

按照官方文档来，首先是导包，`torch.nn` 就是主要的网络包。

- 设置设备位置，是 GPU 还是 CPU
 - 定义网络，所有的层都需要继承 `nn.Module` 这个父类。
 - 这个网络通常是需要实现两个模块，`init` 和 `forward`，初始化和前向网络。
 - `init` 方法流程
-

```
class NeuralNetwork(nn.Module):
    def __init__(self):
        super().__init__()
        self.flatten = nn.Flatten()
        self.linear_relu_stack = nn.Sequential(
            nn.Linear(28*28, 512),
            nn.ReLU(),
```

```

        nn.Linear(512, 512),
        nn.ReLU(),
        nn.Linear(512, 10),
    )

    def forward(self, x):
        x = self.flatten(x)
        logits = self.linear_relu_stack(x)
        return logits

```

先调用父类的初始化，然后定义网络。要使用网络首先需要初始化，然后直接 model(x) 即可。

然后是模型的各个层

- Flatten: 两个参数 start 和 end, 从 start 到 end 维度把数据铺平。比如 [3, 2, 3], 如果 start=1,end=-1, 那么会变成 [3, 6]
 - 就是做一个铺平操作
- linear: 这个是线性层，两个参数，输入大小和输出大小。
- Relu: 激活函数，没有任何参数。ReLU 是一个类，而不是方法，方法应该再 functional 那个文件里面
- Sequential: 这个类是模块的容器
- Softmax: 分类必用的函数

4.4 Module

nn.Module 源码，核心方法

- add_module, 添加子模块
- apply, 递归的把方法运用到所有子模块，在自定义 cuda 算子或者自定义函数的时候也用到，目前我比较常用是两个，首先是参数初始化，其次是定义一些函数的 forward 和 backward 的时候用到。
- buffers, 这也是网络的参数，通过 register_buffer 定义的变量会通过这种方式返回。这种方式定义的变量不会参与梯度下降，但是如果你用 state_dict 存储的时候是会保存

下来。如果就只是在网络里面定义一个变量，那不会参与网络也不会存储在 state_dict 中。

- eval, 这个比较重要，还有 train, 区别就是在于 dropout 或者 batchnorm 这些方法
- get_parameters, optimizer 的时候会用到，你得告诉优化器哪些方法需要训练
- load_state_dict, 存储变量参数，上面提到的 buffer 是会存储在这里

上面是一些比较主要的方法了。

4.4.1 Module 源码

首先是 register_buffer 函数，这个函数加入的变量通常不能作为模型的训练参数的，比如 batchnorm 的参数，或者 nerf 里面 bbox 的范围，虽然不需要训练但是他们和当前的场景，网络相关。这里默认通过 Buffer 定义的变量都是持久的，也就是会跟随 state_dict 存储，当然你也可以设置成是不 persistent 的。

```
self.register_buffer('variable_name', torch.tensor(...))
```

就可以定义一个变量。

register_parameters 是比 buffer 更常用的方法，网络里面可训练参数肯定比 buffer 要多的。**parameter** 是一个类，并且是 **Tensor** 的子类，如果在 **Modules** 这个神经网络的类里面定义这个子类，那么会把这个 **parameters** 加入到参数列表里面。所以在网络里面要把这个训练参数写成 parameters 形式。一般的神经网络都会自己定义一个 get_params 的函数，因为不同参数可能需要不同的学习率。

get_submodule 得到模型的子网络。

get_parameters 得到模块的参数

```
module_path, _, param_name = target.rpartition("..")
```

target 是输入的目标字符串，最右边的. 号左边是模块路径，右边是模块名称。然后把路径下的所有的子模块都赋值给 mod 变量

```
mod: torch.nn.Module = self.get_submodule(module_path)
```

mod 存储了这个路径下所有的子模块。然后检查这个子模块集合里面有没有为什么要找的模块

```
if not hasattr(mod, param_name)
```

```
...
param: torch.nn.Parameter = getattr(mod, param_name)
```

最后还要审核一下是不是 Parameter 类，因为这个函数就是返回 parameters。所以这里的得到 submodule 是需要很完整的把整个路径整个过程给写出来。母类. 子类完完整整的写下才能找到。

get_buffer 是获取另一种不参与模型训练的变量。也是基于字符串，路径也要写全，要不然找不着。他的函数逻辑和前面也是一样，先解析 target 得到路径名称和变量名字，知道路径名称就相当于知道在那个位置了，get_submodule 找到所有可能的子模块，然后再子模块里面找 buffer 变量，最后确定一下是不是 buffer 类即可。

后面的一些函数熟悉使用之后基本都知道怎么实现。apply 函数居然是用递归实现的，我还以为是 for 出来的，深度遍历：

```
for module in self.children():
    module.apply(fn)
fn(self)
return self
```

怪不得有很多模型会自定义一个 apply，像 vgg 这种上百层的会消耗的栈内存就太多了。save 方法主要是保存模型参数。保存一般用 save+state_dict，提取模型用 Load_state_dict 加载状态字典。

4.4.2 Sequential

类似一个容器，把一些网络都集合起来只需要调用这个容器就能 forward 这个网络，这个都比较常用了。

init 方法，如果传入的是 orderdict 类型，也就是用 orderdict 把网络模块包裹起来，每一个模块用他的名字命名

```
add_module(key, module)
```

如果不是 orderdict，那就直接按照 01234 标号

```
add_module(str(idx), module)
```

剩下比较重要的就是 forward 了，直接遍历每一个模块过一遍就行。

4.5 autograd

autograd 自动微分，这是 torch 训练的主要组成部分。例子，假设目前存在一个计算图

$$z = wx + b$$

$$\begin{aligned}y &= \sigma(z) \\L &= \frac{1}{2}(y - t)^2\end{aligned}$$

如果需要计算梯度，是反向传播，先计算后面的再计算前面的。先对 L 本身求偏导，把 L 当成变量，求导：

$$\hat{L} = 1$$

然后对 y 求偏导

$$\hat{y} = y - t$$

要乘上一个 L 但是求导是 1 就跳过。最后对 z 求偏导：

$$\hat{z} = \hat{y}\sigma'(z)$$

然后对 w 求偏导，只有一个 x

$$\hat{w} = \hat{z}x$$

还有一个 b，求导是 1，乘上前面的

$$\hat{b} = \hat{z}$$

如果是对于矩阵求导，那就是雅可比矩阵。

`torch.gutograd` 就是自动计算梯度的引擎

```
import torch

x = torch.ones(5) # input tensor
y = torch.zeros(3) # expected output
w = torch.randn(5, 3, requires_grad=True)
b = torch.randn(3, requires_grad=True)
z = torch.matmul(x, w)+b
loss = torch.nn.functional.binary_cross_entropy_with_logits(z, y)
```

官方例子，我们只需要设置哪一些变量是需要求解梯度，把前向传播的等式写好即可，模型会帮我们自动把梯度求解好。`grad_fn` 可以查看当前这个变量的操作到底是什么操作，比如是加法还是减法等。比如 z 的 `grad_fn` 就是一个加法操作。

backward 求梯度。backward 是 tensor 的一个方法，如果直接 loss.backward，那么 loss 需要是一个标量才能跑，如果你的 loss 不是标量，那么你需要 mean 或者 sum 变成标量。如果需要求两次导数，那么就需要在 backward 输入 retain_graph=True。因为 torch 中的计算图是动态图，计算完就销毁了，求不了第二次导数，只能求一次，retain 之后计算图保留就能多求几次导数。

如果不要求梯度，比如推理的时候就 no_grad() 即可，如果模型非常复杂，通常可以省下大量的内存。另一个不要求梯度的方法是 detach，不过 detach 是使得某一个变量不求梯度，而不是全部。

4.5.1 雅可比矩阵

jacobian 就可以求，但是搞这么久科研了好像没见到有用这个，可能 SDF 那块有用到？

4.6 Training model

如何真正的开始训练一个模型？按照官方案例

```
training_data = datasets.FashionMNIST(
    root="data",
    train=True,
    download=True,
    transform=ToTensor()
)

test_data = datasets.FashionMNIST(
    root="data",
    train=False,
    download=True,
    transform=ToTensor()
)
```

首先导入新的数据集，这个数据集可以通过 torch 自带的数据直接导入，然后就是用构建 dataloader

```
train_dataloader = DataLoader(training_data, batch_size=64)
test_dataloader = DataLoader(test_data, batch_size=64)
```

这个 dataloader 有点封装的太好了，有些任务不是很方便，比如 nerf 一次根据一张图片获取射线，这个时候 col fn 要重写，get item 也要大改。

接下来是构建网络，每一个网络类都需要继承 Module 类

```
class NeuralNetwork(nn.Module):
```

然后定义层，再定义参数即可。几乎所有的任务的 train 都可以分为三部分

- forward
- 计算 loss
- backward

还介绍了 embedding 层，这个层相对于是一个 table，把低维数据转换成高维数据，比如 hashgrid 就能用 embedding 实现。

4.7 自动微分 Forward 与 Reverse 模式

如图 2 所示。

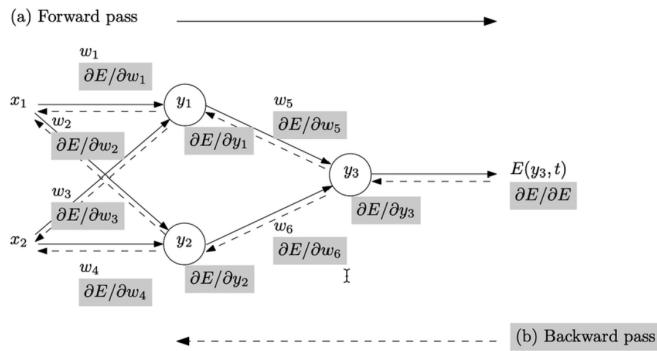


图 2: 前向后向计算示意图

- 符号微分：完整的把公式写出来，通常会非常复杂
- 数值微分：利用求导的定义 $\frac{f(x+h)-f(x)}{h}$ ，但是计算起来不是非常准确，会出现数值的不稳定等问题
- 自动微分：相当于是把符号微分模块化，但是和后向传播有区别，这里是向前算。可以看成是符号微分的分块。

4.8 模型保存

state_dict 是用于保存模型参数的方法，Module 类中实现。两种保存网络方法

- state_dict 方法保存，用 register_buffer 和 parameter 定义的变量都会被保存。

```
torch.save(net.state_dict(), PATH)
```

– 加载的时候先定义一个类，然后 load_state_dict 加载

- 保存整个模型

```
torch.save(net, PATH)
```

– 和上面的区别在于在加载的时候直接加载模型就可以了，不需要先定义类再加载

最常用的还是第一种，因为现在的模型一层嵌套一层，直接存储会消耗很大内存。

但是通常在加载模型的时候我们是想继续训练的，所以更一般的存储方式

```
torch.save({
    'epoch': epoch,
    'model_state_dict': model.state_dict(),
    'optimizer_state_dict': optimizer.state_dict(),
    'loss': loss
}, PATH)
```

然后再分别加载，现在我看到的大部分代码都是这样。

4.9 Dropout

Dropout 有两个，一个是类一个是函数

- nn.Dropout
- torch.nn.functional.dropout

dropout 在 training 的时候会打开，但是在 testing 的时候是不需要的。dropout 类似一个集成学习的范式，一次训练多个不同的模型，但是测试的时候不可能全部 voting，所以 test 的时候直接关闭即可。

对于 MLP 的每一个神经元都会有 p 的概率会失活，所以每一个神经元强度基本要乘上 p ，所以 test 的时候需要最神经元的强度乘上 p ，如图 3 所示。

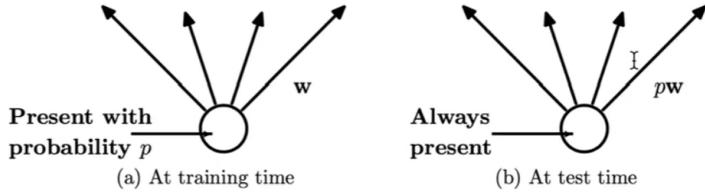


图 3: Dropout

4.9.1 Dropout 源码

Dropout 这个类继承于 `_DropoutNd`，里面只有一个 forward 函数

```
def forward(self, input: Tensor) -> Tensor:
    return F.dropout(input, self.p, self.training, self.inplace)
```

这里面的 `training` 变量在父类和子类都没有实现，他是在 `nn.Module` 这个函数里面使用的。在 `Module` 类的 `init` 函数第一行就写明了 `self.training = True`，默认情况下就是 `true`。当调用 `module` 的 `eval` 函数的时候就会变成 `False`。

老师是看的 C++ 的源码，我看的是 cuda 的。如果是测试阶段，那不需要走 dropout

```
if (is_test_) {
    if (Y != &X) {
        context_.CopySameDevice<float>(
            X.numel(), X.data<float>(), Y->template mutable_data<float>());
    }
    return true;
}
```

然后 `in place` 操作不允许，接着调用 kernel

```
DropoutKernel<<<
    CAFFE_GET_BLOCKS(X.numel()),
    CAFFE_CUDA_NUM_THREADS,
    0,
    context_.cuda_stream()>>>(
```

`X.numel(),`

```
ratio_,  
X.data<float>(),  
Ydata,  
mask->template mutable_data<bool>());
```

x 有多少个元素就调用多少个线程, numel 就是 x 数据的数量。
CAFFE_GET_BLOCKS(X.numel()) 就是

$$\frac{N + threads - 1}{threads} + 1$$

他这里的数据全部都是当成一维处理, 也就是直接 x.data 传进来。

kernel 代码有一个宏定义

```
CUDA_1D_KERNEL_LOOP
```

i 表示当前线程 id, N 表示数据总数。这个是用于处理 grid 数量不够分配线程的情况, 让线程可以循环处理。比如定义了 5 个 block, 每一个 Block 有 2 个线程, 那么第 0 号 block 可以处理第 0 个数据, 正常来说一个线程就干 1 个数据就行, 如果线程不够用就要重复了, 这个时候 0 号线程干 $0+5*2$ 个数据。这种方式可以解决线程数量不够的问题; 注意这种方式并不是循环展开。

```
#define CUDA_1D_KERNEL_LOOP(i, n) \
    for (int i = blockIdx.x * blockDim.x + threadIdx.x; i < n; \
        i += blockDim.x * gridDim.x)
```

所以结合起来就是

```
--global__ void DropoutKernel(  
    const int N,  
    const float ratio,  
    const float* Xdata,  
    float* Ydata,  
    bool* maskdata) {  
    const float scale = 1. / (1. - ratio);  
    CUDA_1D_KERNEL_LOOP(i, N) {  
        maskdata[i] = (Ydata[i] > ratio);  
        Ydata[i] = Xdata[i] * scale * maskdata[i];  
    }  
}
```

4.9.2 dropout 复现

复现比较容易:

```
def train_v1(rate, x, w1, b1, w2, b2):
    layer1 = np.maximum(0, np.dot(w1, x) + b1)
    mask1 = np.random.binomial(1, 1-rate, layer1.shape)
    layer1 = layer1 * mask1
    layer2 = np.maximum(0, np.dot(w2, layer1) + b2)
    mask2 = np.random.binomial(1, 1-rate, layer2.shape)
    layer2 = layer2 * mask2
    return layer2

def train_v2(rate, x, w1, b1, w2, b2):
    layer1 = np.maximum(0, np.dot(w1, x) + b1)
    mask1 = np.random.binomial(1, 1-rate, layer1.shape)
    layer1 = layer1 * mask1 / (1 - rate)
    layer2 = np.maximum(0, np.dot(w2, layer1) + b2)
    mask2 = np.random.binomial(1, 1-rate, layer2.shape)
    layer2 = layer2 * mask2 / (1 - rate)
    return layer2

def test_v1(rate, x, w1, b1, w2, b2):
    layer1 = np.maximum(0, np.dot(w1, x) + b1)
    layer1 = layer1 * (1 - rate)
    layer2 = np.maximum(0, np.dot(w2, layer1) + b2)
    layer2 = layer2 * (1 - rate)

    return layer2

def test_v2(x, w1, b1, w2, b2):
    layer1 = np.maximum(0, np.dot(w1, x) + b1)
    layer2 = np.maximum(0, np.dot(w2, layer1) + b2)

    return layer2
```

把神经元输出的部分值 mask 掉就行。

4.9.3 R-Dropout

这篇 paper 发表在 NIPS 2021 上，虽然说在各种任务上有提升，但是实际上使用不多，因为大部分网络的深度都会很大，两次 forward 无论是存储的 gradient 还是时间都是无法接受的。

- 问题：解决网络在训练和测试过程中存在的 gap 的问题。在训练过程中实际上是训练了多个模型，但是在测试的过程是，是没办法对所有的模型进行集成训练。所以测试的时候通常是把 dropout 关掉，但是这种方式并不能解决 training 和 test 之间的 gap。
- 方法：
 - 两次 forward，对两次 forward 结果做一个双向 KL 散度，使得分布保持一致。
 - 并且还进行完整的理论分析。

具体实现在代码中并不是 forward 两次，而是直接把数据复制一份，然后 forward 过去，两个数据再做双向的 KL 散度。

4.10 残差模块

4.10.1 CNN

主要参数：

- in_channels: 输入特征
- out_channels: 输出特征
- kernel_size: 卷积核
- stride: 步长
- padding: 补长
- group: 对 channels 进行分组，通常来说是全部的 channels 进行，如果设置 group，那么会把 channels 分成两组进行。

对于一个 conv2 的卷积操作：

```
Conv2d(2, 4, 3, 3)
```

意思就是输入 2 个 channels 输出 4 个 channels, kernel 为 3×3 。首先输出 4 个 channels, 那么就需要 4 个 feature map, 一个 feature map 只能生成一个 channels。每一个 feature map 需要 cover 所有的 channels, 所有 channels 就是 2. 所以这个 mox 的 filter 的 weights 是 $[4, 2, 3, 3]$ 。

3×3 的 kernel 比较常用, 许多模型对于 3×3 的 kernel 优化比较好。另外 1×1 的卷积相当于一个 MLP。

4.10.2 残差模块复现

这部分主要是复现一下 google net 的 block, 如图 4 所示。

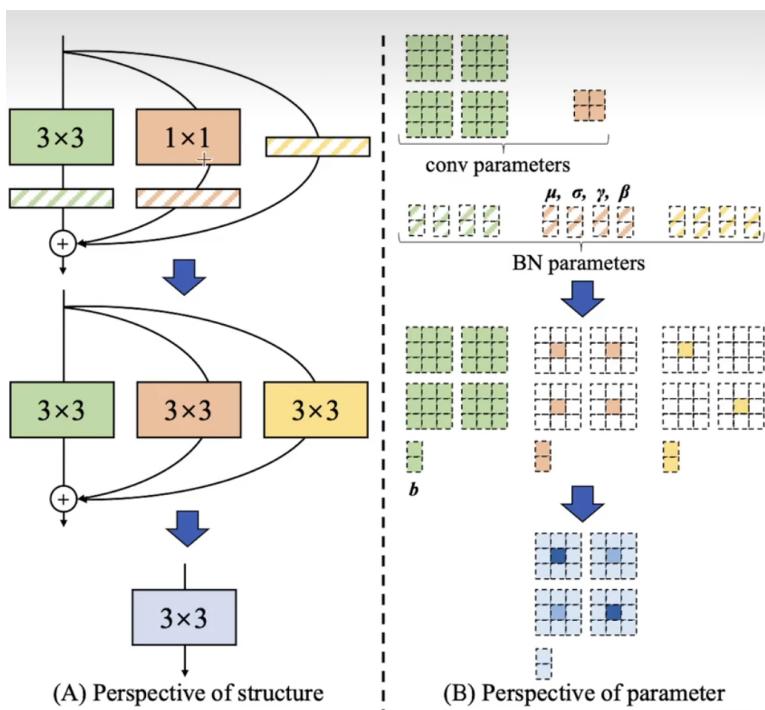


图 4: residual block

三个模块对 x 做卷积之后加起来。正常的实现就是分别用三个卷积处理即可:

```
class ResidualBlockv1(nn.Module):
    def __init__(self, *args, **kwargs) -> None:
        super().__init__(*args, **kwargs)
```

```

    self.conv_2d = nn.Conv2d(
        in_channels=in_channels, out_channels=out_channels,
        kernel_size=kernel_size, padding='same')
    self.conv_2d_pointwise = nn.Conv2d(
        in_channels=in_channels, out_channels=out_channels, kernel_size=1)

def forward(self, x):
    return self.conv_2d(x) + self.conv_2d_pointwise(x) + x

```

老师这里提供了一种计算效率更高的方式，因为卷积的特征，可以先把他们的 filter 相加，再一起做卷积，

- 正常的顺序 filter → addition，先分别对 x 做卷积再做加法
- 由于卷积的特性，我们可以先进行加法再进行卷积 addition → filter

先把这几个卷积的 filter 构建出来，再塞到 conv2d 里面，先构建 1x1 的卷积。

```

self.conv_2d_pointwise = nn.Conv2d(
    in_channels=in_channels, out_channels=out_channels, kernel_size=1)
pointwise_to_conv_weight = F.pad(
    self.conv_2d_pointwise.weight, [1, 1, 1, 1, 0, 0, 0, 0])

```

构建一个 1x1 的卷积，然后把他的权重进行缩放成 3x3 卷积的形状，这样才能加起来。然后把这个权重塞进去

```

self.conv_2d_pointwise = nn.Conv2d(
    in_channels=in_channels, out_channels=out_channels, kernel_size=kernel_size,
    padding='same')
self.conv_2d_pointwise.weight = nn.Parameter(pointwise_to_conv_weight)

```

这里需要重新构建一个 conv2d，因为前面构建的 shape 不对。

然后是构建 identity，这个稍微复杂一点。两个输出的 channel 就需要两个 filter，一个 filter 处理一个维度，另一个维度要是 0 才行。

```

self.zeros = torch.unsqueeze(torch.zeros(kernel_size, kernel_size), 0)
self.stars = torch.unsqueeze(F.pad(torch.ones(1, 1), [1, 1, 1, 1]), 0)

```

一个是第一层一个是第二层，当 zero 在前面，只计算第二层，当 zero 在后面就只计算第一层。

```
self.starts_zeros = torch.unsqueeze(
    torch.cat([self.stars, self.zeros], 0), 0)
# [2, 3, 3]
self.zeros_stars = torch.unsqueeze(
    torch.cat([self.zeros, self.stars], 0), 0)
self.identity_conv2d_weights = torch.cat(
    [self.starts_zeros, self.zeros_stars], 0)
```

这样就构建好了特征。然后塞进 conv2d 里面就行。最后把权重和 bias 全部加起来就行

```
self.conv2d_fusion = nn.Conv2d(
    in_channels=in_channels, out_channels=out_channels, kernel_size=kernel_size,
    padding='same')
self.conv2d_fusion.weight = nn.Parameter(
    self.conv_2d.weight.data + self.conv_2d_pointwize.weight.data +
    self.identity_conv2d.weight.data)
self.conv2d_fusion.bias = nn.Parameter(
    self.conv_2d.bias.data + self.conv_2d_pointwize.bias.data +
    self.identity_conv2d.bias.data)
```

4.11 Transformer

sequence2sequence 模型的 backbone 主要有三类

- CNN: 权重贡献，滑动窗口，对相对位置敏感，对绝对位置不敏感。也就是平移不变性和旋转不变性，这符合图片的特征。
 - Insights: 局部关联性，当前像素和他周围像素关联比较大
- RNN: 对顺序，位置很敏感，运算耗时，串行计算。相对位置敏感，绝对位置也敏感，会有遗忘的问题
 - Insights: 当前时刻需要依赖下一时刻，有序递归建模
- transformer: 对相对位置不敏感，对绝对位置也不敏感（所以需要一个位置编码来指定位置）。计算复杂度相对高一些，是 rnn 的平方级别。
 - Insights: 无局部假设，无顺序假设

如图 5 所示。encoder 由 n 个 block 构成，每一个 block 有两个模块，attention + MLP。decoder 也是 n 个 block 构成，每一个 block 有三个模块，mask attention + attention + MLP 构成。无论是输入还是输出都需要增加一个 PE。

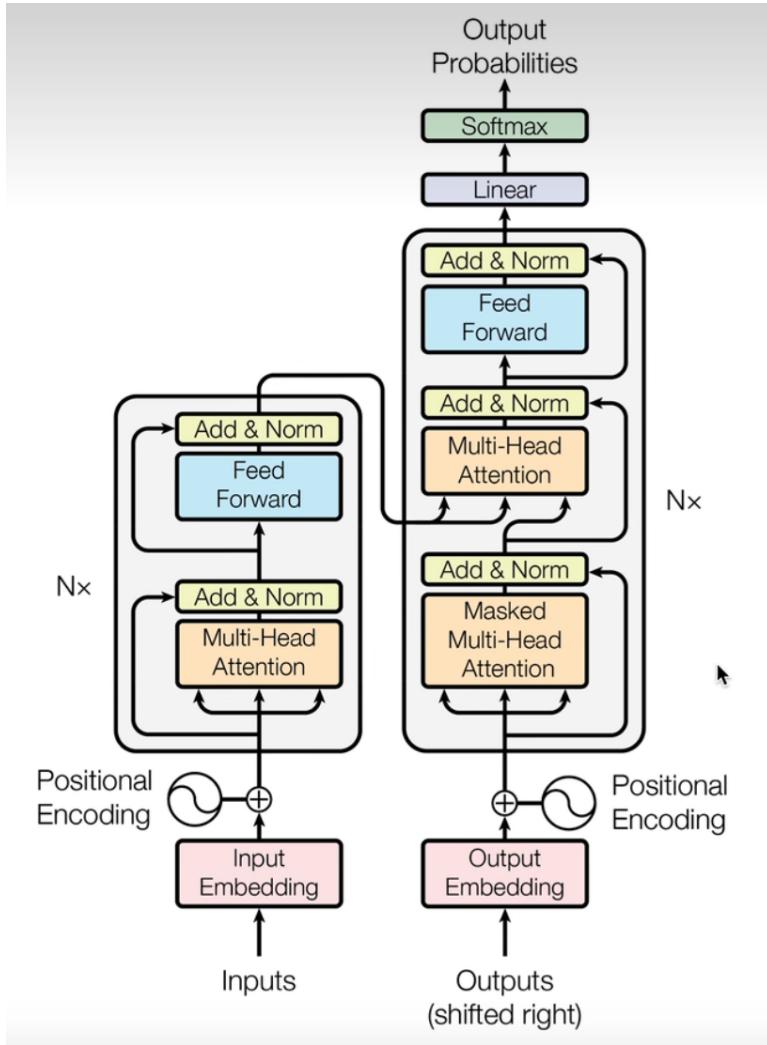


图 5: transformer

4.11.1 Encoder

encoder 的三个部分：

- **input word embedding:** 原来所有的字符都是 1,2,3,4..., 这篇通过一个网络把这些数

字都映射成一个向量。

- position embedding: 这个是因为 transformer 是没有位置假设的，所以需要一个位置编码信息。
 - 每一个位置上编码都需要确定的
 - 句子中的距离要一样
 - 可以推广
 - 作者是用傅里叶编码，我觉得应该把这些位置映射到不同空间的基，然后能够不重合
- multi-head attention: 主要是建模能力更强。注意力机制大部分模型的解释都是用来提取不同位置输入之间的关系。
- feed-forward network: 对每一个位置进行单独建模，前面 attention 是建立各个位置之间的联系，这里就是进行单独建模。

decoder 的五个部分：

- output word embedding
- masked multi-head attention: 加上 Mask 主要是为了保证输出只能看到前面已经输出的
- multi-head attention
- softmax

4.11.2 Transformer 复现

Word embedding 首先是构建数据，这里是构建了两个长度不一样的句子，长度不一样无法组成 tensor 输入模型中，所以需要先 Padding。

```
src_len = torch.randint(2, 5, (batch_size, ))
tgt_len = torch.randint(2, 5, (batch_size, ))

src_seq = [F.pad(torch.randint(1, max_num_words, (L, )),
                 (0, max_src_seq_length - L)) for L in src_len]
tgt_seq = [F.pad(torch.randint(1, max_num_words, (L, )),
                 (0, max_tgt_seq_length - L)) for L in tgt_len]
```

然后构建一个字典，这个字典能够将句子转换成 embedding。

```
src_Embbedding_table = nn.Embedding(max_num_words + 1, model_dim)
tgt_Embbedding_table = nn.Embedding(max_num_words + 1, model_dim)

src_embedding = src_Embbedding_table(src_seq)
tgt_embedding = tgt_Embbedding_table(tgt_seq)
```

输出的 embedding 维度是 [num_src, src_length, embedding_dim]。正常来说应该是还是要加一个结束符的，但是作者这里似乎没有考虑到。

Position embedding 句子的每一个位置都要赋予一个位置编码。位置编码的公式：

$$pe(pos, i) = \sin\left(\frac{pos}{10000^{\frac{2i}{d_{model}}}}\right)$$
$$pe(pos, i) = \cos\left(\frac{pos}{10000^{\frac{2i}{d_{model}}}}\right)$$

主要三个步骤

- 首先构造公式的矩阵
- 用 nn.Embedding 作为载体
- 输入矩阵位置向量或者 pe

构造公式矩阵：

```
pos_matrix = torch.arange(max_position_length).reshape(-1, 1)
# [1, model_dim / 2]
i_matrix = torch.arange(0, model_dim, 2).reshape(1, -1) / model_dim
# [1, model_dim / 2]
i_matrix = torch.pow(1000, i_matrix)
# [max_position_length, model_dim]
pe_embedding_table = torch.zeros(max_position_length, model_dim)
# [max_position_length, model_dim]
pe_embedding_table[:, 0::2] = torch.sin(pos_matrix / i_matrix)
# [max_position_length, model_dim]
pe_embedding_table[:, 1::2] = torch.cos(pos_matrix / i_matrix)
```

塞进 Embedding 里面：

```
pe_embedding = nn.Embedding(max_position_length, model_dim)
pe_embedding.weight = nn.Parameter(pe_embedding_table, requires_grad=False)
```

获取 pe embedding

```
src_positions = torch.stack(
    [torch.arange(max_src_seq_length) for _ in range(batch_size)], 0)
tgt_positons = torch.stack(
    [torch.arange(max_tgt_seq_length) for _ in range(batch_size)], 0)
src_pe_embedding = pe_embedding(src_positions)
tgt_pe_embedding = pe_embedding(tgt_positons)
```

注意这里输入的并不是句子，而是句子每一个单词的位置，我们需要对位置进行编码，而不是句子本身。

4.11.3 Attention

attention 注意力：

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK}{\sqrt{d_k}}\right)V$$

如果是单独的 encoder 和 decoder，这个 Q, K, V 是三个一样的输入进入不同的 MLP，如果是 cross 不同的模块，那么 QK 是不一样的，他的直观解释其实是希望 encoder 去查询 decoder 中的值，当然这只是人赋予他的意义，真正是不是这样不得而知。

除 $\sqrt{d_k}$ 是因为希望概率方差不要变化这么大，并且希望导数不要变成零。其实就保持模型数值的稳定性。如果 softmax 里面的值除去一个数值，可以使得 softmax 的曲线不会这么陡峭。梯度不会变化这么大。通常在计算 softmax 的时候还会减去一个较大的值，结果不会改变但是能防止数值过大。

神经网络只能接收相同规格的输入，所以在输入前通常是会增加 pad，所以在计算的时候需要增加 mask，把那些 pad 的位置 mask 掉，要不然那些 pad 无意义的位置也会加入计算。这里主要有三个 mask 需要进行求解，如图 6 所示，第一个 mask 是 encoder 内部的 mask，第二个是 decoder 自己的 mask，然后就是两个混一起的 mask。

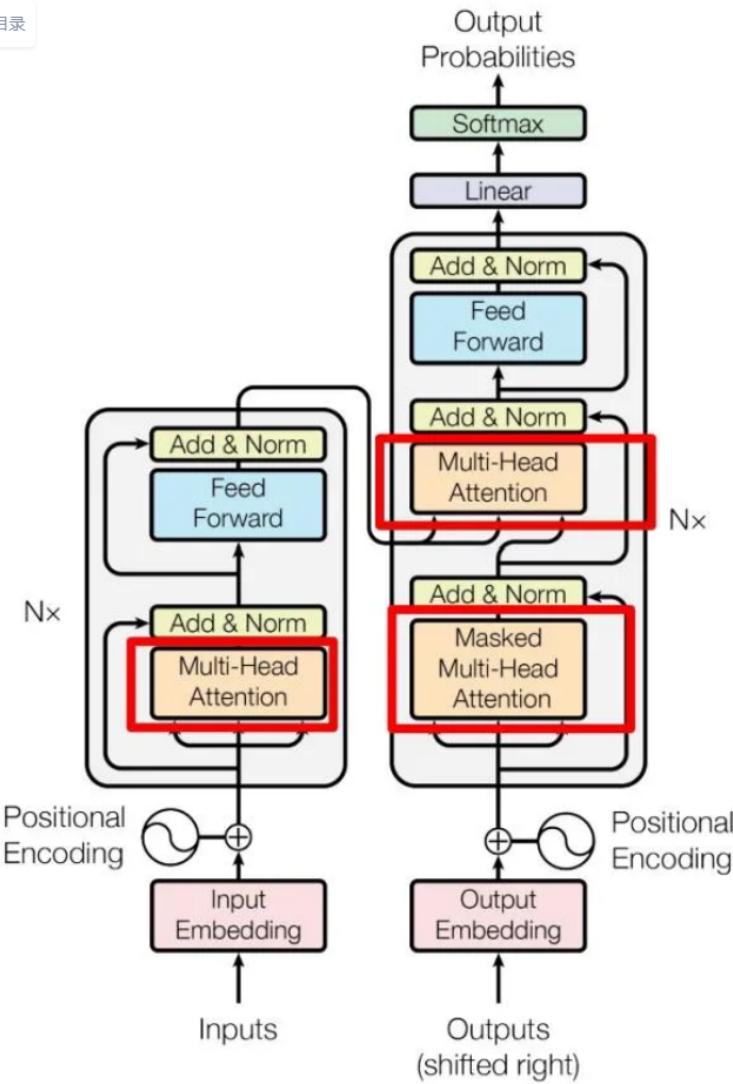


图 6: self attention

encoder 里面的 QKV 是把 src sentence 输入到 MLP 分别生成，所以维度是 $[N, \text{length}, \text{dim}]$, N 是句子的数量, length 是每一个句子的长度, 这个长度是经过 padding 的, 这也是为什么需要 mask, dim 是经过 MLP 后的维度。所以 QK 之后是 $[N, \text{length}, \text{length}]$, mask 也是要相同维度。mask 表示的应该是每一个单词和其他元素之间的关系, 如果是 mask 就是 0, 所以这个矩阵是一个对称矩阵。

```
valid_encoder_pos = torch.stack([
    F.pad(torch.ones(L), (0, max_src_seq_length - L)) for L in src_len], 0)
valid_encoder_pos = torch.unsqueeze(valid_encoder_pos, 2)
valid_encoder_pos_matrix = torch.bmm(
    valid_encoder_pos, valid_encoder_pos.transpose(1, 2))
invalid_encoder_matrix = 1 - valid_encoder_pos_matrix
invalid_encoder_matrix = invalid_encoder_matrix.to(torch.bool)
```

i 行表示 i 个单词对其他单词的关系，有关系就是 1 没关系就是 0。

第二个是 decoder 的 attention, kv 输入 encoder, q 输入 decoder, 所以维度是 [N, target length, src length]。表示当前 i 个输出和其他输入的关系。和之前其实是一样的

```
valid_encoder_pos = torch.stack([
    F.pad(torch.ones(L), (0, max_src_seq_length - L)) for L in src_len], 0)
valid_encoder_pos = torch.unsqueeze(valid_encoder_pos, 2)

valid_decoder_pos = torch.stack([
    F.pad(torch.ones(L), (0, max_tgt_seq_length - L)) for L in tgt_len], 0)
valid_decoder_pos = torch.unsqueeze(valid_decoder_pos, 2)
valid_cross_pos_matrix = torch.bmm(
    valid_decoder_pos, valid_encoder_pos.transpose(1, 2))
invalid_cross_pos_matrix = 1 - valid_cross_pos_matrix
mask_cross_attention = invalid_cross_pos_matrix.to(torch.bool)
```

然后就是 decoder 本身的 mask, 因为作者希望输出的第 i 个单词只和 i-1 个单词相关，所以这个 mask 是一个下三角的。

```
valid_decoder_tril_matrix = torch.stack([F.pad(torch.tril(torch.ones((L, L))), 
                                                (0, max_tgt_seq_length - L, 0,
                                                max_tgt_seq_length - L)) for L in
                                                tgt_len], 0)
invalid_decoder_tril_matrix = 1 - valid_decoder_tril_matrix
invalid_decoder_tril_matrix = invalid_decoder_tril_matrix.to(torch.bool)
```

都是常规操作。在计算 attention 的时候传入不同的 mask 即可

```
def scaled_dot_product(Q, K, V, atten_mask):
    score = torch.bmm(Q, K.transpose(-2, -1)) / torch.sqrt(model_dim)
    masked_score = score.masked_fill(atten_mask, -1e9)
    prob = F.softmax(masked_score, -1)
```

```
context = torch.bmm(prob, V)
return context
```

不过看很多代码写 mask 都是先对其中某一个输入 mask 完再输进去，尽量把 mask 这个操作从算法中解耦，因为 mask 很可能是和阶段，和数据集有关系。

4.11.4 总结

Tranformer 的特点：

- 无先验假设，和 CNN，LSTM 不太一样。相应带来的优势就是具备更强的适应能力。
- 同时由于无先验假设，所以 Transformer 是需要更多的数据训练，而 CNN 本身就符合一定的 bias，比如局部关联性等。而先验假设越多，数据量要求越低，比如一些 few-shot model 如果能注入比较多的先验假设，那效果更好。
- Transformer 的核心在于 attention 机制，他的计算复杂度是和数据规模成正比。

4.12 CNN

神经网络最主要的操作是仿射变换，比如一些向量数据。但是对于图像和语音就不能单纯的就对图像进行拉伸然后当成向量处理，因为这些数据本身是带有一定空间和顺序信息的。pytorch 里面有两个实现，一个类一个方法，但都是一样的实现。核心参数包含

- in channels: 输入通道，

5 CS106B C++ 抽象编程

5.1 Welcome

第一节课主要解决的问题是

- Why take CS106B
- What is an abstraction
- What is CS106B
- Why C++
- What's next

什么是抽象？抽象的定义

Design that hides the details of how something works while still allowing the user to access complex functionality.

手机，汽车就是一种抽象，当我们使用汽车手机的时候，也就是 access，并不需要了解他们的内部构造，也就是 hide the detail 了。另一个例子就是程序语言了，程序语言的抽象核心在于

Through a simpler interface, user are able to take full advantage of a complex system without needing to know how it works or how it was made.

日常业务中常见的前后端分离，解耦编程等等其实都是抽线编程，只需要了解提供的接口参数，就可以方便的使用这些接口定义的方法。抽象的关键在于：如何定义使用和实现，也就是具体实现和接口之间的定义。abstraction boundary 就是定义 usage 和 implementation 的分界线。

课程学习路线如图所示 7。

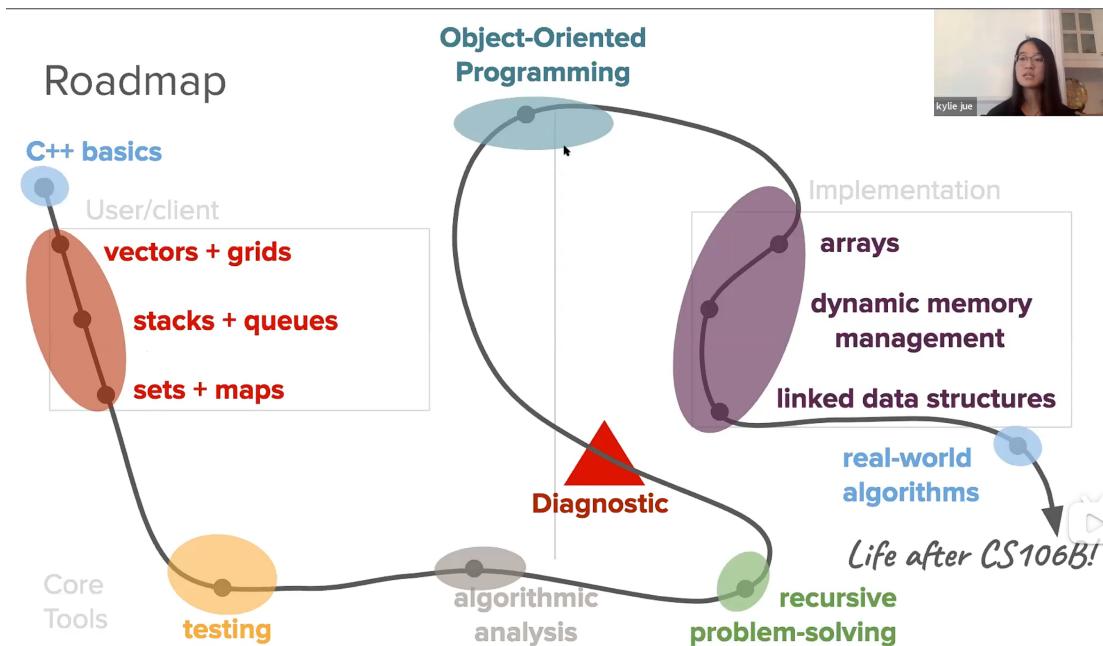


图 7: 课程学习路线

C++ 和 python 的简要对比: C++

```
#include "console.h"
using namespace std;
int main(){
    cout << "Hello world!" << endl;
    return 0;
}
```

Python

```
import sys

def main():

    int __name__ == '__main__':
        main()
```

区别还是很大的，首先是 python 不需要分号，而且 python 是不需要主函数的。但其实代码逻辑和一些基础语法是类似的，只能说一些语法糖有区别而已。

C++ 的特点：

- C++ 是一门编译语言，需要编译成机器语言才能运行
- C++ 可以控制一些低级的计算资源，比如内存，寄存器等
- C++ 和 python 的语言很相似

C++ 的优势和劣势

- 优点
 - C++ 很快
 - C++ 在很多领域都应用广泛，特别是在图形学，计算机视觉领域
 - C++ 强大
- 缺点
 - C++ 复杂
 - C++ 危险，主要是出现在内存上，比如内存泄漏等等。

5.2 C++ 基础

注释，也即是 comments，有两种，单行注释和多行注释。写注释是一种良好的代码习惯。

单行注释

```
// 单行注释
```

多行注释

```
/*
 * 多行注释
 */
```

5.2.1 Include

引用一些在其他程序写好的代码。

In order to make the compiler aware of other code libraries or other code files that you want to use, you must include a header file

有两种方式

```
#include <iostream>
#include "myself.h"
```

5.2.2 console

控制台是我们主要和程序交流的中介，程序的输出结果大部分都会现实在控制台。cout 流把输出推送到控制台的缓冲区。和 python 不一样的是，c++ 的每一行必须要要有分号结束。

5.2.3 变量和类型

变量是程序存储信息的一种方式，变量名称 + 变量数值。这里提一下命名方式，通常在 python 中，命名通常是 A_B，但是 C++ 通常是驼峰命名法。变量是程序中最基本的一种表达形式。

类型是变量的一种表达，在 C++ 是需要事先知道的，常用的几种类型

- int
- double
- string
- char

在 C++ 中，所有的变量，必须显示给出他的类型。python 是不需要的，并且也不能重新定义变量的类型。比如

```
int a = 5;
double a = 5;
```

这样是不行的，意味着我们不能重新在 C++ 中重新定义变量。但你可与 cast 成另一个类型使用。

```
int a = 5;
int a = 5;
```

这样也是不行的，任何变量只能赋值一次。

5.2.4 function and parameter

函数如图 8 所示，主要部分：输入，参数，输出，也就是返回值。

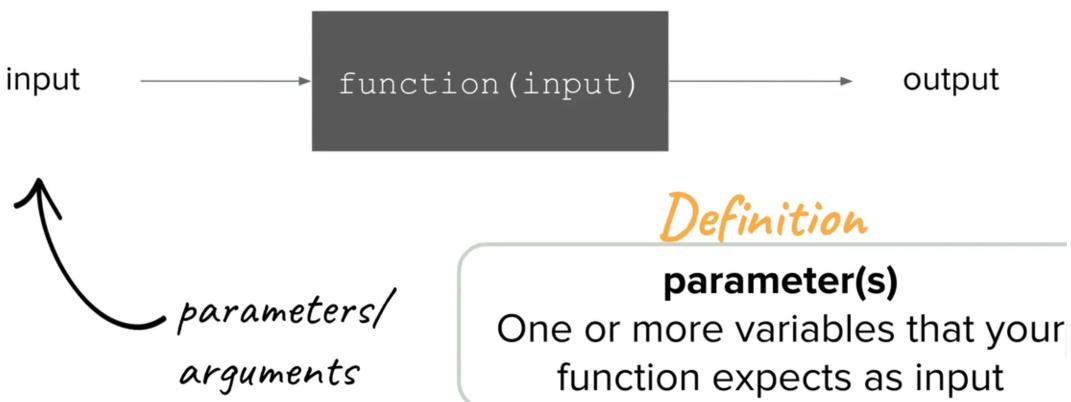


图 8: 函数

函数的格式：returnType functionName(varType parameter1, varType parameter2, ...).

- 函数的参数可以看成是属于这个函数的局部变量，只属于这个函数，离开了这个函数就会失效。相对于把参数复制一份到函数里面。如果是返回值 void 的函数，可以不写返回值或者 return;
- 一个函数必须要定义之后才能调用。
- 这里有一个名称上的差别，定义函数的时候的参数，我们称为 parameters，在调用函数时候的参数我们称为 argument

老师后面给了一个传递参数的错误例子，如图 9所示，输出还是 5，因为输入的参数是局部变量，相对于把这个 myValue 复制进函数，复制进去的函数相对于是一个局部变量。修改一个局部变量，自然是不影响的。

```
// C++:  
#include<iostream>  
using namespace std;  
  
int doubleValue(int x) {  
    x *= 2;  
    return x;  
}  
  
int main() {  
    int myValue = 5;  
    int result = doubleValue(myValue);  
  
    cout << "myValue: " << myValue << " ";  
    cout << "result: " << result << endl;  
}
```

图 9: 传递参数

5.2.5 控制流

这部分就是一些 for, if 等没啥好说的。然后是作业 0.

