CacheLab-Report

学号：21302010042

姓名：侯斌洋

实验过程：

（1）首先进行环境搭建，在 VMware 上安装 Ubuntu20.04 系统，并在此系统上利用 vscode 完成本次 lab。然后阅读附带的文档了解本次 lab 的整体要求以及一些注意事项。

（2）接下来是关于 PartA 以及 PartB 实现的一些细节，包括代码思路以及遇到的问题等。此外，Lab 托管在 GitHub 上，并且每次进行修改后均会上传，可访问 https://github.com/HBY-STAR/ICS_lab/tree/master/cachelab-handout 查看整个实验进行的过程细节以及修改记录。

①PartA：

本实验部分要求实现一个高速缓存模拟器，要求能够识别并处理数据操作并计算每次处理过程中的 hit、miss、以及 eviction 数。

由于该模拟器在运行时要处理命令行参数，根据 PDF 提示包含<getopt.h>头文件，于是去查询 getopt 函数的用法并通过 while 加 switch 语句实现对命令行参数的解析，详见代码中的 void get_argvs(int argc, char **argvs, int *s, int *E, int *b, FILE **fp, bool *isDisplay)函数。

然后是实现高速缓存结构，如下分别定义了行、组和缓存结构。

```c
//行
typedef struct line
{
    bool isValid;  //有效位
    int tag;       //标记位
    int last_time; //上次访问时间
} line;

//组
typedef line *set;

//缓存
typedef struct cache
{
    int lines_per_set; //每组的行数，即E
    int set_num;       //组数，即S
    int b;             //块偏移位数，即b
    int s;             //组索引位数，即s
    set *sets;         //存储结构
} cache;
```

由于高速缓存的大小是由命令行参数给出的，故这里在定义结构的时候要用指针，然后在运行时再申请空间，申请空间与释放空间的函数如下

```c
//创建缓存，为sets申请空间
cache calloc_sets(int s, int E, int b)
{
    int set_num = (1 << s);
    set *sets = calloc(set_num, sizeof(set));
    for (int i = 0; i < set_num; i++)
    {
        sets[i] = calloc(E, sizeof(line));
    }
    cache result = {E, set_num, b, s, sets};
    return result;
}

//释放sets的空间
void free_sets(cache *c)
{
    for (int i = 0; i < c->set_num; i++)
    {
        free(c->sets[i]);
        c->sets[i] = NULL;
    }
    free(c->sets);
    c->sets = NULL;
}
```

然后为了处理操作也定义了操作结构如下：

```
//操作
typedef struct operation
{
    char opt;                 //操作
    unsigned long address; //地址
    int size;                 //大小
} operation;
```

之后实现对于文件中操作的获取函数如下：

```
//通过读取文件中的一行来获取一个操作并返回
operation get_opt(FILE *fp)
{
    char buf[100]; //缓冲区大小为100
    operation opt = {0, 0, 0};
    if (!fgets(buf, 100, fp)) //读取一行并将数据存在buf中
        return opt;
    sscanf(buf, " %c %lx,%d", &opt.opt, &opt.address, &opt.size); //用字符串给opt赋值
    return opt;
}
```

获取了操作之后还要对操作中的 address 进行解析，找出组索引和标记位。

```
//获得组索引
int get_set_index(unsigned long address, int s, int b)
{
    return (address >> b) % (1 << s);
}

//获得标记位
int get_tag(unsigned long address, int s, int b)
{
    return address >> (s + b);
}
```

由此对于文件的操作读取部分以完成，接下来便是对于操作的处理。

由于 load 和 store 在高速缓存上的操作是相同的，故实现为一个函数 void load_store();

