实验报告

Part A

首先根据Block块的定义和所需要的参数，定义一个结构体和一些全局变量：

code

```
typedef struct
{
    int valid;
    int tag;
    int timestamp;
} Block;
int s = -1, E = -1, b = -1, verbosity = 0;
int S = 0, B = 0;
int hit = 0, miss = 0, eviction = 0;
int alltime = 0;
FILE* file = NULL;
Block** cache;
```

下面实现各个函数：

Init函数分配内存，对cache进行初始化： code

```
void Init()
{
    cache = (Block **)malloc(sizeof(Block) * S);
    for (int i = 0; i < S; ++i)
    {
        cache[i] = (Block *)malloc(24LL * E);
        for (int j = 0; j < E; ++j)
        {
            cache[i][j].valid = 0;
            cache[i][j].tag = 0LL;
            cache[i][j].timestamp = 0LL;
        }
    }
}
```

Deal函数读取输入，并根据S、L和M分别执行不同的指令： code

```
void Deal()
{
    unsigned len = 0;
    unsigned long long addr = 0;
    char buf[1000];
    while (fgets(buf, 1000, file))
    {
        if (buf[1] == 'S' || buf[1] == 'L' || buf[1] == 'M')
        {
            sscanf(&buf[3], "%llx,%u", &addr, &len);
            if (verbosity)
                printf("%c %llx,%u ", buf[1], addr, len);
            visit(addr);
            if (buf[1] == 'M')
                visit(addr);
            if (verbosity)
                puts("\n");
        }
    }
}
```

其中的核心函数visit将访问cache并判断访问状态： code

```
void visit(unsigned long long addr)
{
    Block* set = cache[(addr >> b) % S];
    unsigned long tag = addr >> (s + b);
    unsigned int line = 0;
    unsigned long mintime = ~0;
    int i;
    for (i = 0;; ++i)
    {
        if (set[i].tag == tag && set[i].valid)
        {
            hit++;
            if (verbosity)
                printf("hit ");
            set[i].timestamp = alltime++;
            break;
        }
        if (i >= E)
        {
            miss++;
            if (verbosity)
                printf("miss ");
            for (int templine = 0; templine < E; templine++)
            {
                if (set[templine].timestamp < mintime)
                {
                    line = templine;
                    mintime = set[templine].timestamp;
                }
            }
            if (set[line].valid)
            {
                eviction++;
                if (verbosity)
                    printf("eviction ");
            }
            set[line].valid = 1;
            set[line].tag = tag;
            set[line].timestamp = alltime++;
            return;
        }
    }
```

```
}
```

最后完成main函数和其它辅助函数完成对核心函数的组织，注意要进行内存的释放： code

```
int main(int argc, const char** argv, const char** envp)
{
    char op = 0;
    while ((op = getopt(argc, (char* const*)argv, "s:E:b:t:vh")) != -1)
    {
        switch (op)
        {
        case 's':
            s = atoi(optarg);
            S = 1 << s;
            break;
        case 'E':
            E = atoi(optarg);
            break;
        case 'b':
            b = atoi(optarg);
            B = 1 << b;
            break;
        case 't':
            file = fopen(optarg, "r");
            break;
        case 'h':
            Help((char**)argv);
        case 'v':
            verbosity = 1;
            break;
        default:
            Help((char**)argv);
        }
    }
    if (s == -1 || E == -1 || b == -1 || file == NULL)
    {
        printf("%s: Missing required command line argument\n", *argv);
        Help((char**)argv);
    }
    Init();
    Deal();
    fclose(file);
    free(cache);
    printSummary(hit, miss, eviction);
    return 0;
}
```