5.2.6 String

SString 是一种表示字符序列的一种数据结构。string 是一种比 type 更高级的数据结构，不需要像 char 一样在后面增加 0，因为他已经给你弄好了。在 C++ 中双引号表示字符串，

单引号表示字符，和 python 不一样。

因为他是数据结构，所以本身就提供了很多操作，比如加法，大于小于，是比较字符串的首字符码，如果首字符相同就以此类推。

对于 string 的循环有两种，第一种是直接循环索引

```
for(int i = 0; i < str.length(); ++i)
{
    cout << str[i] << endl;
}
```

第二种是直接把值取出来

```
for(char c : str)
{
    cout << c << endl;
}
```

5.2.7 string utility functions

- Built-in C++ char functions
- Built-in C++ string methods
- Stanford string library functions

第一个是 C++ 中 char 的函数，需要引入 `<cctype>` 头文件，函数的种类很多，如图 10 所示。

isalnum: checks if a character is alphanumeric
isalpha: checks if a character is alphabetic
islower: checks if a character is lowercase
isupper: checks if a character is an uppercase character
isdigit: checks if a character is a digit
isxdigit: checks if a character is a hexadecimal character
iscntrl: checks if a character is a control character
isgraph: checks if a character is a graphical character
isspace: checks if a character is a space character
isblank: checks if a character is a blank character
isprint: checks if a character is a printing character
ispunct: checks if a character is a punctuation character
tolower: converts a character to lowercase
toupper: converts a character to uppercase

图 10: cctype function

第二种是 string 方法，这些方法基本上就是要用什么就查什么，如图 11所示

```
s.append(str): add text str to the end of a string  
s.compare(str): return -1, 0, or 1 depending on relative ordering  
s.erase(index, length): delete text from a string starting at given index  
s.find(str): return first index where the start of str appears in this string (returns  
string::npos if not found)  
s.rfind(str): return last index where the start of str appears in this string (returns  
string::npos if not found)  
s.insert(index, str): add text str into a string at a given index  
s.length() or s.size(): number of characters in this string  
s.replace(index, len, str): replaces len chars at index with text str  
s.substr(start, length) or s.substr(start): the next length characters  
beginning at start (inclusive); if length omitted, grabs till end of string
```

图 11: string function

斯坦福自己的库，需要自己下载编译。

5.2.8 Two types of string

主要是指两种，C++ 的 string 和 C 的 string。两个例子

```
string a = "abc" + "abc";
```

这个会直接报错，因为 C 的 string 是没有加法的，如果是两个 string a = "abc"; string b = "abc"; 就可以相加，因为 C++ 的 string 的数据结构是自带了加法的。

```
string a = "abc" + '?';
```

这个不会报错，但是会指向一个垃圾内存，我也不知道为什么。所以 C 和 C++ 之间是需要一些转换的：

- C string 是没有方法的，不能像在 C++ 一样适用 length 这些方法
- C string 个 C++ string 之间可以进行转换

string("text") 把 C string 转换成 C++ string

string.c_str() 把 C++ string 转成 C string

5.3 Assignment 0. Welcome to CS106B!

如果直接打开 CS106B 的课程会发现他们是 23 的课程还没上完，并且作业也是打不开的，需要注册进去。注意到他这个网址，我猜他这个 1 可能是年级的意思，23 是 23 年，8 应该是系号，改成 1228 果然就能进去了 22 年的了，并且 22 年的作业是不需要登录就可以下载的：

- <https://web.stanford.edu/class/archive/cs/cs106b/cs106b.1228/assignments/>

这里就能看到所有作业了，和 23 年是一样的，github 上看到有 16 和 13 年的，但是作业都大变样了。

第一个任务主要就是安装 Qt 为主。安装的时候发现很慢，直接给换了个清华源快到飞起

```
. \qt-unified-windows-x64-4.6.0-online --mirror  
https://mirrors.tuna.tsinghua.edu.cn/qt/
```

在我本地安装这个 Qt 的时候，遇到一个非常奇怪的问题，明明我已经装好了这个 Qt，完了还是遇到找不到 moc 和 rcc 的问题，我自己在 cmd 输入 moc 和 rcc 也有反应，说明环境变量是没问题了。

```
/usr/bin/sh: D:\Qt\6.5.1\mingw_64\bin\rcc.exe: command not found
```

看了一下是发现他这个命令前面多了一个

```
/usr/bin/sh
```

应该是之前全局安装了 cmake 把环境变量给配了的缘故。然后我就选了这个清除全局变量。清除环境变量之后又有别的问题，一直右下角 reading project，读几个小时读不完，卡住了。然后我把这个 reading project 给取消，构建 + 编译就能跑起来了。总的来说这个问题是出现在之前安装了 cmake 或者 visual studio 导致的环境变量混乱的问题。

实验室这台笔记本又没问题了，应该就是这个 cmake 和 visual 的问题。

作业 0 就是安装 Qt 跑一跑。

5.4 Assignment 1. Getting Your C++ Legs

本来作业应该是在 https://web.stanford.edu/class/archive/cs/cs106b/cs106b.1244/about_resources 这的，但发现 24 年这个课还没开始，找不到作业。看了下他这

个 1244，第一个数字应该是学生年级，24 年，最后这个 4 应该是开课月份（概率论是这样命名的），式了下 1224 有课程但找不到作业，1228 找到有作业。作业网址：

- <https://web.stanford.edu/class/archive/cs/cs106b/cs106b.1228/assignments/1-cpp/>

两个任务

- Perfect Numbers
- Soundex Search

首先是他这里面有一个 Testing and the SimpleTest framework

- EXPECT_EQUAL: 格式 EXPECT_EQUAL(reversed("but"), "tub"); 第二个参数是输出的参数。
- EXPECT: 判断是不是真的 EXPECT(isPalindrome("racecar"));
- STUDENT_TEST: 只是表面这是一个学生测试
- TIME_OPERATION: 测试函数时间

问题

- Q1: 自己看表计算要花多少时间计算
 - About 1 second, and four number.
- Q2: 作业中已经存在计算时间的代码了，现在让你在 perfect.cpp 后面增加自己的测试，把数据四次每一次成倍数增加，看看一分钟最多能做完多少？
 - 增加 STUDENT_TEST 就行，就用前面的 test 的代码写。
 - 80000 need 4.12s, 16000s need 17.2s, 280000 need 53.4s。仔细看的话会发现，没增加两倍，时间是增加 4 倍，时间和 scale 并不是一个线性关系，而是一个二次的关系。
- Q3: 在数字 10 上和数字 1000 上计算花费时间是一样的吗？在 1-1000 和 1000-2000 花费的时间是一样的吗？
 - 在 10 和 1000 上肯定时间不是一样的，1000 的时间会更长，1000-2000 的时间更长，因为他们的计算复杂度是 n 平方

- Q4: 问你输出的完美数字什么时候会到达 8128，直接把 8128 的时间算出来减去 496 的就行。
- Q5: 是否有检测失效数字的能力，是有的，因为他这个 for 循环有条件判断是否大于 1 了。
- Q6: 实现 smartSum，这个是比较容易实现的。对于一个数字 N，计算他的平方根，根据平方根的一边计算了一遍即可。注意条件，当 factor=1 的时候，factor 和另一边相等的时候需要额外判断。
- Q7: 就是吧 isPerfectSmarter 和 findPerfектSmarter 实现了，就是把刚刚实现的给加上就行。基本就是照抄，然后测试一下时间。
 - 10000 take 0.218s, 20000 take 0.231s, 40000 take 0.263s, 80000 take 0.323s
 - 对比之前的方法块了很多。尤其是 4w 开始基本上块了几倍。
- Q8: 根据复杂度就算他到底第五个完美数字之前，直接用程序算了一下，第二个到第五个 0.03s
- Q9: 按照他的要求把这个 findNthPerfectEuclid 实现了然后测试。寻找第 n 个 Mersenne prime number 就行。这里发现之前实现的一个问题，这个完数是要求除本身之外的因子之和，所以如果是 1 的时候是需要排除的。在 for 这里加上一个判断

```
long smarterSum(long n) {
    long total = 0;
    long sqrt_n = sqrt(n);
    for (long divisor = 1; divisor <= sqrt_n && divisor < n; divisor++){
        if (n % divisor == 0)
        {
            long divisor_other = n / divisor;
            if (divisor == 1)
            {
                total++;
            }
            else if (divisor_other == divisor){
                total += divisor;
            }
            else
            {
```

```

        total += divisor;
        total += divisor_other;
    }
}
}

return total;
}

```

第二个任务是 Soundex Search，主要是熟悉字符串的操作，实现两个函数

- 实现 take surnames as input and produce phonetic encodings as output.
- 实现 input a surname and then find all matches in a database of Stanford surnames that have the same encoding.

首先介绍 Soundex algorithm，把 surname 转换成 Soundex code。Soundex code 是由一个字母和三个数字组成，类似 Z123 这样。

- 丢弃所有非字母
- 用码表把字母编码
- 去掉相邻重复的编码
- 把第一个数字替换成原来的字母，并大写
- 去掉所有的 0
- 如果长度小于 4，填补 0，否则砍掉超出的
- Q10: Angelou 如何表示成 Soundex code。
 - 首先全部变成数字，0520400，，去掉相邻相同字符把第一个变成大写字母 A520400，A52040，去掉 0，A524，刚刚好。
- Q11: 问你怎么实现，如何分解任务？他上面不是都给步骤了吗，按照步骤分成四个任务呗？没明白这里要干什么。

然后是让你 debug 一下，我设置了两个案例，一个是空白”，另一个是全部都不是字母，都过不了。因为他这里是直接让 result 等于 s 的第一个字符，然后从 index = 1 开始判断。所以第一个基本的没有判断的。直接改成从第一个开始判断就能过了。

然后让你实现一下这个 soundex 方法，这个算法可以用来搜索和语音匹配，但缺点就是只能限制在英语这个语言上。按照之前的想法分成几个步骤实现就行

```
string soundex(string s) {
    /* TODO: Fill in this function. */
    string originString = s;
    string result = removeNonLetters(s);
    result = encodeString(result);
    result = CoalesceAdjacent(result);
    result = replaceFirstChar(result, originString);
    result = removeZero(result);
    result = paddingFour(result);
    return result;
}
```

分成 6 个方法实现就行。逻辑很容易，主要是麻烦。我用一个 map 表示 code 和符号之间的映射关系。

```
void initMaps(map<char, char> &mapString){

    mapString.clear();

    string zeros = "AEIOUHWY";
    string ones = "BFPV";
    string twos = "CGJKQSXZ";
    string three = "DT";
    string fours = "L";
    string fines = "MN";
    string sixs = "R";
    for (char c : zeros){
        mapString[std::toupper(c)] = '0';
        mapString[std::tolower(c)] = '0';
    }
    for (char c : ones){
        mapString[std::toupper(c)] = '1';
        mapString[std::tolower(c)] = '1';
    }
    for (char c : twos){
        mapString[std::toupper(c)] = '2';
        mapString[std::tolower(c)] = '2';
    }
    for (char c : three){
```

```

        mapString[std::toupper(c)] = '3';
        mapString[std::tolower(c)] = '3';
    }
    for (char c : fours){
        mapString[std::toupper(c)] = '4';
        mapString[std::tolower(c)] = '4';
    }
    for (char c : fines){
        mapString[std::toupper(c)] = '5';
        mapString[std::tolower(c)] = '5';
    }
    for (char c : sixs){
        mapString[std::toupper(c)] = '6';
        mapString[std::tolower(c)] = '6';
    }
}

}

```

大小写的映射都放进去，这样后面就不用考虑大小写了。别的可能就相邻方法删除有点麻烦

```

string CoalesceAdjacent(string s)
{
    int N = s.size();
    int cur = 0;
    while(cur < N)
    {
        char curChar = s[cur];
        int i = cur + 1;
        while(i < N)
        {
            if(curChar == s[i])
            {
                s.erase(cur, 1);
                N--;
            }
            else
                break;
        }
    }
}

```

```
    cur++;
}
return s;
}
```

这里没用 for 而是 while，因为这个过程 string 的 size 是不断变化的。对于当前的 cur，需要判断是否是连续相等，i 就表示下一个字符的下标，如果是相等，那就删除，N_，否则就退出。注意这里是不需要 i++ 的，因为你删除之后下一个自动就补上了了。

然后实现好之后需要通过所有测试即可。

最后需要实现一个 soundexSearch，用户输入一个 surname，计算这个 surname 和文件里面的所有 string 的 soundex code，相同的拿出来

```
void soundexSearch(string filepath) {
// The provided code opens the file with the given name
// and then reads the lines of that file into a vector.
ifstream in;
Vector<string> databaseNames;

if (openFile(in, filepath)) {
    readEntireFile(in, databaseNames);
}

cout << "Read file " << filepath << ", "
<< databaseNames.size() << " names found." << endl;

// The names in the database are now stored in the provided
// vector named databaseNames

/* TODO: Fill in the remainder of this function. */
string s;
cout << "Enter a surname (RETURN to quit): ";
cin >> s;
cout << "Soundex code is " << soundex(s) << endl;

Vector<string> matchString;
for (string dataName : databaseNames)
{
    if (soundex(dataName) == soundex(s))
    {
        matchString.add(dataName);
    }
}
```

```
        }
    }

matchString.sort();
cout << "Matches from database: {";
for (int i = 0; i < matchString.size(); ++i)
{
    if (i != matchString.size() - 1)
        cout << matchString[i] << ", ";
    else
        cout << matchString[i];
}
cout << "}";
```

通过作业里面的两个样例即可。

5.5 Console and Vector

主要是围绕两个问题展开

- How do we build programs that interact with users?
- How do we structure data using abstractions in code?

如何交互，如何把数据抽象化表示。

主要内容有几个

- testing
- console
- abstract data type
- pass by reference

测试这个作业一做了基本就全明白了，这一节应该在作业一之前看的。

5.5.1 String

字符串是信息具体表达的一种方式。在生活中有什么问题是比较有意思的能用上字符串解决的？

- Encryption and decryption, 编码和解码
- language translation, 语言翻译
- DNA 的分析, DNA 是由很多的脱氧核苷酸组成, 这些核苷酸是需要配对才能形成 DNA, 比如 A 和 T, C 和 G。也就是你给定了一条 DNA 的链, 他会存在一条互补的链。
 - string complement(string dnaStrand), 给定一条 DNA 的链, 返回这个 DNA 链的互补链。

然后后面让你现场写这个 complement 函数。讲师这里的用的 if 来写, 我还是倾向与用一个 map 的数据结构来完成。

5.5.2 Interact with Users

如何和用户交互, 这个应该是 cin 的用法。交互的平台就是控制台了, 当然更高级一点的有一些游戏引擎自带的界面, 或者是 Qt。

什么是控制台程序?

- A program that uses the interactive terminal as a communication boundary with the user.

如何获取用户的信息? getline 函数, 输入之后需要按下 enter 结束输入, getline 主要做两件事

- cout 输出你参数中的 string
- 接收你输入并返回

所有作业一最后一个任务用 getline 可以一次完成。总结了一下简单的 console program 的写法

- getLine 获取输入, 然后转换成所需的类型。这里接收的输入永远是 string 类型
- while(true) 可以直接循环接收
- console 主要运行在 main 程序

应该听完这节在做作业。

5.5.3 structure data

- 数据结构是一种抽象数据结构，我的理解是他的数据组织方式只是表现给用户看，实际的存储方式还是很简单的。比如二维数组就是一种抽象的数据结构，虽然看起来好像是二维的数据但实际上他在内存中的存储方式是一维的存储的方式。比如 python 的 tensor 也是一样。
- 这种数据结构有自己组织元素的一套方法，不需要用户涉及数据底层。

具体的数据结构

- vector: 在 high-level 的角度上看，vector 是一个顺序集合，可以删除增加元素。
 - 每一个元素都有自己的下标，可以通过索引找到数组元素
 - vector 中所有元素的类型必须一致
 - vector 的数量比较灵活，可以随意增减

类似于 python 的 List 和 java 的 ArrayList，不过 python 的 list 类型是可以不同的。

- create vector，直接就是

```
Vector<int> vec;
```

就行，这里就存储了一个空的数组，数组里面会维护 value 和 index。

- 如果一开始就知道了你的数据多少，那你可以直接 `Vector<int> vec = {1,2,3,4}` 初始化 vector。
- add 方法可以增加一个数据
 - remove 可以移除数据，param 是需要移除数据的下标
 - size 可以获得当前向量的大小

基本上都是一些操作上的问题。

pass by reference，这个应该是指引用了，通过修改 reference value 去修改 origin value。应该是引用传递的意思，意思就是直接传递引用就能直接修改传递的变量。

- 可以通过函数直接编辑数据，这样就不需要返回值了。
- 不需要重新复制数据返回了

- reference 实际上是可以做到多返回值的

Grids, 一种新的数据结构。这不就是数组吗？也可以直接用 vector 实现，vector 里面包一个 vector 就行了。这个数据结构应该是斯坦福这个库带的一个。都是用 vector 或者数组实现的 grid。声明 grid 的三种方式

- Grid<type> gridName;
- Grid<type> gridName(numRows, numCols);
- Grid<type> gridName = { {1, 2, 3, 4... }, ... };

基本方法

- numRows: 行的数量
- numCols: 列的数量
- grid[i][j]: 选择元素
- resize: 重新分配空间
- inBounds: 判断当前行列是否是在边界内

就是一个二维矩阵，使用 grid 的一些注意事项

- grid 在声明的时候需要指定其类型

```
Grid<int> board;
```

- 如果需要直接修改 grid 的值，需要传递引用
- grid 的下标是有顺序的，他是按照屏幕坐标系来定位
- grid 不能获取某一列

```
grid[0]
```

struct 结构体，另一种数据结构。

```
struct GridLocation{
    int row;
    int col;
}
```

- struct 的初始化，可以直接声明的时候初始化也可以在声明之后分别对各个成员初始化

```
GridLocation origin = {0, 0};  
origin.row = 0;  
origin.col = 0;
```

- 用点访问，如果是指针就用箭头。

Quene，队列，这个在 STL 里面也有。先进先出。常用的函数

- enqueue，在队列后面增加一个元素
- dequeue，移除队列第一个元素
- peek 或 front，查看队列的第一个元素，但是不移除
- isEmpty，判断队列是否是空的

stack 和 quene 的区别在于，stack 是先进后出。

- push，添加元素进 stack 里面
- pop，从 stack 后面弹出一个元素
- peek，查看 stack 最顶端的一个元素
- isEmpty，是否为空

基本操作都差不多。

两个使用 stack 或者 quene 的注意事项

- 如果需要在循环中编辑，不能用 size() 作为判断，因为编辑很可能改变队列大小，size 是会变化的。使用 while。
- stack 是不能一边迭代一边又能保证 stack 不被毁坏。因为你要遍历 stack 就会改变元素的存储顺序。quene 也是。

stack 和 quene 相对于其他 ADT 的优缺点

- 缺点：
 - 缺乏随机性，无法随机从 stack 和 quene 中获取数据。只能先进后出或先进先出

- 如果需要遍历 stack 和 queue，只能先移除前面一个数据，也就是说无法再不破坏 ADT 的条件下，遍历 ADT
- 遍历他们的复杂度永远都是 $O(1)$ ，没有别的快速方法
- 优点：
 - 在许多问题上应用广泛
 - 容易操作，容易创建

5.5.4 Unordered Structure

无序数据结构。主要介绍两种

- Sets
- Maps

如图 12 所示，grid 和 vector 分别都是可以索引的，但是 queue 和 stack 是无法索引的，不能在不破坏他们结构的情况下索引他们。

Ordered ADTs with accessible indices	Ordered ADTs where you can't access elements by index
Types:	Types:
<ul style="list-style-type: none"> • Vectors (1D) • Grids (2D) 	<ul style="list-style-type: none"> • Queues (FIFO) • Stacks (LIFO)
Traits:	Traits:
<ul style="list-style-type: none"> • Easily able to search through all elements • Can use the indices as a way of structuring the data 	<ul style="list-style-type: none"> • Constrains the way you can insert and access data • More efficient for solving specific LIFO/FIFO problems

图 12: Order Structures

汉诺塔游戏：

- 三个塔，从下往上分别是大到小。
- 扩展到 N 个塔也是一样
- 我们的目标是把这些塔从一个柱子上移动到另外一个柱子上，在移动的过程中要始终保持下面的比上面的要大，如图 13 所示。

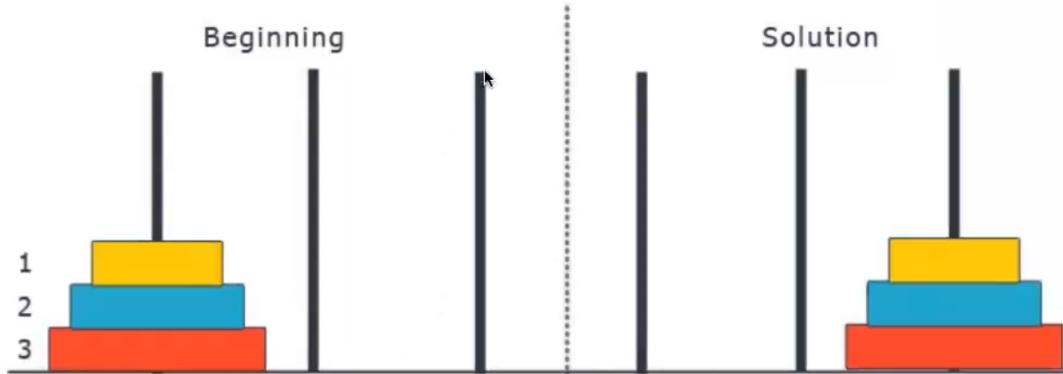


图 13: hanoi game

这其实是一个递归问题，老师用 stack 方式写了，当然递归实际上也是 stack，就练习一下使用 stack。

- 为什么我们需要用无序的数据结构？

首先无序的数据结构在现实生活中使用非常广泛，网页的访问，产品所在类别的集合等。

无序是相对于有序的 ADT 来说的，我们需要一种无序数据结构来模拟生活中出现的数据集合。

Sets 集合

- 集合是一种唯一，无序的数据结构，之前的 stack，queue 这些是可以重复的。
- 集合是比其他的有序数据结构是要更快的
- 集合没有索引，无法索引。

基本方法

- add 添加元素
- contain 是否包含某个元素
- remove 移除元素
- size 查看 set 大小
- isEmpty 是否为空

这些数据结构都是斯坦福自己 lib 里面的。set operation 自然也有，应该是都重写了

- == 判断两个集合是否相等
- != 判断当前两个集合是否有交集
- + 两个集合的并集
- * 返回他们的交集，我一开始看还以为是笛卡尔集合
- - 返回两个集合不相同的集合，包含在第一个但是不在第二个

由于这个 set 是不可能被遍历的，所以只能通过 for(auto a:set) 这种方式访问。

Maps 映射

- 键值对的集合，一个 key 对应一个 value
- 类似于 python 的字典
- 如果把 key 看成整数，那么就是 vector

主要函数

- clear 清楚当前的所有元素
- containkey 是否包含当前的 key
- m[key] 根据 key 读取 value
- get(key) 和上面一样
- isEmpty 判断是否为空
- keys() 输出所有的键值

key =value 添加键值对

- put 也是添加键值对，和上面一样
- remove 移除键值对
- size 查看大小
- values 查看所有的值

和 STL 里面的 map 其实差不多，我比较常用的是 EA 实现的 STL，感觉他们对内存管理更友好一些。

因为 map 也是无序数据结构，所以不能索引遍历，只能通过 `for(auto e: m.keys())` 的方式访问。

unique words program，需要计算文本中每一个单词出现的数目，并且统计单词。set 统计单词 map 统计次数就行了。

5.6 Using Abstraction

这节课主要是使用抽象数据结构。举了一个应用的例子，counting sort，计算当前字符串每一个字符的数量，算法流程

- 遍历整个字符串，生成一个字符的频率映射表
- 遍历 a-z，如果出现再映射表，就按照频率输出到 str 里面
- 返回新的字符串

这个例子可以用三个数据结构解决，用 set 把出现在 str 的字符存储下来，map 存储单词频率，队列进行排序输出。

5.6.1 word ladders

这是一个益智游戏，给定一个 start word 和 end word，从 start word 和 end word 中间会有 n 个中间单词，这些中间单词会缺少一个字母，需要填写这个字母才能到下一个阶梯，如图 14 所示，中间就有四个单词需要填，满足两个要求

- 填写的单词不能和前一个单词相同
- 上下两个单词首尾单词之一需要相同

Word Ladder

Write the missing letter for each word. As you go down the ladder, change one letter to show how the words connect.



图 14: word ladders

如果让人来解决这个问题，通常会用一些已有的先验知识，但是计算机没有。一个有效的算法是 breadth-first search。

Breadth-First Search ，广度优先算法。

- Insight：为了能有组织的进行单词的搜索，先探索所有距离为 1 的单词，然后再探索所有距离为 2 的单词。

比如 map 单词，step 1 阶段生成距离为 1 的单词：rap, man, mop 等。当然会遍历所有距离为 1 的单词。step 2 阶段再生成单词的基础上继续修改一个单词，把再之前出现过的单词都去除，如果没有找到 destination word，那就继续 step 3，直到找到为止。

5.6.2 BFS

正式介绍广度优先。首先我们需要什么

- 首先需要一个数据结构来表示 ladder，也就是游戏环境
- 还需要一个数据结构来探索那些还没被 explore 的 word
- 需要一个数据结构来存储所有已经探索过的 word，在新一轮的探索中如果出现重复是就去除，防止陷入循环。

word 的数据结构应该是 stack 就行，还没被探索的数据结构可以用队列，先进先出可以先把 distance = 1 的 word 探索完，已经探索过的 set 就行了，只需要判断是否 exist。

伪代码：

- 创建一个队列存储还没 expore 的单词
- 如果队列不为空
 - 把队列的第一个单词拿出来，放进 stack
 - 如果结束了推出，否则就用这个单词生成新的 neighbor word
 - * 如果 neighbor word 没有被发现过，复制目前的 ladder 一份，把 neighbor 放进去
 - * 把这个 ladder 丢进队列里面

5.7 Big O notation and algorithm analysis

5.7.1 Nested DS

就是数据结构嵌套，Queue<Stack<string>> 就是一共 Nested DS。

比如一个动物喂食的例子，每一个动物需要在特定的时间吃东西，可以用 Map 来存储，Map<string, Vector<string>>，key 表示动物名称，value 表示喂食的时间等信息。

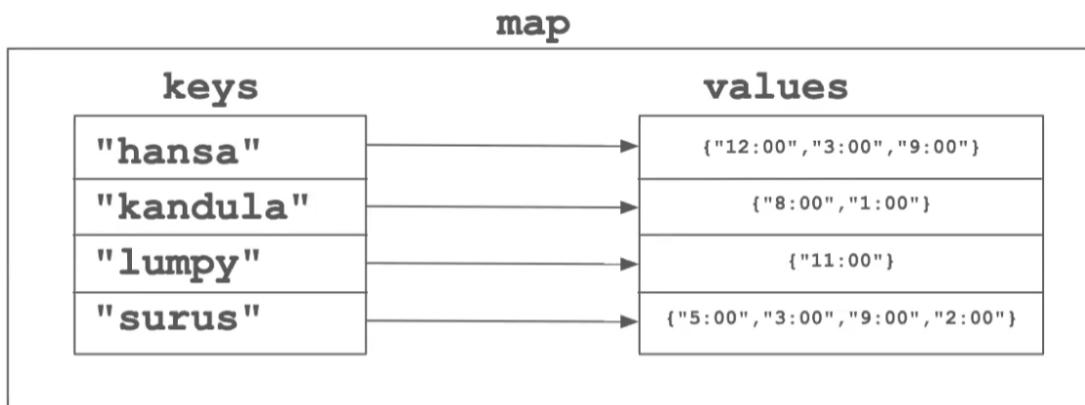


图 15: Nested DS

如图 15 所示。Map 的两个操作符：

`:` Map[key] 会得到当前 key 对应的一个 value 的引用，注意这里是引用，可以直接修改。

- `=`: 使用等号，无论右边是什么，总会复制一份新的。所以 auto a = Map[key]，a 是拷贝而不是引用。

最后还提到了一共 const+ 引用的参数传递，这种方式不存在构造和复制，速度更快。

5.7.2 Big-O

Big-O notation is a way of quantifying the rate at which some quantity grows. 大 O 是用来衡量增长速度的。比如 $O(r^2)$ 平方增长，当输入增加两倍，速度增长四倍。如果两个程序都是 $O(r^2)$ ，这并不意味着两个程序是一样的训练速度，只能说明他们的输入和计算量之间的关系，并且这种关系也是大概的，两个有相同 O 的程序计算量增长也不一定一样，因为 O 是省略了系数和更低阶。

Network Value , 如果增加 n 个用户, 那悲观估计就有增加 n^2 的通信, 所以增长为 $O(n^2)$ 。10,000,000 个用户有 10,000,000 的价值, 那么 1000,000,000 需要多少用户? 用户增加 n , 价值增加 n 平方, 那么现在价值增加了 100, 那么用户就加 10, 自然就是 100, 000, 000。

所以 Big-O 会忽略两点信息

- 系数
- 比他低阶的项

然后现在使用这个 O 来分析程序。

Runtime , 运行时间是最常见的一种。但是运行时间并不能综合评价程序的好坏, 因为在不同设备, 不同机器上程序会展示出不一样的结果。更重要的是 **runtime** 只能反应当下程序的运行时间, 并不能反应未来的运行时间。所以提出用 $O(\cdot)$ 来分析程序。

VectorMax 如图 16所示, 最高就是 n 次, 所以是 $O(n)$ 。只要看和 n 相关的语句就行, 数据结构都有讲过了。

vectorMax()

```
int vectorMax(Vector<int> &v) {  
    int currentMax = v[0];  
    int n = v.size();  
    for (int i = 1; i < n; i++) {  
        if (currentMax < v[i]) {  
            currentMax = v[i];  
        }  
    }  
    return currentMax;  
}
```

图 16: VectorMax

数据结构讲的还比这难一点。

5.8 Assignment 2. Fun with Collections

四个任务:

- warmup
- maze

- search engine
- beyond algorithm analysis

5.8.1 warmup

教你如何 debug，没啥好看的。

- Q1. The display of the Stack in the debugger uses the labels ‘top’ and ‘bottom’ to mark the two ends of the stack. How are the contents labeled when the Stack contains only one element?

只有一个输入的时候，top 和 bottem 应该指向什么？top 应该是指向唯一的那一个元素，bottem 应该指向一个空值，比如-1。

下一个 Test duplicateNegatives 这个函数，这个函数有问题，设计一个只有一个元素的 queue 测试一下：

```
STUDENT_TEST("length-1 test on duplicateNegatives"){
    Queue<int> q = {1};
    Queue<int> t = {1};
    duplicateNegatives(q);

    EXPECT_EQUAL(q, t);
}

STUDENT_TEST("length-1 test on duplicateNegatives"){
    Queue<int> q = {-1};
    Queue<int> t = {-1, -1};
    duplicateNegatives(q);

    EXPECT_EQUAL(q, t);
}
```

第一个正数没问题，第二个负数有问题。看一下代码就知道是他这个 size() 出的问题。size 每一次都会增大，而负数也会随之增大，i 也会随之增大，所以如果全部都是负数他会卡死。有一个正数就可以打破这个循环。因为循环的原因是因为 size 和 i 同步增加，每次循环到这个正数，size 和 i 的距离就近 1。

所以，如果想要这个训练终止，就必须要有一个正数，并且这些正数出现的次数要等于一开始的 q.size。比如输入-3, 4, -5, 10, 4 和 10 出现的次数加起来是四次，因为一开始 i 和

size 相差 4，每次抽到负数他们的差都不变，只有抽到正数 size-i 才会少 1。所以 4 和 10 会分别出现两次，那么-3 会循环两回，每一次增加 1，所以有 4 个-3，4 个-5。

1, 8, -5 也一样，1.8 需要出现 3 次，1 两次，8 一次。因为这个队列是先进先出，所以是先遍历完 1，再到 8 再到-5，1 遍历了两次 8 一次说明第二次遍历数字到 1 就结束了，所以 8 和-5 只遍历了一次，所以-5 只会有两个，结果就是 1, 8, -5,-5。

- Q2. For which type of inputs does the function go into an infinite loop?

第二个问题问你啥时候会进入死循环，没有正数的时候。

- Show your edited code for duplicateNegatives that fixes the problem with the infinite loop.

让你修这个函数，直接在 `val < 0` 条件加一个 `++i` 就行。

```
void duplicateNegatives(Queue<int>& q) {
    for (int i = 0; i < q.size(); i++) {
        int val = q.dequeue();
        q.enqueue(val);
        if (val < 0) {
            q.enqueue(val); // double up on negative numbers
            ++i;
        }
    }
}
```

- Q4. What is the better fix to ‘sumStack’ that corrects the bug?

让你修 `sumStack` 这个函数，就是没有判断这个 `empty`，改成 `Total = 0` 就行。

5.8.2 maze

主要是解决迷宫问题，用 `grid` 这个数据结构。这 part 的任务是实现一个 `nest` 算法来解决 `maze`，`maze` 用 `y` 一个二维数组表示，`true` 表示 `open` 可以走，`false` 表示 `wall`。

`generateValidMoves` 获得可以行动的方向，满足三个条件

- N,S,E,W 四个方向
- 必须要在 `grid` 里面

- 这个位置不能是墙

这个比较简单，注意 maze 判断放后面就行，不要用 &，& 是两边都会判断，&& 是左边不行右边就不判断了，免得到时候 maz[..] 超出边界。

```
Set<GridLocation> generateValidMoves(Grid<bool>& maze, GridLocation cur) {
    Set<GridLocation> neighbors;
    /* TODO: Fill in the remainder of this function. */
    int direction[4][2] = {{-1, 0}, {1, 0}, {0, -1}, {0, 1}};
    for (int i = 0; i < 4; ++i)
    {
        GridLocation nextLocation;
        nextLocation.row = cur.row + direction[i][0];
        nextLocation.col = cur.col + direction[i][1];
        if (nextLocation.row < maze numRows() && nextLocation.col < maze numCols()
            && nextLocation.row >= 0
            && nextLocation.col >= 0
            && maze[nextLocation]){
            neighbors.add(nextLocation);
        }
    }
    return neighbors;
}
```

validatePath , 判断一个路径是否是合理路径。

- 路径必须要是左上角为起点
- 路径必须要是右下角为终点
- 每一个点必须是可走的
- 路径不能包含循环

主要就是几个循环就行。

- Q5. In lecture, we noted the convention is to pass large data structures by reference for reasons of efficiency. Why then do you think validatePath passes path by value instead of by reference?

为什么不传递引用而是复制？因为 stack 是无法在不破坏结构的情况下遍历的，如果传引用，遍历之后结构会被破坏。

- Q6. After you have written your tests, describe your testing strategy to determine that your validatePath works as intended.

让你写几个测试样例确定是否正确运行。

solveMaze , 求解 maze。用 BFS 进行遍历，用 queue 实现

- 建立一个 empty queue，存储 path。
- 创立一个只有一个入口的 path，加入队列
- 然后继续 explore

在添加路径的时候主要不能有重复路径即可，然后加个可视化就行。

```
Stack<GridLocation> solveMaze(Grid<bool>& maze) {
    MazeGraphics::drawGrid(maze);
    Stack<GridLocation> path;
    /* TODO: Fill in the remainder of this function. */
    GridLocation mazeExit = {maze numRows() - 1, maze numCols() - 1};
    Queue<Stack<GridLocation>> UnexplorePath;
    Set<GridLocation> ExplorePath;

    GridLocation entry;
    entry.row = 0;
    entry.col = 0;
    Stack<GridLocation> firstPath;
    firstPath.push(entry);
    UnexplorePath.enqueue(firstPath);

    while (!UnexplorePath.isEmpty())
    {
        Stack<GridLocation> curPath = UnexplorePath.dequeue();
        MazeGraphics::highlightPath(curPath, "red", 10);
        if (curPath.peek() == mazeExit)
        {
            // reach rhe target
            path = curPath;
        }
    }
}
```

```

        break;
    }

    Set<GridLocation> nextSteps = generateValidMoves(maze, curPath.peek());
    for(GridLocation next : nextSteps)
    {
        Stack<GridLocation> newPath = curPath;
        if (!PathExistLoop(newPath, next))
        {
            newPath.push(next);
            UnexplorePath.enqueue(newPath);
        }
    }

    return path;
}

```

自己写了一个判断路径是否重复，原先那个 validpath 函数太难用了，直接返回 True false 不就行了吗？

```

bool PathExistLoop(Stack<GridLocation> path, GridLocation curLocation)
{
    while(!path.isEmpty())
    {
        GridLocation cur = path.pop();
        if (cur == curLocation)
        {
            return true;
        }
    }
    return false;
}

```

如果不判断是否路径循环，第二个测试样例过不了，会跑的很慢。

5.8.3 search engine

搞一个搜索引擎。主要用到的就是 Map 和 Set 两个数据结构。首先他的数据集是一个 txt 文件，这个文件每两行是一对网页数据，第一行是 url，第二行是 body text。

cleanToken 函数，输入是网页的 body text，作用就是清楚一些无用的字符，比如符号等。主要做三件事情：

- 把开头和结尾的标点符号去掉
- 确保 str 至少包含了一个字母
- 把字符串转换成 lowercase

其实应该分几个函数写，再组合的，但写一起方便一点

```
string cleanToken(string s)
{
    int start = 0;
    int end = s.length()-1;
    bool return_flag = true;
    while(start < end && return_flag)
    {
        return_flag = false;
        if (ispunct(static_cast<unsigned char>(s[start])))
        {
            ++ start;
            return_flag = true;
        }
        if (ispunct(static_cast<unsigned char>(s[end])))
        {
            -- end;
            return_flag = true;
        }
    }
    s = s.substr(start, end - start + 1);

    bool contain_alpha = false;
    for(int i = 0; i < s.length(); i++)
    {
        char c = s[i];
        if(isalpha(static_cast<unsigned char>(c)))
        {
            contain_alpha = true;
            s[i] = toLowerCase(s[i]);
        }
    }
}
```

```
    }

if (!contain_alpha)
{
    return string("");
}

return s;
}
```

gatherTokens , 做两件事: 1) 把 text 分成单词, 放进 set 里面组成词表。2) 把单词进行 clean 处理

```
Set<string> gatherTokens(string text)
{
    Set<string> tokens;
    Vector<string> split_set = stringSplit(text, " ");
    for (string s : split_set)
    {
        string c_s = cleanToken(s);
        if (c_s != "")
        {
            tokens.add(c_s);
        }
    }
    return tokens;
}
```

buildIndex , 主要三件事: 1) 读取 web 文件。2) 生成 keyword 到 url 的映射。3) 计算一共多少个网页。读取部分参考了 maze.cpp 的实现, 因为他后面要分成 url 和 keyword 的, 这里先分开

```
map<string, string> readWebFile(string filename)
{
    ifstream in;
    if (!openFile(in, filename))
    {
        error("Cannot open file named " + filename);
```

```
    }
map<string, string> webText;
Vector<string> lines;
readEntireFile(in, lines);

for (int i = 0; i < lines.size(); i+=2)
{
    webText[lines[i]] = lines[i+1];
}
return webText;
}
```

返回的就是 url 到 keywords 的映射了。然后再建立 keyword 到 url 的映射就行

```
int buildIndex(string dbfile, Map<string, Set<string>>& index)
{
    int count = 0;
    map<string, string> webText = readWebFile(dbfile);
    for (auto it = webText.begin(); it != webText.end(); ++it)
    {
        ++count;
        string body_text = it->second;
        Set<string> body_text_set = gatherTokens(body_text);
        for (string keyword : body_text_set)
        {
            if (index.containsKey(keyword))
            {
                index[keyword].add(it->first);
            }
            else
            {
                Set<string> url_set;
                url_set.add(it->first);
                index[keyword] = url_set;
            }
        }
    }
    return count;
}
```

findQueryMatches , 根据关键字进行搜索 url 网址, 有几个条件

- 返回网页 url 的集合
- 输入会包含多个关键字, 把这些查询结果都合成一个结果返回
- 输入的关键字会有一些限定符, 包含了一些含义, 比如 +-等

这个比较简单, 但有些细节要注意。他文档里面提示了要用 cleanToken, 然后我就用了这个, 发现 cleanToken 是会把单词前面的 +-都去掉, 所以在 clean 之前先判断是否有 +-号。

searchEngine , 这个没有什么测试, 就是让你实现一个接口。按照他给的样例实现就行

```
void searchEngine(string dbfile)
{
    std::cout << "Stand by while building index..." << std::endl;
    Map<string, Set<string>> index;
    int pages_count = buildIndex(dbfile, index);
    string input;
    while(true)
    {
        std::cout << "Indexed " << pages_count << " pages containing " << index.size()
            << " unique terms" << std::endl;
        std::cout << "Enter query sentence (RETURN/ENTER to quit): ";
        std::cin.ignore();
        getline(std::cin, input);
        if (toLowerCase(input) == "return" || input == "/n")
        {
            std::cout << "All done!" << std::endl;
            return;
        }
        Set<string> search_results = findQueryMatches(index, input);
        std::cout << "Found " << search_results.size() << " matching pages " <<
            std::endl;
        std::cout << search_results << std::endl;
    }
}
```

5.9 递归

之前汉诺塔就是一个典型的递归例子。首先提出一个问题

- How can we take advantage of self-similarity within a problem to solve it more elegantly?

老师举了一个 vee 的例子，有点像是分形几何。

- Definition: A problem-solving technique in which tasks are completed by reducing them into repeated, smaller tasks of the same form.
- 通过把任务分解成多个相同形式的小任务达到分解任务的效果。

递归会存在两个 case

- Base Case, 最简单的版本，不能再被拆分了。并且可以直接得出结果
- Recursive Case, 还是不是最简单的版本，还可以继续拆分

5.9.1 Factorials

阶乘的例子。这个例子也是可以用一个递归实现，因为每一个 $n!$ 的阶乘可以写成 $n * (n-1)!$ 。一直算到最后，0 和 1 就是 base case。

```
int factorial(int n)
{
    if (n == 0) return 1;
    if (n == 1) return 1;
    return n * factorial(n-1);
}
```

每一个递归都会调用新的 stack。

5.9.2 Inverse String

What is the base case and the recursive case? Base case 应该是遍历到最后一个，然后反着加起来就行。return $\text{inver}(i+1) + s[i]$ 。这种任务可以先用 stack 实现一下，然后凡是用栈的位置就是需要调用函数的地方就行了。

老师举的例子差不多，只不过是用字符串替代我这里 index:

- recursive case: $\text{reverse}(\text{str}) = \text{reversive}(\text{without first letter}) + \text{first latter}$

- base case: empty string

当然反过来也可以。

5.9.3 Summary

- 递归，将任务不断拆分成相似重复的任务解决，知道可直接求解为止
- 递归由两部分组成：Base case + Recursive case

5.10 Recursive Fractals

例子，palindrome，如果一个 string 是 palindrome，当且仅当他正过来读和反过来读都是一样的。比如 level，racecar 就是，high 就不是。如果用循环来做只需要 `for(int i = 0, j = str.length(); i < j; ++i, -j)` 就行，但他这里要求就要递归实现。所以这里的递归应该是一次去除两个字母。

```
bool palindrome(std::string str)
{
    if (str.empty())
    {
        return true;
    }
    else if (str.length() == 1)
    {
        return true;
    }
    int start = 0;
    int end = str.length() - 1;
    if (str[start] == str[end])
    {
        return palindrome(str.substr(start + 1, str.length() - 2));
    }
    return false;
}
```

分解成子问题和特殊情况就好了。

对于这个问题来说，what are the base and recursive cases? 我刚刚那个代码，recursive 就是当 str 长度大于等于 2 和前后不相等的时候，其他都是 base。

5.10.1 Fractal

递归其中一个条件就是子相似性，他自己要和他的子部分相似才行。

- self-similarity

如果 object 包含和他自己相同的部分，那么就是 self-similarity。用图像表示递归就是分形算法。如图 17 所示，每一部分单独拿出来都可以看成是一个缩小版的分形树。

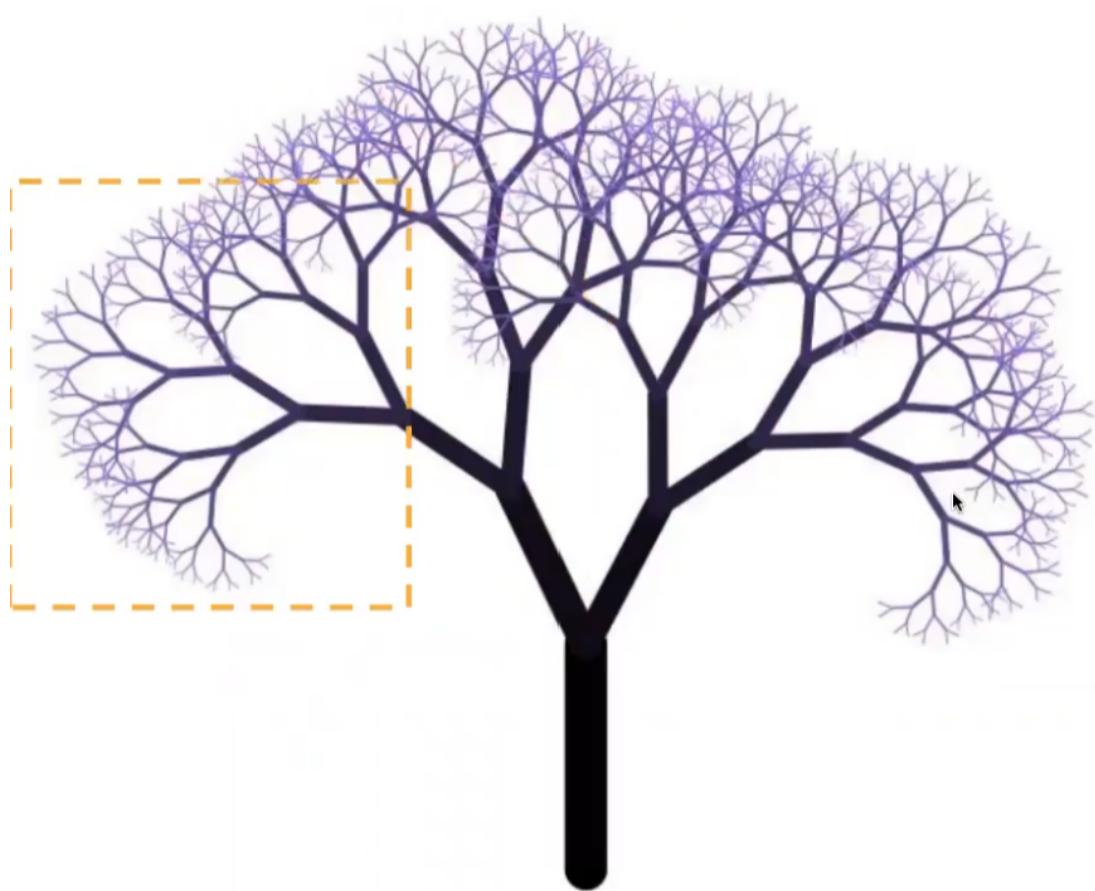


图 17: fractal

后面还举了一个康托三分集的例子，测度为 1 但是可数的集合，典型反例。后面的都比较简单容易，不记录了，感觉将的内容还不如老师上课讲的难。

5.10.2 Why Recursion?

为什么需要递归？

- 首先递归相对循环来说代码更简洁，更易懂
- 效率更高
- 可以解决一些迭代难以解决的问题

一个简单的例子就是汉诺塔，这个样例很难用循环写出来。就通过 B 把圆盘从 A 移动到 C，并且过程中都要保证小圆盘在大圆盘上面。把 $n-1$ 个盘子通过 C 移动到 B 上，然后把最后一个 n 移动到 C 上，然后再把 $n-1$ 个盘通过 A 移动到 C 上。

binary search 是另一个例子，当 $n > 1$ 的时候就是 recursive case，当 $n = 1$ 的时候就是 base case。这种例子用循环写就比较困难，因为他们的相对位置一直在变。相比一般的顺序搜索要更快，复杂度是 $O(\log n)$ 。

这节主要是想说明存在一些问题抽象程度很高，很难用一个迭代完成，这个时候就需要递归。

5.10.3 回溯

回溯算是递归的进一步，他会裁剪一部分不可能的路径，最经典的问题应该是八皇后，不过这里似乎没提及。两种不同类型的 recursion：

- Basic Recursion: 最简单的递归就是把问题不断分解，知道最简单的形式
- Backtracking Recursion: 把所有可能的解通过递归调用，然后把一些不可能的 solution 裁剪掉

感觉不是两种类型，只不过问题难度加大了一个递归无法解决而言。有三种情况是需要使用回溯方法：

- 生成所有可能的解，或者计算所有解的数量
- 找到某个特定的解，或者证明存在某个解
- 找到最好的解

其实就是当这个问题无法被分解成唯一一种情况的时候，这个问题就变成了回溯求解。如果这个问题是可以分解唯一一种情况，那么就只需要 base recursion 就可以解决。

Word Scramble 大概意思就是。首先会提供给你一系列单词，每一个单词会把特定部位圈出来。我们需要对每一个单词进行重新排序，把排序后特定位置的字母拿出来组成新的单词，对新的单词排序即可。如图 18所示。

JUMBLE

Unscramble these four Jumbies,
one letter to each square,
to form four ordinary words.

KNIDY

D I N K Y

©2015 Tribune Content Agency, LLC
All Rights Reserved.

LEGIA

A G I L E

CRONEE

E N C O R E

TUVEDO

D E V O U T

Print answer here:

ADDICTION

Saturday's

Jumbies: EL

D I A I N O D T

Answer: The cyclops' son wanted an action figure for his birthday, so they bought him a — G- "EYE" JOE

THAT SCRAMBLED WORD GAME

by David L. Hoyt and Jeff Knurek

Nick

Check out the new, free JUST JUMBLE app



THE MATH TEACHER HIRED AN ARCHITECT BECAUSE SHE WANTED A NEW —

Now arrange the circled letters to form the surprise answer, as suggested by the above cartoon.