而 modify 的操作是 load 与 store 的组合，实现为一个函数 void modify()。

由于代码很长，故不在此列出，load 和 store 的思路如下：首先根据操作的组索引找到对应的组，在对应的组内遍历所有行寻找该组内是否已有标记位和有效位均吻合的行，若有，则 hit，更新时间后结束操作。若无则 miss，在该组内寻找空余的行，若有空余的行，则该行为新的该操作对应的行，之后更新该行的标记以及有效位之后再更新时间，然后结束操作。若无空余的行则要 eviction，遍历该组寻找上次访问时间最晚的行，执行驱逐更新该行的标记以及有效位之后再更新时间，然后结束操作。modify 的思路如下：直接执行一次 load 然后执行一次 store，在此过程中注意一下 opt 即可。此外函数中的变量 bool isDisplay; 为是否有命令行参数-v 并每次操作都展示 hit、miss 和 eviction 的标记。

最后便是组织上面的操作并实现主函数，如下：

```
int main(int argc, char **argvs)
{
    //变量初始化
    hit_count = 0;
    miss_count = 0;
    eviction_count = 0;
    FILE *fp = NULL;
    bool isDisplay = false;
    int s = 0, E = 0, b = 0;
    operation opt;
    //获取命令行参数
    get_argvs(argc, argvs, &s, &E, &b, &fp, &isDisplay);
    //根据读取的s，E，b创建缓存
    cache cach = calloc_sets(s, E, b);
    //处理操作
    while ((opt = get_opt(fp)).opt)
    {
        switch (opt.opt)
        {
        case 'I':
            continue;
        case 'M':
            modify(&cach, &opt, isDisplay);
            break;
        case 'S':
            load_store(&cach, &opt, isDisplay, false);
            break;
        case 'L':
            load_store(&cach, &opt, isDisplay, false);
            break;
        }
    }

    //进行收尾
    fclose(fp);
    free_sets(&cach);
    printSummary(hit_count, miss_count, eviction_count);
    return 0;
}
```

②PartB：

本次 Lab 的第二部分相对于第一部分代码量会小很多，但需要深入思考的问题却更难了，尤其是对于 64×64 矩阵的优化方案。

（i）32×32

根据 PDF 的提示进行分块处理，这里观察到 cache 的大小正好够填满数组的前 8 行，因此 cache 的映射应该是整个 cache 分别对应 0-7、8-15、16-23、24-31 这四个部分，举例来说就是第 0 行的前 8 个元素块与第 8 行的前 8 个元素块对应 cache 中的一个相同的块。由以上分析分成 8×8 的块，这样对于对角线之外的块 A 和 B 就不会冲突，从而减少了 miss 数。这里为了处理对角线上的 8×8 块 A 和 B 数组同时占用 cache 导致大量不命中的情况，故一次读取 A 中的一行 8 个数据并存入临时变量中，这样 A 每块只有一次冷不命中，而 B 对角线上的块部分有两次不命中，其他位置的块也是只有一次冷不命中，已达到最优要求。具体实现请看代码。

（ii）64×64

由以上 32×32 的分析，这次 cache 的大小正好够填满数组的前 4 行，故首先尝试分成 4×4 的块，此时的 miss 数已经达到了 1600 左右，虽然有了很大的减少但并未达到最优要求。事实上由于 cache 的块大小为 8 个 int 值，故分成 4×4 的块并未充分利用 cache 的空间，想要充分利用 cache 的空间还是要分成 8×8 的块，但块内操作需要思考新的方案，否则还是会有大量不命中。下面是对于 8×8 的块中进行的操作的图示，分别对应于代码中的三个 for 循环的操作：

① 第一个 for 循环

A 的某中一个 8×8 块

a	a	a	a	b	b	b	b
c	c	c	c	d	d	d	d
e	e	e	e	f	f	f	f
g	g	g	g	h	h	h	h
i	i	i	i	j	j	j	j
k	k	k	k	l	l	l	l
m	m	m	m	n	n	n	n
o	o	o	o	p	p	p	p

B 中对应的 8×8 块

a	c	e	g	b	d	f	h
a	c	e	g	b	d	f	h
a	c	e	g	b	d	f	h
a	c	e	g	b	d	f	h

② 第2个 for 循环的单步操作

A 的某中一个 8×8 块

a	a	a	a	b	b	b	b
c	c	c	c	d	d	d	d
e	e	e	e	f	f	f	f
g	g	g	g	h	h	h	h
i	i	i	i	j	j	j	j
k	k	k	k	l	l	l	l
m	m	m	m	n	n	n	n
o	o	o	o	p	p	p	p

