矩阵乘法优化报告

汇编语言第二次大作业

石曜铭

2025 年 7 月 7 日

1 实验整体设计

本实验探究矩阵乘法从朴素实现到底层优化的不同写法，并根据运行时间等结果分析各种优化的作用。

本实验一共探究了六种写法：

1. python 朴素实现；
2. C 语言朴素实现；
3. 在 2 的基础上，添加多线程优化；
4. 在 3 的基础上，考虑 cache 局部性机制，使用分块优化；
5. 在 4 的基础上，使用SIMD向量指令优化；
6. 在 5 的基础上，使用其他优化方法。这里使用比 5 向量化程度更高的 AVX512 实现。

因此实验包含矩阵乘法的多种实现方式，为了验证它们的正确性，我使用 python 的 numpy 包进行答案的计算，并以之为标准与其他方法的输出进行比较，精度不低于 10^{-5} 。

为了优化效果的普遍性，实验在 5 个不同计算机上进行了测试。具体结果见"效果分析"一节。

注：本实验中随机矩阵的方式是使用 C 语言中的 `rand` 函数，并将结果除以 `RAND_MAX` 得到一个 $0 \sim 1$ 之间的浮点数，使用 `double` 存储。

2 源码分析

2.1 代码结构

如右图，整体由一个头文件和多个C语言代码文件和 py 脚本构成。`matrix.h` 定义了结构体 `Matrix`，为了内存连续，降低后续 SIMD 优化中被编译器认为不可优化的可能性，在这里使用一维数组存储矩阵，并在堆区开辟内存。

`matrix_utils.c` 主要包含一些矩阵的基本操作，比如输入输出。其他七个文件从 `v0` 到 `v6` 分别是不同写法的矩阵乘法实现。接下来逐一进行源码分析。

```
.
├── Makefile
├── compile_flags.txt
├── files
├── include
│   └── matrix.h
├── py
│   ├── files
│   │   ├── in.txt
│   │   └── out.txt
│   ├── matrix_mul.py
│   └── np_matrix_mul.py
└── src
    ├── main.c
    ├── matrix_utils.c
    ├── v0_py_numpy_mul.c
    ├── v1_py_simple_mul.c
    ├── v2_c_simple_mul.c
    ├── v3_c_parallel_mul.c
    ├── v4_c_cache_mul.c
    ├── v5_c_SIMD_mul.c
    └── v6_c_multi_optimized_mul.c
```

代码结构图

2.2 python 朴素实现以及调用 NumPy 库得到标准

python 的朴素实现在 `v1` 中。本实验通过系统调用来调用 py 脚本。首先将矩阵输出到指定文件，然后进行 py 脚本的调用。

```c
write_matrix(input_file, A);
write_matrix(input_file, B);
fclose(input_file);

char py_command[] = "python3 py/matrix_mul.py";
if (system(py_command)) {
  printf("error executing py script\n");
  return;
}
```

输出矩阵并调用py脚本

朴素实现的 python 矩阵乘法核心代码：

```python
def matrix_multiply(A, B):
    rowsA = len(A)
    colsA = len(A[0])
    colsB = len(B[0])
    C = [[0.0] * colsB for _ in range(rowsA)]
    for i in range(rowsA):
        for k in range(colsA):
            a_val = A[i][k]
            for j in range(colsB):
                C[i][j] += a_val * B[k][j]
    return C
```

python 矩阵乘法

调用 NumPy 库代码：`C = np.dot(A, B)`.

2.3 C 语言朴素实现

朴素实现的三重循环是可以调换顺序的。这里为了减少 cache miss，选择先枚举 A 的行，再枚举 A 的列，然后枚举 B 的列。这样是在内部循环中是连续访问 B 中元素的，减少了 cache miss。

```
void c_simple_mul(Matrix *A, Matrix *B, Matrix *C) {
  for (int i = 0; i < A->rows; ++i)
    for (int k = 0; k < A->cols; ++k) {
      double tmp = A->data[i * A->cols + k];
      for (int j = 0; j < B->cols; ++j)
        C->data[i * C->cols + j] += tmp * B->data[k * B->cols + j];
    }
}
```