图 18: word scramble

这个问题仅仅用一个递归是解决不了的，而且他无法分解成更小的部分，每一次分解都是一个新问题，并且问题规模并没有减少。

先讨论排序，这个问题需要用到排序，排列其实也是一个递归问题。比如一个 $P(abcd)$ 的全排列， $a+P(bcd)$ 就是下一个子问题，当然这个子问题不止一个，需要一个 for， $b+P(acd)$ 也是一个子问题， $b+P(cd)$ 又是下一个子问题，以此类推。和回溯的区别在于，这里是不需要裁剪的。排序问题算是简化版，我这里自己写的和老师的有点不太一样。首先之前的递归问题都是分解成一个子问题，比如 A 问题

$$A \rightarrow B \rightarrow C \dots$$

所以你只需要递归一次就行，但是在这里递归是分解成多个子问题，A 分解成 B_1, B_2, \dots ，所以这个时候返回的就不是一个字符串而是某一个 collection，并且还需要对这个 collection 遍历，每一次都是指数增加。

```
std::set<std::string> permutation(std::string str)
{
    std::set<std::string> results;
    if (str.length() == 1)
    {
        results.insert(str);
    }
    else
    {
        for (int i = 0; i < str.length(); i++)
        {
            std::string next_str = str;
            next_str.erase(i, 1);
            std::set<std::string> subresults = permutation(next_str);
            for (auto it = subresults.begin(); it != subresults.end(); it++)
            {
                std::string s = str[i] + *it;
                results.insert(s);
            }
        }
    }
    return results;
}
```

用 set 的原因是可能会存在重复的字符，这样可以去掉一些重复的，否则还是需要手动判

断，或者在 for 的时候判断当前字符是否之前重复出现过了。这里 remove 的时候选择用 Index 作为下标移除的原因也是因为可能会有重复的，直接 value 删除可能删到重复的。对比了一下老师写的，我和她其实反过来了，我是递归到底才开始组合，他是一边组合一边递归。

他又跳到 shrinkable words 了，我还以为他讲排序是为了讲前面那个字谜。

Shrinkable words，给定一个单词，每一次去掉一个单词，但是要保证去掉单词之后剩下的字符串仍然是一个单词，如果这个字符串最后能一直删减到只有一个字母，那就是 Shrinkable word。这个问题就需要用到裁剪了，如果某一次的 remove 发现无论去除哪一个都不是单词，那就可以去除，不需要再递归。

```
bool isShrinkable(string str, Lexicon& lex)
{
    if (str.length() == 1 && lex.contains(str))
    {
        cout << str << endl;
        return true;
    }
    for (int i = 0; i < str.length(); i++)
    {
        string substring = str.substr(0, i) + str.substr(i+1);
        if (lex.contains(substring))
        {
            if (isShrinkable(substring, lex))
            {
                cout << substring << endl;
                return true;
            }
        }
    }
    return false;
}
```

Subsets，疯狂举例子，这些例子都是层层递进。遍历一个 set 里面的所有元素，按照是否包含当前元素分成两类，每一类继续递归。前面已经把这个递归过程形式化了，寻找 Base 和 recursion 的过程其实就是构建决策树的过程，如图 19 所示。

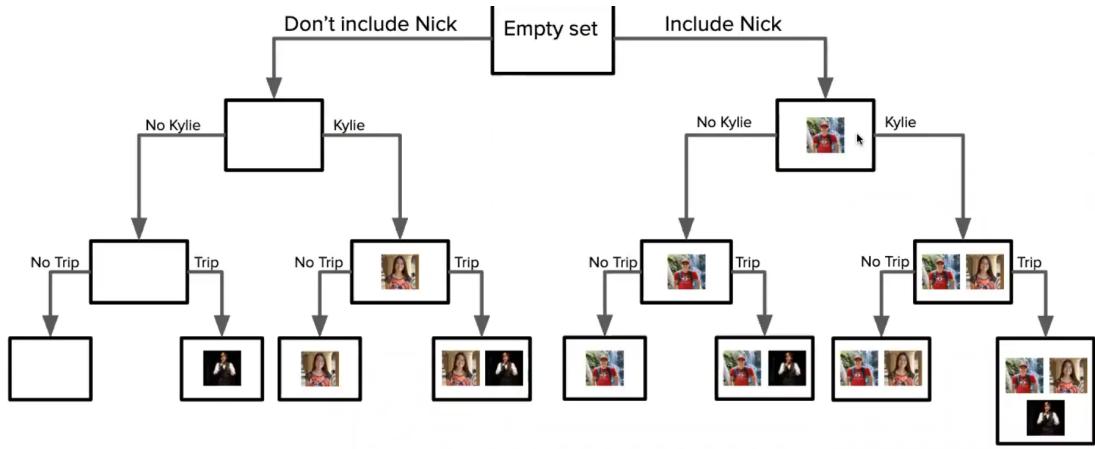


图 19: DecisionTree

后面还举了一个陪审团选择的问题，另一门 61B 专门开了一节课讲，就跳过。

5.11 Recursive Optimization

上几节课主要就是把递归形式化，分成两个过程

- Decision: 决策树每一层如何分支
- Options: 每一个分支做什么

Hard Problems: 之前的问题都是几个类别，并且类别的形式都是类似的。hard 是指多种不同类型的问题，比如 NP-hard。并且这些问题没办法知道生成 solution 的高效路径是什么，只能尝试所有路径。回溯就是解决这类方法。

5.11.1 Knapsack Problems

背包问题。这问题不是动态规划的吗？问题设定：

- 背包可以容纳一定的重量的物体
- 每一个物体包含价值和重量，需要在背包中装入价值最大的物体

最直接的方法就是贪心算法，这种方法并不高效，核心想法还是从重量比较小的物体开始组合。

递归的方法是需要遍历所有重量小于背包重量物体的子集，选择最好的一个。这个问题也包含了子集条件组合，算是上节课问题的进阶。这个问题的 idea 也组合，和上节课的组合问题是一样的，遍历子集贪心选择。

- Recursive Case:
 - 选择没有考虑过的物体
 - 递归包含和不包含这个物体
 - 选择最大值
- Base Case:
 - 重量已经超过
 - 无法再选择

老师代码似乎只考虑了物体只出现一次的情况。这一张最大的知识点应该是就把递归形式化。

5.12 Assignment 3

要写这个代码得用斯坦福自己的库，前面的几个任务没遇到啥问题，这个项目遇到了，可能是用上了一些他自己图形的库了，常见的有几个问题

- 没权限，这个忘记管理员运行了，而且主要是链接库都在 C
- gwindow.h 找不到，你需要再项目，依赖关系选上 cs106B 这个库才行
- 找不到 moc 文件

这个解决方法一般就是清除环境变量，我这里是因为自己设置了 mwing，然后 Qt 又整了一个

还有一个方法是 shadow build 取消构建一次，shadow build 勾上再运行一次

编译后右键清除再编译

- 编译会缺少文件，这个只需要在依赖关系加上 cs106 即可，只需要加一次就行，他后面自动生成 moc_xxx 文件之后就不需要手动再依赖了。
- 作业三还遇到一个问题，filelib.h 没有 readentirefile，我一开始还以为我下错了，但重新下了 cs106 之后发现他确实是没有，于是自己去找了 Stanfordcpplib 把这三个方法搞过来了

- 顺便把后面三个作业也搞下来跑了一遍，除了最后一个哈夫曼出现了函数名称不匹配的情况，其他都能运行

5.12.1 Recursion Warmup

第一个问题是让你调试，第二个问题是尝试-3 的阶乘，结果发现直接不报错直接跳出。因为爆栈了，前面两个问题就是让你了解一下 Qt 的 debug。

- Q3. Describe how the symptoms of infinite recursion differ from the symptoms of an infinite loop.

问你无限循环和无限递归有什么不同，无限递归是会不断地开辟新的栈，系统最终都会出现爆栈的问题，每递归一次就需要新的栈。但是循环只是耗时间而已。

然后让你把这个负数这个问题处理一下，负数就抛出 error，他这个 error 应该是要斯坦福库里面的函数，懒的查了就直接 throw ”n < 0”;

然后是 MyPower 这个函数，这个函数有 Bug，先是跑几个 test case 看看 bug 是啥。发现他给的几个 test 都过了，因为太简单，没有负数。做几个具有负数的 test case

```
PROVIDED_TEST("myPower(), generated inputs with negative inputs") {
    for (int base = -25; base < 25; base++) {
        for (int exp = -10; exp < 10; exp++) {
            EXPECT_EQUAL(myPower(base, exp), pow(base, exp));
        }
    }
}
```

这下就过不了了，exp 是负的没啥问题，base 是负数就出现问题了。

```
Q4. What is the pattern to which values of base number and exponent result in a
test failure?
```

问你啥情况会出现问题，主要就是在这个 base，当 base 小于 0，他就直接默认是-base**exp 了，但这个是要看 exp 是奇数还是偶数，所以当 exp 是偶数就出问题。

```
Q5. Of the existing five cases, are there any that seem redundant and can be
removed from the function? Which do you think are absolutely necessary? Are
there any cases that you're unsure about?
```

MyPower 有五个 case 实现，问你有没有不必要的 case。base case 肯定就是两个等于 0 的时候，剩下的 case 就需要分不同情况考虑。感觉 base < 0 是不需要分出来的，和 base >

0 不都是一样算吗？

后面说了两个 case 可以去掉，除了 negative base 还有 base = 0 的样例。测试的时候发现我自己输出 pow 和 mypower 的值是一样的，但是放进行 test 里面就不一样

```
Incorrect (PROVIDED_TEST, line 88) myPower(), generated inputs with negative inputs
Line 91: EXPECT_EQUAL failed: myPower(base, exp) != pow(base, exp)
myPower(base, exp) = 0.0001079796998164345d
pow(base, exp) = 0.0001079796998164345d
```

输出明明是一样的。完了死活过不了，然后发现是 pow 这里有误差，当你多打出 pow 输出几位之后就会发现和你自己实现的不一样，我一开始因为是 mypower 的问题，试一下 pow(-10, -10) 之后发现 pow 输出 1.00000001e-10d，这明显是循环被裁掉了一部分。所有我怀疑是 pow 用的 float 计算，mypower 这里已经是 double 了，所有后面 mypower 改成 float 之后就可以过了。

```
float myPower(int base, int exp) {
    if (exp == 0) {          // handle zero exp
        return 1;
    }
    else if (exp == 1){
        return base;
    } else if (exp < 0) { // handle negative exp
        return 1.0 / myPower(base, -exp);
    } else {                // both base and exp are positive
        float result = myPower(base, exp / 2);
        result *= result;
        if (exp % 2)
        {
            result *= base;
        }
        return result;
    }
}
```

和样例实现的有点不太一样，把 a^e 拆分成 $a^{\frac{e}{2}} * a^{\frac{e}{2}}$ ，这样算在数据量比较大的时候比较方便。

5.12.2 Balanced Operators

这个任务是检查代码括号是否匹配。比如 () 就是正常匹配。() 就是不匹配，(就是缺失。这个任务应该是栈来做的吧？主要实现两个函数

- operatorsFrom 把括号都提取出来
- operatorsAreMatched 根据提取括号判断是否是匹配的

他要求你第一个 operatorFrom 要用递归实现，有这个必要吗？思想类似于递归，每一次处理小一点。

然后还有要求

- 不能用 loop，也不能用 ds 存储中间结果。我用了一个 set 但我不是存中间结果，是判断是否是括号，你直接几个 & 判断是否括号也行
- 自己再提供测试样例
- operatorsAreMatched 只能包含括号，不能包含其他非括号的内容，否则返回 false

```
string operatorsFrom(string str) {
    /* TODO: Implement this function. */
    set<string> brackets = {"()", "[]", "{}"};
    if (str.length() == 0)
    {
        return "";
    }
    if (brackets.count(str.substr(0, 1)))
    {
        return str[0] + operatorsFrom(str.substr(1));
    }
    else
    {
        return operatorsFrom(str.substr(1));
    }
    return "";
}
```

Match 也是要用递归实现

```
bool operatorsAreMatched(string ops) {
```

```

/* TODO: Implement this function. */
if (ops.length() == 0)
{
    return true;
}
if (ops.find("(") != string::npos)
{
    int index = ops.find("(");
    return operatorsAreMatched(ops.substr(0, index) + ops.substr(index + 2));
}
else if (ops.find("[")) != string::npos)
{
    int index = ops.find("[");
    return operatorsAreMatched(ops.substr(0, index) + ops.substr(index + 2));
}
else if(ops.find("{}") != string::npos)
{
    int index = ops.find("{}");
    return operatorsAreMatched(ops.substr(0, index) + ops.substr(index + 2));
}

return false;
}

```

- Q6. Compare your recursive solution to the iterative approach used for the Check Balance problem in Section 2. Which version do you find easier to read and understand? In which version did you find it easier to confirm the correct behavior?

对比一下递归和迭代的实现，迭代版本就是 stack 实现的版本。那我觉得还是迭代好，一次 $O(n)$ 就能完成所有操作。但如果是可读性上看还是递归比较好阅读。

5.12.3 Sierpinski

这个内容应该是分形几何的内容，每一个递归都把边分一半画三角形。需要实现 drawSierpinskiTriangle 这个函数，每一次分别需要花三个黑色三角形，每一个顶点对应一个三角形。

Gpoint 看了一下源码好像没有重载加法，那只能把 x 拿出来自己算了。

```
int drawSierpinskiTriangle(GWindow& window, GPoint one, GPoint two, GPoint three, int
order) {
/* TODO: Implement this function. */

if (order == 0)
{
    fillBlackTriangle(window, one, two, three);
    return 0;
}

GPoint first((one.x + two.x) / 2, (one.y + two.y) / 2);
GPoint second((one.x + three.x) / 2, (one.y + three.y) / 2);
GPoint thrid((two.x + three.x) / 2, (two.y + three.y) / 2);
drawSierpinskiTriangle(window, one, first, second, order - 1);
drawSierpinskiTriangle(window, two, first, thrid, order - 1);
drawSierpinskiTriangle(window, three, second, thrid, order - 1);
return 0;
}
```

没注意他还要计算三角形数量，我还纳闷为什么要返回 int。那只需要递归加上就行了：

```
int drawSierpinskiTriangle(GWindow& window, GPoint one, GPoint two, GPoint three, int
order) {
/* TODO: Implement this function. */

if (order == 0)
{
    fillBlackTriangle(window, one, two, three);
    return 1;
}

GPoint first((one.x + two.x) / 2, (one.y + two.y) / 2);
GPoint second((one.x + three.x) / 2, (one.y + three.y) / 2);
GPoint thrid((two.x + three.x) / 2, (two.y + three.y) / 2);
int result = drawSierpinskiTriangle(window, one, first, second, order - 1) +
drawSierpinskiTriangle(window, two, first, thrid, order - 1) +
drawSierpinskiTriangle(window, three, second, thrid, order - 1);
return result;
}
```

5.12.4 Binary merge

给定两个递增的序列，把这两个序列合在一起还是递增，注意事项

- 输入的序列可能是空序列，需要特殊考虑
- 使用迭代实现，而不是递归实现

设定两个 Index 表示当前处理到的队列的位置，两个队列一起迭代，一次只处理一个，，甚至这个 length_a 都可以不定义 (因为 a.size() 是如果 pop 出数据之后是会变的)，改成 while(!a.isEmpty() && b.isEmpty()) 就行。

```
while(iter_a < length_a && iter_b < length_b)
{
    if (a.peek() > b.peek())
    {
        result.enqueue(b.dequeue());
        ++iter_b;
    }
    else if (a.peek() < b.peek())
    {
        result.enqueue(a.dequeue());
        ++iter_a;
    }
    else
    {
        result.enqueue(a.dequeue());
        result.enqueue(b.dequeue());
        ++iter_a;
        ++iter_b;
    }
}
```

这个循环跑完之后很可能两个都完了，或者还剩下一边没玩

```
while(iter_a < length_a)
{
    result.enqueue(a.dequeue());
    ++iter_a;
}
while(iter_b < length_b)
```

```
{  
    result.enqueue(b.dequeue());  
    ++iter_b;  
}
```

最后就是：

```
Queue<int> binaryMerge(Queue<int> a, Queue<int> b) {  
    Queue<int> result;  
    /* TODO: Implement this function. */  
  
    int iter_a = 0;  
    int iter_b = 0;  
    int length_a = a.size();  
    int length_b = b.size();  
    while(iter_a < length_a && iter_b < length_b)  
    {  
        if (a.peek() > b.peek())  
        {  
            result.enqueue(b.dequeue());  
            ++iter_b;  
        }  
        else if (a.peek() < b.peek())  
        {  
            result.enqueue(a.dequeue());  
            ++iter_a;  
        }  
        else  
        {  
            result.enqueue(a.dequeue());  
            result.enqueue(b.dequeue());  
            ++iter_a;  
            ++iter_b;  
        }  
    }  
    while(iter_a < length_a)  
    {  
        result.enqueue(a.dequeue());  
        ++iter_a;  
    }  
}
```

```
while(iter_b < length_b)
{
    result.enqueue(b.dequeue());
    ++iter_b;
}
return result;
}
```

不能一次处理两个，比如 $a < b$ 之后 a, b 都放进 $result$ 里面，不行的，因为你不知道 b 是不是小于 a 的下一个元素。

- Q7. Give a rough estimate of the maximum length sequence that could be successfully merged on your system assuming a recursive implementation of binaryMerge.

问如果是递归实现，粗略估计一下能合并的最大长度是多少。最坏的情况是当 a 全部遍历完了， b 还一个没动，如果 a 长度是 n ， b 长度是 m ，那么需要 $O(n+m)$ 的复杂度，假设系统最大能给的栈是 L ，那么 $n+m = L$ 就是最大长度。和前面两个题接上了，但前面 debugger 我没仔细看。

- What would be the observed behavior if attempting to recursively merge a sequence larger than that maximum?

如果超出了最大长度会怎么样，分情况，如果两个队列很对元素相同的，那么说明在递归的时候很多情况是两个元素一起操作，那么问题不大；如果是最坏情况，那就会爆栈了。

然后需要增加一个输入检查，判断输入是否都排好序的，给了两个方案

- 输入前进行检查
- 在合并的过程中进行检查

我是在每次添加 $result$ 的时候进行判断是否前面的小于后面一个。发现一个 `queue` 的问题，`peek` 这个函数在队列是空的时候会出现异常，所以每一次 `peek` 我都要判断一次。之前没检查的代码不需要是因为 `iter` 帮我判断了。这份代码写的冗余。

- Q9. Include the data from your execution timing and explain how it supports your Big O prediction for binaryMerge.

每一次添加一个元素，复杂度肯定是 $O(m+n)$ 。test 了一个样例，1000000 是 0.152，2000000 是 0.298，4000000 是 0.568，刚好差不多就是两倍。

然后他把 `binaryMerge` 封装成一个大的 `merge`，让你分析效率

- Q10. Include the data from your execution timing and explain how it supports your Big O prediction for naiveMultiMerge.

他的代码里面 k 表示多少个 queue 要合并， n 表示合并之后一共多少个元素。naiveMultiMerge 里面是有一个 k 次的循环，所以肯定要乘 k ，每次合并都是合并 n/k ，实际上内部合并还是 $O(n)$ ，所以一共应该是 $O(nk)$ ，也就是 $O(n^2)$ ，因为 k 也是 n 来表示。然后让你实验证， $n=11000$, $k=n/10$ 的时候， 0.657 , $n=22000$, $k=n/10$, 2.596 ，当 n 增加两倍，时间增加四倍，刚好是 n^2 , $n=44000$, 10.337 , n 增加两别，时间刚好也是四倍。

接下来就是实现递归版本；

- 把 k 个序列分成两半
- 左边一半调用 recMultiMerge，右边一半也一样
- 最后剩下两个序列就调用 binaryMerge

二分合并，这个也比较容易

```
Queue<int> recMultiMerge(Vector<Queue<int>>& all) {
    Queue<int> result;
    /* TODO: Implement this function. */

    if (all.size() == 1)
    {
        return all[0];
    }

    Vector<Queue<int>> left = all.subList(0, all.size() / 2);
    Vector<Queue<int>> right = all.subList(all.size() / 2);
    Queue<int> left_merge = recMultiMerge(left);
    Queue<int> right_merge = recMultiMerge(right);
    result = binaryMerge(left_merge, right_merge);
    return result;
}
```

- Q11. Include the data from your execution timing and explain how it demonstrates $O(n \log k)$ runtime for recMultiMerge.

让你验证一下这个复杂度。首先保持 n 两倍增长， $n = 90000$, 0.443 , $n=180000, 0.913$, $n = 360000$, 2.018 ，如果 n 增长 2 倍，应该增长 2.15 的时间，刚好对得上。后面的问题都比较简单，不记录。

5.12.5 Backtracking Warmup

- Q16. What is the value of totalMoves after stepping over the call to moveTower in hanoiAnimation?

moveTower 走完之后 totalMoves 这个变量是多少， totalMoves = 15.

- Q17. What is the value of the totalMoves variable after stepping over the first recursive sub-call? (In other words, within moveTower just after stepping over the first recursive sub-call to moveTower inside the else statement.)

当第一个递归走完之后 totalMoves 的值是多少。是 2

- Q18. After breaking at the base case of moveTower and then choosing Step Out, where do you end up? (What function are you in, and at what line number?) What is the value of the totalMoves variable at this point?

在 moveTower 这个函数的 base case 设置断点，问你 step out 之后去哪里，这个 totalMoves 是多少， step out 之后还是回到 moveTown，然后 totalMoves 是 1。

下一个任务是 buggy subset sum，他给了一个正确的实现函数 countZeroSumSubsets，计算加起来是 0 的子集。

- Q19. What is the smallest possible input that you used to trigger the bug in the program?

这个 bug 要什么样的输入能测出来。问题在于 `sumSoFar += v[index]`，在传参之后，当前这个位置的值也发生改变，也就是说

```
buggyCount(v, index + 1, sumSoFar += v[index])
```

这句是没问题

```
buggyCount(v, index + 1, sumSoFar)
```

这句就不对了， `sumSoFar` 出问题了，如果没有加起来是 0 的子集那不会出问题，有加起来是 0 的子集，比如 3, -3, 0 这样的，他会把 3 也输出，因为当第一项 `buggyCount(v, index + 1, sumSoFar += v[index])` 找到 3 和 -3 的时候，`sumSoFar` 就是 0 了，所以他以为 3 也是 0。Q20 也解释了。

5.12.6 Boggle Score

类似于数独，找到 valid word，一个单词需要满足下列条件才是合法的

- 至少四个字符长度
- 单词必须是有效单词
- 字符之间需要相邻，需要注意的是这里不仅仅是可以上下左右，斜方向也是可以的，也就是一共八个方向
- 每一个单词只能走一次，即使是两条不一样的路线

需要实现 scoreBoard 这个函数，递归回溯实现 + 标记走过的方块。到底是方块不能走两次还是单词不能算两次？前面说 Each word is scored only once，这里又是 You don't want to visit the same letter cube twice。会错意了，他是说单词只能出现一次，每一个路径不能有重复的 letter 出现，但是不同路径是可以出现重复 letter 的。我还以为是要贪心算，枚举 + 剪枝就行。

还有一个比较重要的是 pruning，判断当前的 word 是否是前缀，不是就可终止了。最后对单词进行几分，长度为 4 是 1 分，往后没多一个长度就多一分。

- Base Case：判断当前位置是否可以走，不能走直接就返回 0 了。如果可以走就需要判断加上字符之后是不是出现过，是不是一个单词
- Recursive Case：当前的单词是不是前缀

```
int scoreBoard_backtracking(Grid<char>& board, Lexicon& lex,
    int x_cur, int y_cur,
    string curStr, Grid<bool>& mark, Set<string>& words)
{
    if (x_cur < 0 || x_cur >= board numRows() || y_cur < 0 || y_cur >= board numCols())
    {
        return 0;
    }

    if (!isAlpha(board[x_cur][y_cur]))
    {
        return 0;
    }
```

```

if (!mark[x_cur][y_cur])
{
    return 0;
}

int scores = 0;
string newStr = curStr + board[x_cur][y_cur];
mark[x_cur][y_cur] = false;
if (lex.contains(newStr) && !words.contains(newStr))
{
    scores += points(newStr);
    words.add(newStr);
}

if (lex.containsPrefix(newStr))
{
    scores += scoreBoard_backtracking(board, lex,
        x_cur + 1, y_cur,
        newStr, mark, words);
    scores += scoreBoard_backtracking(board, lex,
        x_cur - 1, y_cur,
        newStr, mark, words);
    scores += scoreBoard_backtracking(board, lex,
        x_cur, y_cur + 1,
        newStr, mark, words);
    scores += scoreBoard_backtracking(board, lex,
        x_cur, y_cur - 1,
        newStr, mark, words);
    scores += scoreBoard_backtracking(board, lex,
        x_cur + 1, y_cur + 1,
        newStr, mark, words);
    scores += scoreBoard_backtracking(board, lex,
        x_cur + 1, y_cur - 1,
        newStr, mark, words);
    scores += scoreBoard_backtracking(board, lex,
        x_cur - 1, y_cur - 1,
        newStr, mark, words);
    scores += scoreBoard_backtracking(board, lex,
        x_cur - 1, y_cur + 1,

```

```

        newStr, mark, words);
    }
mark[x_cur][y_cur] = true;
return scores;
}

```

判断 mark 和 exist word 用的 grid 和 set, set 是引用传递, 但是 grid 我也是引用传递, 因为可能这个 grid 很大, 如果比较大的话每一次 copy 很慢, 所以还是引用传。所以在这过程 mark 完之后需要再 mark 回来,要不然会影响其他 path。

5.12.7 Voting Power

像是背包问题, 求每一个投票者的重要性。直接枚举所有可能性, 每一个可能性把他的关键人都算一遍就行。有几个注意事项

- 严格大于, 等于不算
- 算出来的值需要四舍五入
- 如果一旦大于 50 了, 那就不必要继续了, 因为后面的加入的不会是关键投票
- 少用按值传递
- 不要考虑直接构建 power set。
- 相同的 weight 具有相同的 critical number

这也是递归解决

```

void countingCritical(Vector<int>& blocks, Vector<int>& results, Vector<int>
votingIndex, int votingCount, int scores)
{
    for (int i = 0; i < votingIndex.size(); ++i)
    {
        if (votingCount - blocks[votingIndex[i]] <= scores)
        {
            results[votingIndex[i]]++;
        }
    }
}

```

```

void computePowerIndexes(Vector<int>& blocks,
int index, int votingCount,
Vector<int>& results, Vector<int> votingIndex, int scores)
{

    if (votingCount > scores && !votingIndex.isEmpty() &&
        votingIndex[votingIndex.size() - 1] == index - 1)
    {
        // cout << votingIndex << votingCount << endl;
        countingCritical(blocks, results, votingIndex, votingCount, scores);
    }

    if (index < blocks.size())
    {
        computePowerIndexes(blocks, index + 1, votingCount, results,
                            votingIndex, scores);
        Vector<int> nextVotingIndex = votingIndex;
        nextVotingIndex.add(index);
        computePowerIndexes(blocks, index + 1, votingCount + blocks[index],
                            results, nextVotingIndex, scores);
    }
}

Vector<int> transformCritical(Vector<int> results)
{
    float scores = 0;
    for (int i = 0; i < results.size(); ++i)
    {
        scores += results[i];
    }
    for (int i = 0; i < results.size(); ++i)
    {
        results[i] = (results[i] / scores) * 100;
    }
    return results;
}

```

```

Vector<int> computePowerIndexes(Vector<int>& blocks)
{

    int scores = 0;
    for (int i = 0; i < blocks.size(); ++i)
    {
        scores += blocks[i];
    }
    scores /= 2;

    Vector<int> result(blocks.size(), 0);
    int index = 0;
    int votingCount = 0;
    Vector<int> votingIndex;
    computePowerIndexes(blocks, index, votingCount, result, votingIndex, scores);
    result = transformCritical(result);
    return result;
}

```

计算复杂度实际上是指数级别的，一开始看他的 note-Duplicates/re-calculating 提到相同的 voting weight 是可以做简化的，我以为是可以终止，把 path prune 掉，但发现不行。因为虽然后面的 target block 不会成为 critical，但是前面的那些原本的 block 是会算进去的，只能在 countingCritical 的时候简化一下，但杯水车薪就没做这个简化。所以我这个并没做简化，这个问题应该是动态规划处理，在《ACM 程序样例》里面是作为动态规划的例子出的。

5.13 Object-Oriented Programming

面向对象编程，这节课开始进入抽象编程。抽象的表现形式：随着抽象程度越来越高，从中获取的信息和细节越来越少，相对于只是一个和用户对接的接口。

- Abstraction Definition: Design that hides the details of how something works while still allowing user to access complex functionality.

日常见到的 vector, Maps 这些数据结构就是抽象数据结构，也是 class 实现。每一个 class 包含接口和实现：

- interface: 用户通过接口操作

- Implementation: 具体实现

struct 和 class 的区别: **class** 可以将成员设置成不同访问级别，私有友元等；**struct** 只能默认 **public**。**class** 比 **struct** 有更好的封装特性。

创建一个类需要定义三个属性：

- 成员变量
 - 在类里面定义变量，也就是属性
 - 这些变量或属性不能直接被外界 crud
- 成员函数
 - 调用对象的方法
- 构造器
 - 如何构造实例，通常需要定义很多个不一样的

应该还有一个析构器。这节课后面一半的内容都是将 C++ 的基本语法了感觉。类分成声明和实现，声明通常就是接口了，写在.h 文件里面，通常扫一眼 h 文件就知道有些什么接口可以调用。具体的实现会写在 cpp 里面，h 文件会链接 cpp 的实现。不过有一些简单操作也会写在.h 文件里面，写成 inline 函数，比如日志类等，nvidia 就喜欢轻量级的一些部件写进 h 里面，用 inline。

header.h 文件通常会定义这个类

```
#pragma once
class abc
{
public:
    void add();

private:
    int a;
}
```

第一句宏是告诉你只需要调用一次，因为井号引入这个 h 文件的时候是会复制这个头文件的内容，如果不加这个那么有可能这个 h 文件会被多个 cpp 复制过去，导致程序在链接的时候不知道找谁。我常用 define 这个宏来代替 pragma，因为有时候我会定义 static 变量，h 文件里面可能只有一部分我是需要 once copy 而已。

```
#include "header.h"
void abc::add()
{
    return;
}
```

实际上通常还会有一个命名空间，因为可能会出现同名函数，比如 EA 有自己的 vector，stanford 也有自己的 STL 库，这就有可能重复。

5.14 Dynamic Memory and Arrays

Array，前面有一个和 array 相似的数据结构 vector。Array 会分配数个连续的块，这些块之间的地址是连续的。

5.14.1 动态内存分配数组

- 声明一个指针指向一个分配数组，什么类型的数组就什么类型的指针。比如 string 的数字就是 string 类型的指针。但是这个字符类型实际上并不是指针的真实类型，double* 的指针还是 4 个字节，只不过代表指向的内容类型而已。
- 通过 new 分配内存空间。

动态分配内存和静态分配内存有几个比较大的区别：

- 动态分配内存不是在编译的时候分配内存，是在运行时分配内存。静态内存是在编译的时候就分配好了，所以静态内存中指定的数量一定要是 const，因为 const 也是在编译的时候分配，不能是变量。
- 动态分配内存是在堆上进行，他是会占用 cpu 的内促，但是静态是栈上的内存。
- 动态内存是不能被自动回收的，静态内存的可以。

动态内存分配的注意事项

- C++ 速度和简洁性优先
- new 分配的内促是固定大小，不会增长
- new 是没有边界检查，使用指针访问可能越界

Stack 和 Heap 的区别：

- 目前为止的变量都是定义在栈上，如果是通过指针动态分配的变量，比如 `int* a = new int()` 会分配到堆上
- 栈是静态分配使用的内存空间，堆是动态分配所使用的内存空间
- 存在栈的变量访问非常快，堆没有这么快，但是存在堆的变量更好控制
- 在栈上的变量会自动回收，在堆上的变量需要自己回收

5.15 Implementing an ADT

这节课是需要实现一个数据结构，这个应该比较简单。根据前面的设计类的原则，先是变量，接口，然后构造器。

- 变量：`element` 指针，需要动态分配内存，`allocatedCapacity` 数组大小，`numItems` 目前存储了多少个变量。其实 STL 里面的数组是栈定义的，也就是直接 `arr[NUM]`，并不是用堆分配。
- 然后是定义一些接口，写在 `h` 文件里面，和 `vector` 一样的基本操作了。
- 构造器初始化，`allocatedCapacity` 就是 `new` 分配的大小，`element` 就是要用 `new` 分配，`numItems` 有效数量。

这里用的方法是 `new` 的方式去分配内存，程序是不会自动回收内存的，所以我们需要一个析构器去释放内存，如果是 `string` 这些栈的内存就不需要了。这些操作里面 `add` 比较特殊，如果加入的超过了 `capacity`，那需要重新分配内存然后再赋值了。这两节课没啥好记录的，如果是语法操作的话其实另外一门 EA 大佬讲的更好。

5.16 Priority Queues and Heaps

优先队列。这一节主要是 multi-levels of Abstraction，也就是利用现有的数据结构构建更高级别的数据结构，优先队列就是一种数据结构，这节课应该就是利用之前提到的数据结构去构建这个优先队列吧。

- Abstract Data Structures ← Data Organization Strategies ← Fundamental C++ Data Storage

首先什么是优先队列，队列是先进后出，优先队列是在队列的基础上加上优先级，自动对 `element` 进行排序。优先队列的接口

- `enqueue` 加入元素，插入的元素需要给定优先级，根据优先级插入

- dequeue 删除最高优先级的元素
- peek 返回有最高优先级的元素

priority queue 和一般的 queue 区别在于他需要根据优先级维护数据的有序性。优先队列分别对应的是

- Abstract Data Structures ← Data Organization Strategies ← Fundamental C++ Data Storage
- Priority Queue ← Sorted Array, Binary Heap ← arrays

主要使用的还是 binary heap。

5.16.1 Binary Heap

Binary Heap 的性质

- parent 的优先级比孩子的优先级要高。
- 每一个节点有两个孩子，每一个父母节点如果有孩子，要么两个孩子，要么只有左边的孩子，不能只有右边的孩子。
- 两种类型，Min-heap 越小的数字表示越高的优先级，Max-heap 越大的数字表示越高的优先级。

heap 用数组存储，如果 parent 是第 i 个，那么 left child 是第 $2*i+1$ 个，right child 是第 $2*i+2$ 个。如果已知孩子节点是第 i 个，那么父亲节点是第 $(i-1)/2$ 向下取整。

heap 主要的三个操作

- peek: 只需要选择输出优先级最高的数值就行。
- enqueue: 插入比较麻烦，需要维护这个序列的有序性
 - 包含两步，首先插入，其次排序，也就是 bubble up。一开始先将元素插入底层节点，然后把这个节点和父母节点进行比较，判断是否要往上走。只需要和父节点进比较就行。注意的是只需要和父节点进行比对，不需要和兄弟节点进行比对，因为 heap 只能判断父节点和子节点之间的关系，兄弟节点的关系是无法确定的。
 - array 的插入只需要包含进新元素即可，复杂度是 $O(1)$ ，这里存在一个 Bubble up 的过程，复杂度是 $O(\log n)$

- `dequeue`: 也是类似的，弹出一个新元素之后是需要对列表进行维护。这里的维护操作是 bubble down，和 enqueue 反过来了，其实也很简单，把最底层的子节点挪动到优先级最高的地方，然后往下排就行。
 - 数据结构里面提供了一种基于线索二叉树的方法，这种方法可以直接基于 $O(1)$ 把他 bubble down 了

5.17 Memory and Pointers

内存和指针，这节课相对重要。

内存和持久化存储（硬盘）都是用于存储信息，但是内存断电则丢失。内存，也是就 RAM，类似于很多箱子随机化存储信息，每一个箱子都有自己的内存地址，根据这个内存地址可以读取信息。

```
string pet = "cat";
```

cat 存储在 RAM 中，他会有一个地址指向这个 cat，我们可以知道这个地址是什么，但是无法决定存储在哪里，RAM 并不是可编程的。

Memory Organization Summary

- 每个变量都会有一个地址
- 每一个地址是唯一的
- 计算机会知道程序变量的地址
- 给定一个内存地址，能够根据地址找到变量

程序员和计算机内存通过指针操控，也就是通过指针对接。指针是一种新的数据类型，能够定义一个指针。

```
string* a;
int* b;
char* c;
```

指针类型只是表示当前指向内存的类型，并不代码指针的类型就是如此，`string*`, `int*`, `char*` 指针的类型都是整型，`sizeof` 可以看到都是四个字节。后面的内容主要围绕几个事情

- 指针可以使得我们和内存直接交互，但这也是危险的，对于无用的指针最好设置为 `nullptr`

- 指针动态分配是需要手动回收的

这种基础语法还是 CS106L 或者 cherno 讲的比较好。

5.18 Assignment 4. Priority Queue

五个任务，这次作业都比较简单。

5.18.1 Warmup

第一个任务主要还是让你调试一下，我也是醉了。

- Q1: allballs[0] 这个实例里面的 xyid 是怎么变的

x 和 y 分别表示当前这个球的坐标，同时还会维护一个方向 vx 和 vy，每一次迭代坐标就加上方向。如果到达了边界那方向反过来即可。id 是实例的唯一标识，这是不会变的。

- Q2: How do the values of the member variables of the stuck ball change from iteration to iteration? Contrast this to your answer to the previous question.

应该是问你球卡出了变量咋变，方向反过来，x 和 y 的方向会反过来。

- Q3: After forcing the stuck ball to position (0, 0), does the ball move normally from there or does it stay stuck?

没卡住

- Q4: On your system, what is the observed consequence of these memory errors

当 index outside 直接就是超出访问范围，delete memory twice 是内存错误，access deleted memory 是不会报错，但是会指向一个完全错误的区域。

5.18.2 PQArray and PQHeap

- Q5: There are extensive comments in both the interface (pqarray.h) and implementation (pqarray.cpp). Explain how and why the comments in the interface differ from those in the implementation. Consider both the content and audience for the documentation.

cpp 和 h 中的注释有什么不同，cpp 中的注释更详细，涉及到了一些具体的实现方法，用什么数据结构等。h 的注释只是告诉你输入输出，隐藏了实现细节。

- Q6: numAllocated 和 numFilled 有什么区别。

numAllocated 表示这个数据结构能够容纳多少元素， numFilled 表示这个数据结构目前有多少的有效数据。

- Q7: 访问数据结构是可以直接访问变量获取，比如可以直接访问 numFilled 知道目前有多少元素，或者直接通过 array[index] 得到元素，为什么还是设计了一些类似 size(), isEmpty() 这些函数？

主要是为了安全考虑，不能让用户直接 access 这些资产，比如数据库，提供一个接口就行了。同时这也是 C++, java 这些语言的特性。

实现注意事项

- pqarray 中的数据是从大到小
- 一旦超出容量，重新分配的容量是之前的两倍
- 核心操作就是在插入的时候找到属于当前元素的位置，一个比较简单做法是先插入到最后，然后再交换过来
- 不能对数组重新排序

```

void PQArray::enqueue(DataPoint elem) {
    /* TODO: Implement this function. */
    if (isFull()) {
        realloc();
    }
    _elements[size()] = elem;
    for (int i = _numFilled; i > 0; --i) {
        if (_elements[i].priority > _elements[i - 1].priority) {
            swap(i, i - 1);
        }
    }
    _numFilled++;
}

```

主要做三件事，判断是否已满，加入新元素，新元素排序。

- Q8: Give the results from your time trials and explain how they support your prediction for the Big-O runtimes of enqueue and dequeue.

enqueue 的复杂度是 $O(n)$, 排序的时候最差就是排到第一个, dequeue 是 $O(1)$, 因为只需要把 numfilled 处理一下就行。

5.18.3 PQueue Client and Data Science Demos

- Q9; Based on the Big O of enqueue/dequeue, what do you expect for the Big O of pqSort if using a PQArray? Run some timing trials to confirm your prediction, and include that data in your answer.

问这个 PQsort 的复杂度是多少, 一共插入 n 个数据, 每一个数据最大复杂度是 $O(n)$, 所以一共是 $O(n^2)$ 。

实现一下 topk, 也就是保留前 k 个优先级最高的元素, 首先判断是否满 k 个了, 如果不满直接加入就行, 他自己会会排序, 如果满了就比较 peek 的优先级, 如果是小于当前 peek 的优先级, 那就直接去除即可。

- Q9:Based on the Big O of enqueue/dequeue, what do you expect for the Big O of topK in terms of k and n if using a PQArray? Run some timing trials to confirm your prediction, and include that data in your answer.

问复杂度。首先这里用的是 heap, heap 的插入优先级是 $O(\log n)$, 而里面最多 k 个元素, 所以就是 $\log k$ 。最外面的 while 是 $O(n)$ 复杂度, 因为 n 个元素, 里面的两个 if 和 else, 最高就是 $\log k$, 所以复杂度一共是 $O(n \log k)$

5.19 Linked Lists

链表, 一个和数组差不多的数据结构。国内的数据结构课程都是从难的开始学, 后面就简单了, 国外这些好像都是从简单的开始搞。

5.19.1 Linked List

列表特点

- 链表是一系列节点的有序组织
- 每一个节点包含两个信息
 - 任务所需数据
 - 指向下一个节点的指针, 通过指向下一个节点的指针能够遍历整个链表

链表的大致结构:

- (Data + Link Pointer) → (Data + Link Pointer) ...

通常链表一个节点的定义

```
struct Node{  
    string data;  
    Node* next;  
}
```

链表操作

- Traversal, 遍历
- Rewriting, 重新排列
- Insertion, 插入新元素
- Deletion, 删除元素

老师用 linked list 实现一个 stack

- push: 需要注意一下, 先让新节点指向后继节点, 然后再把 top 节点指向新节点, 如果 top 先指向新节点, 那么后继节点会被丢失。
- pop: 把第一个链表的节点去掉, 注意需要手动释放, 都是 new 分配。最好不要直接让 `top = top -> next`, 因为这样会丢失下一个节点, 而下一个节点是需要释放的, 所以需要用一个临时变量缓存。
- isEmpty: 判断当前这个 Top 的指针是否指向空即可。
- Deletion: 删除也是需要 $O(n)$ 复杂度一个一个的遍历删除, 直接调用 pop 其实就可以。

链表的特点就是添加元素, 删除元素相对容易, 但是遍历或者查找特定的元素不简单。

5.19.2 Linked List Operation

上一节是用 list 模拟一个 stack, 这节课是对 list 主要操作进行阐述。What is a linked list, 如图 20所示。

- a chain of nodes

- 每一个节点包含两组信息
 - 当前节点的数据
 - 下一个节点的指针信息
- 通过第一个节点可以直接遍历整个链表
- 最后一个节点是特殊的节点，一般就是 Nullptr

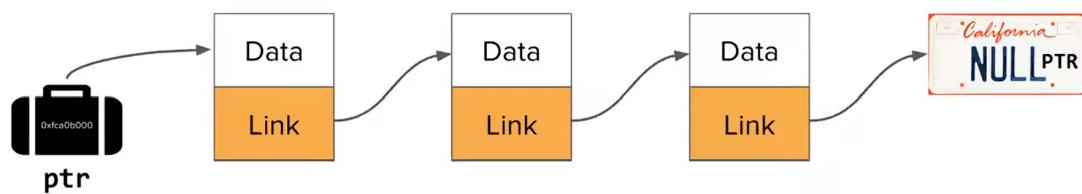


图 20: 链表

链表的核心操作

- Traversal, 遍历所有的元素
- Rewriting, 元素顺序重排
- Insertion, 插入新的元素
- Deletion, 特定位置的元素

Traversal 遍历，也就是打印列表。首先尝试直接打印：

```
Node* list = createlist();
cout << list << endl;
```

list 本质上还是一个指针，输出还是十六进制的数字。需要写一个函数一步步遍历，当指针指向空的时候就结束。计算链表的大小的时候也是只能通过遍历计算或者添加时计算，因为我们只能拿到 linkedlist 的头指针，类似于窗口的句柄这样。

Free 释放链表内存，链表的每一个节点都是独立分配，所以每一个节点的内存都是动态分配内存，需要手动释放。而释放内存是需要遍历整个链表。释放链表的时候注意 next 的指向顺序。

Linked List and Recursion，链表可以看成是一个递归结构，一个节点不是指向下一个节点就是空指针。

```
void print(Node* list){  
    if (list == nullptr)  
        return;  
    cout << list -> data << endl;  
    print(list->next);  
}
```

这样写代码会更优雅，但是每一次递归所需要的栈都是会存储上下文空间，需要的内存更大。

Insertion，在链表的首位插入元素

```
void prependTo(Node* list, string data)  
{  
    Node* newNode = new Node;  
    newNode -> data = data;  
    newNode -> next = list;  
    list = newNode;  
}
```

这段代码运行后不会有任何的改变，因为传入的 list 是值传递而不是指针或引用传递。应该传 `Node**` 或者直接引用传递，或者返回这个指针然后接收。否则在函数处理完之后 `newNode` 和 `list` 都会被删除掉。

- 没有特殊指代，C++ 传递的参数都是值传递。
- 一个函数传递指针会把指针的值复制进来
- 能够在函数内部改变指针的值，但是无法改变原始指针。

老师后面用引用传递解决了这个事情。在尾部插入，其实是一样的，把前一个当成是头指针，用 `prepend` 就行。主要需要分情况，空列表和非空列表的情况。

5.19.3 LinkedList sorted

在链表的最后加上一个数据的复杂度是 $O(n)$ ，如果用前面的 `appendTo`，也就是在最后一个位置增加元素的方式，那么按照给定数值列表创建一个新的链表是 $O(n^2)$ ，老师希望改进这种方式。其实接着上次添加的位置在后面加就行， $O(n)$ 复杂度。

后面提的是按条件添加，其实差不多。他这里的排序有点笨，选择一个最小的元素丢到首位，然后从第二个位置开始往后选择最小的元素放到第二个位置。这样的复杂度是 $O(n^2)$ 。但其实还有更简单的，直接从中间把这个链表拆开，然后归并复杂度只有 $O(log n)$ ，或者直接创建一个新链表一个个插进去，这样复杂度 $O(n)$ 。

老师选了个冒泡排序，冒泡其实也是 $O(n^2)$ 的复杂度。

5.20 Assignment 5. Linked Lists

5.20.1 Memory Debugging Warmup

这个主要就是熟悉一下 debug 流程。

- What does the yellow background for a test case indicate in the SimpleTest result window?
 - 问你运行之后出现了什么错误，出现了内存泄漏的问题。
- What is the observed consequence of a test that uses `delete` on a memory address that has already been deallocated?
 - 对已经释放掉内存的变量使用 `delete` 的结果是什么？当然是内存错误了。释放一个没有被使用的地址。
- On your system, what is the observed consequence of `badDeallocate`? Under what circumstances (if any) did the buggy code trigger an error or crash?
 - 问你 `badDeallocate` 在什么情况下会崩溃，观察到的结果是什么。其实大部分情况下是不会出现问题的，因为 `delete` 和 `win` 上的回收是一样的，他只会把这个位置标记成可以覆盖，而不是直接清空这个地址的内容。比如 `delete a`，他会把 `a` 内存标记为可以覆盖，但并不会删除里面的东西。当你的程序内存非常小的时候，这是没有问题的，因为内容还在，但当你需要占用的内存非常多，`delete` 掉之后就会被其他内容立刻写入。这个时候就会出问题。
 - 当你把 `size` 调到十万的时候就会出现 invalid memory access 的问题了，因为这个时候内存不够他需要立刻回收并占用那些被 `delete` 的内存。

- How is a segmentation fault presented on your system?
 - 没明白他想问啥？我这写的是内存非法访问出现的。

5.20.2 Labyrinth Escape

- Q5: What is a different legal path through the example labyrinth that gathers all three needed items?
 - 他应该是让你重新找一条合法路径出来，其实只要能收集到这三个物体即可。

实现 isPathToFreedom 这个函数，就判断这个路径是否合理。

- 每一步是能走的
- 走完后，或者在这个过程中需要把 needs 里面的 string 都收集好
- 并且输入的路径需要合法

```

bool isPathToFreedom(MazeCell* start, string moves, Set<string> needs) {
    /* TODO: Your code here */
    if (start->contents != "" && needs.contains(start->contents))
    {
        needs.remove(start->contents);
        if (needs.size() == 0)
        {
            return true;
        }
    }
    string dir = "NSWE";
    for (char c : moves)
    {
        if (dir.find(c) == -1)
        {
            error("Reports errors if given illegal characters");
        }

        if (c == 'N' && start->north != nullptr)
        {
            start = start->north;
        }
    }
}

```

```

else if (c == 'E' && start->east != nullptr)
{
    start = start->east;
}
else if (c == 'W' && start->west != nullptr)
{
    start = start->west;
}
else if (c == 'S' && start->south != nullptr)
{
    start = start->south;
}
else
    break;

if (start->contents != "" && needs.contains(start->contents))
{
    needs.remove(start->contents);
    if (needs.size() == 0)
    {
        return true;
    }
}
return false;
}

```

我这里写了两个 if, 第一个 if 是因为头一个位置也需要判断是不是包含了 needs 的。最后他还要你自己写一个合法路径出来, 需要自己起名字, 然后他会用 hash 编码一下生成一个新的路径, 这个比较麻烦要进 debug 里面看。

5.20.3 Sort linked list

链表排序。Q6 问你能承受多少次递归, 作业三没认真做, 这里忘记怎么算了。他还着重强调了不要用递归, 用迭代循环, 他要用大数测试。这个作业应该听完 Adavnce sort 在写的, 但之前都学过。要求

- 需要在原链表上进行复写, 不能分配新空间
- 不能使用 STL

- 不能使用额外的数据结构
- 泛化性强，就是能处理更多更长的数据。

然后注意功能函数不能用递归实现。

知道为什么要写 concate 了，因为以前学的时候这个快排的输入是有一个 start 和 end 的，这里没有 end，这就是以前学的单链表排序。首先加了两段测试，分别是 divide 和他最后在测试时间的时候没有测试是否正确。

```

STUDENT_TEST("Test divide"){
    Vector<int> values = { 5, 4, 3, 1, 6, 7, 4, 9, 11, -9};
    ListNode* list = createList(values);
    ListNode* end = list;
    while(end->next != nullptr)
    {
        end = end->next;
    }
    divide(list, end);
    printList(list);
}



---


PROVIDED_TEST("Time linked list quicksort vs vector quicksort") {
    int startSize = 50000;

    for(int n = startSize; n < 10*startSize; n *= 2) {
        Vector<int> v(n);
        ListNode* list = nullptr;

        /* Create linked list and vector with the same random sequence. */
        for (int i = n-1; i >= 0; i--) {
            v[i] = randomInteger(-10000, 10000);
            list = new ListNode(v[i], list);
        }

        /* NOTE: This test does not check correctness of the linked list sort
         * function. It only times the two operations to compare relative speed. */
        TIME_OPERATION(n, quickSort(list));
        TIME_OPERATION(n, v.sort()); /* Standard vector sort operation is backed
                                     with quicksort algorithm. */
        EXPECT(areEquivalent(list, v));
    }
}

```

```
    deallocateList(list);
}
}
```

最后这里加了个比较是否正确。首先是 divide, divide 主要就是按照 pivot 分成大于和小于两部分, 这里两个指针 left 和 right, left 指向的就是 pivot 的位置, left 前面都是比 pivot 小的。当 right 比 pivot 小, 说明这个位置是需要交换的, 因为在 Left 后面的都应该比 pivot 大。

至于为什么不直接交换 left 和 right 而是要把 left 往前移动一个位置再交换, 是因为 left 本身指向的是 pivot 的位置, 这个位置是空的。从这里也可以看出 quicksort 是不稳定的, 排完人可能不会按照相对顺序了。

```
ListNode* divide(ListNode*& front, ListNode* end)
{
    if (!front || front->next == nullptr)
    {
        return front;
    }

    int pivot = front->data;
    ListNode* left = front;
    ListNode* right = left->next;
    while (right != end)
    {
        if (right->data < pivot)
        {
            left = left->next;
            swapData(left, right);
        }
        right = right->next;
    }

    swapData(front, left);
    return left;
}
```

quicksort 在 divide 之后需要分别对两边进行排序, 需要递归, quicksort 的输入只有 start, end 只能通过是否为空判断。所以需要断开

```
void quickSort(ListNode*& front) {
```

```

/* TODO: Implement this function. */
if (!front || !(front->next))
{
    return;
}
ListNode* end = front;
while(end != nullptr)
{
    end = end->next;
}
ListNode* mid = divide(front, end);

ListNode* left = front;
ListNode* right = mid->next;
mid->next = nullptr;

quickSort(left);
quickSort(right);
concat(left, right);
}

}

```

其实 quicksort 增加一个输入都不需要这么麻烦加了 concat。

- Q7: 为什么要传指针的引用？

上课老师讲过了，但其实我猜老师的交换是把整个 node 换过来，我这里是只换 data，这个 node 是不变的，所以实际上我这只传值也是没问题的。如果你是换节点，那就是需要引用了。