Part B

32*32

由于直接进行映射会造成大量的缓存miss，故考虑分块转置来提高效率。由于每个set有32bytes（即可以存放8个int），故考虑将原矩阵按照8 * 8的小矩阵进行转置。同时为了减少miss数，将A矩阵的一行8个数用8个临时变量存储，再放入B矩阵的一列中即可。
miss数分析：采用临时变量转存后，A矩阵的第0、8、16、24列整列都是冷不命中，B矩阵除了A矩阵的所有对应位置不命中外，在对角线上的所有元素也会出现miss，故miss数为32 * 4+15 * 4+8 * 12=284 < 300，满足要求。 code

```
int i, j, k, a0, a1, a2, a3, a4, a5, a6, a7;
for (i = 0; i < 32; i += 8)
{
    for (j = 0; j < 32; j += 8)
    {
        for (k = i; k < i + 8; ++k)
        {
            a0 = A[k][j];
            a1 = A[k][j + 1];
            a2 = A[k][j + 2];
            a3 = A[k][j + 3];
            a4 = A[k][j + 4];
            a5 = A[k][j + 5];
            a6 = A[k][j + 6];
            a7 = A[k][j + 7];
            B[j][k] = a0;
            B[j + 1][k] = a1;
            B[j + 2][k] = a2;
            B[j + 3][k] = a3;
            B[j + 4][k] = a4;
            B[j + 5][k] = a5;
            B[j + 6][k] = a6;
            B[j + 7][k] = a7;
        }
    }
}
```

64*64

采取同32 * 32的分析，可以得到划分的子块为8 * 4，这样可以保证B数组每四个cache块不会重复载入，但却似的A数组只利用了4个int数据，从而实际结果出现了较多miss，因而需要改进方法。
不妨考虑将8 * 8和4 * 4分块的优点结合起来，即依然按照8 * 8的整体分块，但在内部进行4 * 4块的转置。上文提到8 * 4可以避免重复载入，因此我们先将A数组8 * 8分块的上半部分通过临时变量转置到B矩阵的上半部分，不过此时B的右上4 * 4块还需要平移到左下角才能完成转置。因此，接下来我们再通过临时变量将A的左下角一列一列地转置到B的右上角，同时将B的右上角平移到其左下角，最后再将A的右下角直接转置到B的右下角即可。 code

```
int i, j, k, a0, a1, a2, a3, a4, a5, a6, a7;
for (i = 0; i < N; i += 8)
{
    for (j = 0; j < M; j += 8)
    {
        for (k = i; k < i + 4; ++k)
        {
            a0 = A[k][j];
            a1 = A[k][j + 1];
            a2 = A[k][j + 2];
            a3 = A[k][j + 3];
            a4 = A[k][j + 4];
            a5 = A[k][j + 5];
            a6 = A[k][j + 6];
            a7 = A[k][j + 7];
```

```
            B[j][k] = a0;
            B[j + 1][k] = a1;
            B[j + 2][k] = a2;
            B[j + 3][k] = a3;
            B[j][k + 4] = a4;
            B[j + 1][k + 4] = a5;
            B[j + 2][k + 4] = a6;
            B[j + 3][k + 4] = a7;
        }

        for (k = j; k < j + 4; ++k)
        {
            a0 = A[i + 4][k];
            a1 = A[i + 5][k];
            a2 = A[i + 6][k];
            a3 = A[i + 7][k];

            a4 = B[k][i + 4];
            a5 = B[k][i + 5];
            a6 = B[k][i + 6];
            a7 = B[k][i + 7];

            B[k][i + 4] = a0;
            B[k][i + 5] = a1;
            B[k][i + 6] = a2;
            B[k][i + 7] = a3;
            B[k + 4][i] = a4;
            B[k + 4][i + 1] = a5;
            B[k + 4][i + 2] = a6;
            B[k + 4][i + 3] = a7;
        }

        for (k = i + 4; k < i + 8; ++k)
        {
            a0 = A[k][j + 4];
            a1 = A[k][j + 5];
            a2 = A[k][j + 6];
            a3 = A[k][j + 7];

            B[j + 4][k] = a0;
            B[j + 5][k] = a1;
            B[j + 6][k] = a2;
            B[j + 7][k] = a3;
        }
    }
}
```

61*67

感觉没看出来有什么规律.....不过对miss数的要求还比较宽，因此可以尝试一下各种分块的miss数，拿16 * 16试一下发现可以通过，但拿左右数值继续进行尝试后发现17 * 17的miss数应该是最少的。 code

```
int i, j, k, r;
for (i = 0; i < N; i += 17)
{
    for (j = 0; j < M; j += 17)
    {
        for (k = i; k < i + 17 && k < N; ++k)
        {
            for (r = j; r < j + 17 && r < M; ++r)
            {
                B[r][k] = A[k][r];
            }
        }
    }
}
```