B 中对应的 8×8 块

a	c	e	g	i	k	m	o
a	c	e	g	b	d	f	h
a	c	e	g	b	d	f	h
a	c	e	g	b	d	f	h
b	d	f	h				

③ 第2个 for 循环

A的某中一个 8×8 块

a	a	a	a	b	b	b	b
c	c	c	c	d	d	d	d
e	e	e	e	f	f	f	f
g	g	g	g	h	h	h	h
i	i	i	i	j	j	j	j
k	k	k	k	l	l	l	l
m	m	m	m	n	n	n	n
o	o	o	o	p	p	p	p

B中对应的8×8块

a	c	e	g	i	k	m	o
a	c	e	g	i	k	m	o
a	c	e	g	i	k	m	o
a	c	e	g	i	k	m	o
b	d	f	h				
b	d	f	h				
b	d	f	h				
b	d	f	h				

④ 第3个 for循环

A的某中一个 8×8 块

a	a	a	a	b	b	b	b
c	c	c	c	d	d	d	d
e	e	e	e	f	f	f	f
g	g	g	g	h	h	h	h
i	i	i	i	j	j	j	j
k	k	k	k	l	l	l	l
m	m	m	m	n	n	n	n
o	o	o	o	p	p	p	p

B中对应的8×8块

a	c	e	g	i	k	m	o
a	c	e	g	i	k	m	o
a	c	e	g	i	k	m	o
a	c	e	g	i	k	m	o
b	d	f	h	j	l	n	p
b	d	f	h	j	l	n	p
b	d	f	h	j	l	n	p
b	d	f	h	j	l	n	p

（iii）61×67

根据 PDF 的提示进行分块处理，由于给出的最优要求的 miss 数较多，达到了 2000，且该缓存大小并不规则，故先对不同分块进行测试，当分块为 17×17 时已达到要求，并不需要额外优化了，故就此结束。具体实现请看代码。

本地运行结果：（如测试结果不同请联系我，谢谢）

```
hby@ubuntu:~/Desktop/ICS_lab/cachelab-handout$ ./driver.py
Part A: Testing cache simulator
Running ./test-csim
                        Your simulator     Reference simulator
Points (s,E,b)    Hits  Misses  Evicts    Hits  Misses  Evicts
     3 (1,1,1)       9       8       6       9       8       6  traces/yi2.trace
     3 (4,2,4)       4       5       2       4       5       2  traces/yi.trace
     3 (2,1,4)       2       3       1       2       3       1  traces/dave.trace
     3 (2,1,3)     167      71      67     167      71      67  traces/trans.trace
     3 (2,2,3)     201      37      29     201      37      29  traces/trans.trace
     3 (2,4,3)     212      26      10     212      26      10  traces/trans.trace
     3 (5,1,5)     231       7       0     231       7       0  traces/trans.trace
     6 (5,1,5)  265189   21775   21743  265189   21775   21743  traces/long.trace
    27


Part B: Testing transpose function
Running ./test-trans -M 32 -N 32
Running ./test-trans -M 64 -N 64
Running ./test-trans -M 61 -N 67

Cache Lab summary:
                        Points   Max pts      Misses
Csim correctness          27.0        27
Trans perf 32x32           8.0         8         287
Trans perf 64x64           8.0         8        1179
Trans perf 61x67          10.0        10        1970
          Total points    53.0        53
```

总结：

在本次 lab 中主要加深了对于高速缓存的理解，在 PartA 中弄清了每次数据处理时高速缓存是如何工作的，hit、miss 以及 eviction 时高速缓存内部的状态，此外还实现了一个高速缓存模拟器。在 PartB 中加深了对高速缓存与主存关系的理解，并注意到在处理大矩阵时分块策略在高速缓存层面上对于函数性能的提高，此外不断优化函数的过程也是一次充满挑战的经历。