- Q8: 验证时间复杂度

测试样例就能看出来了， $5w$ 和 $10w$ 的时间应该相差 $\frac{5w \log(5w)}{10w \log(10w)} = 2.128$ ， $5w$ 的时候是 0.015 乘上刚刚好就是 0.319 约等于 0.32。算下去分别是 0.067 - 0.069, 0.181 - 0.192

最坏的情况当然是极端情况的时候了

- Q9: 对比 STL。说错了不是 STL，他这个库是斯坦福自带的。

quicksort 块多了，vector.sort 用的应该是选择排序或者冒泡，明显是 n^2 复杂度的排序算法。

5.21 Advanced Sorting

这次课主要讲的是排序，主要会提到两个算法：

- Merge Sort, 归并排序
- Quick Sort, 快速排序

前面讲了 selection sort，每一次选择最小的一个元素，把他放在首位， n 个元素就需要选择 n 个，时间复杂度是 $O(n^2)$ 。另一个提到的方法的 insertion sort，也就是冒泡排序，第 i 个元素和前 $i-1$ 个元素比较，放到正确的位置上。复杂度也是 $O(n^2)$ ，相比选择排序他算好的了，因为选择排序无论什么情况都是需要比较 n 次，一直都是 n^2 次。

Divide and Conquer 是更高级的一类算法，比如递归，分治等。一个 n 长的数列，用前面的算法进行排序，复杂度是 n^2 ，但是拆分成两个数列，他们的复杂度分别是 $(\frac{n}{2})^2$ 。从 Divide-and-Conquer algorithm 受到启发：

- 排 N 个数的数列需要时间 t ，排两个 $\frac{N}{2}$ 的数列需要 $\frac{t}{2}$ 的时间，我们可以通过分成更小的数列加快排序。

所以核心思想是将数列分成多个子列，分别排序，然后合并。事实上这些子列可以并行，比如 cuda 的 radix sort 就是归并排序的并行版本。

5.21.1 Merge Sort

归并排序的递归版本

- base case: 空列表
- recursive case: 主要做两件事
 - 分成两个更简单的列表
 - 把这两个列表排序完成后重新结合

但其实有时候并不需要分到很小，在数量比较小的时候用冒泡更有性价比，比如 STL 里面的归并就是这样，分到一定程度了你用冒泡他更快的。归并就比较容易了。

拆分和合并复杂度都是 n ，串行运行，一共 $\log n$ 次递归，所以复杂度是 $O(n \log n)$ 。目前基于比较的排序算法没有能在平均复杂度下超过 $O(n \log n)$ 的了。

5.21.2 QuickSort

他这的块排和我学的基本一样，但伪代码有一点区别，但差不多。

- 将 List 基于 pivot 分成三组，一般都是 list 的第一个元素，这是第一步算法，在作业里面需要实现 partition 算法。
 - 小于 pivot 的一组，大于 pivot 的一组，相同的一组
- 递归对大于和小于的两组 list 进行排序。
- 然后 concatenate 三组即可。

对应作业 5 就需要实现两个功能函数：partition 和 concatenation。感觉很麻烦，其实两个指针 left 和 right 就能完成 partition，甚至都不需要 concat。我感觉他这里应该是默认你在 partition 的时候会链表断开了，但实际上直接在原始链表上操作就行。快排总体的思路

- divide: 根据 pivot 拆分 list
- join: 把这三部分合起来
- 递归进行剩下的两部分

和 merge sort 不同的是，在 merge sort 中 merge 这个过程是排序的核心阶段。但是在快排中 divide 才是核心阶段。QuickSort 也是 $O(n \log n)$ ，但在最坏情况下会退化成 n^2 ，

5.22 Tree

Linked list 的一些问题：

- 链表的优势很明显，删除和插入非常简单，只需要调整指针位置即可。
- 链表的缺点是由于他的链式结构，想要到达某一行元素只能通过遍历。

树是一种层次结构，包含一个根节点和其他一些非空子树，同时他也是一种递归结构。树能快速索引的原因是他对数据进行了分类，比如八叉树，把空间分成在不同位置。树的一些定义：

- root node: 根节点，有且只有一个
- children node: 孩子节点，如果节点被箭头指向，他可以成为孩子节点
- 父母节点：如果从该节点有延伸出的箭头，那么该节点可以称为父母节点

- 叶子节点：如果他不是父母节点，那就是叶子节点，只要他没有孩子。
- siblings node: 同一个父母的节点互相称为兄弟节点
- path: 两个节点直接的连接称为路径
- depth of node: 节点深度，从 root node 到当前节点的路径长度

总结：

- 非空的树的顶端都会存在一个根节点
- 每一个节点都会有 0 个或者多个孩子，这里没有特指二叉树，所以是可以多个节点
- 每一个节点最多只有一个父母
- 通过 path 可以从父节点到子节点，algorithm 这本书里面提到过一个双向二叉树，但是后面被链式二叉树淘汰了
- 深度和高度的定义，深度是边的数量，高度是节点的数量

看了下后面居然没讲红黑树，平常我用的比较多就是红黑树，八叉树。

5.22.1 Tree Properties

- 任何节点最多只能有一个父节点
- 树不能有循环，要不然停不下来了，其实就是单向箭头

理论上，子节点可以有很多个，但是在计算机里二叉树是用的最多的，一个节点最多有 2 个孩子，分别是左孩子和右孩子。

5.22.2 Tree Traversal

三种遍历方式，先序，中序和后序遍历，都是递归遍历。还是有层序遍历。

pre-order 先序遍历，先对当前节点操作，然后遍历左，再遍历右。

in-order 中序遍历，左中右，

post-order 左右中。就是一个顺序的不同。

5.23 Binary Search Tree

也就是 set 和 map 的底层数据结构。对于一个集合来说，BST 有很多种，如图 21 所示，这是比较好的情况，两边的数量差不多， $O(\log n)$ 的复杂度。

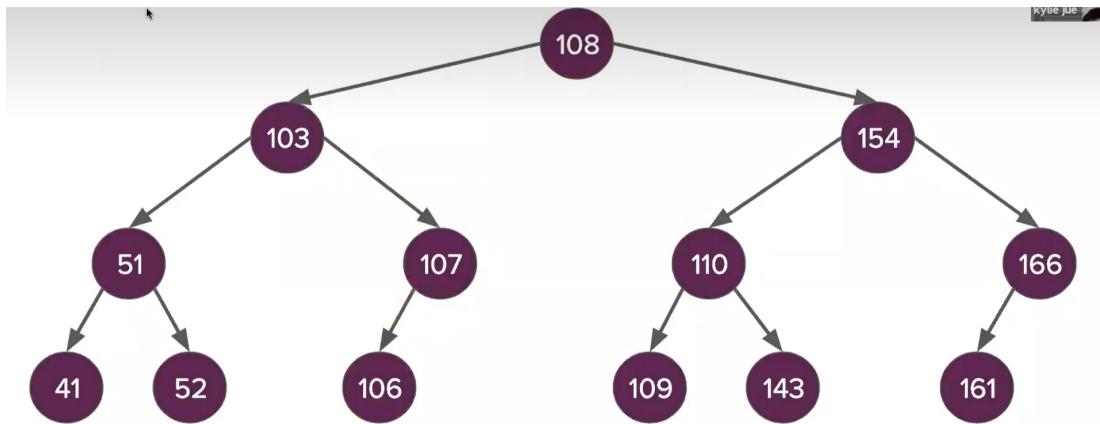


图 21: BST

如果是情况比较差的时候，也就是刚刚好是选择最小元素构建，如图 22 所示，复杂度是 $O(n)$ 。

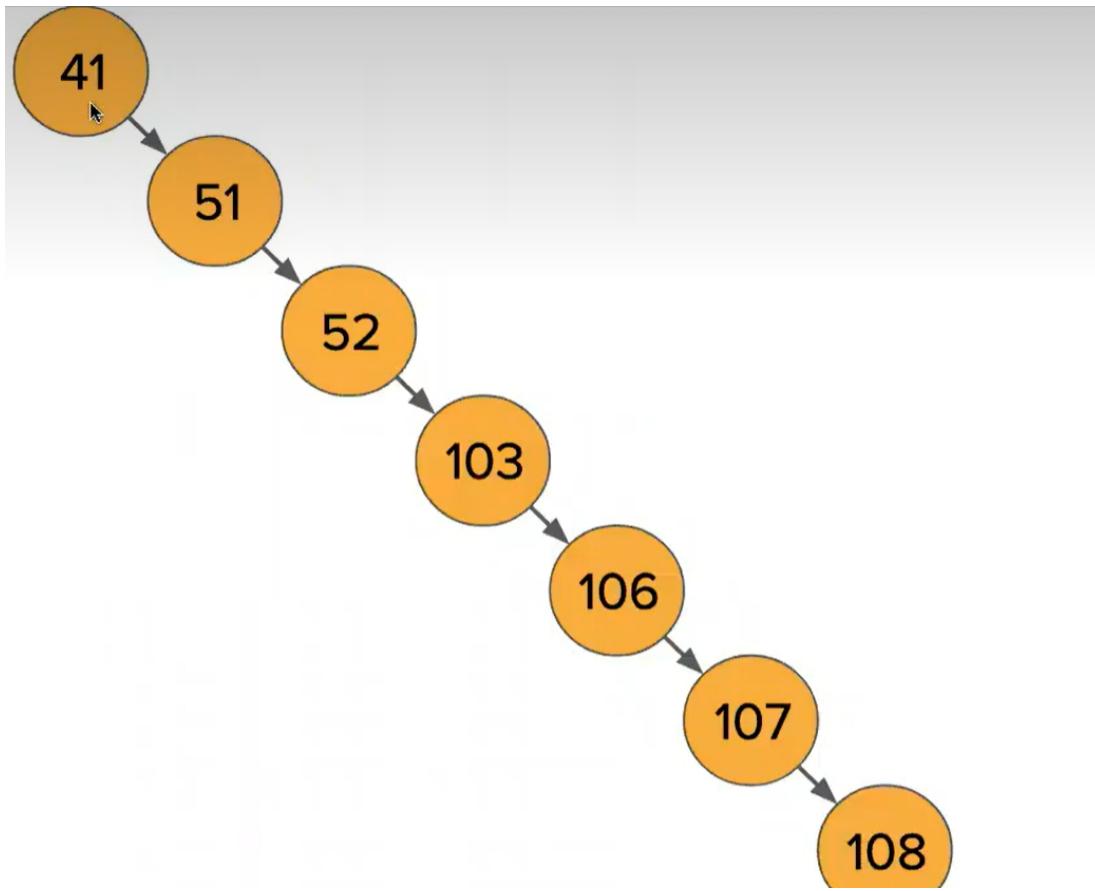


图 22: BST

一颗 BST 的最好情况就是平衡，也就是 logn 的情况，如果是随机插入，有很大概率是平衡的，当然也可以自行调整为平衡状态。

后半部分都是在实现 BST 的基础接口，自己写一个 set，没什么记录的必要了。

5.24 Huffman Coding

前面一堆铺垫讲编码，按照信息论里面的定理，哈夫曼编码是最佳码。哈夫曼编码的流程：

- 构建每一个字符的频率表
- 初始化空队列

- 每一次按照频率选择 n 个字符构建一个节点，一般是几进制就选几个节点。然后删除掉被选择的节点。
- 以此类推直到只有一个节点

5.25 Hashing

哈希编码，instantngp 典型的应用，后面还延伸出了 geo-hashing，InstantNGP 的 occ grid 就是用这种方式存储，将空间规划成 z 字型。Hash 是为了解决链表存储空间过大的问题，换句话说解决稀疏数据存储的问题。0, 1, 1000 用数组存储需要 1000 长度的数字（如果我们想达成 O(1)）复杂度的搜索的话，但是这会产生许多无用的空间浪费。而 hash 可以解决这个问题，当然相应带来的问题就是 hash 冲突了。

hash 函数的两个特点：

- 总结来说就是单射。没有两个特点。

典型的 hash 如图 23 所示，他这里处理冲突的方式就是直接塞进链表里面。

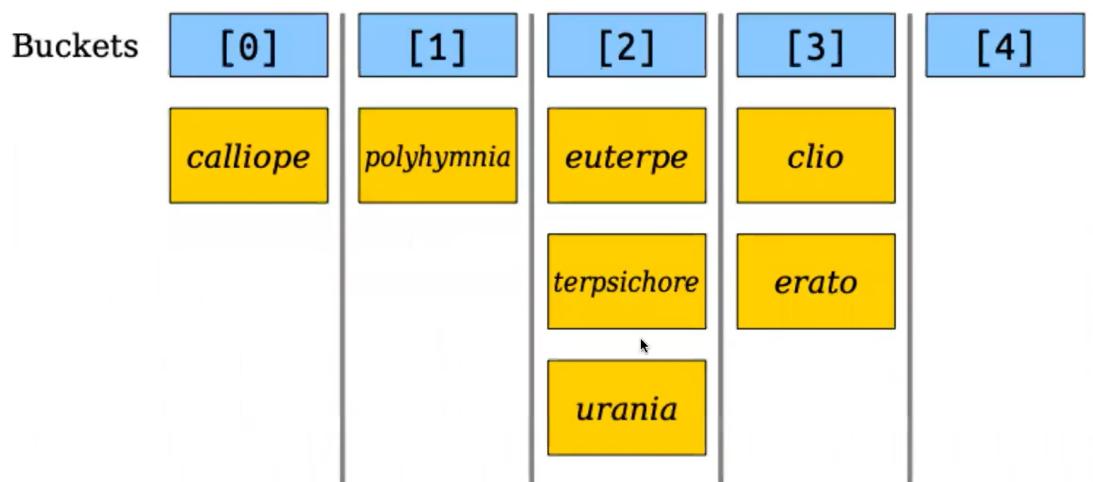


图 23: Hashcode

假设有 b 个空间和 n 个元素，称 $\alpha = \frac{n}{b}$ 为 load factor，如果 α 比较大，那么 hash 速度降低，如果 α 比较小，那么 hash 会浪费很多空间。此时，查询的复杂度为 $O(\frac{n+1}{b})$ 。一般当 $\alpha > 2$ 的时候，就需要重新分配空间。

冲突的处理方式还蛮多的，简单的有直接上链表，或者 +1，或者直接把多余的元素投射到其他空间。

5.26 Assignment 6. Huffman Coding

两个任务：

- Warmup
- Huffman

5.26.1 Warmup

- Q1. Use the above encoding tree to decode the bit sequence 0101100011.
 - 第一个问题让你解码 0101100011，哈夫曼编码都是前缀码，直接看前缀就行。看他上面的图，O 对应 1，N 对应 00，M 对应 010，S 对应 011，所以是 010-1-1-00-011 = MOONS
- Q2. Prepare a table for the above encoding tree that lists each character with its assigned bit sequence. Use your table to encode the string "SONS".
 - 编码 SONS, 011 1 00 011
- Q3. Huffman codes obey the prefix property: no character's encoded bit sequence is a prefix of any other. What feature of an encoding tree demonstrates that it obeys the prefix property?
 - 哈夫曼编码具有各个编码前缀不同的特点，问你为什么他会具有这样的特点。从信息论上看，满足 kraft 不等式，当然只是一个必要条件；主要是他在构造的过程中，他的字符只存在叶子节点上，非叶子节点是不可能带有字符信息。如果两个字符的编码前缀要相同，带字符的编码需要存在中间节点上，但这不可能的。
- Q4. Flatten the encoding tree above on the right into its sequence of bits and sequence of characters.
 - 让你仿造前面的做法对 hash tree 进行编码。分两种，一种是对字符的编码，也就是叶子节点，后序遍历，一种是树结构的编码，也就是把所有的节点都一起编码进去，中序遍历。这个题目有点像那种给你中序，后续，让你把树的结构给整出来的题目，只给先序和后序是不行的。

- 树结构的编码：1101000。字符编码：NMSO
- Q5. Unflatten the sequences 110100100 and FLERA to reconstruct the original encoding tree.
 - 这就是以前数据结构期末考试那种题。给了先序中序求树结构。如图 24所示。

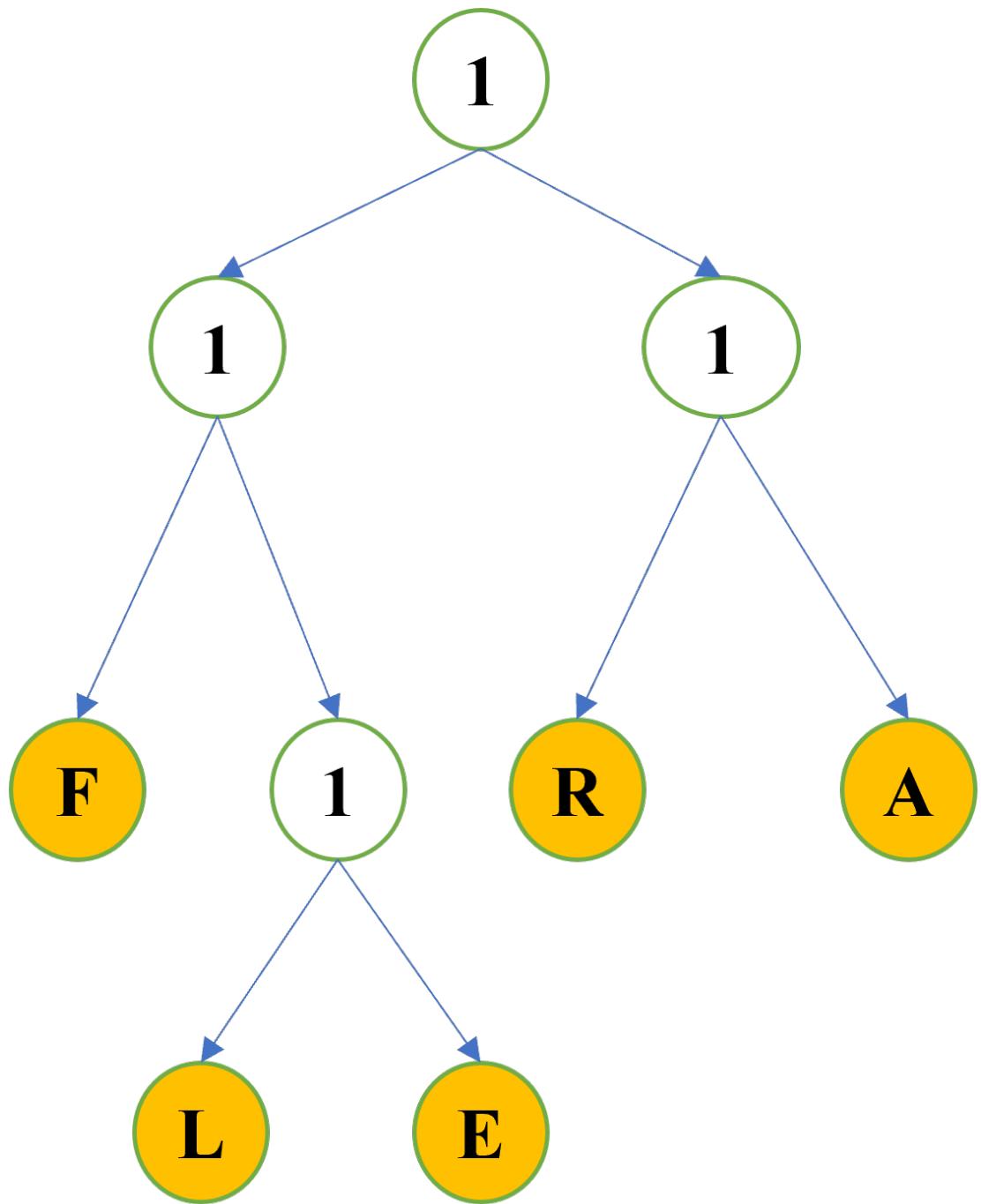


图 24: encodingTree

- Q6. Construct a Huffman coding tree for the input "BOOKKEEPER".
 - 先统计词频: B-1, O-2, K-2, E-3, P-1, R-1
 - 选择 BP 做为一组, 剩下 BP-2, O-2, K-2, E-3, R-1
 - 选择 OR 为一组, 剩下 BP-2, OR-3, K-2, E-3
 - 选择 BP, K 为一组, 剩下 BPK-4, OR-3, E-3
 - 选择 OR, E 为一组, 剩下 BPK-4, ORE-6
 - 最后两组合并即可
- Q7. A node in a Huffman coding tree has two non-null children or no children. Why does it not make sense for a node in a Huffman tree to have just one non-null child?
 - 首先是他的构造过程就是每次选择两个元素合并成新节点
 - 其次如果存在一个节点只有一个孩子节点, 那么说明这个节点是可以删去的, 因为有孩子的节点不会存储编码字符
- Q8. Describe the difference in shape of a Huffman coding tree that will lead to significant savings for compression versus one that will achieve little to no compression.
 - 如果一颗哈夫曼树很平衡, 左右节点孩子数量几乎一样, 那么压缩效率会很低。
 - 如果一颗哈夫曼树偏科, 一边很多节点一边没有, 那么效率会很高。
 - 因为哈夫曼树在压缩过程中出现次数越多会给予更短的编码, 出现次数越少的给予更长的编码。当字典中字符频率差异很大的时候, 编码效率很高, 但是当字符频率差不多的时候几乎就不会压缩了, 也就是平衡的情况。
- Q9. 举一个例子说明不压缩的情况。
 - 比如 AABBCCDDEE, 就几乎不会压缩, 因为都一样频率

然后要求实现几个工具函数。

createExampleTree 这个函数只是创建一颗特殊的 example tree 而已, 直接创建几个节点合成就好了。

deallocateTree 释放树的空间, 后序遍历删除就行。

areEqual 需要对比两棵树的结构和叶子，也就是作业里面的 tree shape 和 tree leaves，分别用先序和中序。一定要有中序，先序和后序是不能唯一确定一颗树的。然后对比序列即可：

```
void preorderString(EncodingTreeNode* root, string& returnString)
{
    /*
     * tree shape
     **/


    if (root == nullptr)
    {
        return;
    }
    if (root->isLeaf())
    {
        returnString += '0';
    }
    else
    {
        returnString += '1';
    }
    preorderString(root->zero, returnString);
    preorderString(root->one, returnString);
}

void inorderString(EncodingTreeNode* root, string& returnString)
{
    /*
     * tree leaves
     **/


    if (root == nullptr)
    {
        return;
    }
    if (root->zero == nullptr)
    {
        returnString += root->ch;
    }
}
```

```
    inorderString(root->zero, returnString);
    inorderString(root->one, returnString);
}
```

然后根据他的要求写 5 个测试样例就行。

5.26.2 Huffman

主要就是实现对文本的解码和编码。decodeText 根据一棵树对编码进行解码，也就是 warmup 里面的 Q1，给一棵树相当于给定了每一个字符的编码，然后给定一堆编码让你解。常规有两个做法：1) 先把树对应的编码都写出来，比如 S=101 这样，然后再对着解码。2) 直接接着编码走一遍。比如 25 所示。

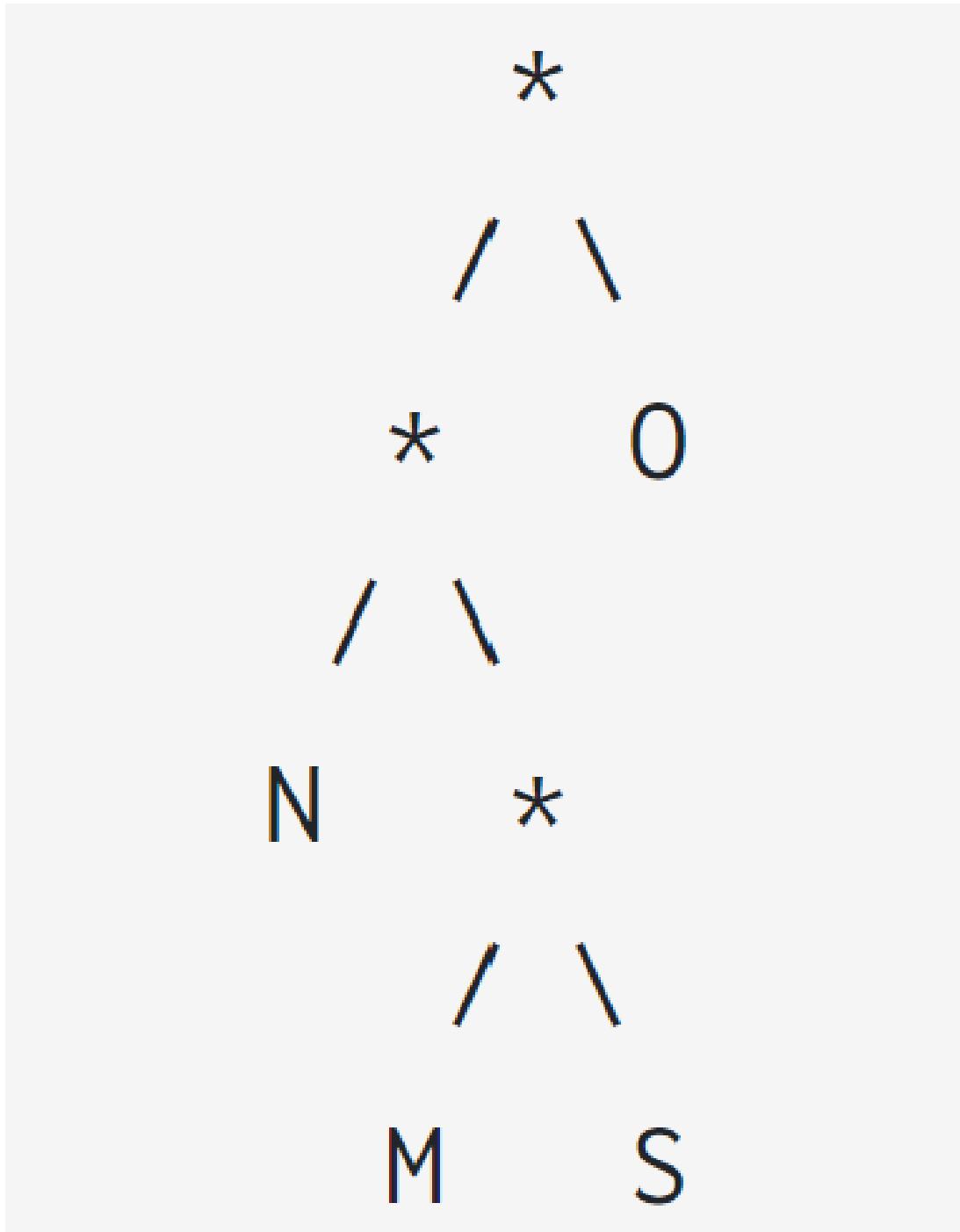


图 25: encodingTree example

给定编码 0101100011，那你只需要按照编码，0 往左走，1 往右走，走到叶子就记录 char，重新根节点开始。所以这可以用递归，在这里 010 1 1 00 011 这个四个字符的内部是迭代实现，字符之间是递归实现

```
void decodeText_(EncodingTreeNode* tree, Queue<Bit> messageBits, string& returnString)
{
    if (messageBits.isEmpty())
    {
        return;
    }

    EncodingTreeNode* temp_root = tree;
    while (!temp_root->isLeaf())
    {
        if (messageBits.dequeue() == 0)
        {
            temp_root = temp_root->zero;
        }
        else
        {
            temp_root = temp_root->one;
        }
    }

    returnString += temp_root->ch;
    decodeText_(tree, messageBits, returnString);
}

string decodeText(EncodingTreeNode* tree, Queue<Bit>& messageBits) {
    /* TODO: Implement this function. */
    string returnString = {};
    decodeText_(tree, messageBits, returnString);
    return returnString;
}
```

Unflatten 先序遍历和后序遍历构造一颗树。我的想法是先用先序遍历把 shape 生成出来，在后序遍历赋值，两个递归实现。首先是递归把树给构造出来，base case 是两个情况：1) 当 queue 是 empty 的时候，2) 当弹出的值是 0 的时候，说明是叶子节点不需要递归。不需要判断节点是否为空，因为 queue 里面的元素就能判断了。

```
void preorderReconstruction_(Queue<Bit>& treeShape, EncodingTreeNode* &root)
```

```

{
    if (treeShape.isEmpty())
    {
        return;
    }

    if (treeShape.dequeue() == 0)
    {
        root = new EncodingTreeNode('A');
        return;
    }
    else
    {
        root = new EncodingTreeNode(nullptr, nullptr);
    }
    preorderReconstruction_(treeShape, root->zero);
    preorderReconstruction_(treeShape, root->one);
}

EncodingTreeNode* preorderReconstruction(Queue<Bit>& treeShape)
{
    EncodingTreeNode* root = nullptr;
    preorderReconstruction_(treeShape, root);
    return root;
}

```

基本就是按照遍历输出的顺序，只不过输出变成创建节点，因为节点是要创建的，所以不用加上 `root == nullptr` 的判断。然后 inorder 把叶子节点填上即可

```

void inorderReconstruction(Queue<char>& treeLeaves, EncodingTreeNode* root)
{
    if (root == nullptr)
    {
        return;
    }
    inorderReconstruction(treeLeaves, root->zero);
    if (root->isLeaf())
    {
        root->ch = treeLeaves.dequeue();
    }
}
```

```

    }
    inorderReconstruction(treeLeaves, root->one);
}

EncodingTreeNode* unflattenTree(Queue<Bit>& treeShape, Queue<char>& treeLeaves) {
/* TODO: Implement this function. */
EncodingTreeNode* root = preorderReconstruction(treeShape);
inorderReconstruction(treeLeaves, root);
return root;
}

```

然后是把 decompress 写了，这个只需要调用前面的 API 就行。只要注意最后需要调用 deallocateTree，我前面好像有几个作业，我好像是把释放写到 test 里面了，应该包装在 API 里面。

后面一部分是编码。encodeText 是利用 tree 对输入文本进行编码。分两步，先把字典导出来，然后根据字典编码。生成字典

```

void buildDictionary_(EncodingTreeNode* tree, Queue<Bit> temp, std::map<char,
Queue<Bit>>& dict)
{
    if (tree->isLeaf())
    {
        dict[toupper(tree->getChar())] = temp;
        return;
    }

    Queue<Bit> temp_0 = temp;
    temp_0.enqueue(0);
    Queue<Bit> temp_1 = temp;
    temp_1.enqueue(1);

    buildDictionary_(tree->zero, temp_0, dict);
    buildDictionary_(tree->one, temp_1, dict);
}

std::map<char, Queue<Bit>> buildDictionary(EncodingTreeNode* tree)
{
    std::map<char, Queue<Bit>> result;

```

```

    Queue<Bit> temp;
    buildDictionary_(tree, temp, result);
    return result;
}

```

前中后序都是可以，反正都是按照父到子的顺序走。

flattenTree 是把树的 shape 和节点编码出来，好家伙我之前 areequal 的时候做了这个事情，难道 isEqual 的时候不用这样做吗？我想了一下确实不太需要这么麻烦，只需要两棵树同步进行遍历就行了，如果是非叶子就判断两者的孩子树是否相同，如果是节点就判断孩子。equal 改成这样

```

if (a == nullptr && b == nullptr)
{
    return true;
}
if (a == nullptr || b == nullptr)
{
    return false;
}
if (a->isLeaf() && b->isLeaf())
{
    if (a->getChar() == b->getChar())
    {
        return true;
    }
    else
        return false;
}
if (a->isLeaf() || b->isLeaf())
{
    return false;
}

if (a->zero != nullptr && b->zero != nullptr)
{
    return areEqual(a->zero, b->zero);
}
if (a->one != nullptr && b->one != nullptr)
{

```

```

        return areEqual(a->one, b->one);
    }
    return false;
}


```

flatten 的那两个函数之前就写了。buildHuffmanTree 建立哈夫曼树，用优先队列。注意的是在自定义比较函数的时候，他默认是稳定排序。

- 统计词频
- 建树

```

class cmp{
public:
    bool operator()(std::pair<EncodingTreeNode*, int> & a,
                     std::pair<EncodingTreeNode*, int>& b){
        return a.second >= b.second;
    }
};

EncodingTreeNode* generateTree(std::map<char, int> dict)
{
    std::priority_queue<std::pair<EncodingTreeNode*, int>,
                        std::vector<std::pair<EncodingTreeNode*, int>>, cmp> q;
    for (std::map<char, int>::iterator it = dict.begin(); it != dict.end(); it++)
    {
        EncodingTreeNode* node = new EncodingTreeNode(it->first);
        std::pair<EncodingTreeNode*, int> p(node, it->second);
        q.push(p);
    }

    while(q.size() >= 2)
    {
        std::pair<EncodingTreeNode*, int> l = q.top();
        q.pop();
        std::pair<EncodingTreeNode*, int> r = q.top();
        q.pop();
        EncodingTreeNode* node = new EncodingTreeNode(l.first, r.first);
        std::pair<EncodingTreeNode*, int> newP(node, l.second + r.second);
        q.push(newP);
    }
}


```

```
    }
    return q.top().first;
}
```

统计词频比较简单，建树这块没什么难点，也可以自己写个 search 函数把最小值找出来，但要注意的是，他的排序是稳定排序，比如 12321 排序，排完之后相同数字的次序需要相同。这里用的是优先队列。还有就是不需要考虑大小写，我前面都默认是变成大写然后处理，然后最后一个例子你得记录大写的位置是哪些，然后解码的时候调整。我就懒得考虑了，反正不考虑也能过测试。

5.27 Graph

后面图这块找不到作业了。图的应用

- 社交网络
- 化学组成
- 地铁线路等

图的一些基本组成和性质

- 两个节点如果存在边链接，那么就是邻居
- loop 表示起点和终点相同的边，cycle 是指起点和终点相同的路径
- 如果每一个节点都存在路径能到达其他节点，那么这个图就是 connected 的。他这里给的图是无向图，如果是有向图需要额外考虑。
- 如果每一个节点都有一条边能直接到达另一个节点，那么就是完全图，也就是 complete

总结

- 图的组成
 - 邻居
 - 路径
 - cycle
 - loop

- 图的性质

reachable

connected

complete, 完全图和链接性应该都是建立在 reachable 上, 因为需要区分有向图和无向图。

图的类别

- 按照边是否有方向

– 有向图

– 无向图

- 按照边是否有权重

– 加权图

– 无权图

还有其他不同分类方式, 比如根据性质分, 二部图等。链表, 树和图都是 linked data Structure

- linked list: 一个节点最多只能和一个节点关联
- tree: 一个节点只能有一个父节点, 不能有环
- graph: 无限制

图的表达有两种方式, 一种是邻接矩阵, 另一种是链表。

- 链表: 图中多少个节点就多少个链表节点, 每一个节点存储当前节点信息和他要链接的下一个节点信息
- 邻接矩阵: $n \times n$ 的矩阵, 每一个位置存储边的信息

这两种表达都可以表示有向图和无向图。

5.27.1 Iterating over a Graph

tree 有三种表达先序, 中序和后序。图的一种简单遍历方式是 BFS, 广度优先, 树也是特殊的图, 广度优先对 tree 的遍历就是层次遍历。可以用来寻找当前节点之间的最短路径, 队列实现。按照邻居的距离, 从一条边到两条边以此类推。

BFS 的性质:

- 只能在无权图进行遍历
- 如果边有权则不能用 BFS

5.27.2 Dijkstra 算法

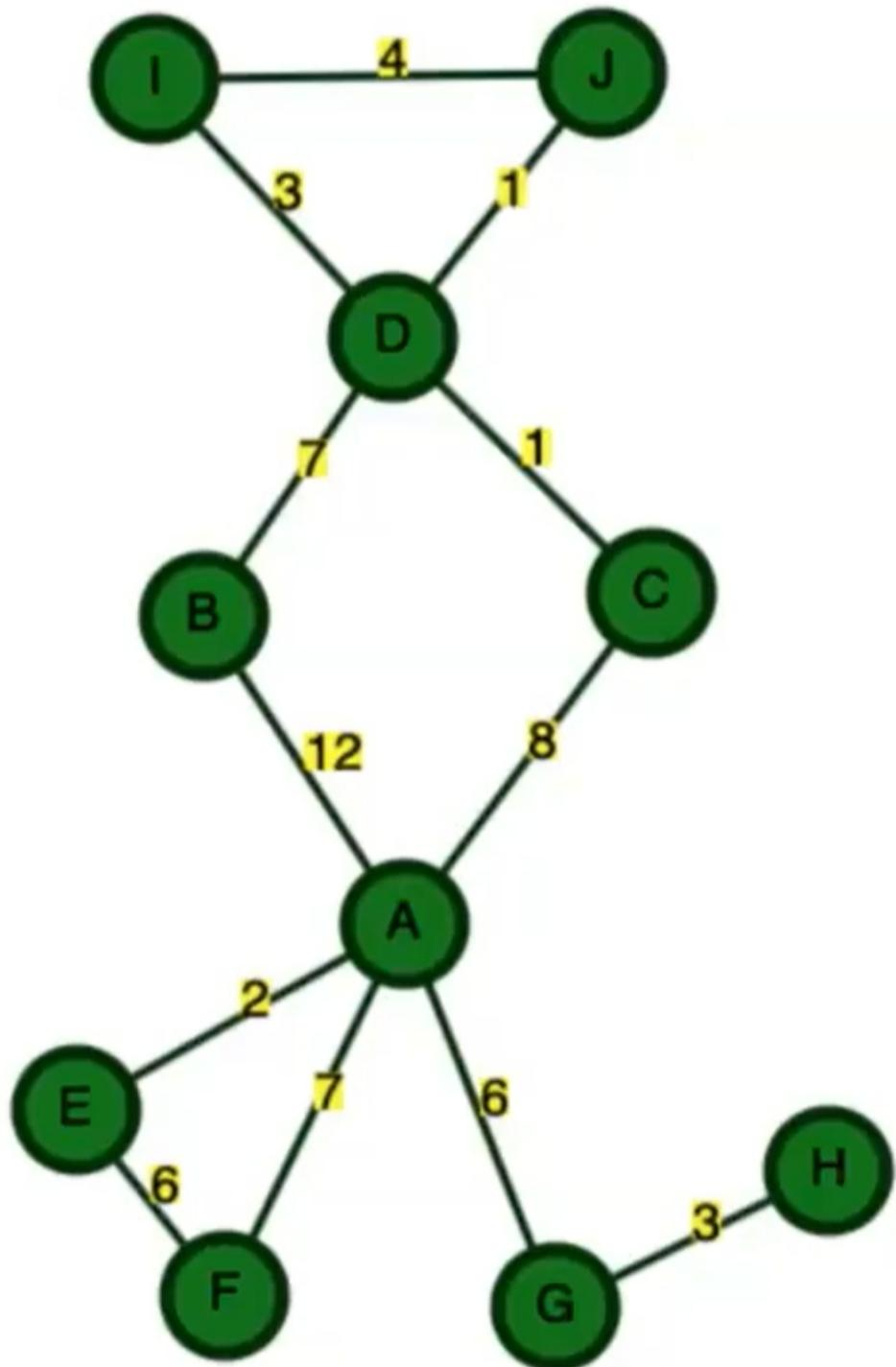
有权图的最短路径算法。在 BFS 的基础上加上了权重的比较。

- 可以在有权图找到最短路径，负数权重是不行的。
- 在一些特殊情况，Dijkstra 可以利用额外信息加快搜索。

这里简化了算法，具体流程：从原点开始选择距离原点距离最近的节点，然后查看通过该节点到达其他节点是否比从原点到达更短。这里存在一个选择最近距离定点的过程，这个过程如果出现负权重会出现死循环。因为当选择了一个最小的边，这个边指向节点 A，而 A 存在负权重指向 B，那么此时，存在一种可能，原点通过 B 到达 A 是可能比原先到达 A 的路径更短的，而 Dijkstra 是不会二次考虑到 A 的距离，他一次做完决定之后就不会再回头了。

5.27.3 A*

如果没有任何的额外信息，Dijkstra 算法应该是最好的。A* 相比 Dijkstra 算法利用了方位的信息。比如 26 所示。



187

图 26: Astar Example

假设从 A 到 J，那大概率是向上走而不是向下。

- 这种使用额外信息的算法称为启发式算法
- 因为他没有评估所有的路径，只是根据先验开发一部分路径，所以通常会 underestimate 路径，也就是局部最优
- 把先验信息加入到优先级中
- 距离通常用曼哈顿距离

Graph 没细讲，就简单提了一些概念。

6 CUDA 官方文档

<https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html> 只是粗略介绍了一下，大概知道是什么，能干什么。

6.1 Introduction

相比 CPU，GPU 比 CPU 的吞吐量更大，内存更大。之所以两者的能力不同，是因为两者设计的初衷就不一样。CPU 更多是为逻辑服务，通过一个个线程去执行，更多是希望能进行一些复杂的逻辑运算。GPU 则是并行为主，适合那些重复上千次的简单计算。

如图 27所示，GPU 基本上把所有的 CPU 中的流控制和缓存都换成了计算单元，因为在程序设计中就默认了 GPU 只进行一些简单的计算。

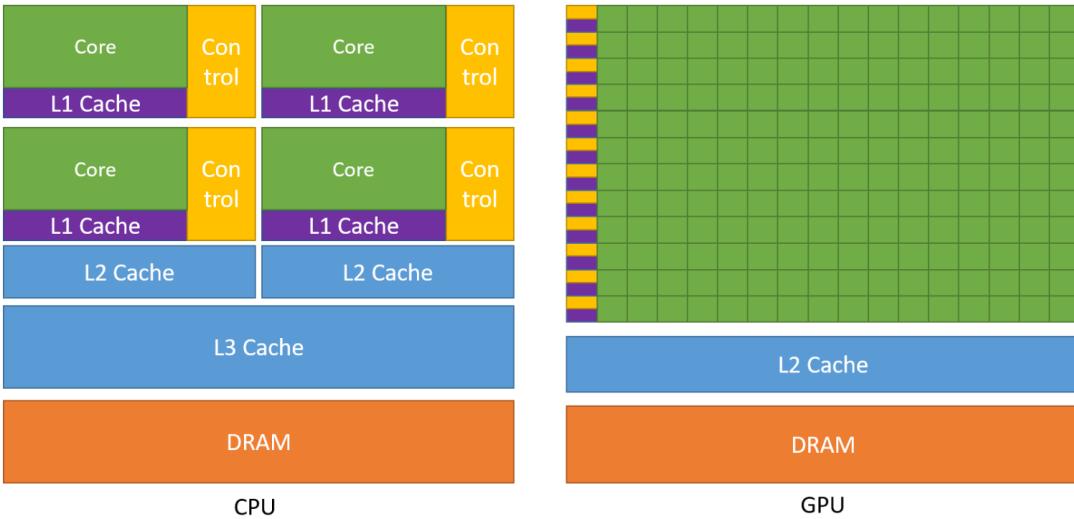


图 27: CPU 和 GPU 的架构差异

一般的应用程序会包含并行和串行部分，并行部分用 GPU 实现能得到更高的性能。
2006 年，NVIDIA 发布了 cuda，GPU 的一个接口，API。允许使用 C++ 作为编程语言。

- 并行编程的难点在于如何把程序并行化设计，利用 GPU 的多核加快运算
- cuda 编程使用 c 或 c++ 这种程序员熟悉的语言，减少开发成本
- cuda 的核心抽线: a hierarchy of thread groups, shared memories, barrier synchronization

对于程序员来说最重要的就是抽象，对问题分而治之。

6.2 Programming Model

主要讨论编程模型。

6.2.1 kernel

- **Kernel:** 内核，cuda 中定义的 C++ 函数称为内核 kernel，和一般的函数不一样，他会在 N 个机器上执行 N 次，一般的 C++ 函数只会执行一次。

使用关键字 `__global__` 指定，调用的时候需要指定要多少个线程去执行他。

```
--global__ void VecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}

int main()
{
    ...
    // Kernel invocation with N threads
    VecAdd<<<1, N>>>(A, B, C);
    ...
}
```

global 定义了 kernel，把向量相加并存储在 C 中。调用的时候应该是 N 个线程执行 1 次，我猜的。每一个线程恰好对应一个位置相加。

6.2.2 Thread Hierarchy

- **Thread Hierarchy:** 就是线程索引的方式

threadIdx 是一个三维向量。cuda 中先有一个 grid，grid 中会有很多个 block，每一个 block 里面会有多个线程，block 的线程可以只是一维，也可以是二三维。这些线程的 id 就可以看成是对应元素计算的索引，比如向量，矩阵等。

1D block 线程索引就 block 内的索引。2D block 就和二维数组一样，如果 block 大小是 (DX, DY)，那么对于在 (x, y) 的线程，她的索引就是 $x + yDx$ ；3D block 就和三维数组一样，如果 block 大小是 (DX, DY, DZ)，那么对于在 (x, y, z) 的线程，她的索引 $x + yDx + zDyDz$ 。这里是先填满 xy，再到 z，xyz 三个坐标按顺序走 xyz。

```
// Kernel definition
__global__ void MatAdd(float A[N][N], float B[N][N],
                      float C[N][N])
{
    int i = threadIdx.x;
    int j = threadIdx.y;
    C[i][j] = A[i][j] + B[i][j];
}
```

```
int main()
{
    ...
    // Kernel invocation with one block of N * N * 1 threads
    int numBlocks = 1;
    dim3 threadsPerBlock(N, N);
    MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
    ...
}
```

用一个 block 来计算这个二维矩阵。block 是 $N \times N$ 的，刚刚好和矩阵是一样的。emmmmm
那么如果矩阵数量更大，是不是直接设定一个和矩阵一样的 block 就行？后面说了，因为
不同线程之间是共用一个处理器的内存，一个 block 最多是 1024 个线程。

grid 也有维度之分，可以是一维，二维，三维的 block

```
--global__ void MatAdd(float A[N][N], float B[N][N],
float C[N][N])
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    if (i < N && j < N)
        C[i][j] = A[i][j] + B[i][j];
}

int main()
{
    ...
    // Kernel invocation
    dim3 threadsPerBlock(16, 16);
    dim3 numBlocks(N / threadsPerBlock.x, N / threadsPerBlock.y);
    MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
    ...
}
```

类型可以是 int 或者 dim3，我还不太求出这个 dim3 是什么类型，但他们都是用 dim3，可
能表示就是 three dimension 吧。threadsPerBlock 表示当前的 block 就是 2 维， 16×16 。
numBlocks 表示 grid 里面 block 的数量，当分配好 block 和线程的数量，那么剩下就是如
何分配线程的 index 了。就是把局部的坐标放缩到全局坐标，每一个线程的所在位置和整

个矩阵的元素是一一对应的。

这些线程的计算并不一定就是并行的，会按照仍任何顺序进行计算，所以线程可以以任意的顺序调度线程。

块内的线程可以通过共享内存共享数据，并通过同步执行来协调内存访问。

```
--syncthreads()
```

同步线程的方法，在这个 block 中如果使用了这个函数，那么 block 里面的所有线程都要执行完才能进行下一步。

6.2.3 Thread Block Clusters

- **Thread Block Clusters:** made up of thread blocks.

也就是由 block 组成的有一单元，但是由 grid 统一管理。文章里说是为了调度提出的。cluster 和 block 的维度也是一样，可以是一二三维度。一个 cluster 最多 8 个 thread。如果使用了 cluster，那么 blockDim 表示的还是 block 的数量，Cluster Group 可以得到 cluster 中 block 的数量。

```
--global__ void __cluster_dims__(2, 1, 1) cluster_kernel(float *input, float*
    output)
{
}

int main()
{
    float *input, *output;
    // Kernel invocation with compile time cluster size
    dim3 threadsPerBlock(16, 16);
    dim3 numBlocks(N / threadsPerBlock.x, N / threadsPerBlock.y);

    // The grid dimension is not affected by cluster launch, and is still enumerated
    // using number of blocks.
    // The grid dimension must be a multiple of cluster size.
    cluster_kernel<<<numBlocks, threadsPerBlock>>>(input, output);
}
```

threadsPerBlock 表示每一个 block 里面是 16x16 的 thread，numBlocks 里面表示 grid 有多

少个 block, `__cluster_dims__` 表示 cluster 的数量, 这里表示就 2 个 block 一个 cluster。这里要注意的是这种情况下定义的 cluster 一定要固定写死, 要不然会报错。template 不知道行不行。

那能不能在 runtime 的时候再定义 size 呢? 也行, `cudaLaunchKernelEx`。

```
// Kernel definition
// No compile time attribute attached to the kernel
__global__ void cluster_kernel(float *input, float* output)
{

}

int main()
{
    float *input, *output;
    dim3 threadsPerBlock(16, 16);
    dim3 numBlocks(N / threadsPerBlock.x, N / threadsPerBlock.y);
    cluster_kernel<<<numBlocks, threadsPerBlock>>>();
    // Kernel invocation with runtime cluster size
    {

        cudaLaunchConfig_t config = {0};
        // The grid dimension is not affected by cluster launch, and is still enumerated
        // using number of blocks.
        // The grid dimension should be a multiple of cluster size.
        config.gridDim = numBlocks;
        config.blockDim = threadsPerBlock;

        cudaLaunchAttribute attribute[1];
        attribute[0].id = cudaLaunchAttributeClusterDimension;
        attribute[0].val.clusterDim.x = 2; // Cluster size in X-dimension
        attribute[0].val.clusterDim.y = 1;
        attribute[0].val.clusterDim.z = 1;
        config.attrs = attribute;
        config.numAttrs = 1;

        cudaLaunchKernelEx(&config, cluster_kernel, input, output);
    }
}
```

他这里和前面最大的区别是，把一个 kernel 变成一个用 cluster 去执行的 kernel。

6.2.4 Memory Hierarchy

- **Memory Hierarchy**

每一个 thread 有自己的 memory，每一个 thread block 都会有 shared Memory，global memory 所有的线程都是可以访问。

- local Memory: 每一个 thread 都有，只有自己的 thread 能访问
- shared Memory: 每一个 Block 有一个，再一个 block 内的 thread 都能访问
- global Memory: 所有 thread 都能访问，无论是哪一个

如图 28所示，他的内存是金字塔形状的

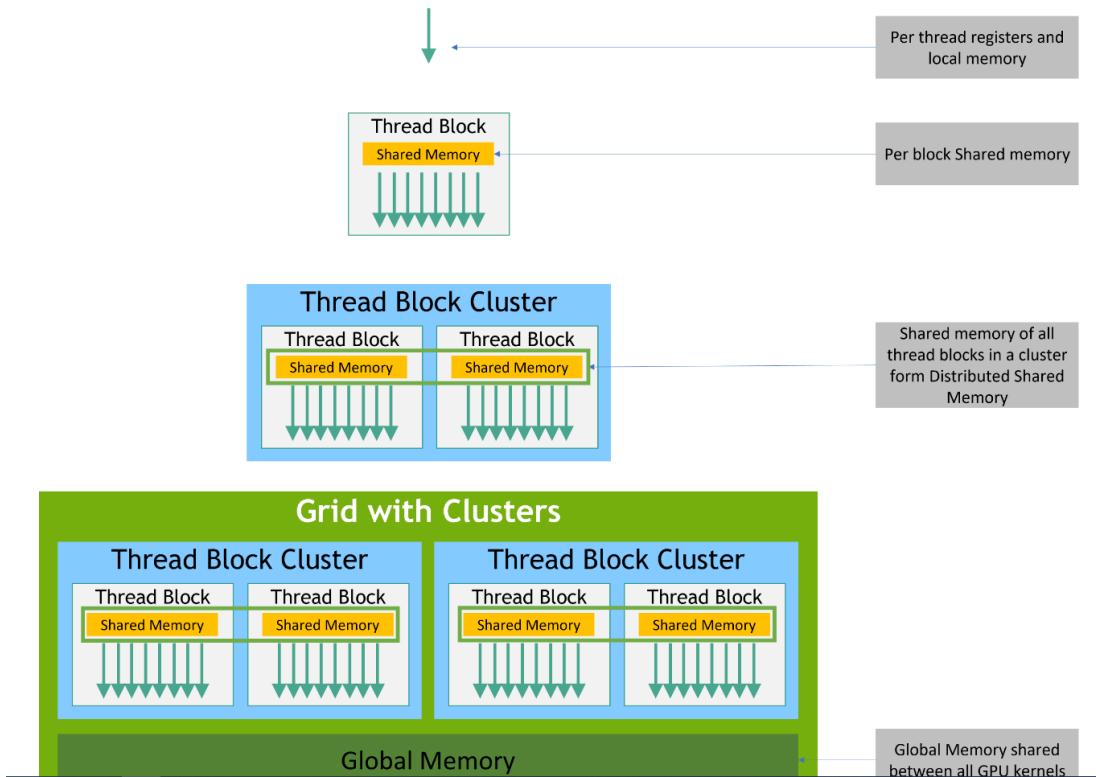


图 28: CUDA 的内存

6.2.5 Heterogeneous Programming

- Heterogeneous Programming

两条假设：

- the CUDA programming model assumes that the CUDA threads execute on a physically separate device that operates as a coprocessor to the host running the C++ program.
- The CUDA programming model also assumes that both the host and the device maintain their own separate memory spaces in DRAM, referred to as **host memory and device memory, respectively**.

第一个假设意思是 C++ 程序中的主程序和线程程序是分开不同设备执行，主程序一般是在 CPU，而 thread 可以看成是主程序的协程在 GPU 执行，他们之间的内存不相通。第二个假设的意思是主程序和协程之间的内存不互通。host 指的就是主程序。如图 29所示。

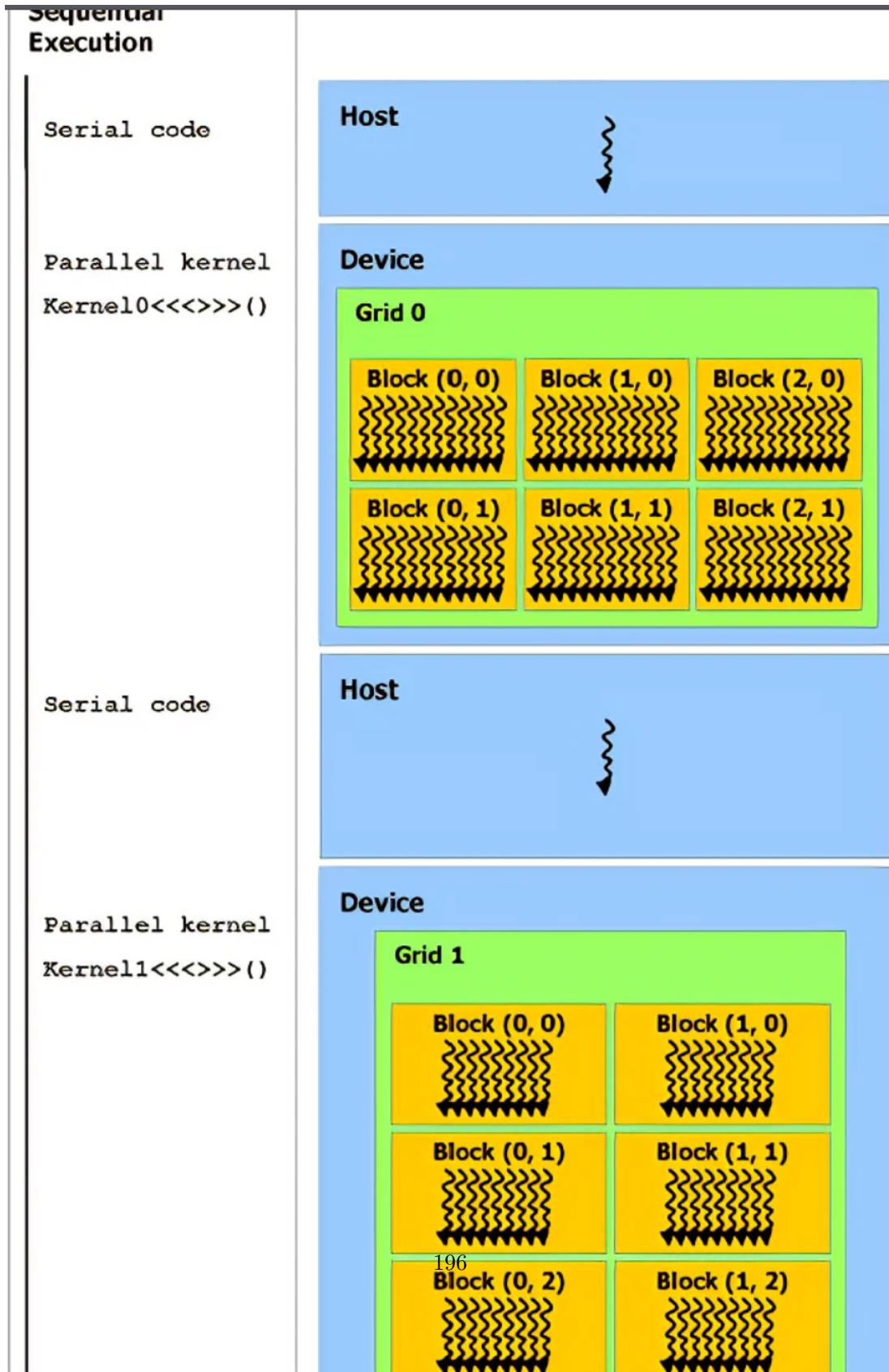


图 29: CUDA 的主程序和协程

6.2.6 Asynchronous SIMT Programming Model

- **Asynchronous SIMT Programming Model**

提到了 CUDA 的异步操作，我感觉这个对于程序员的要求比较高。Asynchronous Operations 没看太懂这部分，他的意思应该是这个主线程和 Thread 的异步。一般来说这个 host 是要等待 thread 执行完毕，异步可以使得分开执行。

6.2.7 Compute Capability

- **Compute Capability**

主要就是 cuda 的版本号了，第一个数字是什么架构，第二个数字表示版本。cuda 向下兼容，往高了装就行。

6.3 Programming Interface

程序由 C++ 扩展和一系列 runtime library 组成。C++ 扩展需要 nvcc 编译才能使用，runtime 的主要功能

- 分配回收内存
- 迁移数据
- 在不同设备之间管理系统
- etc

编译代码，kernel 可以由 PTX 和 C++ 来写，PTX 就是刚刚那堆 global，可以由 C++ 调用。

6.3.1 Compilation Workflow

- **Compilation Workflow**

offline 的编译流程

- 把 host code 和 device code 分开
- 把 host code 中 «<»> 替换成 runtime 中定义的对象。这里本身填的就是线程的一些参数。

- 修改后的 host code 要么输出为 c++ 代码，由另一个工具编译，要么直接输出为目标代码，让 nvcc 在最后的编译阶段调用 host 的编译器。

他这里还提了一个 Just-in-Time Compilation，但这两种编译方式结果是不变的。

后面这部分的编译看的不是很懂，不再记录。

6.4 Hardware Implementation

多处理器（SM）以 32 个并行线程组为一个 warp，以 warp 为单位进行创建，管理，调度和执行线程。如果 warp 中的线程执行遇到了分支指令，cuda 会尽可能的执行所有分支中的指令，将那些不符合分支条件的指令标记无效。

多处理器的结构是 SIMT 结构，同时还有一个 Hardware Multithreading 的技术。SIMT 总结来说就是一个指令应用到多个线程上提高并行能力，使用相同的指令流，但是在不同数据上执行相同程序。Hardware Multithreading 就是通过交替执行实现并行计算。当一个 kernel 被执行的时候，grid 中的线程块分配到多处理器上。SM 是由很多个 cuda 组成，一个 core 只能执行一个 thread，warp 就是 cuda core 的一个子集，也就是 SM 里面的 core 会自己分成很多个 warp 的组合，一个 warp 包含 32 个 thread，也就是 warp 一次并发 32 个线程。一个 warp 的线程必定都属于同一个 block。

6.5 Performance Guidelines

效率指导，就是告诉你应该如何编程使得程序更高效。

- Maximize parallel execution to achieve maximum utilization，并行数尽可能大
- Optimize memory usage to achieve maximum memory throughput，内存吞吐量尽可能大，也就是说每一条 thread 执行的命令尽可能简单
- Optimize instruction usage to achieve maximum instruction throughput，指令的吞吐量大，指令尽可能简单
- Minimize memory thrashing，内存置换尽可能少

6.5.1 Maximize Utilization

Maximize Utilization，在高层次上，应用程序应该通过使用异步函数调用和异步并发执行中描述的流来最大化主机、设备以及连接主机和设备的总线之间的并行执行，这句话我直接翻译的。有三个层次上的设计

- Application Level: 设计准则，serial workloads to the host; parallel workloads to the devices. 串行工作尽可能在 CPU 上，并行工作在 GPU 上。
 - 不能并行的原因可能是需要数据同步，文章举了两个例子，如果是两个线程是在相同的 block，那么可以使用 `__syncthreads()`，那么可以通过 shared memory 在 block 内交互内存，因为每一个 threads 有自己的 local memory，需要进入一个共享内存才能进行交互。如果是不同 block，他这里说 kernel invocation，就是不同的 kernel 调用，kernel 调用需要先定义 block 和 threads，那肯定是不同的 block，那就只能去 global memory 进行交换了。
- Device Level: kernel 足够多，调用的处理器足够多也能提高利用率。
- Multiprocessor Level: 处理器级别应该是最底层的了。之前 Hardware Multithreading 没仔细看，不知道这个 resident warps 是啥。warps 是指一组大小固定的线程，resident warps 是指一组已经被指定到处理器的线程，所以并行的效率和 resident warps 的数量息息相关。
 - 在每一条指令发出时，warp scheduler 会选择一条准备执行的指令，可以是同一个线程的指令也可以是不同线程的指令。**Warp scheduler** 准备执行下一条指令所需时间称为延迟 (**latency**)，如果所有 warp 在延迟时间都有指令要发出，那么此时几乎没有延迟了，这个时候并行效率最高。就像交通灯一样，如果刚刚好到达这个路口就能绿灯，那效率自然是最高的。
 - 如果操作数是在寄存器上，那么这个时间是很快的，他延迟就是上一条指令执行的时间。
 - 如果操作数是在内存上，那么这个时间会很大，特别如果是在 CPU 上，IO 时间是很长的。
 - 如果存在某些情况，warp 并没有立刻执行下一条指令的原因可能是存在内存分配问题和线程同步命令。
 - kernel 调用寄存器的数量对于 resident warps 的影响很大，并且线程选择应该是 warp 的倍数，warp 一般是 32 个线程。

总的来说就是尽量使用寄存器吧。

后面提到了如何根据寄存器选择 block 和 thread 的数量，也就是说这个 block 和 thread 既要满足计算需要，也要满足寄存器的数量。以下有几个重要函数

- `cudaOccupancyMaxActiveBlocksPerMultiprocessor`: 根据 block size 和 shared memory usage 预测处理器目前并发 block 的数量。也就是处理器一共有多少个正在计算

的 block，乘上每一个 Block 需要是 warp 数量，就是处理器中正在并发 warp 的数量，除以所有的 warp 就是 warp 的占比了。

- cudaOccupancyMaxPotentialBlockSize 和 cudaOccupancyMaxPotentialBlockSize-VariableSMem：应该是计算最多可利用的 block
- cudaOccupancyMaxActiveClusters：计算 cluster 的数量

看一个计算利用率的例子

```
int main()
{
    int numBlocks;      // Occupancy in terms of active blocks
    int blockSize = 32;

    // These variables are used to convert occupancy to warps
    int device = 0;
    cudaDeviceProp prop;
    int activeWarps;
    int maxWarps;

    cudaGetDevice(&device);
    cudaGetDeviceProperties(&prop, device);

    cudaOccupancyMaxActiveBlocksPerMultiprocessor(
        &numBlocks,
        MyKernel,
        blockSize,
        0);

    activeWarps = numBlocks * blockSize / prop.warpSize;
    maxWarps = prop.maxThreadsPerMultiProcessor / prop.warpSize;

    std::cout << "Occupancy: " << (double)activeWarps / maxWarps * 100 << "%" <<
        std::endl;

    return 0;
}
```

device 表示 cuda 的编号，cudaGetDevice 表示获得 cuda 的编号。cudaGetDeviceProperties

获得当前 cuda 的熟悉。cudaOccupancyMaxActiveBlocksPerMultiprocessor 计算当前这个 cuda 正在运行的 block, numBlocks 乘上 blocksize 表示当前线程有多少, 除以 warpssize 计算出 warp 有多少个。然后除以 maxwarp 计算 warp 的利用率。

这段代码给出当前通过计算空闲 block 得出当前适合 kernel 的最佳 blocksize 和 gridSize, 然后计算

```
int launchMyKernel(int *array, int arrayCount)
{
    int blockSize; // The launch configurator returned block size
    int minGridSize; // The minimum grid size needed to achieve the
                     // maximum occupancy for a full device
                     // launch
    int gridSize; // The actual grid size needed, based on input
                  // size

    cudaOccupancyMaxPotentialBlockSize(
        &minGridSize,
        &blockSize,
        (void*)MyKernel,
        0,
        arrayCount);

    // Round up according to array size
    gridSize = (arrayCount + blockSize - 1) / blockSize;

    MyKernel<<<gridSize, blockSize>>>(array, arrayCount);
    cudaDeviceSynchronize();

    // If interested, the occupancy can be calculated with
    // cudaOccupancyMaxActiveBlocksPerMultiprocessor

    return 0;
}
```

cudaOccupancyMaxPotentialBlockSize 表示根据 Mykernel 内核计算最佳的 blocksize 是多少。第一个和第二个参数是计算出的数值。第三个是函数指针。所以你直接定义

```
void (*funcP)(int*, int) = MyKernel;
```

把 *funcP 传进去也行。

后面那个程序也差不多，不过改成了 runtime 赋值即可。

6.5.2 Maximize Memory Throughput

Maximize Memory Throughput, 最大化应用程序的整体内存吞吐量的第一步是最小化低带宽的数据传输。意思就是应该尽可能的减少 CPU 和 GPU 之间的传输，因为 host 通常是在 cpu 上运行，thread 是在 gpu 上运行，IO 的时间通常很长。减少 global memory 和 device 之间的信息传输本质上就是增加 on-chip memory 的利用率，on-chip memory 通常指集成芯片的内存，类似于缓存吧，速度更快。也就是 shared memory 和 caches，shared memory 是指 block 共享的内存。

1. Shared memory: 共享内存相当于用户管理的缓存，shared memory 是用户可以管理，并且可管理的。具体过程
 - (a) 首先数据会从 gpu 的内存流向 shared Memory
 - (b) 同步 block 中的所有线程
 - (c) 处理 shared memory 中的数据
 - (d) 同步 block 中的所有线程
 - (e) 把结果写回 device memory

这里提到了一个内存传输的高效技术：Mapped Memory，应该和 OS 里面的一样，直接把内存交换过去。

Device Memory Accesses, 这部分内容可能重要一些，gpu 内存的访问。内存分类

- global memory: 是指 GPU 中申请，并且所有线程都能访问的一块内存
- device memory: 是指 GPU 中所有的内存，包括了 global, shared, constant, texture, local 所有内存
- shared memory: Block 内线程共享的内存
- constant memory: GPU 中常量存储的内存，一般用 `__constant__` 修饰
- texture memory: 纹理内存，通常永固存储图像和其他类型的网格数据。内存中使用了四叉树数据结构优化。
- local memory: 每一个线程独有的内存，互相不共享

global memory, 当 warp 执行一条指令需要访问 global memory 的时候, 会先把每一个线程所需要的内存联合成一个或者多个事务, 然后一起执行。所以理论上讲, 内存事务需要越多, 浪费的内存就越多。所以为了尽可能的提高内存的使用效率, 有以下几种优化方法

- 遵循一些给定架构的优化方案
- 使用对齐的数据类型
- 对二维数据做一些 padding

对于第二种 Size and Alignment Requirement, 对于内存的吞吐, 只有 1, 2, 4, 8, 16 字节的字, 并且会自动对齐。如果不满足对齐, 程序会交错访问。

```
struct __align__(8) {
    float x;
    float y;
};
```

可以通过强制指定对齐, 这个技术 C 语言也是这样做的。这里还提到一个注意事项

Reading non-naturally aligned 8-byte or 16-byte words produces incorrect results (off by a few words), so special care must be taken to maintain alignment of the starting address of any value or array of values of these types.

对于第三种 Two-Dimensional Arrays, 意思就是访问线程块的数量和数组元素数量都要是 warp 的倍数才行。

Local Memory, 在 local memory 的变量只有三种可能

- 不能确定一个数组是否可以用常数索引
- 可能占用很多寄存器的结构
- 寄存器溢出

Shared memory, 共享内存也是 on-chip memory, 他的速度比一般的内存要快很多。为了实现更高的 bandwidth, 我觉得应该是并发量的意思。shared memory 会分成多个 bank, 这些 bank 大小相同, 可以同时访问。所以如果刚刚好 n 个线程访问的位置就是不同的 bank, 那么一次就可以完成读取。否则就需要串行读取。

这种优化我觉得相对来说比较困难。

Constant Memory 和 **texture Memory** 没看太懂是干什么用的。

6.5.3 Maximize Instruction Throughput

最大指令吞吐量。最大化指令吞吐量主要有三点

- 尽量减少使用低吞吐量的指令。比如使用内部函数而不是定义的函数，单精度而不是多精度数
- 减少 warp 内的分支
- 减少指令的数量

算术指令，单精度一般是 32 位表示，23 位表示尾数，8 位表示指数，一位符号。双精度是 64 位。

- 单精度范围正负 10^{38} ，小数精度为 6-7 位
- 双精度范围正负 10^{308} ，小数精度位 15-16 位

一般单精度其实就够。

- Single-Precision Floating-Point Division: 单精度除法，`__fdividef(x, y)` 单精度除法
- Single-Precision Floating-Point Reciprocal Square Root: `1.0/sqrt()` 被编译器优化为 `rsqrt()`，直接调用 `rsqrt()`
- Sine and Cosine: `sin` 三角函数运算成本较高。
- Integer Arithmetic: 整数除法/和模的运算% 成本比较高，可以用位运算替代。`i / n` 等于 `i >> log2(n)`, `i % n` 等于 `i & (n - 1)`
- Half Precision Arithmetic: 16 位精度加法
- Type Conversion: 转换成精度更低的类型

上面提到的这些其实就是把不必要的运算精度都缩减了，使得运算加快，把除法取模运算都用位运算替代。

Control Flow Instructions，任意的流控制指令都会影响指令的处理效率。如果 warp 存在不同 Thread 有不同分支，那么 warp 就需要对每一个线程去尝试所有分支，这个过程是线性的，自然就慢了。

Synchronization Instruction，这个应该是让你少点用同步线程控制。

Minimize Memory thrashing，通常在分配内存和回收内存的过程中会发现，内存调配通常会不断变慢。NVIDIA 的建议

- 不要用 cudaMalloc / cudaMallocHost / cuMemCreate 分配内存，这种方式强制内存空闲，阻止其他程序使用
- 在运行早期分配内存，减少 cudaMalloc 和 cudaFree 的次数

6.6 C++ Language Extensions

终于到核心编程了。

6.6.1 Function Execution Space Specifiers

- `__global__`: 在 device 上执行，也就是在 gpu 上执行，在 host 上调用，在主程序上调用，如果是 cuda 版本高于 5 是可以在 device 上调用
 - 返回类型必须是 void，所有他计算的结果，返回值都得引用或者别的方式获得
 - 调用的时候必须指定他的 execution configuration，也就是线程，block 这些
 - 异步调用，调用 global 函数后，主程序会继续运行
- `__device__`: 只能在 device 上执行，也只能在 device 上调用
- `__host__`: 只能在 host 上执行，在 host 上调用

`__noinline__` and `__forceinline__`

- `__noinline__`: 防止编译器对函数进行内联展开，即强制编译器不对该函数进行优化，使函数调用可被跟踪或调试。如果对改函数的执行有问题，可以用这种方式跟踪函数的执行。
- `__forceinline__`: 说明该函数是 inline。

这两个修饰我还不确定他和 C++ 里面的是不是一样的，inline 在 C++ 里面就是直接把调用 inline 函数的位置复制粘贴整个函数进去，防止过多的栈调用。这里我不确定是不是这个意思。

6.7 Variable Memory Space Specifiers

`__device__`, `__shared__` and `__constant__` 分别指定这个变量所在的位置。
如果不指定创建在哪个位置，默认

- 定义在 global memory

- 他的生命周期都会在 cuda 里面度过
- 不同设备会有不同的对象
- 可以被所有线程访问，因为他是 global memory 里面的
- `__device__`: 后面的只能有一个和 device 一起用，因为 device memory 包含了他们
- `__shared__`: 创建在某一个 block 的内存里面，只有 block 里面的线程才可以访问
- `__constant__`: 在 constant memory 里面，所有在 grid 的线程都可以访问

补充一下 shared，文章特意提了一嘴

All variables declared in this fashion, start at the same address in memory, so that the layout of the variables in the array must be explicitly managed through offsets.

意思就是这个 shared memory 的变量无论怎么创建，都是从相同的内存地址开始。我们想获得如下数组

```
short array0[128];
float array1[64];
int array2[256];
```

只能根据 index 获取

```
extern __shared__ float array[];
__device__ void func() // __device__ or __global__ function
{
    short* array0 = (short*)array;
    float* array1 = (float*)&array0[128];
    int* array2 = (int*)&array1[64];
}
```

首先 array0 指向就是第一个地址，第一个地址往后的 128x2 个字节都是 array0。array1 指向 256 个字节之后的一个，以此类推，也就是说他这个存储是连在一起的。字节一定要对齐，不对齐不行。

`__restrict__` 告诉编译器该指针所指向的内存区域在生命周期中不被任何其他指针访问，也就是它是唯一拥有访问该内存区域权利的指针。一般是用于最后优化。比如

```
void foo(const float* a,
         const float* b,
         float* c)
{
    c[0] = a[0] * b[0];
    c[1] = a[0] * b[0];
    c[2] = a[0] * b[0] * a[1];
    c[3] = a[0] * a[1];
    c[4] = a[0] * b[0];
    c[5] = b[0];
    ...
}
```

在传递参数的时候可能会出现 a 和 c 是指向相同的地址，所有这个时候计算就可能出现问题。

```
void foo(const float* __restrict__ a,
         const float* __restrict__ b,
         float* __restrict__ c);
```

这样就不会出现问题。但这一般是在优化的时候才需要使用。

6.7.1 Built-in Vector Types

深度学习常用 tensor，自然要定义相关的变量了。make_ 函数构建向量，比如

```
int2 make_int2(int x, int y);
```

一共是可以有 1234 长度的向量。

- dim3: 这是基于 uint3 的向量，make_uint3 函数创建。没有指定的位置都初始化为 1，这个类型一开始定义 block 的时候用过了。
- Built-in Variables: 这类变量是线程，block 和 grid 自带的，用于查看 block 等的属性
 - blockDim, 这个变量是 grid 的各个参数，是 dim3 类型的变量
 - blockIdx, 类型是 uint3, 表示当前 block 的索引
 - blockDim, 表示当前 block 的维度是多少，xyz 的 scale 是多少

- threadIdx, 表示当前 thread 的索引
- warpSize, 一个 warp 的大小, 一般是 32

6.7.2 Memory Fence Functions

CUDA 是弱顺序模型, 并不是严格按照执行顺序访问内存的, 这个函数就是帮助程序使得线程能按照一定顺序访问内存。这里两个线程, 一个线程写, 一个线程读

```
__device__ int X = 1, Y = 2;

__device__ void writeXY()
{
    X = 10;
    Y = 20;
}

__device__ void readXY()
{
    int B = Y;
    int A = X;
}
```

因为线程的运行不具备顺序性, 所以 XY 的值会出现任何情况。Memory Fence Functions 就可以强迫这些线程按照一定顺序进行。Memory Fence Functions 可以用于强制指定内存访问的顺序, 防止出现随机情况。但是 Memory Fence Functions 只是确定读写顺序, 具体读什么内存的顺序他不关心, 对于所有的内存都可以使用。

- `__threadfence_block()`: 在 block 中的线程如果需要写, 在调用 `threadfence block` 之前的函数都执行完, 再执行调用之后的函数. 读也一样。
- `__threadfence`: 上面那个带 block 的函数是一个 block 的函数, 而不是所有的函数。内存的写入是先到缓存, 缓存再到内存, 在调用了 `threadfence` 之后的写入操作不会呗其他线程观测到, 没看明白。
- `__threadfence_system`: 确保在调用 `__threadfence_system` 之前的所有线程写操作都能被 device 上线程观测到, 也就是在 `__threadfence_system` 调用之后这个写入会立刻从缓存写到内存。

这个 threadfence 看的真的没看太懂，自己上手做了一下，觉得他应该解决的是读写平衡的问题，因为内存是先写到缓存，再写到内存，threadfence 的意思就是到 threadfence 这等一下，让上面的内容全部写到内存开始读。

6.7.3 Synchronization Functions

- `__syncthreads`: waits until all threads in the thread block have reached this point and all global and shared memory accesses made by these threads prior to `__syncthreads()` are visible to all threads in the block. 在 syncthreads 之前所以 thread block 对于 global 和 shared memory 的访问都要完成，也就是都要写进内存让其他线程看见。
 - 注意 syncthreads 的对象是对于相同 block 的线程。
 - 特意提到了 conditional 语句，在一个 Block 中所有的条件都相同才好，要不然会有问题
- `__syncthreads_count`: 效果和 syncthreads 相同，但是他需要返回一个满足条件的 Thread 的数量，满足啥条件没看明白。
- `__syncthreads_and`: 这个也是一个增加统计额外的数值
- `__syncthreads_or`: 这个也是一个增加统计额外的数值
- `__syncwarp`: 有一个参数 mask，等待 warp 中带有 mask 的线程执行完毕

这几节的同步我都没看太懂。

6.7.4 Texture Object API

可以说这节都不知道干嘛用的，texture object 是啥都不知道。

6.7.5 Surface Object API

同上

6.7.6 Read-Only Data Cache Load Function

只支持 5.0 以上的版本。主要就一个函数 `__ldg`。

`ldg` 函数，看起来好像是 Load global device 的缩写，主要就是在全局内存中读取数据，将全局内存中的数据丢入到缓存。

```
T __ldg(const T* ptr)
```

T 表示读取的数据类型，ptr 表示数据地址，另外 ldg 只能读取全局的内存中的单个数据，不能读数组。

6.7.7 Load Functions Using Cache Hints

也是加载数据的函数，和 ldg 的区别就是 ldg 是不使用缓存读取数据，速度稍微慢一点，ldcg 是使用缓存读取。

- ldca 是读取数组数据，将数据送到缓存中
- ldcs 是读取标量
- ldlu 是读取本地内存
- ldcv 读取纹理数据

6.7.8 Store Functions Using Cache Hints

反过来，存储。参数分别是数据和存储的地址。

6.7.9 Time Function

```
clock_t clock();
long long int clock64();
```

6.7.10 Atomic Functions

主要职能是在 global memory 或者 shared memory 中修改 32 和 64 位数据，也就是先读，再改，后写。

- atomicAdd(): 先从内存中读取数据，然后加上一个数字，再写回。

例子：

```
--global__ void mykernel(int *addr) {
    atomicAdd_system(addr, 10); // only available on devices with compute capability 6.x
}
```

```
void foo() {
    int *addr;
    cudaMallocManaged(&addr, 4);
    *addr = 0;

    mykernel<<<...>>>(addr);
    __sync_fetch_and_add(addr, 10); // CPU atomic operation
}
```

这两个分别是 GPU 和 CPU 的自动加法，分别是在不同内存进行，就算 addr 相同，他们也不是相同的地址，一个是在 GPU，一个是在 CPU。

后面是一些具体的 API，用到再看。

6.7.11 Address Space Predicate Functions

类似于 python 那些 is_cuda... 那些函数，判断数据是再哪里。

- isGlobal: 判断是否是在 global memory 上
- isShared: 判断是否是在 shared memory 上
- isConstant: 判断是否是在 constant memory 上
- isGridConstant: 是否被初始化
- isLocal: 判断是否是在 local memory 上

6.7.12 Address Space Conversion Functions

内存转移，不同内存间的数据转移。

6.7.13 Alloca Function

```
--host__ __device__ void * alloca(size_t size);
```

在 stack frame 分配空间，原话是说在 stack frame 分配空间，并不是 global memory。当调用者返回，那么内存自动回收。和 C++ 里面的好像不一样。

6.7.14 Compiler Optimization Hint Functions

这部分是编译器优化，先放一下。

6.7.15 Warp Vote Functions

意思应该是：输入一个 int predicate，然后 warp 的每一个线程和他进行对比，对比完之后其中有一个 active thread 将这些返回值进行整合返回。其实就是投票了，不过确认最后返回值的策略有不同。

- __all_sync: 所有线程都返回 true，就返回 true
- __any_sync: 有其中一个线程返回 true，就返回 true

6.7.16 Warp Match Functions

判断是否存在某个特定的值。

6.7.17 Warp Reduce Functions

没看懂主要有啥用，就是一个统计作用吧感觉。

6.7.18 Warp Shuffle Functions

主要是 warp 线程内的数据交换。

案例

- Broadcast of a single value across a warp

```
#include <stdio.h>

__global__ void bcast(int arg) {
    int laneId = threadIdx.x & 0x1f;
    int value;
    if (laneId == 0)      // Note unused variable for
        value = arg;      // all threads except lane 0
    value = __shfl_sync(0xffffffff, value, 0); // Synchronize all threads in warp, and
                                                // get "value" from lane 0
    if (value != arg)
        printf("Thread %d failed.\n", threadIdx.x);
}

int main() {
    bcast<<< 1, 32 >>>(1234);
```

```
    cudaDeviceSynchronize();  
  
    return 0;  
}
```

0x1f 是 00011111，取当前线程编号的后五位。一个 warp 包含 32 个线程，laneId 就是得到当前线程在 warp 的位置。因为一个 warp 中的线程只能处理同一个 block 中的线程。如果一个 block 的线程有 35 个，那么他会用两个 warp 处理，一个 warp 处理 32，另一个处理 3 个 thread。当然这仅限于一维的 block。

lanID==0，把第一个变量单拎出来，`__shfl_sync` 是将 warp 中的线程的值进行同步到其他线程上。0xffffffff 就是 mask 参数，warp 多少个线程就多少位，这里 32 位全是 1，说明就全部参加置换。

```
value = __shfl_sync(0xffffffff, value, 0)
```

把 warp 中 index 为 0 的 thread 的 value 的值广播给其他变量。（value 不是一个 Int 吗？为什么能表示一个变量。）

`cudaDeviceSynchronize` 是确保 GPU 的工作都做完了，在 `cudaDeviceSynchronize` 之前会把所有 GPU 工作干完了，要不然 `return` 的时候 GPU 还没做完。

6.7.19 Nanosleep Function

sleep 函数吧

6.7.20 Warp Matrix Functions

用于解决 $A \cdot B + C$ 的矩阵求解问题

6.7.21 Asynchronous Barrier

这里主要提了个 sync，感觉和前面的那几个同步差不多。

后面基本都是 API，说实话我看完也记不住。

6.7.22 Dynamic Global Memory Allocation and Operations

global 中的内存分配。

```
--host__ __device__ void* malloc(size_t size);  
__device__ void *__nv_aligned_device_malloc(size_t size, size_t align);
```

```
--host__ __device__ void free(void* ptr);
```

global memory 一开始会分配一块固定大小的堆，上面的三个函数就是去申请，释放的。

6.7.23 Execution Configuration

配置线程的参数。«< Dg, Db, Ns, S »», 都是 dim3 的类型，grid 的大小，block 的大小，每一个 Block 分配多少的 shared memory，默认是 0。s 没看懂是什么。

6.8 Cooperative Groups

其实就是线程之间的通信和协作。虽然 cuda 已经提供了一个单一，简单的线程块内协作的线程间的同步函数，但是 cuda 程序员通常需要定义线程块内的配置。Cooperative Groups 提供了跨线程块的同步模式。

Cooperative Groups 的编程模型由以下元素组成

- 协作线程组的数据类型
- 获取由 CUDA Launch API 定义的隐式线程组的操作
- 用于将现有组划分为新组的集体操作
- 数据移动和修改的集体算法
- 同步组内线程操作
- 检查线程组属性的操作
- **Cooperative Groups Fundamentals**

首先要引入头文件

```
#include <cooperative_groups.h>
```

基本类型是 Thread Groups，可以看成是一组线程的句柄，只能有所拥有的组内成员访问。

- **Thread Group Collective Operations**

所有线程之间执行集体操作，集体操作是需要再一组指定的线程之间进行同步或者通信操作。获得线程大小 size() 方法，获得组内的调用线程 thread_rank()，索引就是 0 到 size - 1 之间。

```
using namespace cooperative_groups;
__device__ int reduce_sum(thread_group g, int *temp, int val)
{
    int lane = g.thread_rank();

    // Each iteration halves the number of active threads
    // Each thread adds its partial sum[i] to sum[lane+i]
    for (int i = g.size() / 2; i > 0; i /= 2)
    {
        temp[lane] = val;
        g.sync(); // wait for all threads to store
        if(lane<i) val += temp[lane + i];
        g.sync(); // wait for all threads to load
    }
    return val; // note: only thread 0 will return full sum
}
```

这是归约求和。thread_group g 表示 g 这个线程组。temp 是一个临时缓存，用来存储每一个线程的值，val 是最后计算的值。

```
int lane = g.thread_rank();
```

得到当前执行的 thread id 是几。

- 每一个线程都会经过这个 for 循环，每一次循环，active 的线程都会减半。
- 假设 8 个线程，那么一开始 $i = 4$ ，此时 8 个线程都会独立的走这个函数，一共只有 0, 1, 2, 3 这四个线程跑，他们会把 4, 5, 6, 7 这四个线程的值都加到前面 0, 1, 2, 3 上去，所以后面这四个线程就没用了。
- 第二次循环的时候就只加 0, 1，把 2, 3 加到 0, 1 上，以此类推。

所以其实这里面有一半的线程就存储了以下数值，没参与什么计算。

- **create thread block**

线程块是 cuda 并行的基本单元，warp 是 cuda 执行的基本单元。

```
thread_block block = this_thread_block();
```

可以得到一个线程组的句柄。对于一组 group 来说

```
dim3 group_index();
dim3 thread_index();
```

组的 index，组内的 index，相当于 grid 和 block 的 blockIdx 和 threadIdx。

6.8.1 Partitioning Groups

tiled_partition() 将线程块划分成多个组。

```
thread_group tile32 = cg::partition(this_thread_block(), 32);
```

32 个线程为一组，返回一个句柄。

```
thread_group tile4 = tiled_partition(tile32, 32);
```

tiled_partition 返回的就是线程组。

6.8.2 Modularity

模块化是 Cooperative Groups 的核心。将线程组当作显示参数传递当参数的时候，就会出现模块化。

```
--device__ int sum(int *x, int n)
{
    ...
    __syncthreads();
    ...
    return total;
}

--global__ void parallel_kernel(float *x, int n)
{
    if (threadIdx.x < blockDim.x / 2)
        sum(x, count); // error: half of threads in block skip
                        // __syncthreads() => deadlock
}
```

syncthreads 会等待所有线程到达既定位置，然后再进行下一步。但问题是并不是所有线程都运行了 sum，这就出现了死锁。

如果把线程组弄成一个显示代码，那么就不会有问题

```
--device__ int sum(thread_block block, int *x, int n)
{
    ...
    block.sync();
    ...
    return total;
}

__global__ void parallel_kernel(float *x, int n)
{
    sum(this_thread_block(), x, count); // no divergence around call
}
```

6.8.3 Optimizing for the GPU Warp Size

除此之外，还有一个静态分组，在编译阶段就完成

```
thread_block_tile<32> tile32 = tiled_partition<32>(this_thread_block());
thread_block_tile<4> tile4 = tiled_partition<4>(this_thread_block());
```

相比前面的 partition，这种是在编译阶段就直接定义好了，在编译的时候就能了解分组大小，提供更好的优化机会。

6.9 CUDA Dynamic Parallelism

动态并行变成是 cuda 编程的一种扩展。直接在 gpu 上创建数据可以减少主机和设备之间的 IO，加快速度，数据也可以在 gpu 中生成，不过 gpu 和 cpu 之间的类型有所差别。我感觉这个功能主要是使得程序能够在 gpu 就使用并行技术。

6.9.1 Execution Environment and Memory Model

首先是介绍一些术语

- grid: 网格是线程的集合
- Thread Block: 同一个处理器上执行的一组线程
- Kernel Function: 并行程序

- Host: 调用并行程序的环境，通常就是 cpu
- Parent: 父线程、线程块或网格是已启动新网格、子网格的一种。直到所有启动的子网格也完成后，父节点才被视为完成。
- Child: 由父网格启动的线程
- Thread Block Scope: block 的生命周期
- Device Runtime: 设备运行时

后面都是写高级特性，先放放。

写几个例子

6.9.2 CUDA 实现 dropout

dropout 核心就是神经元失活

```
self.mask = n.random.rand(*x.shape) > self.dropout_radio
```

如果当前的这个维度的概率是大于某个给定的值，就设置成是 1，否则是 0。

7 CUDA 总结

7.1 CUDA 基础

主要分成几个部分

- 异构并行计算
- CUDA 编程模型
- CUDA 执行模型
- 内存（重要）
- 流和并发
- 指令集原语
- GPU 加速库
- 多 GPU 编程

7.1.1 并行计算

并行计算涉及了两个领域

- 计算机架构

硬件的主要目的就是为软件提供更快的计算速度

- 并行程序设计

软件的主要目的就是为了尽可能压榨硬件的计算资源

传统的计算机分成三部分

- 内存
- CPU
- IO

如图 30 所示。

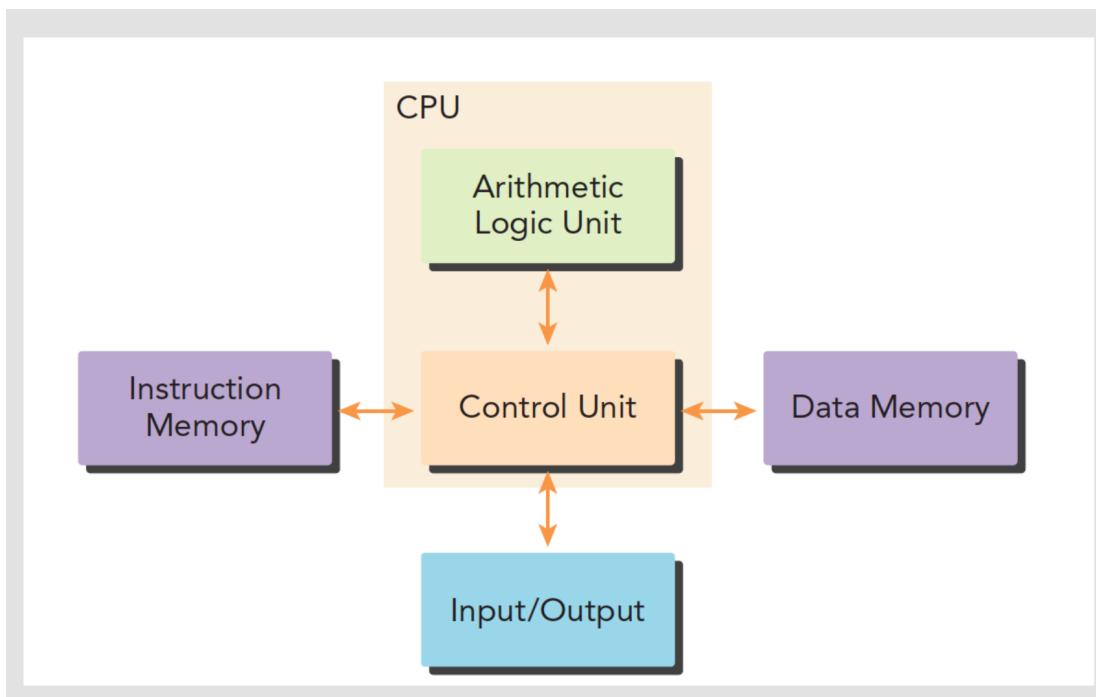


图 30: cpu 架构

7.1.2 并行性

并行有两种

- 指令并行
- 数据并行

cuda 的设计更关心数据并行。数据并行是第一步，把数据划分成小块或者是数据周期执行。比如把数据划分成 n 个小块，每一个块的执行顺序是随机的，每一次选择一个线程执行。周期执行则是指线程按照一定顺序周期执行，别 1-6 线程执行块 1-6，下一个时间线程 1-6 执行 7-12。数据划分是 cuda 并行里面的最重要的部分，针对不同的问题和不同计算机结构，我们要通过理论和试验共同来决定最终最优的数据划分。

7.1.3 计算机架构

- Flynn's Taxonomy

Flynn's Taxonomy 是一种划分不同计算机结构的方法，根据指令和数据进行划分

- SIMD: 单指令多数据
- MIMD: 多指令多数据
- SISD: 单指令单数据
- MISD: 多指令单数据

SIMD 就是用的比较多的并行架构，指令唯一，但是数据不同。

如何提高并行性？三个手段，在官方文档也有提到

- 降低延迟，主要是内存，减少 device 和 host 之间的 IO
- 提高带宽，官方文档提到了尽量把 block 的线程数量设置成和 warp 一样大
- 提高吞吐量，包括数据和指令
- 根据内存划分
- 分布式内存的多节点系统
- 共享内存的多处理器系统

第一个分布式，其实就是多个系统，缓存，内存，处理器都是独立的，如图 31所示。第二个就是多个处理器共享一个内存，但是缓存是独立的 32所示。

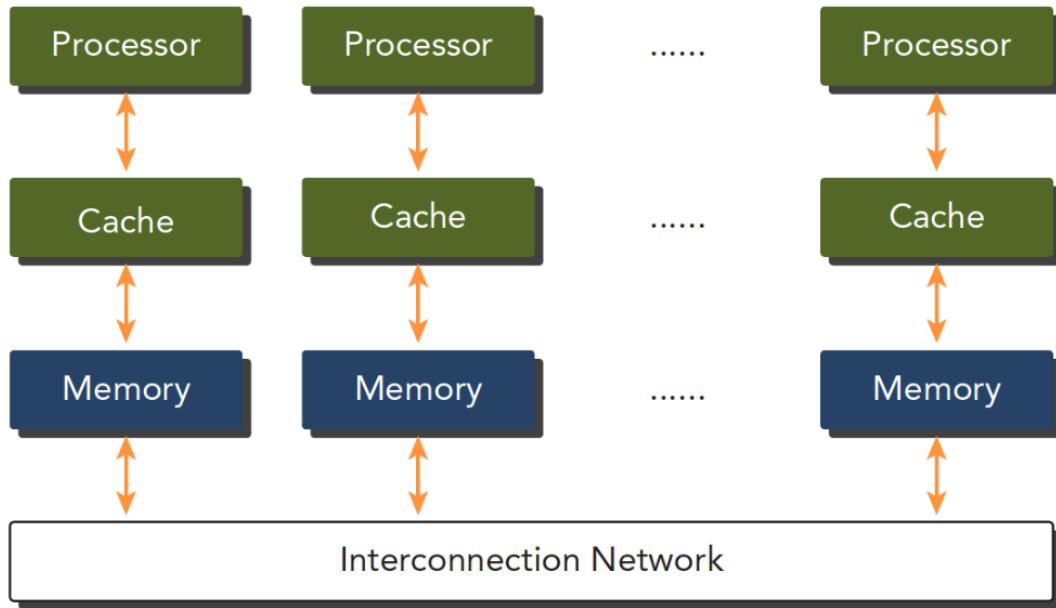


图 31: 分布式

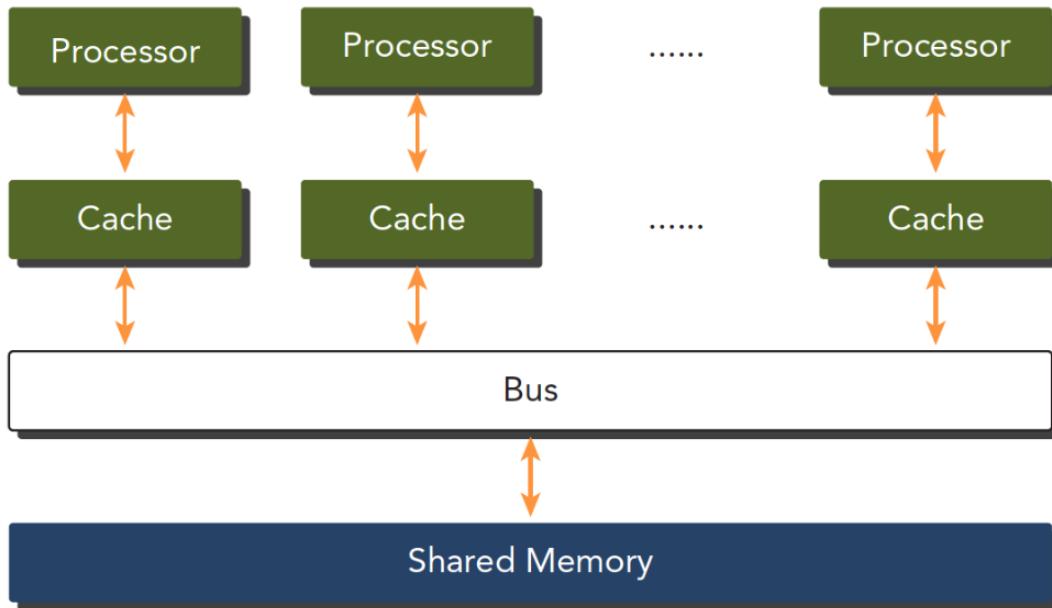


图 32: 多处理器共享同一内存

7.2 异构计算与 CUDA

GPU 本来的任务是做图像处理，把数据处理成是图形图像，图像的特点就是并行性很高。

7.2.1 异构架构

CPU 可以看成是一个指挥者，host，完成大量计算的是 GPU，如图 33 所示。

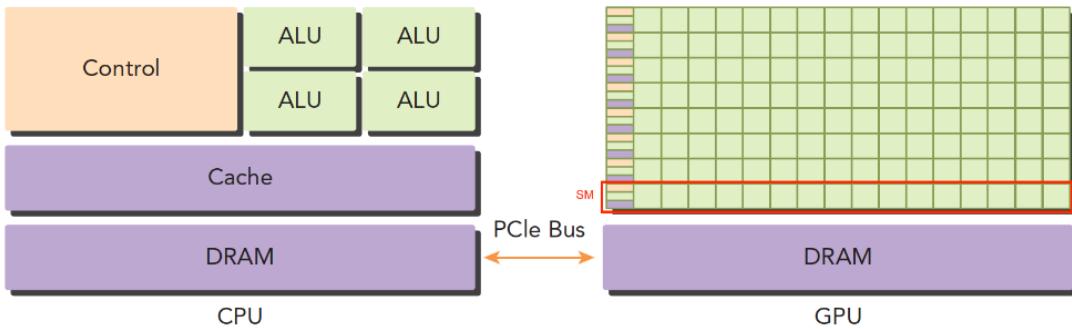


图 33: GPU 和 CPU 联系

- CPU 一般由四个 ALU 组成，ALU 是完成逻辑计算的核心。DRAM 是内存，Cache 缓存。
- GPU 是由更多的多处理器 SM 组成，每一个 SM 的 ALU 和控制单元能力变弱了，并且 cache 也独立。所以对于逻辑复杂的任务 GPU 处理效率是很低的。

CPU 和 GPU 之间通过总线连接，总线的数据传输是需要排队调度的，这部分的 IO 会很慢，cuda 也有 dgp 这些函数改进。一个异构包含了两个以上的架构，所以代码也包括不止一部分

- host code
- device code

host code 在主机运行，device code 在设备运行，这两者的代码是独立运行，自己跑自己的。主机端主要是整体程序的运行，包括复杂的指令逻辑等，设备端主要是用于计算，做一些子任务。

GPU 通过两个指标衡量计算能力

- 核心数量，越多越好
- 内存大小，越大越好

GPU 的性能指标

- 峰值计算能力
- 内存带宽

总的来说，如果是低并行，高逻辑，那适合 cpu，如果是高并行，低逻辑，那适合 gpu，如图 34 所示。

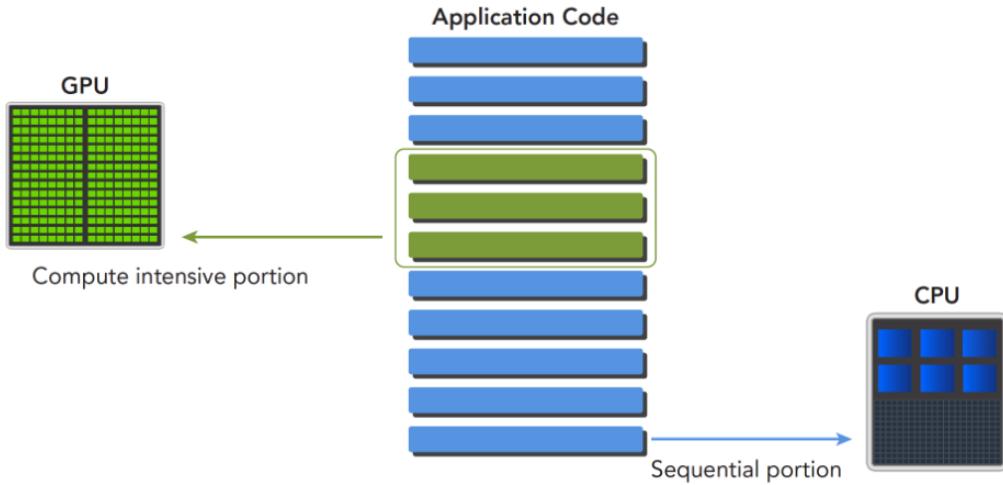


图 34: 串行和并行

CPU 和 GPU 线程的区别：

- CPU 线程包含的内容很多，切换上下文需要保存的内容非常多，切换成本很高
- GPU 的线程是轻量级，一般可以创建成千上万的线程，多数是在排队状态，切换基本没开销
- CPU 是减少一个线程的运行时间，而 GPU 是大量线程提高吞吐量

7.2.2 CUDA

CUDA 是一种异构计算平台，对于 API 有两种不同层次，如图 35 所示。

- 驱动 API，更底层
- 运行 API，更高层

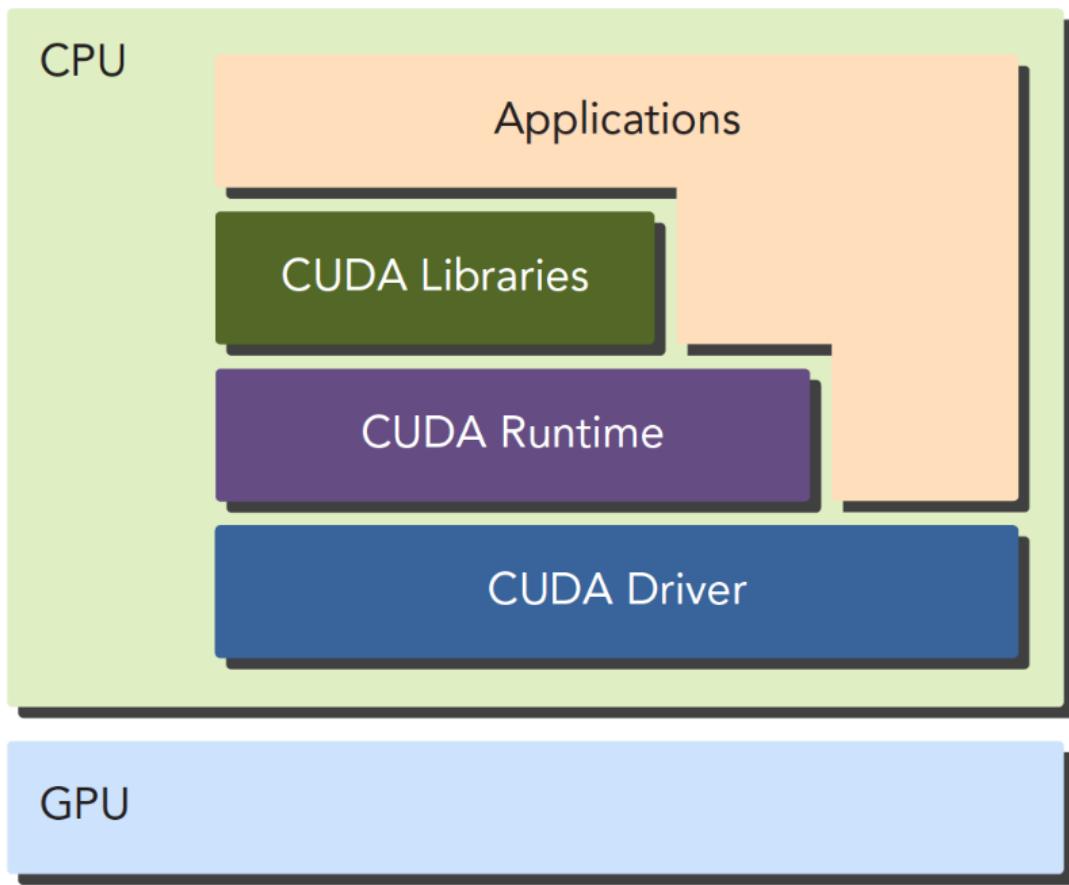


图 35: CUDA 的 API

CUDA nvcc 编译器会自动分离你代码里面的不同部分，如图中主机代码用 C 写成，使用本地的 C 语言编译器编译，设备端代码，也就是核函数，用 CUDA C 编写，通过 nvcc 编译，链接阶段，在内核程序调用或者明显的 GPU 设备操作时，添加运行时库。

看一个例子

```
#include <iostream>

__global__ void hello_world(void)
{
    printf("GPU: hello world \n");
}
```

```
int main() {
    hello_world<<<1, 10>>>();
    cudaDeviceReset();
    return 0;
}
```

____global____ 表示这个编译器是在设备上执行的核函数，«<1, 10»» 表示线程和 block 的数量。cudaDeviceReset 表示 cpu 和 gpu 的同步，因为 gpu 和 cpu 的执行并不是同步的，很可能说 cpu 已经执行完了，gpu 还在跑，完了 cpu 走完就退出了。所以需要这个函数进行同步。一般的 cuda 程序分成几步

- 分配内存
- 拷贝内存到设备
- 调用 cuda 核函数执行
- 计算完后把数据返回 host
- 销毁内存

cpu 和 gpu 的编程主要区别在于对 gpu 架构的熟悉，理解机器的结构是对编程效率影响非常大的一部分，了解你的机器，才能写出更优美的代码，而目前计算设备的架构决定了局部性将会严重影响效率。主要的几个对象

- 线程组的层次结构
- 内存层析结构
- 同步

线程和内存是主要研究对象。

7.3 编程模型概述

cuda 类似于一个硬件接口，为应用和硬件设备之间的桥梁。如图 37 所示。

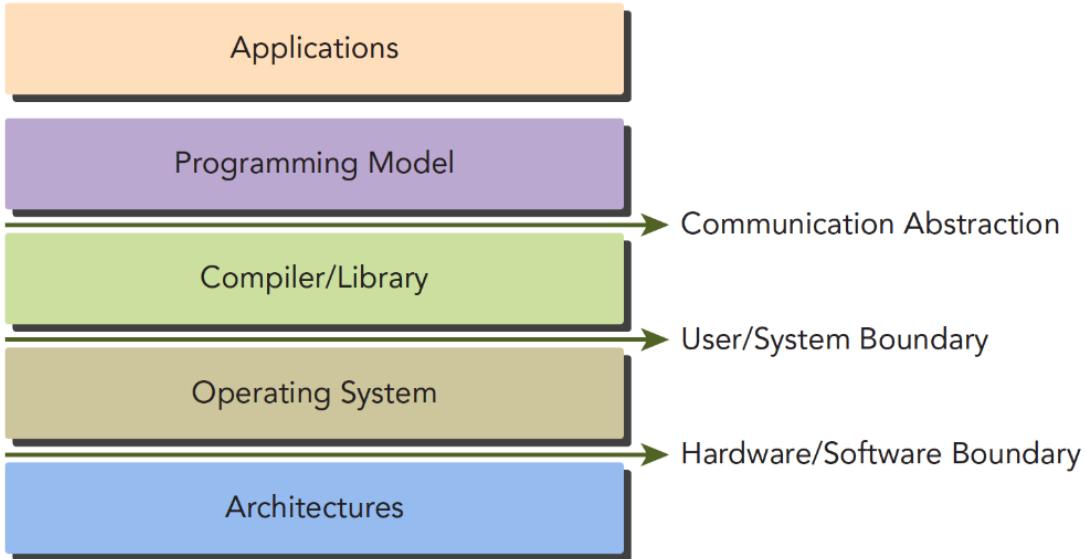


图 36: 编程模型

编程模型就是用到的语法，内存结构，线程结构等这些我们需要自己写的部分，控制了异构模型的计算工作模式就是编程模型。gpu 大致分成

- 核函数
- 内存管理
- 流程管理
- 流

7.3.1 CUDA 编程结构

异构环境通常包含了两个不同的环境

- CPU 和内存
- GPU 和内存

这两个内存从硬件到软件都是隔离的，一个完整的 cuda 程序如图 37所示。

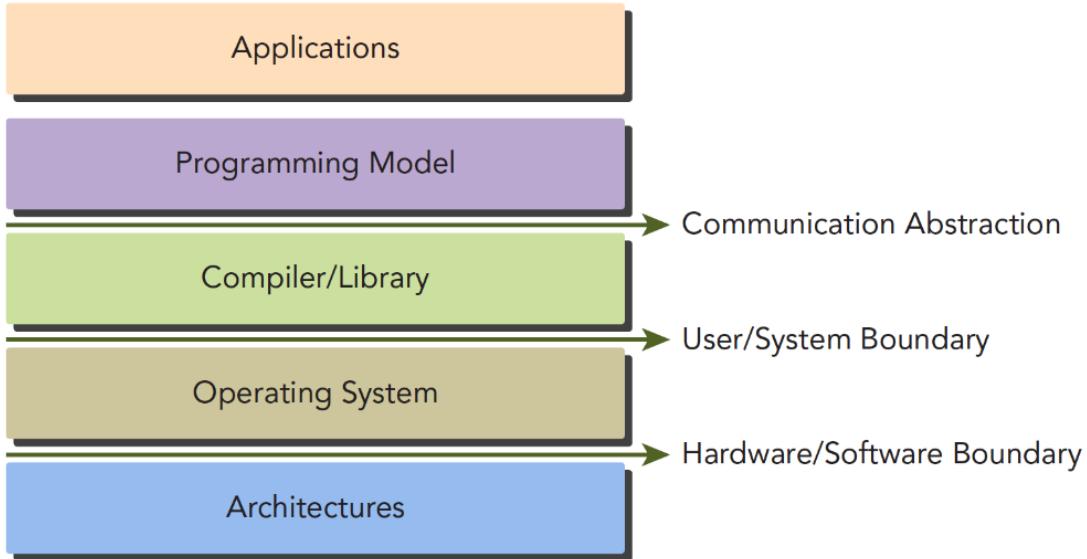


图 37: 编程模型

注意这里的并行和串行都是异步的，很可能第二段 host 执行，第一段 device code 还在执行。

7.3.2 内存管理

在 CPU 编程中，也就是一般的 C++ 代码，寄存器和栈空间是由机器自己分配，堆空间是用户控制分配。cuda 也是一样的，但是 cuda 上的 API 既可以用于管理 host 的内存，也可以用于管理 device 的内存。内存管理函数

- `cudaMalloc`: 内存分配，对标 `malloc`
- `cudaMemcpy`: 内存复制，对标 `memcpy`
- `cudaMemset`: 内存设置，对标 `Memset`
- `cudaFree`: 内存释放，对标 `free`

首先看 `cudaMemcpy`，他可以完成从 Host 到 host 的拷贝，host 到 device 的拷贝，device 到 host 的拷贝，device 到 device 的拷贝。内存是分层次的，如图 38 所示。

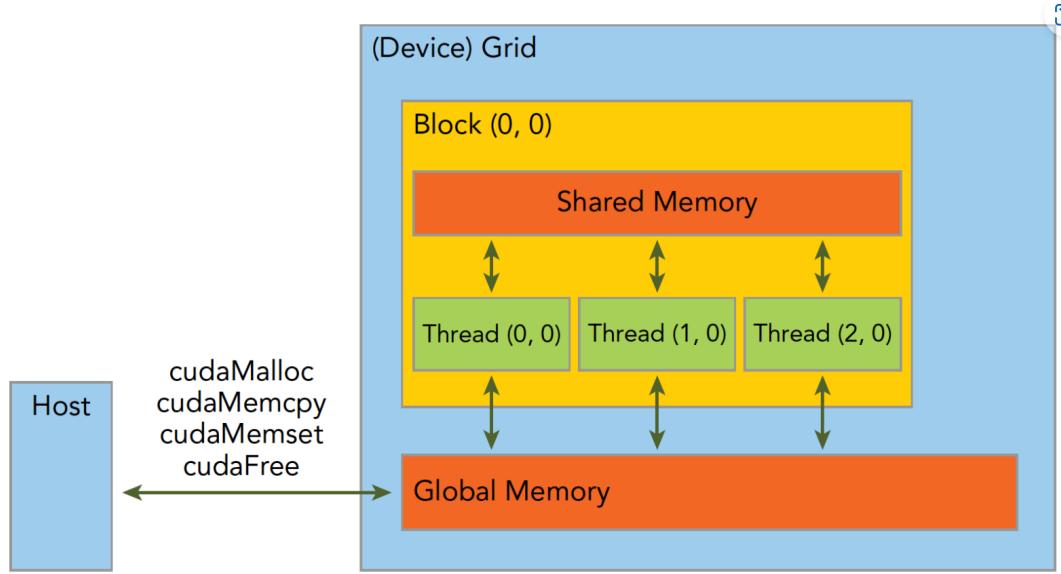


图 38: cuda 的内存层次结构

7.3.3 线程管理

一个核函数只能有一个 grid, 一个 grid 可以有很多个块, 每一个块可以有很多个线程。如图 39所示。

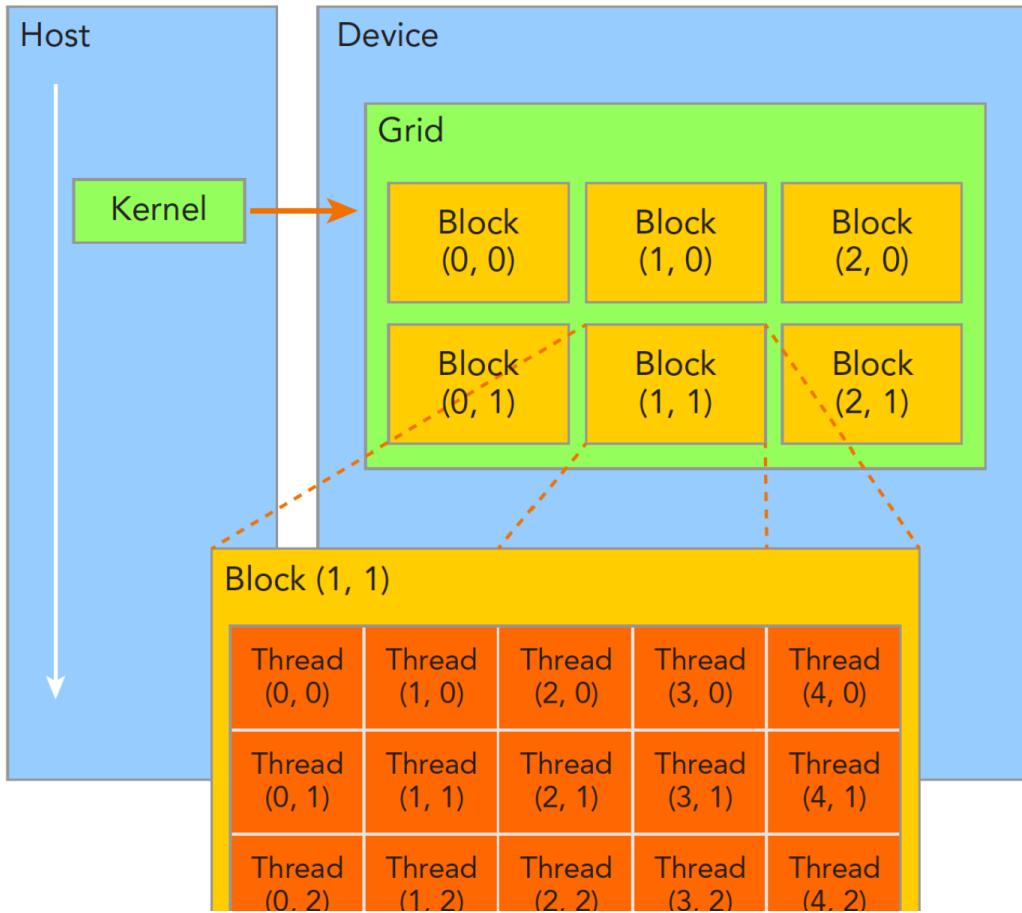


图 39: 核函数的 grid

一个线程块之间可以完成同步，内存共享，他们之间有一个 shared memory。但是不同块之间的线程是不可以互相影响的，因为他们是物理隔离的。不同块运行在不同的 warp 上。然后就是要给每一个线程一个编号了，因为 cuda 他是单指令，多数据，也就是 SIMD，所有每一个线程执行的指令都一模一样的。依靠

- blockIdx
- threadIdx

编号，最多可以三个维度，也就是

- blockIdx

- blockIdx.x
 - blockIdx.y
 - blockIdx.z
- threadIdx
 - threadIdx.x
 - threadIdx.y
 - threadIdx.z

自然也需要有字段来存储维度

- blockDim
 - blockDim.x
 - blockDim.y
 - blockDim.z
- gridDim
 - gridDim.x
 - gridDim.y
 - gridDim.z

grid 和 block 一般是二维是三维，也就是 grid 一般是分成二维的块，block 一般是三维的块。核函数主要是几个内容

- 启动核函数
- 编写核函数
- 验证核函数
- 错误处理

先正确写出，然后再进行优化。调用核函数代码

```
kernel_name<<<grid,block>>>(argument list);
```

对标 C++ 就是函数调用。«<grid,block»> 表示要执行这段代码的 grid 核 block, 通过对 grid 和 block 维度的改变, 可以调整的内容有两个

- 线程的数量
- 线程的布局

注意的是, 线程的并行和 host code 的运行是异构的, 线程调用后会立刻回到 Host code 执行, host code 的执行不会等到核函数执行完.

如果想要主机段等待执行完毕, 可以通过一下指令

- cudaDeviceSynchronize
- 但是似乎 cudaDeviceReset 也有同步的功能, 当然 cudaDeviceReset 的作用是释放 device 上的所有上下文, 包括分配的内存, 一般的放在最后面, 但他似乎也是有同步这个功能的

定义核函数

```
--global__ void kernel_name(argument list);
```

前面有几个限定符

- global: 设备端执行,global 是可以 host 调用的, 也可以 device 上调用,global 其实就说明了他的调用位置了.
- device: 设备端执行, 只能设备端调用
- host: 主机端执行

kernel 代码存在以下几点限制

- 只能访问设备内存, 如果需要访问 Host 内存就需要把内存移动过去
- 必须要有 void 返回类型
- 不支持可变数量的参数
- 不支持静态变量
- 异步行为, 即是在 host 调用的时候,device 不会停止

并行程序中经常做的事情是把 for 并行化, c 语言的代码:

```
void sumArraysOnHost(float *A, float *B, float *C, const int N) {
    for (int i = 0; i < N; i++)
        C[i] = A[i] + B[i];
}
```

cuda 并行

```
--global__ void sumArraysOnGPU(float *A, float *B, float *C) {
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}
```

定义一个一维, 和矩阵数量一样的线程, 然后每一个线程执行一个元素.

7.4 组织并行线程

线程建立矩阵索引, 多线程可以每一个线程处理不同的数据计算, 如何分配线程计算的数据就是一个问题. 比如对于一个二维的 grid 和 block, 如图 40所示.

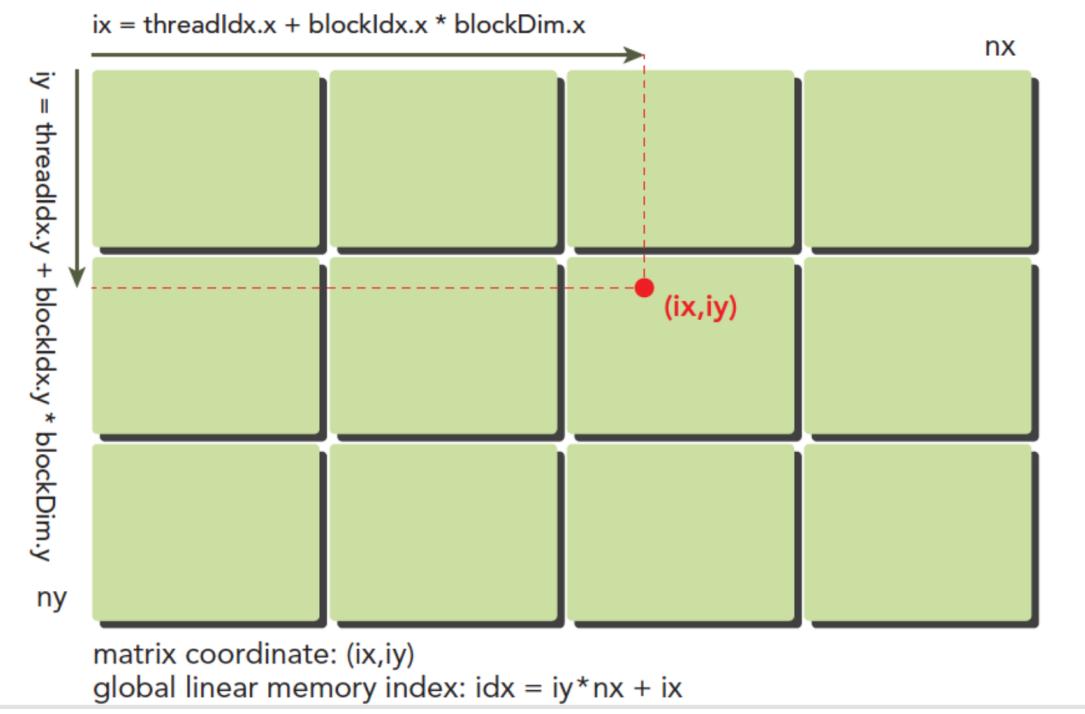


图 40: 索引

图中的 ix 和 iy 就是整个线程模型中一个线程的全局索引, 注意 $threadIdx$ 和 $blockIdx$ 都是局部索引, 我们需要计算全局索引。通过计算 ix 和 iy 就得到了线程中唯一标号。并且运行的时候 kernel 是可以访问这个标号的。如果每个不同的线程执行同样的代码, 又处理同一组数据, 将会得到多个相同的结果, 显然这是没意义的, 为了让不同线程处理不同的数据, CUDA 常用的做法是让不同的线程对应不同的数据, 也就是用线程的全局标号对应不同组的数据。

所以在这里我们主要做的事情就是

- 线程和块索引
- 矩阵中给定点的坐标
- (ix, iy) 对应的线性内存的位置

利用线程计算二维矩阵加法

```
--global__ void sumMatrix(float * MatA, float * MatB, float * MatC, int nx, int ny)
```

```

{
    int ix=threadIdx.x+blockDim.x*blockIdx.x;
    int iy=threadIdx.y+blockDim.y*blockIdx.y;
    int idx=ix+iy*ny;
    if (ix<nx && iy<ny)
    {
        MatC[idx]=MatA[idx]+MatB[idx];
    }
}

```

不过他这里是用一维矩阵表示二维

7.4.1 二维网格二维块

```

int nx = 20;
int ny = 22;
int dimx = 10;
int dimy = 10;
dim3 block_0(dimx, dimy);
dim3 grid_0((nx - 1) / block_0.x + 1, (ny - 1) / block_0.y + 1);
sumMatrix<<<grid_0,block_0>>>(A_dev,B_dev,C_dev,nx,ny);

```

假设我们要计算的矩阵的维度分别是 nx 和 ny, 假设 block 是 dimx 和 dimy, 那么我们需要的 grid 就是 $(nx - 1) / block_0.x + 1, (ny - 1) / block_0.y + 1$, 当然一般的 block 中 thread 的数量是要和 warp 相同.

7.4.2 二维网格一维块

```

dimx=32;
dim3 block_1(dimx);
dim3 grid_1((nxy-1)/block_1.x+1);

```

nxy 就是矩阵的维度

7.4.3 二维网格一维块

也一样的索引, 只不过这个 y 永远都是 1 而已.

7.5 CUDA 执行模型

核心部分，也就是学习 cuda 最终的目的。

7.5.1 gpu 架构

GPU 的架构是围绕一个多处理器 SM 扩展而成。如图 41所示.

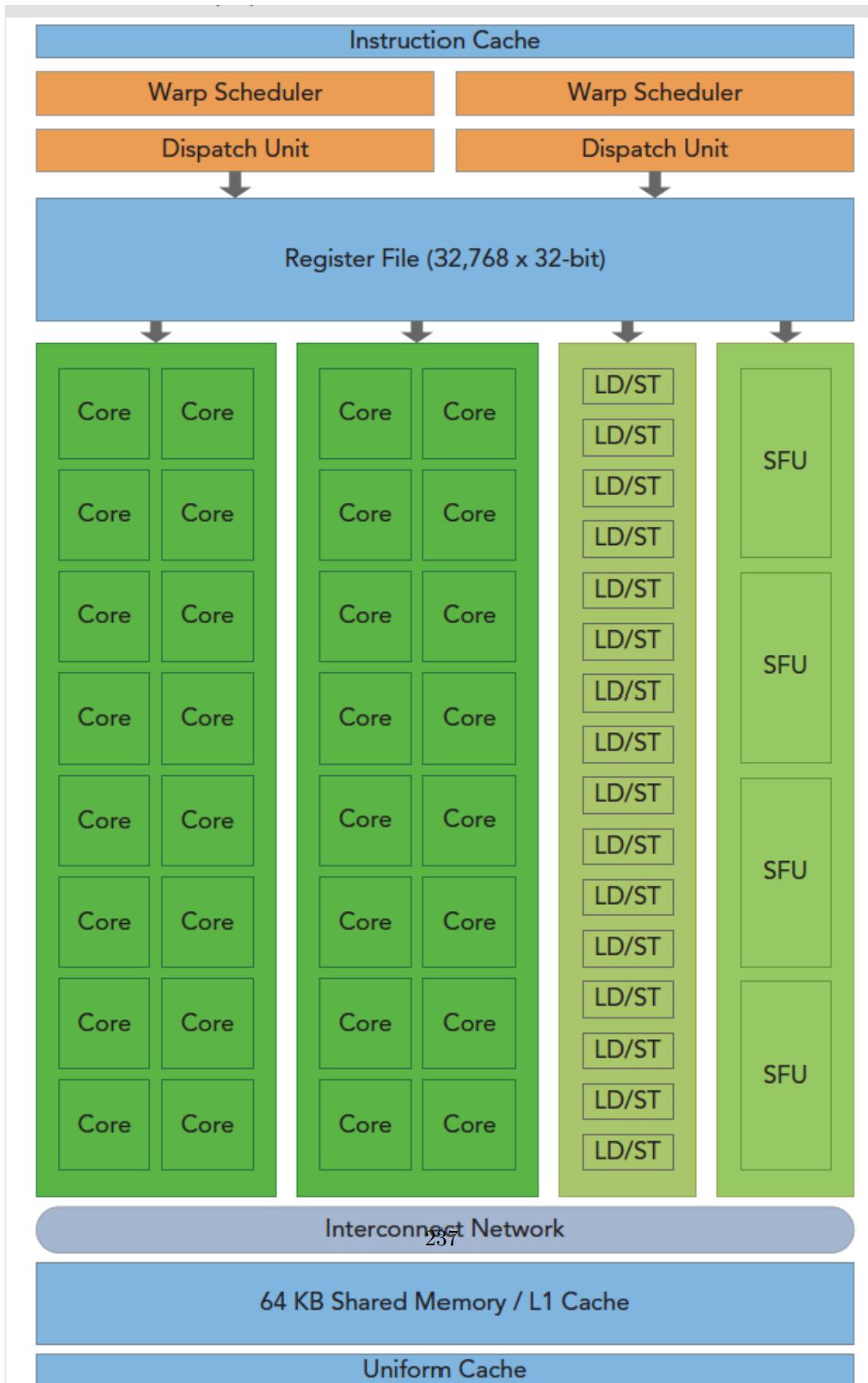


图 41: gpu 架构

指令到 warp 调度器，到分配执行单元，寄存器，到 SM。关键部件包含

- CUDA 核心
- 内存，缓存
- 寄存器文件
- 加载，存储单元
- 特殊功能单元
- 线程调度器
- SM，每一个 SM 都能支持几百个线程执行，每一个 gpu 通常会有很多很多个 sm，当一个核函数的网格启动的时，他里面的 block 会被分配到 SM 上执行。
 - 当一个 block 被分配给一个 SM 之后，未来就只能在这个 SM 上执行了。一个 SM 是可以执行多个 block 的。
- 线程束，也就是 warp，一般就是 32。不同设备的 warp 是不一样大的，gpu 是以 warp 为单位执行的，一个 warp 只能装载一个 block，也就是说一个 warp 只能调度一个 block，warp 里面的所有线程都只能属于一个 block，但是一个 block 是可以在多个 warp 上去执行的。
- SIMD vs SIMT，单指令多数据这种机器可以完成多次计算，但是特点就是他每一个分支只能做同一个事情，不允许有条件语句。但是 SIMT 就更加灵活了，SIMT 的某些线程是可以选择不执行的，所有 SMIT 更像是线程级别的系统，更精细，对程序员要求也越高；SMID 更像是指令级别的系统，程序员能控制的也就指令了。
 - SIMT 的特性：每一个线程都有自己指令的地址计数器，每一个线程是自己的寄存器状态，每一个线程可以有一个独立的执行路径。
- 32，这个数字很重要，32 是 SM 以 SIMD 方式同时处理的工作粒度，也就是说某些时刻，只能最多有 32 个线程进行执行，在一个 sm 中。

7.5.2 CUDA 编程的组件与逻辑

编程模型和硬件的对应关系，如图 42 所示。

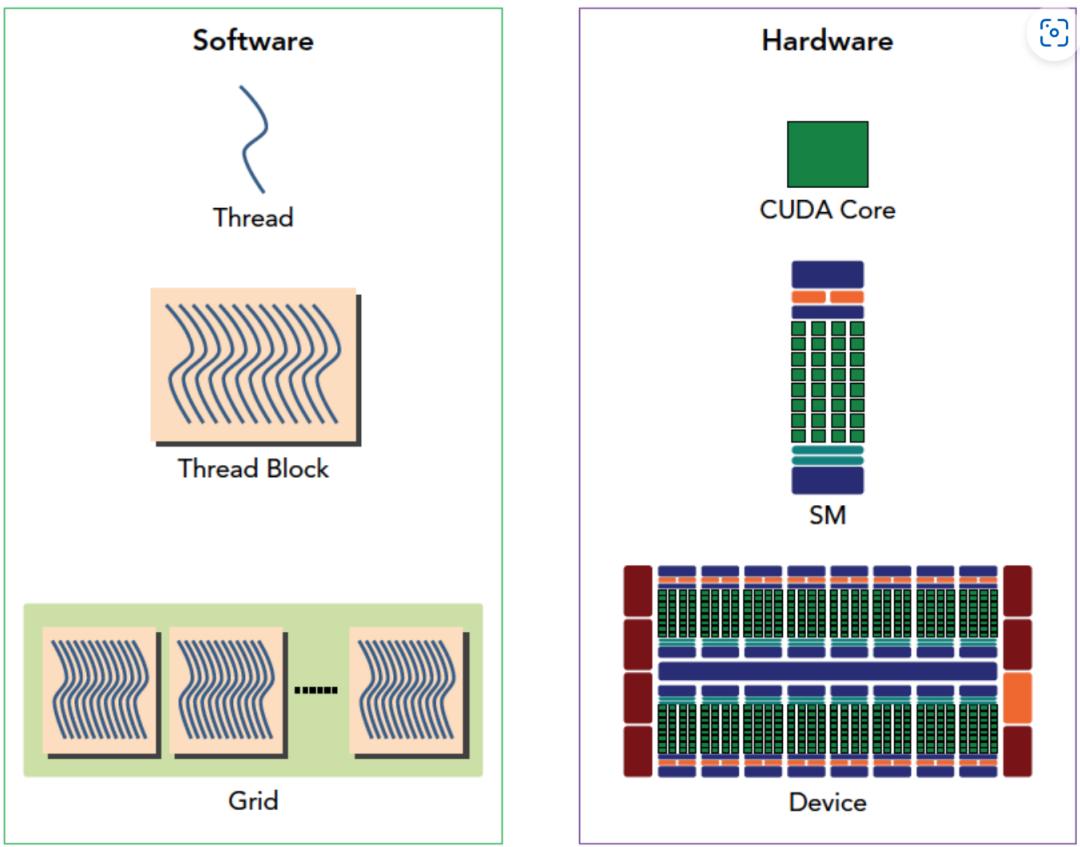


图 42: 编程模型和硬件的对应关系

- 线程对应着就是 cuda 的核心，一个核心处理一个线程
- block 对应一个 sm，每一个 sm 可以处理多个 block，但一个 block 只能在一个 sm 上处理。一个 sm 包含了多个 warp，一个 warp 只能处理一个 block。
- grid 对应着设备

7.5.3 Fermi 架构

Fermi 架构是第一个完整的 GPU 架构，虽然距离现在也有十几年了，但是有很多本质的东西没变。具体参数如下

- 512 个 cuda core

- 每一个 cuda 核心有 ALU 和浮点运算 FPU
- CUDA 核心在 16 个 SM 线程上
- 6 个 384-bits 的 GDDR5 的内存接口
- 支持 6G 的全局机载内存
- 分配线程块到 SM 线程束调度器上
- 768KB 的二级缓存，被所有 SM 共享

其实很多参数和现在的 gpu 差不多，内存在这个时候就已经是层次设定了。SM 包括

- 执行单元，也就是 core
- 调度器和调度单元，也就是 warp Scheduler
- 共享内存，这个内存我不知道是不是 shared memory

SM 有两个线程束调度器，和指令调度单元，当一个线程块被指定给一个 SM 时，线程块内的所有线程被分成线程束，两个线程束选择其中两个线程束，在用指令调度器存储两个线程束要执行的指令，线程束是在 sm 上交替执行。还是以 warp 为执行单元，如图 43所示。

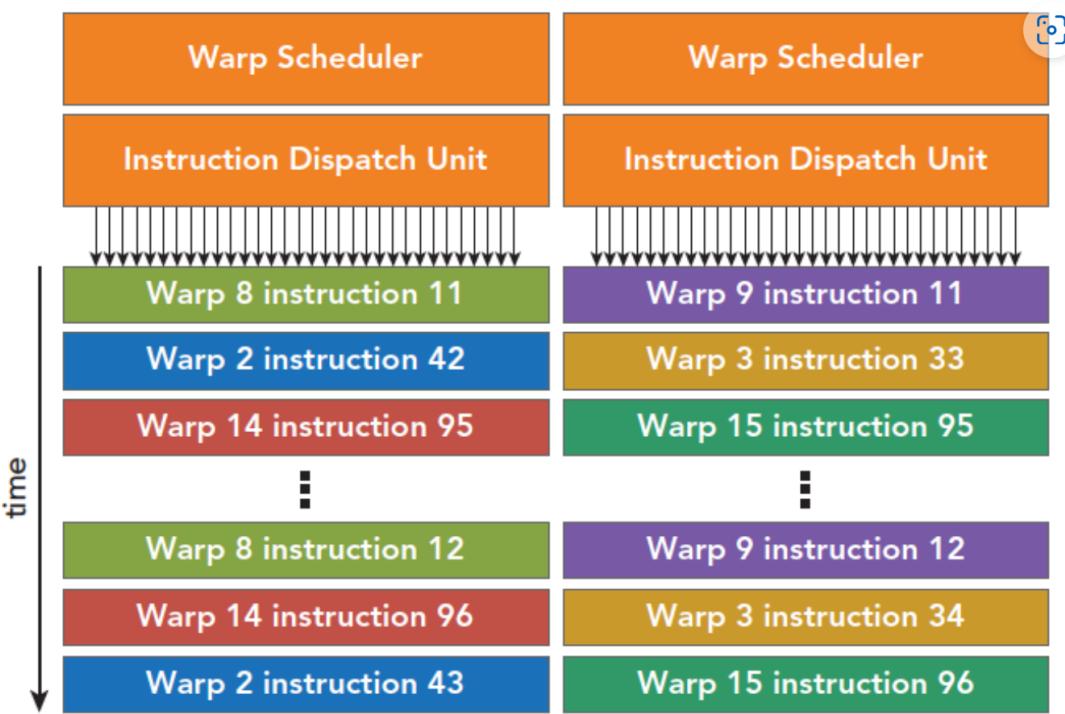


图 43: Fermi

7.5.4 Kepler 架构

Kepler 架构是 Fermi 架构的进化版，有三点突破

- 更强大的 SM
- 动态并行技术
- Hyper-Q 技术

还有其他的参数上也提升了不少，比如 cuda core 的数量等。首先是 SM 上，SM 里面的 cuda core 可以启动 core，也就是 gpu 的内核可以启动内核，只用 gpu 可以做到递归了。Hyper-Q 主要是是 cpu 和 gpu 之间的硬件连接，增加了 gpu 的并行性。

另外还有两个性能分析工具

- nvvp

- nvprof

cuda 还是得对性能可视化。

7.6 Warp 执行的本质

cuda 的编程模型并没有给每一个线程赋予执行顺序，他们都异步。但实际上硬件的资源是有限的，不可能说你分配多少个线程就同时执行多少个线程。所以实际上还是有先后顺序的。

7.6.1 线程块和线程束

线程束是 SM 中的基本执行单元，当一个网格启动，就相当于一个内核启动了，网格中包含的线程块会被分配到某一个 SM 上，然后在 SM 上又会分配到多个线程束，也就是 warp。每一个 warp 一般是 32，warp 中所有线程按照 SIMT 方式执行。

线程块并不是真实在 gpu 中存在的，他是一个逻辑产物，因为计算机中内存总是一维线程，写的时候可以是二维三维这样，方便写程序。在 block 中，每一个线程有唯一的编号 threadIdx，在 Grid 中每一个块有唯一的编号 blockIdx。

7.6.2 warp 分化

- 再 cpu 处理的时候，当程序包含大量的分支，意味着该程序的逻辑是复杂的。所以目前的 cpu 都是使用了分支预测技术。
- 对于 gpu 来说，逻辑部分是难点。假设一段 if..else 代码，32 个线程中可能有一部分走 if 一部分走 else，这个时候 warp 就分化了。
 - 解决问题的方法是都走一遍，比如这个 If..else，符合条件的就走，不符合条件你就看别人走。这样严重影响效率，也违背了线程设计的初衷。

例子

```
--global__ void mathKernel1(float *c)
{
    int tid = blockIdx.x* blockDim.x + threadIdx.x;

    float a = 0.0;
    float b = 0.0;
    if (tid % 2 == 0)
```

```
{  
    a = 100.0f;  
}  
else  
{  
    b = 200.0f;  
}  
c[tid] = a + b;  
}
```

每一个线程要执行两次分支，把 if 和 else 的都执行完。一个 warp 内需要执行两次不同的条件，最好的做法应该是要保证一个 warp 内的线程都走同一个分支。

```
--global__ void mathKernel2(float *c)  
{  
    int tid = blockIdx.x* blockDim.x + threadIdx.x;  
    float a = 0.0;  
    float b = 0.0;  
    if ((tid/warpSize) % 2 == 0)  
    {  
        a = 100.0f;  
    }  
    else  
    {  
        b = 200.0f;  
    }  
    c[tid] = a + b;  
}
```

tid/warpsize 在 0-31 为 0，在 31-63 为 1，同一个线程内都是同一个分支，速度会快很多。但是在实际运行的过程中，mathKernel1 和 mathKernel2 的运行效率都是 100，因为编译器做了优化。

所以最好的做法，肯定就是避免线程束分化了。

7.6.3 资源分配

每一个多处理器的执行基本单位是 warp，也就是说单指令会通过指令调度器广播给 warp 的全部线程，这些线程同一个时刻会执行同一个命令。从 warp 的角度上讲，一个时刻内如果有多个 warp，这些 warp 不可能同时执行，所以可以把这些 warp 分成三类

- 已经执行的 warp
- 激活的 warp，已经在 SM 上准备就绪，但是还没轮到她跑
- 未激活的 warp，可能分配到 sm 上了，但是还没到片上

SM 上有多少个 warp 可以被激活取决于三个因素

- 程序计数器
- 寄存器
- 共享内存

注意：warp 一旦被激活来到片上，那么他就不会再离开 SM 直到执行结束。每一个 SM 都有 32 位的寄存器组，不同架构数量不一样。共享内存是固定大小，会再 block 之间分配，称为 block 的 shared memory，block 中的 thread 是可以共享这些 shared memory 的。

当寄存器和共享内存分配给了线程块，这个线程就处于活跃状态，称为活跃 warp，这种线程也分成三类

- 选定的 warp
- 阻塞的 warp
- 符合条件的 warp

选定的 warp 是指这个 warp 被 SM 选好要执行了，也就是准备就绪的 warp，激活的 warp；不符合条件没准备好的就是阻塞的 warp。满足以下几个条件才是符合条件的

- 32 个 cuda core 可以用来执行
- 执行所需的资源都就绪了

所以 warp 按照执行进行分类

- 正在执行的 warp，如果有分支，会把所有分支执行一遍
- 未开始执行的 warp
 - 激活的 warp，也就是已经被 SM 选好准备就绪的 warp
 - 符合条件的 warp，准备要执行的 warp
 - 不符合条件的 warp，比如确实 core 或者内存等

7.6.4 延迟隐藏

延迟就是计算机运行一个算法所需要等待的时间，延迟隐藏，就是要求计算机在执行的时候所花的时间尽量小，也就是说利用率尽可能的高，使得尽可能的利用上所有资源。对于 GPU 来说延迟隐藏极为重要，延迟分两种

- 算术指令：算术操作从开始到产生结构之间的时间，这个时间段是只有计算单元进行工作，ALU 处于空闲
 - 算术延迟 10-20 个时钟周期
- 内存指令：主要是 IO，比如 CPU 到 GPU，或者 global memory 到 local memory 等
 - 内存延迟 400-800 个时钟周期

至少需要多少 warp 来保证最小化延迟？所需 warp = 延迟 x 吞吐量

并行是延迟隐藏增加的技术，指令级并行和线程级并行。指令隐藏的关键目的是使用全部的计算资源，而内存读取的延迟隐藏是为了使用全部的内存带宽，内存延迟的时候，计算资源正在被别的线程束使用，所以我们不考虑内存读取延迟的时候计算资源在做了什么，这两种延迟我们看做两个不同的部门但是遵循相同的道理。

7.6.5 占用率

占用率是一个 SM 中活跃 warp 的数量，占 SM 最大支持 warp 数量的比。

- 小的线程块：线程块的线程太少，导致 warp 空闲的线程比较多
- 大的线程块：太多线程，可用硬件资源少

7.6.6 同步

- 线程块内同步
- 系统级别

线程块内的同步就是

```
--syncthread();
```

7.7 并行性表现

利用 8192x8192 的矩阵计算来测试效率、

```
int ix = threadIdx.x + blockDim.x * blockIdx.x;
int iy = threadIdx.y + blockDim.y + blockIdx.y;
int idx = ix + iy * ny;
MatC[idx] = MatA[idx] + MatB[idx];
```

增强并行性的一个重要做法，就是避免分支分化。

7.7.1 并行规约问题

问题：一组特别多数字通过计算得到一个数字。当计算有以下特点

- 结合性
- 交换性

可以用并行归约的方法处理他们。规约包含三个步骤

- 将输入向量划分到更小的数据块中
- 用一个线程计算一个数据块内的部分和
- 对每个数据块部分和再求和

数据分块保证我们可以用一个线程块来处理一个数据块。一个线程处理更小的块，所以一个线程块可以处理一个较大的块，然后多个块完成整个数据集的处理。cpu 版本的实现

```
int recursiveReduce(int *data, int const size)
{
    // terminate check
    if (size == 1)
        return data[0];
    // renew the stride
    int const stride = size / 2;
    if (size % 2 == 1)
    {
        for (int i = 0; i < stride; i++)
        {
            data[i] += data[i + stride];
        }
    }
```

```

        data[0] += data[size - 1];
    }
    else
    {
        for (int i = 0; i < stride; i++)
        {
            data[i] += data[i + stride];
        }
    }
    // call
    return recursiveReduce(data, stride);
}

```

这个还是比较好理解的，和官方文档那个案例的意思差不多，只不过这里是增加了奇数的处理。

内核实现：

```

__global__ void reduceNeighbored(int * g_idata,int * g_odata,unsigned int n)
{
    //set thread ID
    unsigned int tid = threadIdx.x;
    //boundary check
    if (tid >= n) return;
    //convert global data pointer to the
    int *idata = g_idata + blockIdx.x*blockDim.x;
    //in-place reduction in global memory
    for (int stride = 1; stride < blockDim.x; stride *= 2)
    {
        if ((tid % (2 * stride)) == 0)
        {
            idata[tid] += idata[tid + stride];
        }
        //synchronize within block
        __syncthreads();
    }
    //write result for this block to global mem
    if (tid == 0)
        g_odata[blockIdx.x] = idata[0];
}

```

}

核心技术就是把数据分成多个块，每一个块用一个 block 计算，如图 44所示。

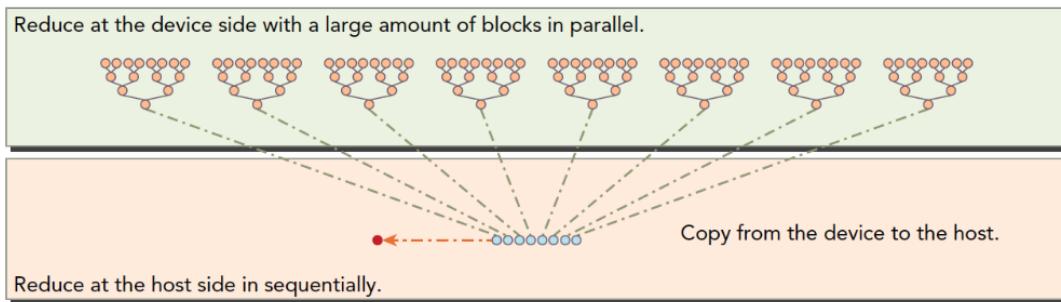


FIGURE 3-22

图 44: 分块规约

```
unsigned int tid = threadIdx.x;
```

得到一个块里面他的线程位置。

```
if (tid >= n) return;
```

一个块就算 n 个，超出了不算。

```
int *idata = g_idata + blockIdx.x*blockDim.x;
```

每一个块应该算的数据位置的起始位置，这里是一维的 block。

```
for (int stride = 1; stride < blockDim.x; stride *= 2)
{
    if ((tid % (2 * stride)) == 0)
    {
        idata[tid] += idata[tid + stride];
    }
    //synchronize within block
    __syncthreads();
}
```

分组相加， $\text{stride}=1$ 就是相邻的元素相加，02468 这些元素相加。 $\text{stride}=2$ ，048 相加，以此类推。最后 $\text{tid} = 0$ 存储了此块的和。

感觉这些编程最难的应该是两点：1) 线程的索引，局部索引和全局索引之间的变换。2) 寻找满足条件的线程进行相加。

--syncthreads

是 block 内线程同步。如果分块的时候一个 block 处理不了，那这个同步函数就失效了。不同块是不行的。求和过程如图 45 所示。

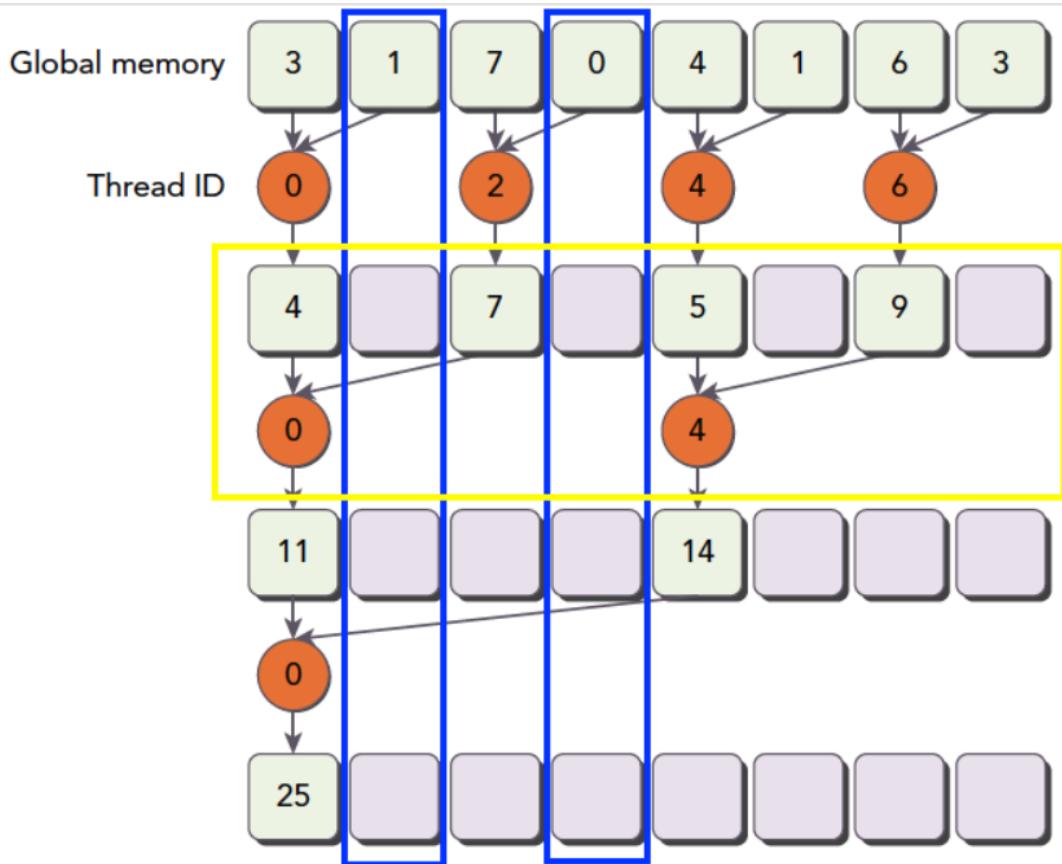


FIGURE 3-21

图 45: 求和过程

0 线程执行了 01 加法，2 线程执行了 23，以此类推。

7.7.2 改进规约分化

这里存在一个问题，这个 for 循环存在了一半的 if，导致一半的线程阻塞了。第一轮一半的线程没用，第二轮 3/4 的线程用。在第一轮 1357... 这些线程都没什么用，他们的值分别加到了 1246... 上面。

首先我们保证在一个块中前几个执行的线程束是在接近满跑的，而后半部分线程束基本是不需要执行的，当一个线程存在分支，分支都不需要执行，硬件会停止他调用其他，这样就节省了资源。作者说的好像不是很清楚，我重述一遍，为什么要保证一个块前几个线程是跑满呢？因为一个 Block 他会分在不同的 warp 里面工作，这个例子数量比较少，如果数量多了，然后前边的线程又跑满，比如一共 600 个线程，前面一半跑满，那么按照 Warpsize = 32，前面跑满的就会分到同一个 warp。所有尽量让前面的线程都跑满是为了使得 warp 的利用率高。

```
--global__ void reduceNeighboredLess(int * g_idata,int *g_odata,unsigned int n)
{
    unsigned int tid = threadIdx.x;
    unsigned idx = blockIdx.x*blockDim.x + threadIdx.x;
    // convert global data pointer to the local point of this block
    int *idata = g_idata + blockIdx.x*blockDim.x;
    if (idx > n)
        return;
    //in-place reduction in global memory
    for (int stride = 1; stride < blockDim.x; stride *= 2)
    {
        //convert tid into local array index
        int index = 2 * stride *tid;
        if (index < blockDim.x)
        {
            idata[index] += idata[index + stride];
        }
        __syncthreads();
    }
    //write result for this block to global mem
    if (tid == 0)
        g_odata[blockIdx.x] = idata[0];
}
```

int index = 2 * stride *tid; 这一句是关键，让 0,1,2,3,4 这些线程去做，把执行的线程都提

前了。

7.7.3 交错配对

如图 46 所示。

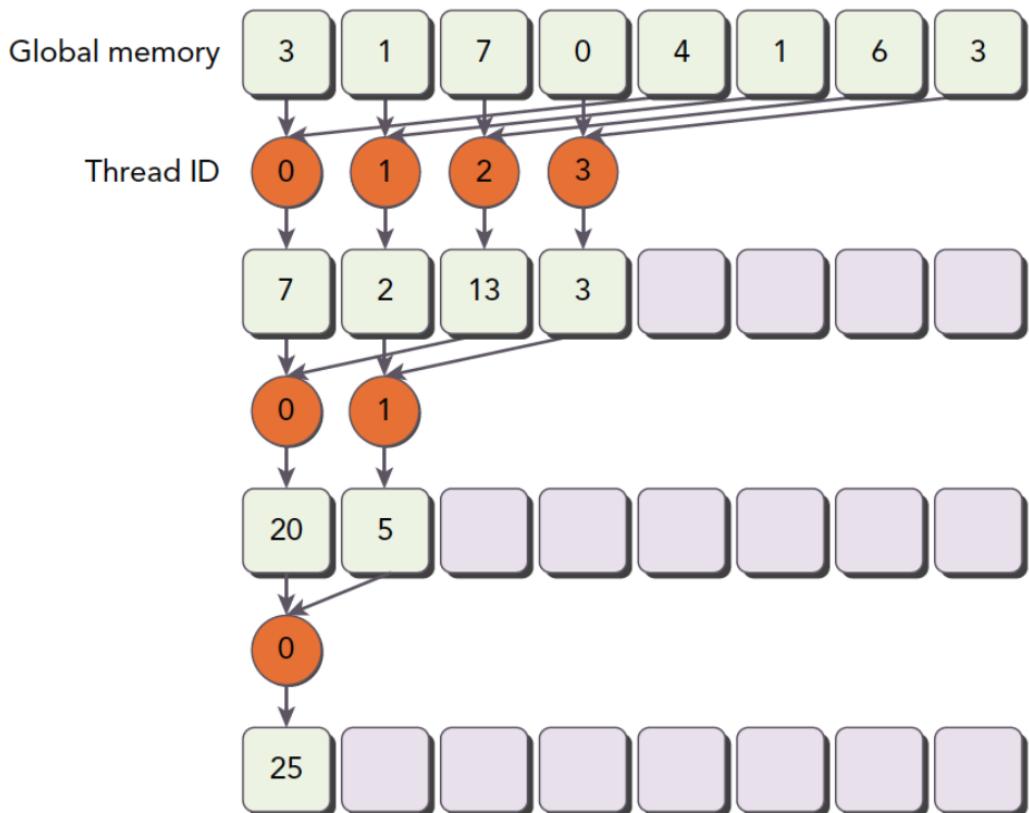


图 46: 交错配对

本身就比较优了，跳步相加就行。

```
--global__ void reduceInterleaved(int * g_idata, int *g_odata, unsigned int n)
{
    unsigned int tid = threadIdx.x;
    unsigned idx = blockIdx.x*blockDim.x + threadIdx.x;
    // convert global data pointer to the local point of this block
    int *idata = g_idata + blockIdx.x*blockDim.x;
```

```
if (idx >= n)
    return;
//in-place reduction in global memory
for (int stride = blockDim.x/2; stride >0; stride >>=1)
{
    if (tid <stride)
    {
        idata[tid] += idata[tid + stride];
    }
    __syncthreads();
}
//write result for this block to global mem
if (tid == 0)
    g_odata[blockIdx.x] = idata[0];
}
```

这哥们很喜欢用位运算。这一节的主要内容就是加强并行，主要的 insight 就是要考虑如何把 warp 中的线程利用率拉满，这里的做法就是尽可能的把前面部分的线程跑满，也就是把跑满的放一起，这样在同一个 warp 中这些线程就不会跑到不同的分支。

7.8 展开循环

GPU 喜欢做一些确定的东西，GPU 没有分支预测的能力，所以每一个分支他都执行。

```
for (int i=0;i<100;i++)
{
    a[i]=b[i]+c[i];
}
```

如果对这个代码进行循环展开：

```
for (int i=0;i<100;i+=4)
{
    a[i+0]=b[i+0]+c[i+0];
    a[i+1]=b[i+1]+c[i+1];
    a[i+2]=b[i+2]+c[i+2];
    a[i+3]=b[i+3]+c[i+3];
}
```

这种展开是可以加快模型的运行速度，但实际上 C++ 的一些编译器会帮助我们优化，也就是说这两种写法的机器代码是差不多一样的。但是目前的 CUDA 编译器不具备这样的能力，不能做这样的优化，而人为展开核函数内的循环是可以非常大提高内核性能。

展开循环的目的还是两个

- 减少指令消耗
- 增加更多的独立调度指令

7.8.1 展开规约

交错规约只是一个数据处理一个块，8 个数据他还是需要用 4 个线程处理。但如果说以前是一个线程对应一个数据，现在一个线程对应两个数据。

```
--global__ void reduceUnroll2(int * g_idata,int * g_odata,unsigned int n)
{
    //set thread ID
    unsigned int tid = threadIdx.x;
    unsigned int idx = blockDim.x*blockIdx.x*2+threadIdx.x;
    //boundary check
    if (tid >= n) return;
    //convert global data pointer to the
    int *idata = g_idata + blockIdx.x*blockDim.x*2;
    if(idx+blockDim.x<n)
    {
        g_idata[idx]+=g_idata[idx+blockDim.x];

    }
    __syncthreads();
    //in-place reduction in global memory
    for (int stride = blockDim.x/2; stride>0 ; stride >>=1)
    {
        if (tid <stride)
        {
            idata[tid] += idata[tid + stride];
        }
        //synchronize within block
        __syncthreads();
    }
    //write result for this block to global mem
```

```
if (tid == 0)
    g_odata[blockIdx.x] = idata[0];

}
```

大概看懂了

```
unsigned int idx = blockDim.x*blockIdx.x*2+threadIdx.x;
```

线程的全局索引，以前是一个 block 的数据处理一个块的数据，现在是一个 block 的数据处理两个块的数据。比如 ABCD 三个块的数据，应该是需要用四个 block 计算，但是现在只需要两个 block，第一个 block 处理 A 的数据，首先把 A 的数据和 B 的数据做一个向量加法，然后对 A 的数据做一个交替归并。

```
int *idata = g_idata + blockIdx.x*blockDim.x*2;
```

数据位置，需要跳过两个 block。

```
if(idx+blockDim.x<n)
{
    g_idata[idx]+=g_idata[idx+blockDim.x];
}
```

相当于做了一个向量加法。后面就是和一搬的交替归并是一样的。如图 47 所示。

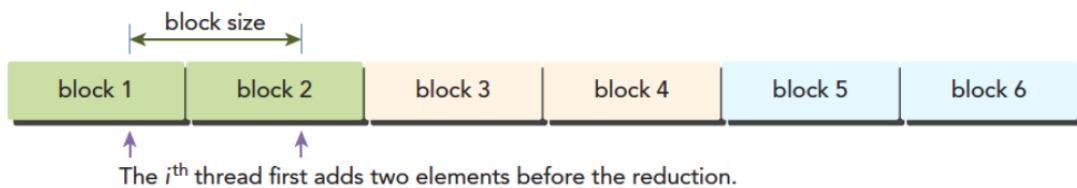


FIGURE 3-25

图 47: rolling2

对比了一下两者效率

```
nvprof --metrics dram_read_throughput ./cuda_projects
```

首先是一般的交替，如图 48 所示，吞吐量大概就是 18 上下。

```
Device "NVIDIA GeForce GTX 1060 6GB (0)"
  Kernel: reduceInterleaved(int*, int*, unsigned int)
    1          dram_read_throughput      Device Memory Read Throughput 18.961GB/s 18.961GB/s 18.961GB/s
PS E:\cuda_projects\cmake-build-debug>
```

图 48: reduceUnroll1 的测试

2 个 block 展开的规约，如图 49 所示，吞吐量大概就是 34，差不多是双倍的吞吐量了。

```
Invocations           Metric Name           Metric Description   Min   Max   Avg
Device "NVIDIA GeForce GTX 1060 6GB (0)"
  Kernel: reduceInterRollingTwo(int*, int*, unsigned int)
    1          dram_read_throughput      Device Memory Read Throughput 34.201GB/s 34.201GB/s 34.201GB/s
PS E:\cuda_projects\cmake-build-debug> |
```

图 49: reduceUnroll2 的测试

这里 window 可能会出问题，出现 4390:99 的问题，这个问题是你权限没设置到位。写了一些一个指令同时计算 4 个 block 的算法：

```
--global__ void reduceInterRollingFour(int* iData, int* oData, unsigned int n)
{
    unsigned int tid = threadIdx.x;
    unsigned int idx = threadIdx.x + blockDim.x * blockIdx.x * 4;

    if (idx >= n)
    {
        return;
    }

    int* iData_process = iData + blockDim.x * blockIdx.x * 4;

    if(idx + blockDim.x < n)
    {
        iData[idx] += iData[idx + blockDim.x];
        if (idx + blockDim.x * 2 < n)
            iData[idx] += iData[idx + blockDim.x * 2];
        if (idx + blockDim.x * 3 < n)
            iData[idx] += iData[idx + blockDim.x * 3];
    }
}
```

```

    }

    __syncthreads();

    for(int stride = blockDim.x/2; stride > 0; stride >>= 1)
    {
        if(tid < stride)
        {
            iData_process[tid] += iData_process[tid + stride];
        }
        __syncthreads();
    }

    if(tid == 0)
    {
        oData[blockIdx.x] = iData_process[0];
    }
}

```

在中间添加三个向量加法就行。Nsight System 性能分析更好用，同样也可以写出 roll8 的。只不过在中间向量加法的时候有所改变

```

if (idx + blockDim.x < n)
{
    iData[idx] += iData[idx + blockDim.x];
    if (idx + blockDim.x * 2 < n)
    {
        iData[idx] += iData[idx + blockDim.x * 2];
    }
    if (idx + blockDim.x * 3 < n)
    {
        iData[idx] += iData[idx + blockDim.x * 3];
    }
    if (idx + blockDim.x * 4 < n)
    {
        iData[idx] += iData[idx + blockDim.x * 4];
    }
    if (idx + blockDim.x * 5 < n)
    {
        iData[idx] += iData[idx + blockDim.x * 5];
    }
}

```

	reduceUnroll1	reduceUnroll2	reduceUnroll4	reduceUnroll8
Duration	893	493	361	296
计算吞吐量	28	28	23	18
内存吞吐量	27	45	51	62

```

if (idx + blockDim.x * 6 < n)
{
    iData[idx] += iData[idx + blockDim.x * 6];
}
if (idx + blockDim.x * 7 < n)
{
    iData[idx] += iData[idx + blockDim.x * 7];
}
}

```

四种实现的效果对比：可以看到，内存的吞吐量是很高的，但是这个计算吞吐量却不大，我觉得可能是后面的 if。因为我发现作者在增加向量的时候并没有考虑界限问题，很可能超出，所以我需要每一次相加都做一个判断。

后面想了一下，我添加向量那步没必要增加 if 判断，因为就算最后只有 6 个，不到 8 个，那两个多出来的数字也不需要额外判断，因为他们不加进去就是了，所以在 reduceUnroll8 中直接一个 `idx + blockDim.x * 7 < n` 就行。

7.8.2 完全展开的归约

展开第二个 for stride 的求和。写了一下但是好像没有什么影响，就算了。

7.8.3 模板函数的归约

这部分没看太懂。

这部分真的受益匪浅，主要就是两个技术

- 尽量让同一个 warp 里面的线程能跑满
- 尽量让一个指令多执行数据，提升指令的吞吐量

在宿舍那台 3080 laptop 上用 Nsight compute 就很直观的看出 kernel 的效率，但是在实验室这个 1060 这里就用不了这个 Nsight，不知道为什么？

7.9 动态并行

目前位置所有的内核都是在 host 中进行调用，能否在内核中调用内核？这个就是动态并行的。

- 优点是可以让内核变得有层次。动态并行的另一个好处是等到执行的时候再配置创建多少个网格，多少个块，这样就可以动态的利用 GPU 硬件调度器和加载平衡器了，通过动态调整，来适应负载。并且在内核中启动内核可以减少一部分数据传输消耗。
- 缺点就是程序会更复杂，并且更不好控制

7.9.1 嵌套执行

用 gpu 中的内核启动内核，在 cpu 并行中有一个类似的概念，就是父子线程。host code 启动了一个 kernel，kernel 在执行过程中又启动一个新 kernel，等待新的 kernel 结束后 kernel 才能结束。

- 调用父线程，也就是父 kernel 后，每一个线程都可以再调用子 kernel，这些子网格又相同的线程块，可以同步。block 中所有的线程创建的所有子网格完成后，block 才会执行完。这和 host 不太一样，host 和 device 完全是异步执行。
- 父子线程之间存在一个隐式同步，也就是所有的父线程都要等待子线程执行完成后，才能退出。

主机内启动的网格，如果没有显式同步，也没有隐式同步指令，那么 cpu 线程很有可能就真的退出了，而你的 gpu 程序可能还在运行，这种就是没用隐式同步。内存竞争是动态并行：

- 父 kernel 和子 kernel 共享相同的全局和常量内存，这个全局内存不确定他说的是不是 shared memory 还是 global memory，global memory 肯定共享
- 父 kernel 和子 kernel 有各自的 local memory
- 有了子网格和父网格间的弱一致性作为保证，父网格和子网格可以对全局内存并发存取。
- 有两个时刻父网格和子网格所见内存一致：子网格启动的时候，子网格结束的时候，也就是说子网格结束后会立刻将缓存中的内容写入内存，再把指挥权交给父 kernel
- 共享内存和局部内存分别对于线程块和线程来说是私有的，也就是说父 block 的 shared memory 和子 block 的 shared memory

- 局部内存对于线程来说私有

7.10 全局内存

在执行模型中，核，内存都会影响性能，但影响最大的还是内存。从前面交错规约的例子就可以看出，内存吞吐率增大，计算吞吐量减小了但是计算效率缺高了不少。

7.11 内存层析结构的优点

程序的局部性特点

- 时间局部性

就是一个内存位置的数据某时刻被引用，那么在该时刻附近也有可能被引用，随着时间流失，该数据被引用的可能性也会降低。

所以通常一个数据被调用之后会先存储进缓存中，等待下一侧被调用。

- 空间局部性

如果某一内存的数据被引用，那么附近的数据也有可能被引用

缓存其实就是基于这两点设计的。如图 50所示。

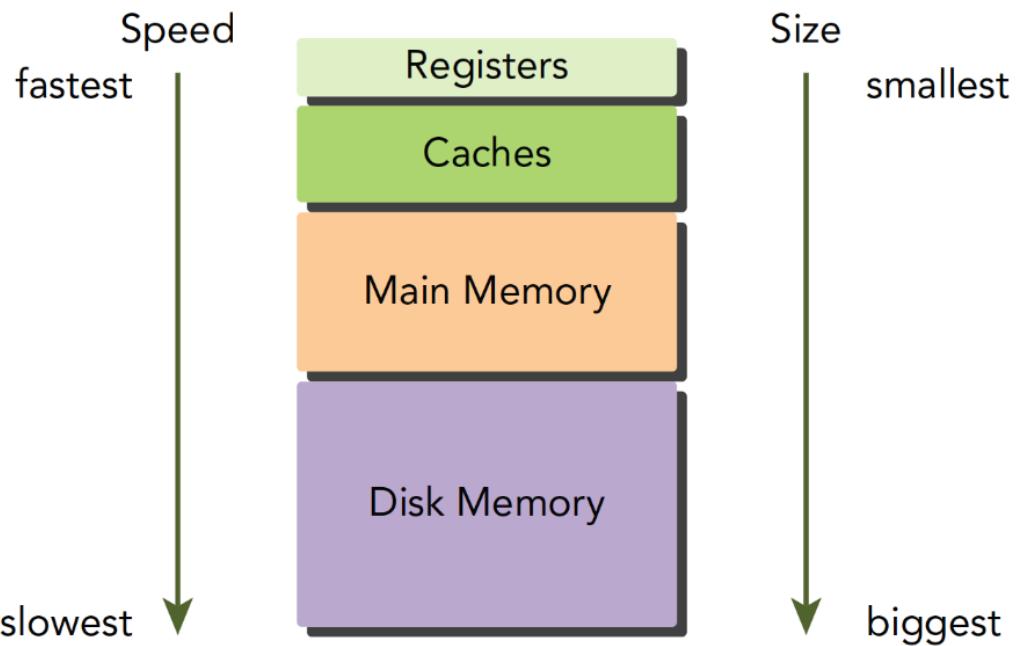


FIGURE 4-1

图 50: 内存模型

速度最快的肯定就是寄存器了，他可以和 CPU 同步进行，接着就是缓存，在 CPU 片上，然后就是主存储器，最后就是常见的内存条和硬盘了。越快的内存越小，越慢的内存越大。内存模型层次越往下：

- 每一个比特的价格更低
- 容量更大
- 延迟更高
- 访问频率更低

最后一层的硬盘是可以永久存储，但其访问频率也更低。

7.11.1 CUDA 内存模型

可编程内存，和不可编程内存两种。相比 CPU，GPU 的内存模型有很多了

- 寄存器
- 共享内存
- 本地内存
- 常量内存
- 纹理内存
- 全局内存

很多都是为了图形处理服务的。每一个线程有自己的本地内存，也就是 **local memory**，**block** 有自己的共享内存，**shared memory**；所有线程都可以访问读取常量内存和纹理内存，也就是 **constant memory** 和 **texture memory**，但不能写只能读；全局内存，常量内存和纹理内存有相同的生命周期；整体的内存图如图 51所示。

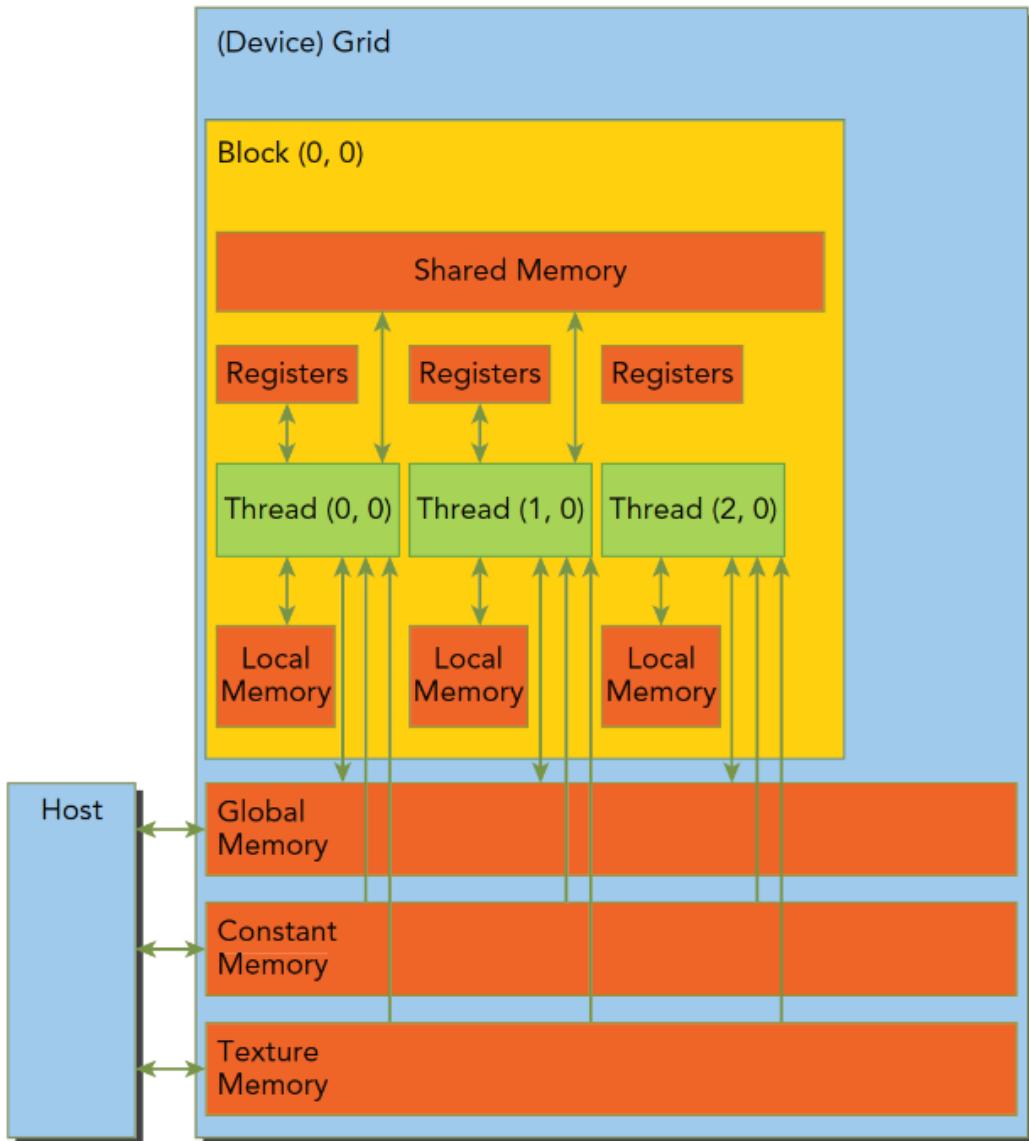


FIGURE 4-2

图 51: 内存结构

所有按照内存的访问范围从上到下，首先是 Global memory, constant memory 和 texture memory，他们可以被任何线程访问。然后就是 shared memory，他只能被 block 内的线程共享，local memory 只能被线程单独访问。

7.11.2 寄存器

寄存器无论是在 cpu 还是 gpu 都是访问速度最快的内存设备，但是 cpu 比 gpu 的寄存器数量更多，当我们不加修饰直接用类似 `int a` 这种方式在核函数内部声明，这个变量会存储在寄存器中。这和 cpu 有些许不同，在 cpu 中声明的变量都是存储在内存中，计算的时候才会传输到寄存器计算，剩下的时候都待在内存中。核函数中定义的常数长度的数组，也是在寄存器中。

寄存器对于每一个线程是私有的，生命周期和核函数一致，从开始到结束，执行完之后寄存器就不能访问了。寄存器是 SM 的稀缺资源，所以会出现两种情况

- 如果一个线程用到的寄存器少，那么一个 block 中就可以并发更多的线程
- 但如果用到的变量太多了，寄存器存不下，本地内存就过来帮忙存储与，这个时候就会有一些效率上的问题

7.11.3 本地内存

本地内存存储的条件是：在核函数中被使用到，或者在核函数中定义了，但是寄存器放不下了，就可以放到本地内存，所以一般放在 `local memory` 的有

- 使用未知索引引用的本地数组
- 可能会占用大量寄存器空间的较大本地数组或者结构体
- 任何不满足核函数寄存器限定条件的变量

本地内存实质上是和全局内存一样在同一块存储区域当中的，其访问特点——高延迟，低带宽。所以使用本地内存并不是一个好方法，因为本地内存本质上和全局内存是同一个地方，所以 IO 的时间是很长的。

既然寄存器可以和本地内存之间做互通，那么这个 `local memory` 应该是对于线程不可见的，也就是不可编程。

7.11.4 共享内存

如果需要在共享内存下运行，那需要加限定词，每个 SM 都有一定数量的由线程块分配的共享内存，共享内存是片上内存，跟主存相比，速度要快很多，也即是延迟低，带宽高。其类似于一级缓存，但是可以被编程。共享内存相比本地内存来说，更快，和 `global memory` 相比，更快，延迟低，带宽高。本地内存反而比共享内存要慢我是没想到。共享内存存在核

函数内声明，生命周期和线程块一致，线程块运行开始，此块的共享内存被分配，当此块结束，则共享内存被释放。

因为共享内存是块内线程可见的，所以就有竞争问题的存在，也可以通过共享内存进行通信，当然，为了避免内存竞争，可以使用同步语句：

```
void __syncthreads();
```

所以共享内存也会存在内存竞争的问题。

7.11.5 常量内存

常量内存设备内存中，SM 是有专门的常量内存缓存。常量内存核函数外，全局范围内声明，对于所有设备，只可以声明 64k 的常量内存，常量内存静态声明，并对同一编译单元中的所有核函数可见。常量内存是指对核函数不可修改，但是对于 host code 是可以修改的。初始化

```
cudaError_t cudaMemcpyToSymbol(const void* symbol, const void *src, size_t count);
```

就是把 host 的内容复制到 symbol，也就是常量内存里面。

常量内存的读取机制是一次读取会广播给所有 warp 里面的线程。所有如果所有线程都要相同地址的内容，那么效率会很高。如果不同线程要的是不同地址，那么每一个不同的变量都要不断广播出去。

7.11.6 纹理内存

纹理内存也是在全局内存中，纹理内存是对二维空间局部性的优化，感觉是对 2D 图片专门设计的。

7.11.7 全局内存

这是 GPU 上的最大内存，也是延迟最高的内存。和本地内存是一个地方。

```
--device--
```

就可以定义。当多个核函数同时执行，如果使用到了同一个全局变量，可能会存在内存竞争。因为全局内粗的读取特性就访问对齐，也就是说一次他会读取固定大小的内存，所有读取的时候通常要满足两个条件：

- 跨线程的内存地址分布

- 内存事务的对齐方式

一般来说，读取的需求越大，数据的吞吐量就会降低，因为对齐机制会导致模型传输过多的无用数据。

之前看文档的时候，`device memory` 和 `global memory` 是不一样的，但是这里是相同的，`device` 关键字声明的，就是全局。另外，意思就是除了共享缓存，其他的缓存都是在全局内存上。

7.11.8 GPU 缓存

四种缓存

- 一级缓存
- 二级缓存
- 只读常量缓存
- 只读纹理缓存

每一个 SM 都有一个一级缓存，所有 SM 公用一个二级缓存。与 CPU 不同的是，CPU 读写过程都有可能被缓存，但是 GPU 写的过程不被缓存，只有加载会被缓存。

CPU 和 GPU 的最大区别就在于缓存一致性，CPU 中当你更新了一个缓存，那么其他核的缓存也要跟着更新，这很大程度上限制了 cpu 核的数量。但 GPU 是不会通知你的。

我猜测一下这个读取过程，现在也没时间去弄明白他了。读取的时候应该是一个线程只更新他的的 L1 缓存，也不会通知其他线程的缓存。所以很可能出现，在内存中相同地址的数据，在两个线程的缓存不一样，可能 A 线程更新了他自己的缓存，并且写回了内存，但是 B 线程在读取的时候发现这个地址的数据在一级缓存中了，就不去内存里找，这导致 B 线程读取的还是原来的数据。

7.11.9 静态全局内存

和 CPU 一样，动态分配在堆上进行，静态分配在栈上进行。动态分配一般都需要 new 这些关键字。

```
#include <cuda_runtime.h>
#include <stdio.h>
__device__ float devData;
__global__ void checkGlobalVariable()
{
```

```
printf("Device: The value of the global variable is %f\n",devData);
devData+=2.0;
}
int main()
{
    float value=3.14f;
    cudaMemcpyToSymbol(devData,&value,sizeof(float));
    printf("Host: copy %f to the global variable\n",value);
    checkGlobalVariable<<<1,1>>>();
    cudaMemcpyFromSymbol(&value,devData,sizeof(float));
    printf("Host: the value changed by the kernel to %f \n",value);
    cudaDeviceReset();
    return EXIT_SUCCESS;
}
```

首先在全局内存定义一个变量，devData，他前面的关键字 device 就说明他是在全局内存了。这个时候 devData 对于 Host 主机来说就是标识而已，是无法访问的。cudaMemcpyToSymbol(devData,&value,sizeof(float)); 就是把 host 的变量复制过去。常量内存是在 global memory 里面，所以一开始创建的变量在全局内存应该是没问题的。这个声明只是告诉 host 说我在全局内存中有一个变量，这个变量就是常量变量，存储在常量内存中。然后在复制的过程中找到在全局内存的常量内存，复制过去。

主机代码不能直接访问设备变量，设备也不能访问主机变量，这就是 CUDA 编程与 CPU 多核最大的不同之处。如果非要这样做，只能说先获得 gpu 中这个常量的地址：

```
float *dptr=NULL;
cudaGetSymbolAddress((void**)&dptr,devData);
```

拿到地址之后根据地址复制过去

```
cudaMemcpy(dptr,&value,sizeof(float),cudaMemcpyHostToDevice);
```

其实我有个疑问，全局变量和常数变量是怎么区分的？他这里一开始就在全局内存里面了，那怎么知道这个变量是常量呢？我觉得代码他可能写错了，这个 cudaMemcpyToSymbol 的函数他应该根据你这个字节，找到常量内存中的一个区域，把值放进去，再把指针返回。

7.12 内存管理

主要了解两个内容

- 如何分配释放设备内存
- 如何再主机和设备之间进行传输内存

7.12.1 内存分配和释放

内存分配

```
cudaError_t cudaMalloc(void ** devPtr, size_t count)
```

第一个参数是一个指针的指针。先声明后调用

```
float * devMem=NULL;  
cudaError_t cudaMalloc((float**)&devMem, count)
```

devMem 是一个指针，初始化指向了 NULL，cudaMalloc 修改 devMem 的值指向一个分配好的在 cuda 的内存，如果把 devMem 直接传递，他还是 null。这其实和 C++ 的引用有点像，你要改变一个变量，你要传一个地址，这是在 C 语言的情况，如果是 C++，那就是引用。

内存初始化可以用

```
cudaError_t cudaMemset(void * devPtr, int value, size_t count)
```

不过要求初始化的内存要处在 gpu 上。内存释放就是 cudafree。

7.12.2 内存传输

C++ 分配完之后就可以直接写了，但是这里还需要把数据传到 gpu 上去。

```
cudaError_t cudaMemcpy(void *dst, const void * src, size_t count, enum cudaMemcpyKind  
kind)
```

主要是后面那个传输类型，分成几种

- cudaMemcpyHostToHost，cpu 到 cpu，不明白为什么要这个，直接赋值或者深度拷贝不就行了吗？
- cudaMemcpyHostToDevice，cpu 到 gpu 的传输
- cudaMemcpyDeviceToHost，gpu 到 cpu 的传输
- cudaMemcpyDeviceToDevice，gpu 到 gpu 的传输

基本上并行编程都需要 cpu 到 gpu, gpu 再回到 cpu 输出。CPU 到 GPU 是总线传输, 8GB/s, 但是 GPU 到 GPU 内存是 144GB/s, DDR5 内存。所以减少 device 和 host 之间的 IO。

7.12.3 固定内存

HOST 是分页内存管理, 把一些物理内存分成页, 然后一个引用程序给一大块内存。对于应用来说, 他们看到的地址永远是虚拟的, os 会把物理地址映射到虚拟地址上, 当内存不够, os 可以随时调整这个物理地址。但是把数据从 host 到 device 这个过程是需要很长时间, 其中经过总线还需要调度。所以如果你传输的时候, 页面发生移动, 那么就出现了问题了。所以在传输数据前, 会先锁页, 再传输。

cudaMallocHost

之前没看懂, 现在回来补一下, 锁页内存解决的问题, 是 host 拷贝到 device 花费时间长的问题, 原先的 host 到 device 上需要经历两个阶段, 第一个阶段是可分页内存到锁页内存上, 第二个阶段是锁页内存到 device。那么锁页内存技术就是直接把变量分配在锁页内存上, 只有拷贝只需要一次了。需要注意的是, 锁页内存设备是可以直接访问的, 当他是存储在 cpu 上

```
CHECK(cudaMallocHost((float**)&a_d,nByte));
CHECK(cudaMallocHost((float**)&b_d,nByte));
CHECK(cudaMallocHost((float**)&res_d,nByte));

initialData(a_h,nElem);
initialData(b_h,nElem);

CHECK(cudaMemcpy(a_d,a_h,nByte,cudaMemcpyHostToDevice));
CHECK(cudaMemcpy(b_d,b_h,nByte,cudaMemcpyHostToDevice));

dim3 block(1024);
dim3 grid(nElem/block.x);
sumArraysGPU<<<grid,block>>>(a_d,b_d,res_d);
printf("Execution configuration<<%d,%d>>\n",grid.x,block.x);
```

ad 和 bd 都是在 cpu 上, 可以直接当成是 gpu 的变量访问。

但我觉得这个 cudaMallocHost 的作用应该是用来加快 cpu 到 gpu 的复制的, 这个示例代码从 cpu 到 cpu 有啥用呀?

7.12.4 零拷贝内存

前面已经提到过，host 不能访问 device 的内存，device 也不能拷贝 host 的内存，但是随着 cuda 版本和 gpu 的迭代，现在是可以做到了。GPU 线程是可以直接访问零拷贝内存，这部分在 host 的内存中，使用零拷贝内存有下面几种情况：

- 当设备内存不足的时候，可以利用主机内存
- 避免主机和内存之间的显示内存传输
- 提高总线的传输

设备和主机如果同时访问同一个设备地址，我们需要注意内存竞争。

首先，零拷贝内存是固定内存，不可分页。

```
cudaError_t cudaHostAlloc(void ** pHost, size_t count, unsigned int flags)
```

最后一个标志符，有如下选择

- cudaHostAllocDefalt：这个函数和 cudaMallocHost 一样，在固定页面分配内存
- cudaHostAllocPortable：这个函数是返回能被 cuda 使用的固定内存
- cudaHostAllocWriteCombined：高效的内存
- cudaHostAllocMapped：产生零拷贝内存

虽然写着是 cuda，但这部分的变量是在 cpu 的锁页内存上。一般的 gpu 上分配的内存默认都是锁页内存，而 CPU 上的内存分配一般都是可分页的。

零拷贝内存虽然不需要显式的传递到设备上，但是设备还不能通过 pHost 直接访问对应的内存地址，设备需要访问主机上的零拷贝内存，需要先获得另一个地址，这个地址帮助设备访问到主机对应的内存。

但我个人觉得没有什么用吧？你要访问主机的内存，最终还是要走总线，不还是慢吗？

7.12.5 统一虚拟寻址

统一虚拟寻址，也就是 os 会把 gpu 和 cpu 的物理地址映射到同一套虚拟地址上。

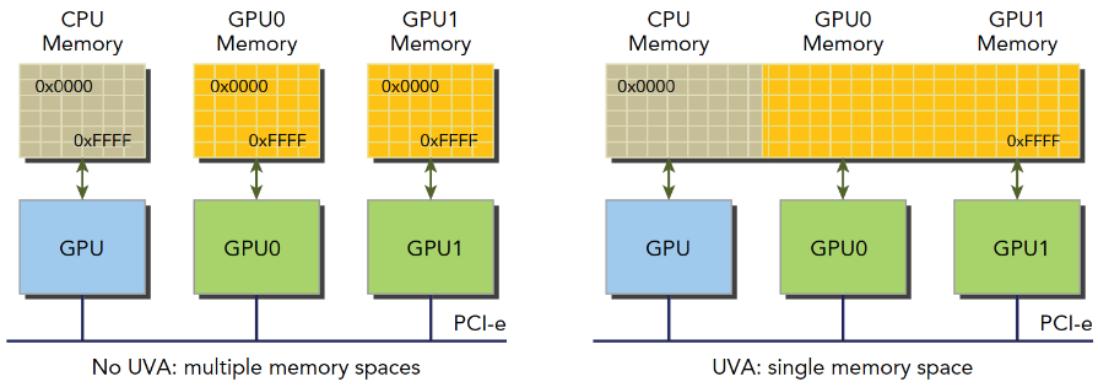


FIGURE 4-5

图 52: 统一虚拟寻址

如图 52 所示，右边就是统一寻址。感觉这部分内容我还是不怎么会用到。
之前如果要获得零拷贝内存的指针函数

`cudaHostGetDevicePointer`

但是统一虚拟地址之后这个函数没用了，直接就可以使用

```

float *a_host,*b_host,*res_d;
CHECK(cudaHostAlloc((float**)&a_host,nByte,cudaHostAllocMapped));
CHECK(cudaHostAlloc((float**)&b_host,nByte,cudaHostAllocMapped));
CHECK(cudaMalloc((float**)&res_d,nByte));
res_from_gpu_h=(float*)malloc(nByte);

initialData(a_host,nElem);
initialData(b_host,nElem);

dim3 block(1024);
dim3 grid(nElem/block.x);
sumArraysGPU<<<grid,block>>>(a_host,b_host,res_d);

```

开头先在 cpu 的锁页内存上创建两个变量，分配空间，另一个变量是在 cuda 上，然后直接就可以把锁页内存上的变量传递到核函数中。而之前没统一寻址是不行的，需要

`CHECK(cudaHostGetDevicePointer((void**)&b_dev,(void*) b_host,0));`

找到内存。cudaHostAlloc 和 cudaHostAlloc 感觉没区别了，都可以被 gpu 直接访问，而且也是在锁页内存上。

后面还有一个统一内存寻址，先放一下。

7.12.6 总结

总结一下：

- 锁页内存：锁页内存解决的问题 cpu 和 gpu 之间的数据传输问题。可以由 cudamallocHost 和 cudaHostAlloc 创建，这两个函数都是创建在锁页内存上，并且可以被 gpu 和 cpu 访问。
- 零拷贝内存：也是解决 cpu 和 gpu 之间的数据传输，相比锁页内存来说，他是想彻底解决，锁页内存只是想把二阶段的拷贝编程一阶段，他是想彻底不要拷贝。但是出现了统一寻址之后这两者似乎没区别了。

这部分内容还有点迷糊，还需要看看。这一张主要观点就是锁页内存更快。

7.13 内存的访问模式

CUDA 的执行模型是以 warp 为单位，内存访问也是以 warp 为基本单位发布和执行的，存储也是一致。

7.13.1 对齐与合并访问

全局内存是一个逻辑层面的模型，编程的时候需要思考两种模型，一种是逻辑层面，比如写 grid, block 这些；另一种是硬件角度的，比如电信号等。核函数从全局内存中读取数据只有两种粒度的

- 128 字节
- 32 字节

用哪个粒度还是要看读取方式了，如果是使用一级缓存那就是 128，否则就是 32。**SM** 是通过 warp 进行线程的访问内存，当一个 warp 执行的某个线程需要读取内存，那么其他在 warp 的线程也是需要访问内存的，如果当前一个线程要读取一字节，那么整个 warp 就要读取 32 字节，这也是为什么最低就是 32 字节的原因。在优化内存的时候，需要关注两个特性

- 对齐内存访问

当内存访问地址是缓存粒度的偶数倍的时候，比如 64, 256，称为对齐访问。

- 合并内存访问

当一个 warp 内的线程访问的内存都在一个内存块里的时候，就会出现合并访问。

最理想的情况是对齐内存访问，对齐是指刚刚好就在一个地址段内访问，合并是指都在同一个 128 或者 32 大小的字节内存块内。例子如图 53 所示。

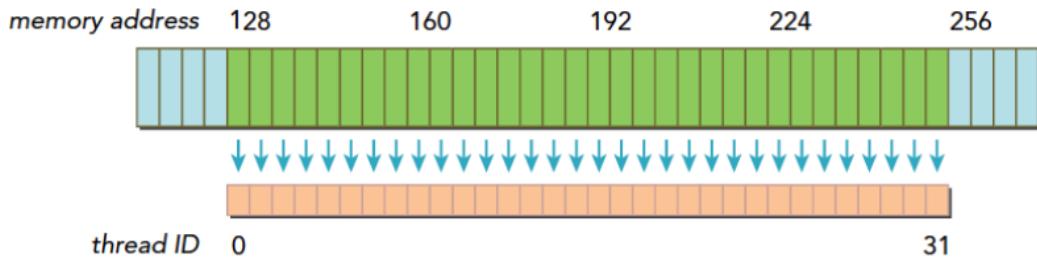


FIGURE 4-7

图 53: 访问缓存

- 一个 warp 加载数据，在加载的时候使用一级缓存。如果是对齐合并访问，那就像 53 一样一次读取就完成了。
- 如果一个事务不是在一个对齐的地址段上，会出现两种情况：
 - 连续但不在一个对齐的段上。比如要访问 1-128，首先起始地址就不是 32, 128 的偶数倍，那么第一次要传 0-127，第二次要传 128-255 两次传递。
 - 不连续的，页不在一个对齐段上，比如 0-63, 128-191，这个也是第一次传 0-127, 128-255 就行。
 - 完全分散开如图 54 所示，所需数据需要三个不同的内存块上，需要读取三次。极端情况下如果刚刚好 32 个线程需要的全部都是在不同块，那基本就要读 32 次了。

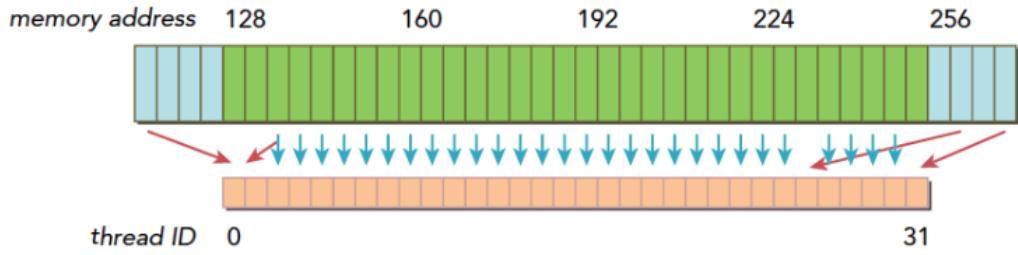


FIGURE 4-8

图 54: 分散读取

所以，优化的核心在于用最少的事务次数满足最多的内存请求。

前面在交错归并的时候就使用到了这点。

7.13.2 全局内存读取

SM 加载数据，根据不同设备和类型分成三种类型

- 一级和二级缓存
- 常量缓存
- 只读缓存

当启用一级缓存，SM 全局加载请求首先会尝试一级缓存，如果一级缓存缺失，就进入二级缓存，如果二级缓存缺失，就直接找 DRAM 让他读取并且同时载入缓存。

内存加载可以分成两类：

- 缓存加载
- 没有缓存的加载

7.13.3 缓存加载

首先需要注意的是：CPU 中的缓存是不可编程的，必须要用上；再 GPU 中可以编辑一级缓存。也就是说无论你咋用访问总是要读取缓存的，区别就是一级还是二级，如果是一级就 128 二级就 32.

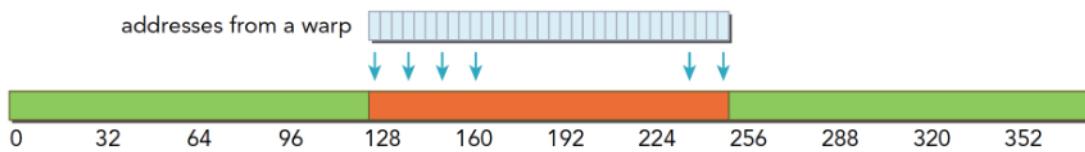


FIGURE 4-9

图 55: 一级缓存对齐合并访问

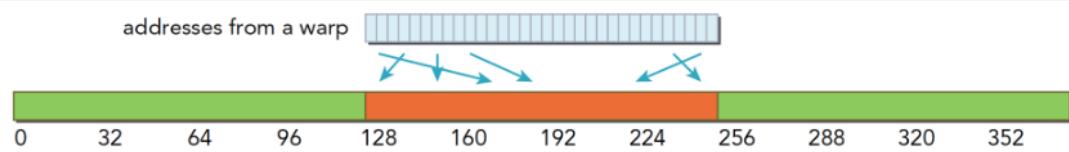


FIGURE 4-10

图 56: 一级缓存对齐合并但不连续访问

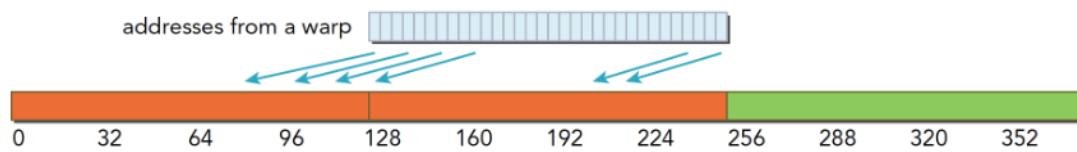


FIGURE 4-11

图 57: 一级缓存连续非对齐的访问

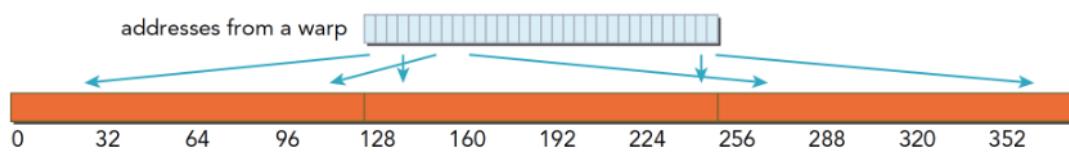


FIGURE 4-13

图 58: 每个线程束内的线程请求的都是不同的缓存行内

- 对齐合并的访问，如图 55 所示。

- 有一级缓存，那就是 128，对齐是指刚刚好指令是挨个挨个地址逐级递增的，一号线程对应一号地址，二号对应二号这样，并且所读取的变量刚刚好也是在同一个内存块上。这个时候只需要读取一次就好了。（这里默认是每一个线程读取 4 个字节，32 个刚刚好就是 128）
- 对齐不连续的，如图 56 所示。
 - 虽然是刚刚好对齐，但是并没有连续，但这样起始和对齐合并情况一样的，线程按照自己的需求检索就行
- 连续非对齐的访问，如图 57 所示。
 - 这是需要多次缓存了，跨多少个块就多少次内存事务了。
- 如果都同一个地址那也是一样的情况。
- 如果都是不同的地址，如图 58 所示。
 - N 个线程就 N 次访问了，这种情况是最坏的。

CPU 和 GPU 的一级缓存有显著的差异，GPU 的一级缓存可以通过编译选项等控制，CPU 不可以，而且 CPU 的一级缓存是的替换算法是有使用频率和时间局部性的，GPU 则没有。

7.13.4 没有缓存的加载

没一级缓存了，这个时候就 32 粒度了。和前面 128 的区别在于，不理想的情况下，读取的字节会少不少。

7.13.5 只读缓存

只读缓存也支持使用全局内存加载代替一级缓存，可以通过只读缓存从全局内存中读取数据。两种方式从只读缓存读取：

- 使用函数 `_ldg`
- 在间接引用指针上使用修饰符

只读缓存是只能由 cpu 写入，gpu 读出的。

7.13.6 全局内存写入

写入和读取完全不同，写入简单一点。**一级缓存不能用在 Fermi 和 Kepler GPU 上进行存储操作**，也就是说他只会经过二级缓存。其他的情况就和上面一样了。

7.13.7 性能调整

优化设备的两个目标

- 对齐合并内存访问，减少带宽浪费，确保每一次读取的数据尽可能的用上
- 足够的并发，隐藏指令，内存延迟

前面提到的优化指令的方式

- 增加每一个线程中执行独立内存操作的数量，这个就是循环展开了
- 增加 SM 并行性

对于第一点，用一个向量加法的例子：

```
--global__ void sumArraysGPU(float *a, float *b, float *res, int n) {
//int i=threadIdx.x;
int i = blockIdx.x * blockDim.x + threadIdx.x;
if (i < n)
    res[i] = a[i] + b[i];
}
```

每一个线程负责一次对应元素的加法。展开循环

```
--global__ void sumArraysGPU_expand(float *a, float *b, float *res, int offset,
const int size) {
// 相当于把每一个线程都往前移动了4个block
int i = threadIdx.x + blockIdx.x * blockDim.x * 4;
int k = i + offset;
if (k + 3 * blockDim.x < size) {
    res[i] = a[k] + b[k];
    res[i + blockDim.x] = a[k + blockDim.x] + b[k + blockDim.x];
    res[i + blockDim.x * 2] = a[k + blockDim.x * 2] + b[k + blockDim.x * 2];
    res[i + blockDim.x * 3] = a[k + blockDim.x * 3] + b[k + blockDim.x * 3];
}
}
```

每一个线程完成四次加法，增大了每一个线程的内存吞吐量。不过这里优化不应该把缓存读取考虑进去吗？这里的 block 默认 512，展开的时候，在 $res[i + blockDim.x]$ 和 $res[i + blockDim.x * 2]$ 都没法在同一个内存里面。但是如果缩小，并行性又不够了。

7.14 核函数可达到的带宽

多个 SM 需要工作，需要什么数据就需要从内存里面拉取，这些内存中获取的数据就会从总线传输到 SM。

前面提到两种技术

- 最大化 Warp 的数量，循环展开来隐藏内存延迟，维持更多的正在执行的内存访问使得总线利用率提高。
 - 最大化 warp 的数量提高并行性，每个线程如果需要读取内存，那么总线利用率会提高
 - 循环展开本质上是一种综合技术，首先是减少指令消耗，用更少的指令做更多的事情；其次减少循环判断；增加独立调度指令提高内存吞吐量。
- 通过适当的对齐和合并访问，提高带宽效率

主要还是通过吞吐量优化。

7.14.1 内存带宽

并行对内存带宽很敏感，SM 并行了很多线程，但是因为数据没到，导致没法工作。所以去哪请求内存中数据的安排方式和 warp 的访问方式都对带宽有显著影响，一般有两种带宽

- 理论带宽：这个是由硬件限制了
- 有效带宽：核函数是实际达到的带宽，测量的实际带宽：

$$\text{有效带宽} = \frac{\text{(读字节数 + 写字节数)} * 10^{-9}}{\text{运行时间}}$$

如何通过调整核函数提高有效带宽。

7.14.2 矩阵转置

在 cpu 中实现：

```
void transformMatrix2D_CPU(float * MatA, float * MatB, int nx, int ny)
{
    for(int j=0; j<ny; j++)
    {
        for(int i=0; i<nx; i++)
        {
```

```
    MatB[i*ny+j]=MatA[j*nx+i];  
}  
}  
}
```

矩阵转置如图 59所示。

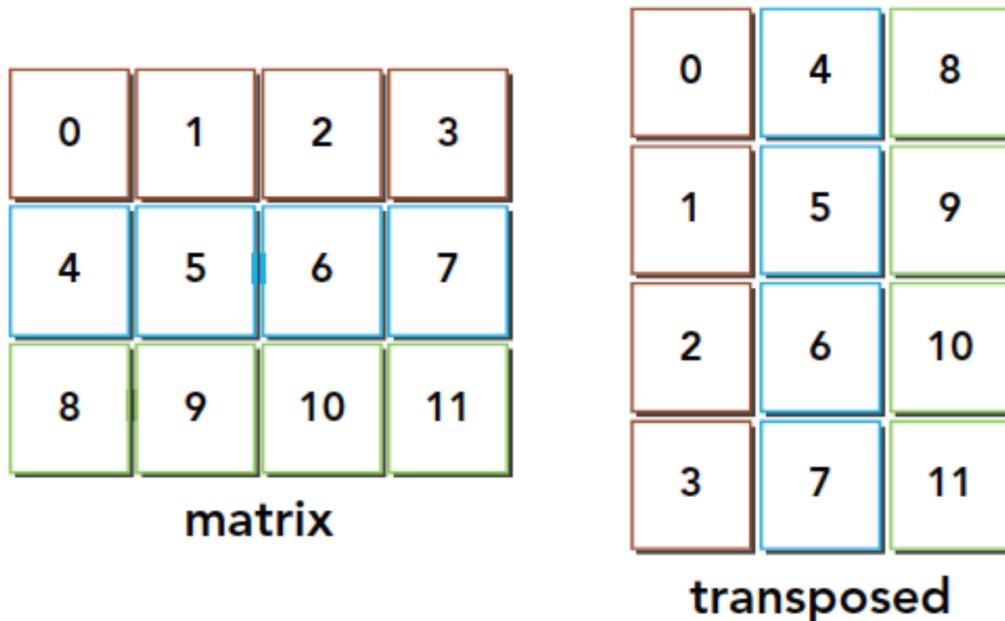


图 59: matrix transpose

这是串行的解决方法，但所有的数据在硬件层都是一维排布的，如图 60所示。

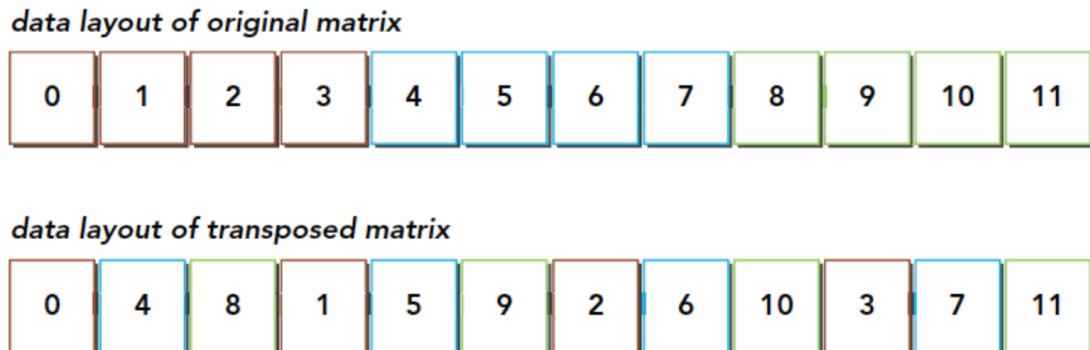


图 60: matrix 的存储结构

由此可以得到两个结论:

- 读取: 原矩阵行进行读取, 请求的内存是连续的, 并且是可以合并访问
- 写入: 写入到转置矩阵的时候是交叉写入, 访问是交叉的

交叉访问是使得内存访问变差的罪魁, 图中 59看不出什么, 就算不用一级缓存, 只用二级也是能存 32 个字节, 当你读取第一个位置的时候, $32/4 = 8$ 个元素的都会丢进来。后面的 01234567 都会放进来, 也就是说每两个元素就需要和内存做一个 IO, 和 global device 做一个 io。

补充一下一级缓存, 在核函数加载请求的时候, 会先去一级缓存看看有没有, 有的话就命中了, 这和 cpu 缓存一样, 在缓存中命中了就不需要再内存里面读取了。在 GPU 中, 列读是不太合理的, 读取之后这一行的数据会存储在缓存中, 后面的数据会被使用。一次读取 128 字节, 其中只有一个单位是有用的, 但剩下不会被立刻覆盖, 因为一个缓存是几 k 甚至更大。所以按列读取数据虽然只用了一个, 但是下一列的时候所有读取元素都在一级缓存中。

7.14.3 转置上下限

行读行存是上限, 列读列取就是下限了。如图 61 和 62 所示。

```
6.69% 742.39us      1 742.39us 742.39us 742.39us copyRow(float*, float*, int, int)
```

图 61: 行读行存核函数时间



图 62: 列读列存核函数时间

上限是 742.39ns，下限是 3.7079ms。

7.14.4 朴素转置：读取行与读取列

考虑转置

```
--global__ void transformNaiveRow(float* MatA, float* MatB, int nx, int ny)
{
    int ix = threadIdx.x + blockDim.x * blockIdx.x;
    int iy = threadIdx.y + blockDim.y * blockIdx.y;
    int idx_row=ix+iy*nx;
    int idx_col=ix*ny+iy;
    if (ix < nx && iy < ny)
    {
        MatB[idx_col]=MatA[idx_row];
    }
}

--global__ void transformNaiveColumn(float* MatA, float* MatB, int nx, int ny)
{
    int ix = threadIdx.x + blockDim.x * blockIdx.x;
    int iy = threadIdx.y + blockDim.y * blockIdx.y;
    int idx_row=ix+iy*nx;
    int idx_col=ix*ny+iy;
    if (ix < nx && iy < ny)
    {
        MatB[idx_row]=MatA[idx_col];
    }
}
```

两种不同方式

- `transformNaiveRow` 行读列写， 2.8465ms
- `transformNaiveColumn` 列读行写， 1.5917ms

列读的效果更好，

GPU 没有维护时间局限性，只是维护了空间的一致性，这导致缓存不会因为使用频率的增加而增加变量存储的时间。如图 63 和 64 所示。

Kernel: transformNaiveColumn(float*, float*, int, int)	gld_throughput	gst_throughput	Global Load Throughput	322.066GB/s	322.066GB/s	322.066GB/s
1			Global Store Throughput	49.064GB/s	49.064GB/s	49.064GB/s

图 63: transformNaiveColumn throughput

Kernel: transformNaiveRow(float*, float*, int, int)	gld_throughput	gst_throughput	Global Load Throughput	27.439GB/s	27.439GB/s	27.439GB/s
1			Global Store Throughput	180.116GB/s	180.116GB/s	180.116GB/s

图 64: transformNaiveRow throughput

列读的吞吐量更高，但是行写的吞吐量就低；行读的吞吐量更低，但是列写的吞吐量就高。我感觉可能是这个列读的时间更小，和写入关系不大，反正写入也不会经过缓存。

7.14.5 展开转置

增加指令的吞吐量，让一个指令可以执行更多的操作

```
--global__ void transformNaiveRowUnRolling(float* MatA, float* MatB, int nx, int ny)
{
    int ix = threadIdx.x + blockDim.x * blockIdx.x * 4;
    int iy = threadIdx.y + blockDim.y * blockIdx.y;
    int idx_row=ix + iy*nx;
    int idx_col=iy + ix*ny;
    if (ix < nx && iy < ny)
    {
        MatB[idx_col]=MatA[idx_row];
        MatB[idx_col + ny * 1* blockDim.x] = MatA[idx_row + blockDim.x * 1];
        MatB[idx_col + ny * 2* blockDim.x] = MatA[idx_row + blockDim.x * 2];
        MatB[idx_col + ny * 3* blockDim.x] = MatA[idx_row + blockDim.x * 3];
    }
}
```

```

__global__ void transformNaiveColumnUnRolling(float* MatA, float* MatB, int nx, int
ny)
{
    int ix = threadIdx.x + blockDim.x * blockIdx.x;
    int iy = threadIdx.y + blockDim.y * 4 * blockIdx.y;
    int idx_row=ix + iy * nx;
    int idx_col=ix * ny + iy;
    if (ix < nx && iy < ny)
    {
        MatB[idx_row] = MatA[idx_col];
        MatB[idx_row + nx * 1 * blockDim.y]=MatA[idx_col + 1 * blockDim.y];
        MatB[idx_row + nx * 2 * blockDim.y]=MatA[idx_col + 2 * blockDim.y];
        MatB[idx_row + nx * 3 * blockDim.y]=MatA[idx_col + 3 * blockDim.y];
    }
}

```

每一个线程干四个 block 的转置。结果好像不是很好

- transformNaiveRowUnRolling:2.7876ms
- transformNaiveColumnUnRolling:2.2544ms

transformNaiveRowUnRolling 是比之前好了，但是 transformNaiveColumnUnRolling 差了，我觉得可能是展开方式。我这里是列的方向做的 unroll，也就是一次会跳过几列，导致你缓存的数据可能被跳过了。所以列是不能跳过，行可以跳

```

__global__ void transformNaiveColumnUnRolling(float* MatA, float* MatB, int nx,
int ny)
{
    int ix = threadIdx.x + blockDim.x * blockIdx.x * 4;
    int iy = threadIdx.y + blockDim.y * blockIdx.y;
    int idx_row=ix + iy * nx;
    int idx_col=ix * ny + iy;
    if (ix < nx && iy < ny)
    {
        MatB[idx_row] = MatA[idx_col];
        MatB[idx_row + 1 * blockDim.x]=MatA[idx_col + 1 * blockDim.x * ny];
        MatB[idx_row + 2 * blockDim.x]=MatA[idx_col + 2 * blockDim.x * ny];
        MatB[idx_row + 3 * blockDim.x]=MatA[idx_col + 3 * blockDim.x * ny];
    }
}

```

}

transformNaiveColumnUnRolling 变成 2.0637ms，还是差了一点。

7.15 使用统一内存的向量加法

统一内存的基本思路就是减少指向同一个地址的指针。经常需要在本地分配内存，然后传输到设备，然后从设备传输回来。使用 cudaMallocManaged 分配内存，这种内存和之前提到了 zero-copy 的效果相似，在 device 和 host 都能访问。

但似乎用起来效率不是很高。

7.16 共享内存和常量内存

shared memory 是少数几个在片上速度很快的内存，如图 65 所示。

存储器	芯片上/芯片外	存取	范围
寄存器	片上	R/W	一个线程
本地	片外	R/W	一个线程
共享	片上	R/W	block 内所有线程
全局	片外	R/W	所有线程+主机
常量	片外	R	所有线程+主机
纹理	片外	R	所有线程+主机 知

图 65: GPU 的内存

7.17 CUDA 共享内存概述

GPU 的内存按照类型口语分成两种

- 板载内存

- 片上内存

全局内存是较大的板载内存，延迟高，共享内存是片上的较小的内存，延迟低，带宽高，本地内存也是在全局内存上。共享内存的用途

- 块内线程通信的通道，因为他对于一个 block 内的所有线程都可见
- 用于全局内存数据的可编程管理的缓存
- 告诉暂存存储器，用于转换数据来优化全局内存访问模式

7.17.1 共享内存

共享内存使得同一个线程块中可以互相协同，便与片上的内存可以被最大化利用。并且，共享内存是可以变成的，可以用代码控制。

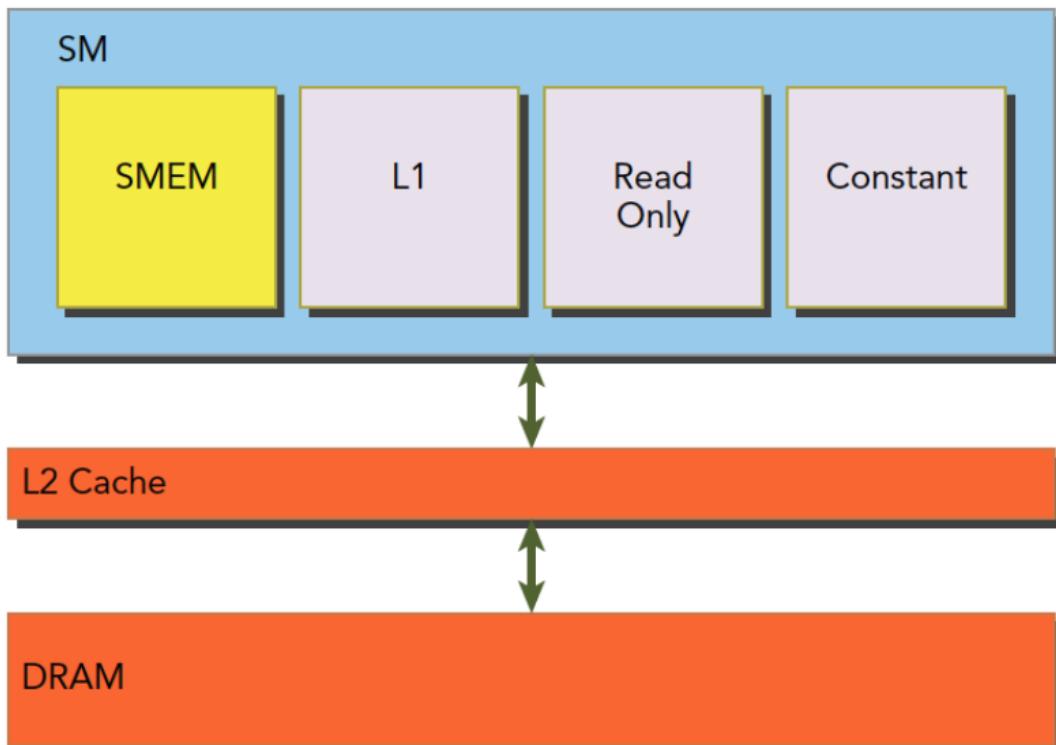


FIGURE 5-1

图 66: 共享内存 在 SM 的结构

如图 66 所示，共享内存是在 SM 里面，所以更接近 cuda core 的 SMEM，L1，read-only memory 和 constant memory 会有更快的速度。共享内存是在他所属的线程块被执行时建立，线程块执行完毕后共享内存释放，线程块和他的共享内存有相同的生命周期。

对于共享内存的访问有三种类型：

- 最理想的情况是每一个线程都访问一个不冲突的共享内存，大家各干各的互不干扰。
- 当线程之间是有访问冲突的适合，多少个线程就需要多少个事务。
- 如果 Warp 的线程全部都访问一个地址，那么这个线程访问完后以广播的形式告诉大家。

7.17.2 共享内存存储体和访问模式

- 内存存储体

共享内存是一个一维的地址空间，和前面的其他内存一样都是线性的，共享内存有一个特殊形式是会分成 32 个同样大小的内存体，可同时访问，分别对应 32 个不同的线程，这些线程在访问的时候如果只访问属于自己的那部分内存，那就可以保证没有冲突，如果内存事务就可以完成，否则就需要多个内存事务了。

- 存储体冲突

当多个线程访问同一个存储体的时候，就会冲突。只要访问同一个存储体就会冲突，访问同一个地址反而不会冲突，因为他会广播开。所以访问存在多种方式

- 并行访问，多地址访问多存储体
- 串行访问，多地址访问同一个存储体
- 广播访问，单一地址访问

并行访问最常见，是效率比较高的一种。如果是完全无冲突，这种方式是最理想的。当有小部分冲突的时候，大部分不冲突的可以同过一个内存事务完成，其他冲突的就另一个事务，这样要两个内存事务，效率低一点。如果是完全冲突就是最差的场景了，所有线程访问同一个存储体，并且还是同一个存储体的不同地址，这个时候就变成串行访问了。

广播访问是出现在所有线程都只访问一个地址的场景下，一个内存事务执行完毕就把这个数据广播出去。所以提升效率的做法就算要把线程所需要的线程块都放到不同的存储体上去。

共享内存的分配模式，如图 67 所示。

Byte address	0	4	8	12	16	20	24	28	32	36	40	44	48	52	56	60	
4-byte word index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	
Bank index	Bank 0	Bank 1	Bank 2	Bank 3	Bank 4	Bank 5	Bank 6	Bank 7	Bank 8	Bank 9	Bank 10	Bank 11	Bank 28	Bank 29	Bank 30	Bank 31
4-byte word index	0	1	2	3	4	5	6	7	8	9	10	11	28	29	30	31
	32	33	34	35	36	37	38	39	40	41	42	43	60	61	62	63
	64	65	66	67	68	69	70	71	72	73	74	75	92	93	94	95
	96	97	98	99	100	101	102	103	104	105	106	107	124	125	126	127

FIGURE 5-5

图 67: 共享内存拆分成存储体

32 个线程就有 32 个桶，如果你的设备计算能力是四字节，那么就以四个字节为单位存储，如果是 8 字节那就 8 个单位存储。这里以四字节为例子。0 单位的 4 字节放到 0 个存储体，1 单位的四字节放到 1 个存储体以此类推，所以 0, 32, 64, 96 就是放在第一个桶里面。比如 103 号， $103 \% 32 = 7$ ，那么就是在第 8 个罐子里面。

- 内存填充

存储体冲突会严重影响共享内存效率，那么当我们遇到严重冲突的情况下，可以使用填充的办法让数据错位，来降低冲突。就是设置多几个位置占位就行。

7.17.3 同步

内存栅栏，所有调用线程必须等到全部内存修改对其余线程可见时才继续进行，也就是强制同步。CUDA 采用宽松的内存模型，也就是内存访问不一定按照他们在程序中出现的位置进行的。宽松的内存模型，导致了更激进的编译器。

目前比较常用的

```
void __syncthreads();
```

他的作用如下

- 保证在一个线程块内的线程都要到达当前位置才能继续往下执行

- 同一个线程块在此同步函数之前的所有操作对于后面的线程都是可见的
- 这个只能解决同一个线程块内的内存竞争的问题
- 使用不当会出现死锁，如果写在 if 这些语句中可能永远走不到，卡死

有三种栅栏

- 线程块内

```
void __threadfence_block();
```

和前面的 syncthreads 没什么区别感觉。

- 网格级内存栅栏

```
--threadfence();
```

- 系统级栅栏，跨系统，包括主机和设备

```
void __threadfence_system();
```

7.18 共享内存的数据布局

7.18.1 方形共享内存

```
--shared__ int x[N][N];
```

这个 x 就是存储在共享内存里面，如果是用 A[threadIdx.x][threadIdx.y]; 访问，那么第一个 warp 是 00, 10, 20, 30 等以此类推，访问的就都是同一个位置了。

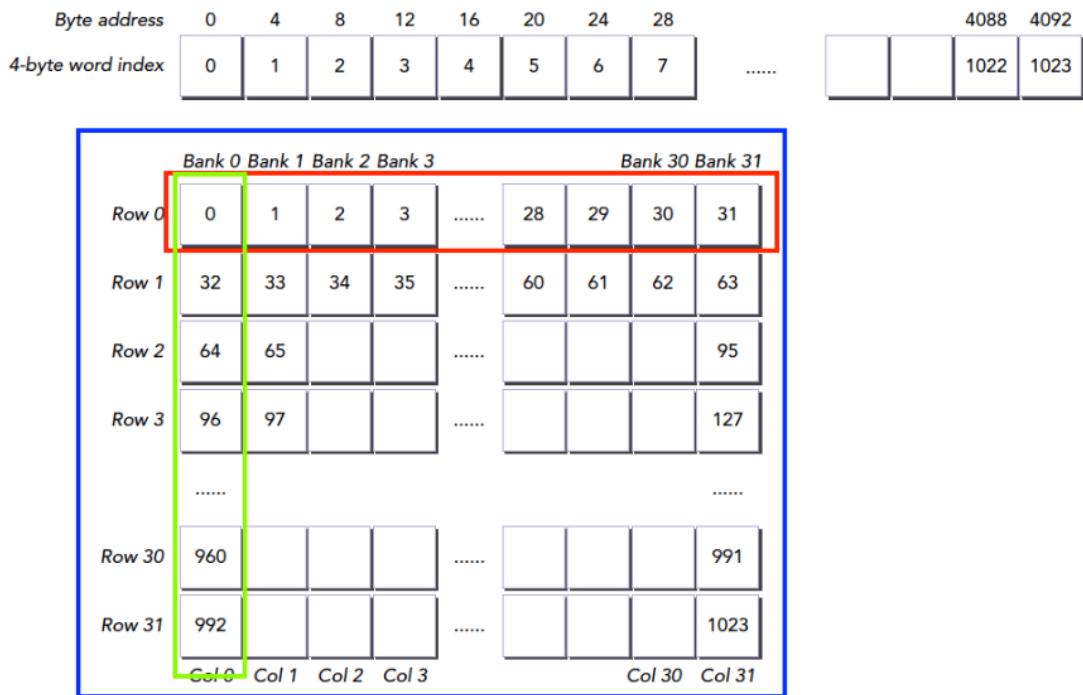


FIGURE 5-12

图 68: 矩阵存储

如图 68 所示，每一个线程基本就是竖着来，访问同一个存储体，这就肯定冲突了。如果是 $A[threadIdx.y][threadIdx.x]$ 那就没有问题了。

7.19 合并的全局内存访问

这是 5.4 节，感觉很重要，但是目前还不太需要就算了。

后面还有流这一章节，听课的时候再看了。

8 AI 葵 cuda 课程笔记

- 地址: https://www.youtube.com/watch?v=l_Rpk6CRJYI&list=PLDV2GyUo4q-LKuiNltBqCKd09GH4SS_ec
- 一共六次课

使用 cuda 的场景

- 非平行运算：比如体渲染，每一条射线采样不同数量的点，每一条射线都要做独立的体渲染计算，就没有办法用到 pytorch 的平行运算。这里感觉很奇怪，你如果采样点数量不同，每一个 thread 就要做 if 条件，那这样一个 warp 出现分支不还是会全部走完？
- 大量的串列计算

8.1 pytorch, C++ 和 CUDA 的关系

pytorch 先呼叫 C++ 的程序，C++ 再去呼叫 CUDA 核函数。所以他们之间的关系

$$pytorch \rightarrow C++ \rightarrow CUDA$$

所以重要部分是在 CUDA，C++ 只是一个桥梁。

8.1.1 简单的例子

如何通过 python 调用 cpp。

我们再处理传过来的 tensor 之前，首先得让 cpp 知道 tensor 是什么，首先要引入包

```
#include <torch/extension.h>
```

写一个简单的程序

```
torch::Tensor trilinear_interpolation(
    torch::Tensor feats,
    torch::Tensor point
) {
    return feats;
}
```

传入两个参数，特征和点位置，直接返回特征。有了 cpp 之后就要指定如何用 python 调用。

```
PYBIND11_MODULE(TORCH_EXTENSION_NAME, m){
    // 左边是python如何call这个函数，右边是call哪个函数
    m.def("trilinear_interpolation", &trilinear_interpolation);
}
```

这个是 pybind 包，`trilinear_interpolation` 表示 python 调用函数的名字，第二个参数就是函数地址。

最后就是要准备一个 `setup.py` 编译安装这个包了。主要是两段

```
ext_modules=[  
    CppExtension(  
        name='cppcuda_tutorial',  
        sources=['coarse_1.cpp'])  
],  
# 告诉程序我们需要Build  
cmdclass={  
    'build_ext': BuildExtension  
}
```

指定编译的程序，编成什么名字，引用就引这个名字，第二段是告诉程序要去 build。然后在当前位置 `pip install .` 即可。

这里还有一个坑，在安装 visual studio 的时候，高于 16 版本的注意安装顺序，需要先安装 visual studio 再安装 pytorch。因为高于 16 版本的 visual studio 带的 MSVC 是 17 以上了，17 以上有 bug，你必须装好 MSVC 之后，重新安装 pytorch，要么你就直接 `cmake . -B build` 全部编译。所以这里如果你先装了 pytorch 再装 visual studio 是有问题的，我的是 2022。后面重新装 pytorch 就没问题了。

调用也很容易，先引入 `torch` 这个包，再引 `cppcuda_tutorial`，调用即可。

8.1.2 核函数

host code 调用 device coder，device code 调用不同线程执行相同指令。通常调用路径是：

$$grid \rightarrow block \rightarrow thread$$

直接 `grid` 也是可以调用这个 `Thread` 的，为什么要经过这个 `block`? up 主说是一个 Grid 只能生成 1024 个 `thread`，所以需要 `block` 让他生成 `thread` 数量更多。我感觉不对，几个原因吧

- Thread 之间的通信和 shared memory 的分配，grid 里面所有的线程都进行通信成本高
- 执行线程的单位是 warp，一个 warp 一般是 32 个线程，而一个 warp 只能处理同一个 block 里面的 thread，如果没有 Block，一次调用 grid 就需要准备很多个 warp 去

处理，对于线程调度很困难。

看一个简单的例子：三线性差值

8.1.3 Trilinear Interpolation

现在有一个立方体，这个立方体有八个顶点。现在在立方体内部有一个三维点，根据这个三维点在这个立方体的位置，算出这个点相对这八个点的特征。也就是说八个点的位置和特征分别是 $\{(x_i, f_i)\}_{i=1}^8$ ，求

$$f = \text{trilinear}(f_1, \dots, f_8)$$

用一个 2D 的插值作为解释，如图 69 所示。

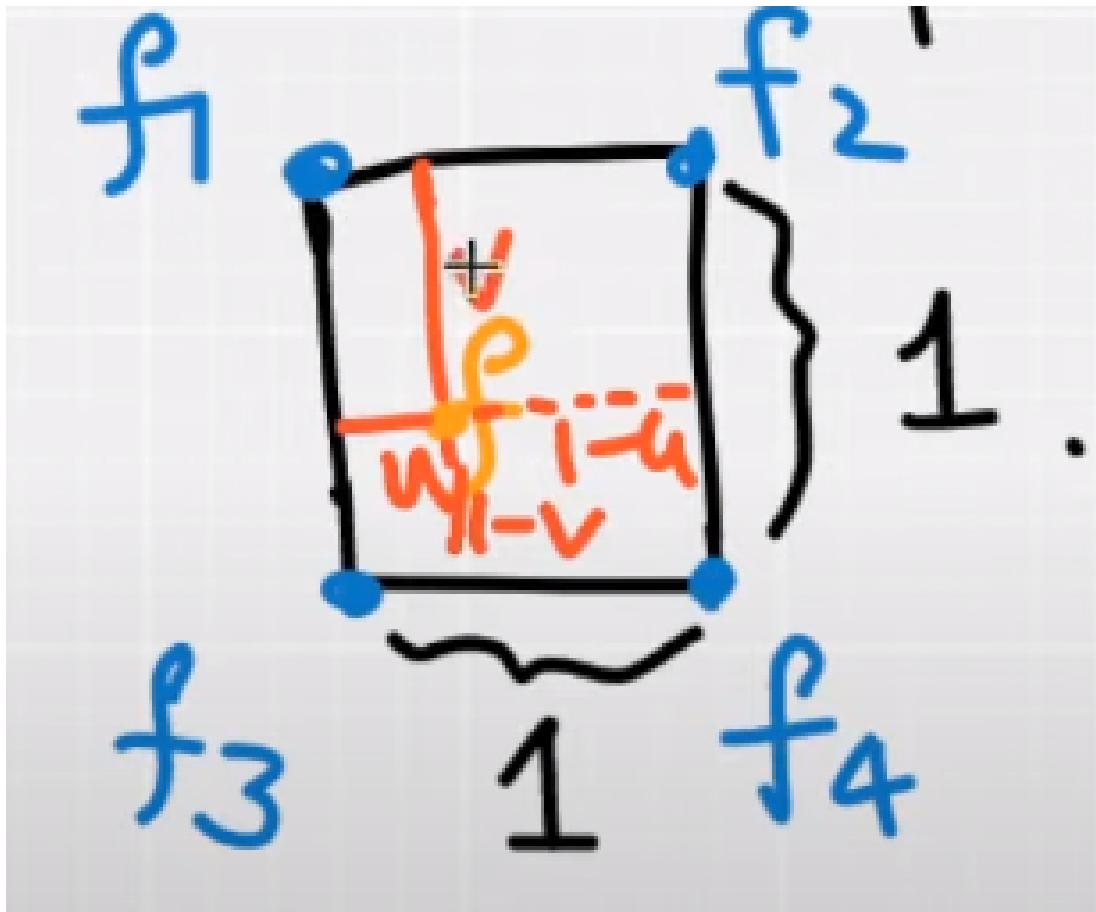


图 69: 2D 的线性插值

计算这四个点对于当前点的贡献。对于 f_1 点，离 f_1 点越近那么贡献越大，所以他的贡献就应该是对面的那面积 $(1-u)(1-v)$ ，以此类推。则

$$f = (uv)f_4 + (1-u)vf_3 + u(1-v)f_2 + (1-u)(1-v)f_1$$

3D 那更是一样，就把面积换成体积就行。

输入: `feats(N, 8, f)`, 一个 N 个立方体要计算, 每一个立方体 8 个点, 每一个点的特征的 `f`. `point(N, 3)`, N 个点对应一个立方体。并没有给出立方体每一个点的坐标, 因为可以通过 `point` 算出来。平行计算的插入点有两个

- N 个点可以平行计算

- 8 个特征也可以并行计算

在写 kernel 前，需要先对输入进行测试，这个比较重要

```
// 检查这个输入是不是再在cuda上
#define CHECK_CUDA(x) TORCH_CHECK(x.is_cuda(), #x " must be a CUDA tensor")
// 检查这个输入是不是连续存储，也就是flatten之后的存储顺序是不是不变。因为cuda的线程模型就是这种contiguous
#define CHECK_CONTIGUOUS(x) TORCH_CHECK(x.is_contiguous(), #x " must be contiguous")
// 检查数据的合理性
#define CHECK_INPUT(x) CHECK_CUDA(x); CHECK_CONTIGUOUS(x)
```

主要检查两个属性

- 是不是在 cuda 上，只有 cuda 上才能调动
- 数据 flatten 之后是不是内存不变，因为在 cuda 的 block 都是这种连续的存储，比如一个二维的矩阵实际在内存是一维的存储，在内存的存储顺序实际上是 00, 01, 02, 03, ..., 10, 11, ...，所以如果你的数据不是 contiguous 的，那么 cuda 的线程索引就不对了。

既然使用了 cuda，那么 setup 也全部都要改成 cuda 了，重新安装就行。

补充：

在初始化 tensor 的时候，如果裡面所有的值都會在 kernel 中被正確填入（大部分情況下是），那用 `torch::empty` 會比 `torch::zeros` 來得更好（速度更快一點）。因為 `torch::zeros` 是先做 `torch::empty` 然後再把所有值歸零，等於是多了一個步驟。

他这里的调用方式也太奇怪了，先 pytorch 调用一个 C++ 程序，这个 C++ 程序又调用一个写在 cu 的 cuda 程序，但这个 cuda 程序又不是并行计算的程序。然后这个写在 cuda 的程序又调用一个 kernel。所以他的调用顺序是：

pytorch → cpp → cpp → kernel

为什么不能直接调用 cpp，让 cpp 直接调用 Kernel 呢？

8.1.4 cpp

pytorch 首先调用 cpp。

```
torch::Tensor trilinear_interpolation(
    torch::Tensor feats,
    torch::Tensor point
) {
    CHECK_INPUT(feats);
    CHECK_INPUT(point);

    return trilinear_fw_cu(feats, point);
}
```

先两个 check 检查存储方式和是否是在 cuda 上。然后直接返回写在 cuda 但不是 Kernel 的函数返回值。

为了让 python 能够调用，需要定义接口

```
PYBIND11_MODULE(TORCH_EXTENSION_NAME, m){
    // 左边是python如何call这个函数，右边是call哪个函数
    m.def("trilinear_interpolation", &trilinear_interpolation);
    m.def("trilinear_interpolation_extend", &trilinear_interpolation_extend);
}
```

用 pybind 绑定。

8.1.5 cu-cpp

然后上面的那个 cpp 又调用一个 cpp 函数。这个函数主要是配置调用核函数的一些条件：

```
const dim3 threads(16, 16);
// 既定公式，计算需要多少个线程
const dim3 block((N - 1) / threads.x + 1, (F - 1) / threads.y + 1);
```

主要是在于调用函数：

```
AT_DISPATCH_FLOATING_TYPES(feats.type(), "trilinear_fw_cu", ([&] {
    trilinear_fw_kernel<scalar_t><<<block, threads>>>(
        feats.packed_accessor<scalar_t, 3, torch::RestrictPtrTraits, size_t>(),
        points.packed_accessor<scalar_t, 2, torch::RestrictPtrTraits, size_t>(),
        feat_interp.packed_accessor<scalar_t, 2, torch::RestrictPtrTraits, size_t>()
    );
}));
```

- AT_DISPATCH_FLOATING_TYPES 表示计算精度，这里是浮点运算，当然也可以改成半精度的计算。
 - 第一个参数表示模板类型，这个宏实际上调用的是一个模板函数。第二个是名称，出问题了会显示哪个出错
- & 表示捕获形式，如果是 = 就是值捕获，& 是引用捕获所有局部变量和参数。这个实际上就是引用捕获而已。
- 然后调用函数。

8.1.6 kernel

kernel 的写法没什么特别。

做了一些优化，主要还是做了展开。

```
cuda : 0.011964559555053711
cuda optimize : 0.007978677749633789
cpu : 0.035902976989746094
True
True
```

不是很明显，快一点吧。我试了一下半精度计算，发现没有影响：

```
cuda : 0.012000083923339844
cuda optimize : 0.007983684539794922
cpu : 0.03489112854003906
True
True
```

然后我用 nvprof 测了不是半精度计算情况下 SM 和内存利用率，如图 70 所示。

		achieved_occupancy	Achieved Occupancy	0.843579	0.843579	0.843579
		gst_efficiency	Global Memory Store Efficiency	12.50%	12.50%	12.50%
Kernel1:	void trilinear_fw_kernel<float>(at::GenericPackedTensorAccessor<float, __int64=3, at::RestrictPtrTraits, __int64>, at::Gen-					
	or<float, __int64=2, at::RestrictPtrTraits, __int64>)					
Kernel2:	void trilinear_fw_kernel_extend<float>(at::GenericPackedTensorAccessor<float, __int64=3, at::RestrictPtrTraits, __int64>, a-					
	rAccessor<float, __int64=2, at::RestrictPtrTraits, __int64>)					
		achieved_occupancy	Achieved Occupancy	0.845705	0.845705	0.845705
		gst_efficiency	Global Memory Store Efficiency	100.00%	100.00%	100.00%

图 70: kernel assess

发现这个时候 SM 的利用率已经 84% 了，内存利用率已经 100%，所以 SM 算的再快也没用，因为内存已经跟不上了。半精度计算能更快，但是内存利用率 100% 了，算再快内存也跟不上，SM 利用率反而还降下来了。我换了宿舍的 3080 算发现有提速，因为宿舍的内存 16G 并且 DDR4。

我觉得能优化的可能就是这个参数这块，算了一下这个参数，加起来 800M，寄存器存不下全部丢到常量内存里面，也就是 global memory。我想这能不能用 shared memory 解决？

8.1.7 反向传播

主要两个步骤

- 计算所有 output 对于 trainable input 的偏微分
- 用 Torch.autograd.function 进行包装

首先是识别输出是什么，知道哪些是可训练的。trilinear 里面只有特征是需要进行梯度计算的， u , v , w 是完全不需要进行计算。只需要计算 $\frac{\delta f}{\delta f_i}$

```
template <typename scalar_t>
// global 表示调用位置表示在 host 调用，device 和 host 分别表示在 Gpu 和 cpu 上执行
__global__ void trilinear_bw_kernel(
    const torch::PackedTensorAccessor<scalar_t, 2, torch::RestrictPtrTraits, size_t>
        dL_feat_interp,
    const torch::PackedTensorAccessor<scalar_t, 3, torch::RestrictPtrTraits, size_t>
        feats,
    const torch::PackedTensorAccessor<scalar_t, 2, torch::RestrictPtrTraits, size_t>
        point,
    torch::PackedTensorAccessor<scalar_t, 3, torch::RestrictPtrTraits, size_t>
        dL_dfeats
)
{
    // 把线程的局部坐标变成全局空间
    // 去除不需要的线程
    const int n = threadIdx.x + blockDim.x * blockIdx.x;
    const int f = threadIdx.y + blockDim.y * blockIdx.y;
    // 去除无用的线程，这是其中一种写法，
    if (n < feats.size(0) && f < feats.size(2))
    {
        // 归一化，输入的坐标都是 -1 到 1，归一化到 0-1 之间
        const scalar_t u = (point[n][0] + 1) / 2;
```

```

const scalar_t v = (point[n][1] + 1) / 2;
const scalar_t w = (point[n][2] + 1) / 2;
const scalar_t a = (1-v)*(1-w);
const scalar_t b = (1-v)*w;
const scalar_t c = v*(1-w);
const scalar_t d = 1-a-b-c;

dL_dfeats[n][0][f] = (1 - u) * a * dL_feat_interp[n][f];
dL_dfeats[n][1][f] = (1 - u) * b * dL_feat_interp[n][f];
dL_dfeats[n][2][f] = (1 - u) * c * dL_feat_interp[n][f];
dL_dfeats[n][3][f] = (1 - u) * d * dL_feat_interp[n][f];
dL_dfeats[n][4][f] = u * a * dL_feat_interp[n][f];
dL_dfeats[n][5][f] = u * b * dL_feat_interp[n][f];
dL_dfeats[n][6][f] = u * c * dL_feat_interp[n][f];
dL_dfeats[n][7][f] = u * d * dL_feat_interp[n][f];
}

}

```

主要是最后有一个封装。

```

class Trilinear_interpolation_cuda(torch.autograd.Function):
    @staticmethod
    def forward(ctx, feats, points):
        feat_interp = cppcuda_tutorial.trilinear_iterp(feats, points)
        ctx.save_for_backward(feats, points)
        return feat_interp

    @staticmethod
    def backward(ctx, dL_dfeat_interp):
        feats, points = ctx.saved_tensors
        dL_dfeats =
            cppcuda_tutorial.trilinear_iterp_bw(dL_dfeat_interp.contiguous(),
                                                feats, points)
        # 回传多少个梯度要看你有多少个输入，再这里坐标是没有梯度的
        return dL_dfeats, None

```

这里有一个 `ctx` 是为了存储一些中间值，作为反向传播使用。这里同样做了循环展开的优化，提升很明显。

cuda : 0.012997865676879883

```

cuda op : 0.0069501399993896484
cpu : 0.032912254333496094
True
True
gpu bw : 0.05884075164794922
gpu op bw : 0.02293872833251953
True

```

nvprof 评估一下，如图 71 和 72 所示，SM 的利用率和线程利用率基本都满了，内存利用率显示很低那也没办法，你再高也处理不来。

Kernel: void trilinear_bw_kernel<float>(at::GenericPackedTensorAccessor<float, __int64=2, at::RestrictPtrTraits, __int64>, at::GenericPackedTensorAccessor<float, __int64=2, at::RestrictPtrTraits, __int64>, at::GenericPackedTensorAccessor<float, __int64=3, at::RestrictPtrTraits, __int64>)
1 achieved occupancy Achieved Occupancy 0.843371 0.843371 0.843371
1 sm_efficiency Multiprocessor Activity 99.97% 99.97% 99.97%

图 71: kernel 反向传播

Kernel: void trilinear_bw_kernel_extend<float>(at::GenericPackedTensorAccessor<float, __int64=2, at::RestrictPtrTraits, __int64>, at::GenericPackedTensorAccessor<float, __int64=2, at::RestrictPtrTraits, __int64>, at::GenericPackedTensorAccessor<float, __int64=3, at::RestrictPtrTraits, __int64>)
1 achieved occupancy Achieved Occupancy 0.952857 0.952857 0.952857
1 sm_efficiency Multiprocessor Activity 99.86% 99.86% 99.86%
1 sysmem_utilization System Memory Utilization Low (1) Low (1) Low (1)

图 72: kernel 反向传播 + 循环展开

9 ECE408 课程笔记

UIUC 的 ECE408 cuda 编程课程，7 次作业，2 次 exam。

9.1 Heterogeneous Parallel Computing

- latency device, 比如 CPU
- throughput device, 比如 GPU

CPU 和 GPU 是不同的两种设计，如图 73 所示。

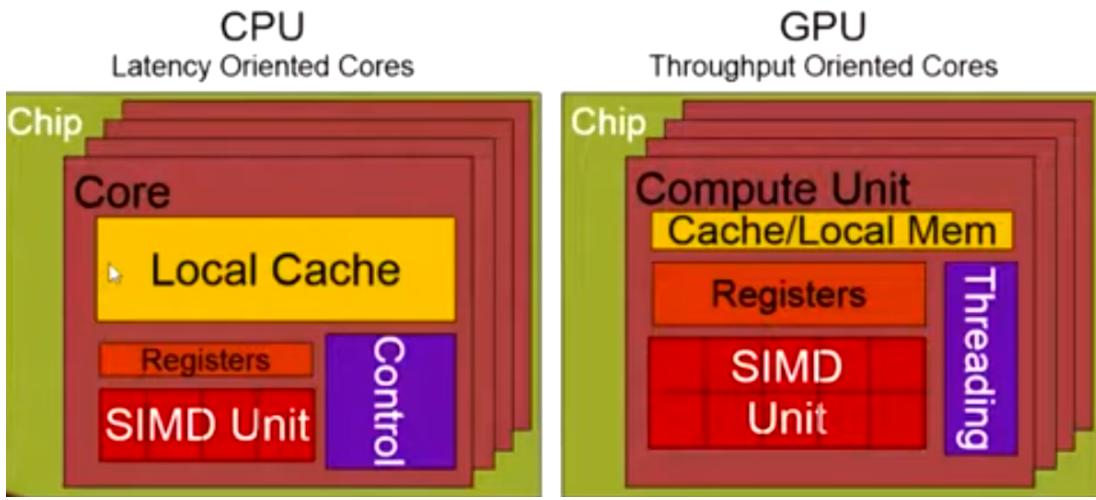


图 73: cpu 和 gpu 的设计

- CPU: 通常具有更大的本地内存，具有的寄存器比较少。CPU 有一块很大的控制单元。
- GPU: 通常具有很小的本地内存，并且通用有更多的寄存器。主要是为了支持大量的线程。同时为了支持线程更多，SIMD 也会更多，也就是处理器。而 GPU 这边是调度器。

go deeper, CPU 的结构如图 74所示。

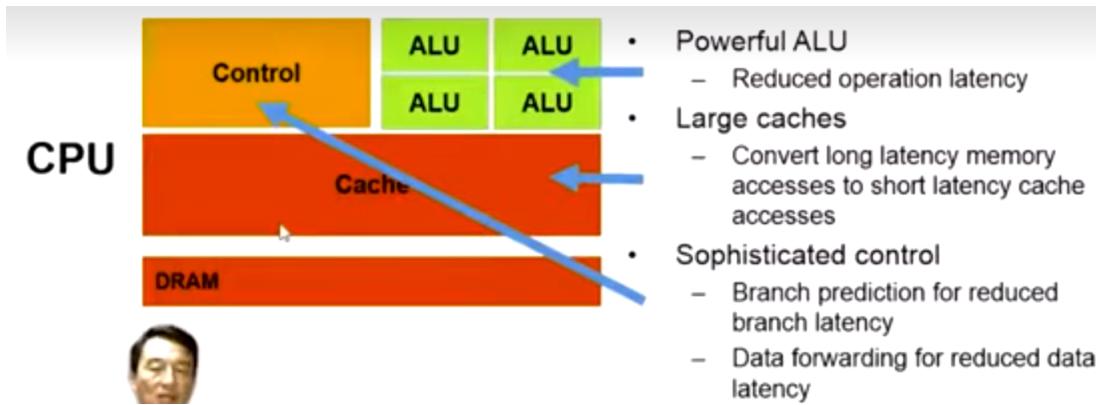


图 74: cpu 的结构

CPU 的三个特点

- Powerful ALU: ALU 数值计算单元非常强。
- Large Caches: 减少内存读取的延时
- Sophisticated Control: 分支预测和数据转发。分支预测可以预测程序分支的走向，GPU 是没这能力的。

第一个解决计算延迟，第二个解决 IO 延迟，第三个解决分支预测的延迟。

GPU 的结构如图 75 所示。

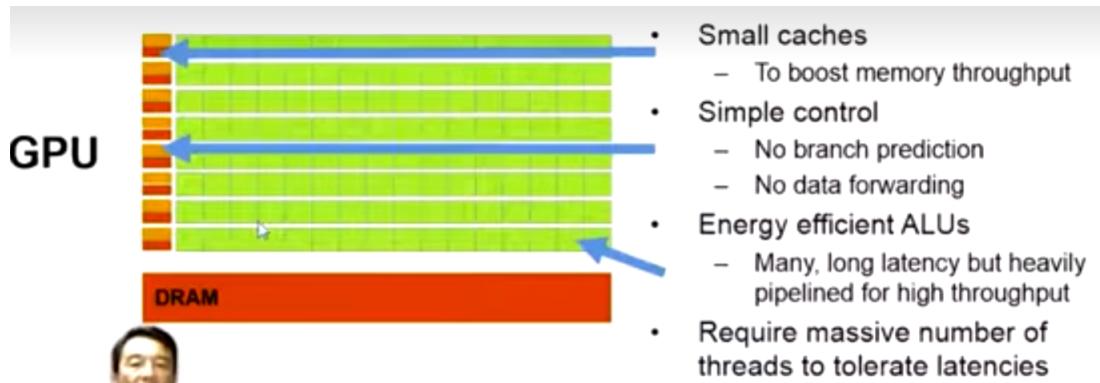


图 75: gpu 的结构

GPU 的特点

- Small Caches: 缓存非常小，增加内存吞吐量。这些缓存是为了线程准备的，老师提到了内存合并，如果 gpu 有一级缓存，那么一次缓存读取是 128 字节，如果、每一个线程刚刚好访问的内容地址都是再这个 128 字节中就可以一起读取。
- Simple Control: 没有数据转发和分支预测，所以最好一组 warp 就不要有分支在里面。
- Energy efficient ALU: gpu 的 ALU 比较差，所以通常需要大量的线程进行并发，这样能提高吞吐量，要不然就一直等，gpu 的 ALU 没有 cpu 那么高效
- massive number of threads: 线程够多

所以这两种机器的设计方向完全不同，一个是冲着延迟去，一个是冲着吞吐量去的。对于一些具有复杂逻辑的串行代码，CPU 是 GPU 的十倍加速。对于一些并行代码，GPU 是 CPU 的十倍速度。