C 语言朴素矩阵乘法

2.4 多线程优化

首先确定要将对 A 不同行之间的处理并行，默认线程数是 8。根据线程数对行进行分块，然后使用统一的线程工作函数计算对应的矩阵乘法结果，然后进行合并。

```
void *threadWorker(void *Arg) {
  ThreadArg *Targ = (ThreadArg *)Arg;
  int st_row = M / NUM_THREADS * Targ->threadId;
  int ed_row = Targ->threadId == NUM_THREADS - 1 ? M : st_row + M / NUM_THREADS;
  Matrix *A = Targ->A;
  Matrix *B = Targ->B;
  Matrix *C = Targ->C;
  //  printf("%d %d %d\n", Targ->threadId, st_row, ed_row);
  for (int i = st_row; i < ed_row; ++i)
    for (int k = 0; k < A->cols; ++k) {
      double tmp = A->data[i * A->cols + k];
      for (int j = 0; j < B->cols; ++j)
        C->data[i * C->cols + j] += tmp * B->data[k * B->cols + j];
    }

  pthread_exit(NULL);
}
```

线程工作函数

```
void c_parallel_mul(Matrix *A, Matrix *B, Matrix *C) {
  pthread_t threads[NUM_THREADS];
  ThreadArg Args[NUM_THREADS];

  for (int i = 0; i < NUM_THREADS; ++i) { // create threads
    Args[i].threadId = i;
    Args[i].A = A, Args[i].B = B, Args[i].C = C;
    pthread_create(&threads[i], NULL, threadWorker, &Args[i]);
  }

  for (int i = 0; i < NUM_THREADS; ++i)
    pthread_join(threads[i], NULL);
}
```

线程的创建

2.5 利用 cache 进行分块优化

在矩阵乘法的过程中，将矩阵拆成能放进 L1-Cache 的小块，可以进一步降低 cache miss，增加加载速度。所以需要根据 L1-Cache 的大小计算出合适的 block 大小。

在分块矩阵乘法的多线程实现中，为了尽可能多的利用闲置的线程，使用动态分配的方法。即对每个线程记录一个指向原子操作的整型代表下一个要处理的行。哪个线程先处理完，就可以给这个原子操作的整型加上 `block_size` 。

```c
void *blockedThreadWorker(void *arg) {
  ThreadArg *targ = (ThreadArg *)arg;
  Matrix *A = targ->A;
  Matrix *B = targ->B;
  Matrix *C = targ->C;
  atomic_int *next_row = targ->next_row;

  int i_b;
  while ((i_b = atomic_fetch_add(next_row, block_size)) < A->rows) {
    int i_e = min(i_b + block_size, A->rows);
    for (int j_b = 0; j_b < B->cols; j_b += block_size) {
      int j_e = min(j_b + block_size, B->cols);
      for (int k_b = 0; k_b < A->cols; k_b += block_size) {
        int k_e = min(k_b + block_size, A->cols);
        for (int i = i_b; i < i_e; ++i)
          for (int k = k_b; k < k_e; ++k) {
            double val = A->data[i * A->cols + k];
            for (int j = j_b; j < j_e; ++j)
              C->data[i * A->cols + j] += val * B->data[k * B->cols + j];
          }
      }
    }
  }
  pthread_exit(NULL);
}
```

分块矩阵乘法的线程函数

2.6 使用 SIMD 指令优化

为了能使编译器做出向量化的优化，需要在代码中给出明确的标志表示这里可以被向量化，比如循环展开。将循环变量每次加 4，一次循环中处理 4 个乘加操作。但是，经过多次尝试，编译器始终认为这里的指针可能存在地址上的重叠和别名，只会做出 SSE 优化，即使用 128 位的 xmm 寄存器，每次处理两个乘加操作。即使使用 `restrict` 关键字修饰也不能做到。

解决方案是，强行在代码中使用 `immintrin.h` 中的内联函数，直接显式声明一个使用 ymm 寄存器的变量，然后进行操作。

此版本和 4 的区别仅在于最内层循环的函数有所改变：

```c
for (int i = i_b; i < i_e; ++i) {
  const double *restrict rowA = A->data + i * A->cols;
  double *restrict rowC = C->data + i * C->cols;
  for (int k = k_b; k < k_e; ++k) {
    const double val = rowA[k];
    const double *restrict rowB = B->data + k * B->cols;
    int j = j_b;
    __m256d val_vec = _mm256_set1_pd(rowA[k]);
    for (; j <= j_e - 4; j += 4) {
      __m256d b_vec = _mm256_loadu_pd(rowB + j);
      __m256d c_vec = _mm256_loadu_pd(rowC + j);
      c_vec = _mm256_fmadd_pd(val_vec, b_vec, c_vec);
      _mm256_storeu_pd(rowC + j, c_vec);
    }
    for (; j < j_e; ++j)
      rowC[j] += val * rowB[j];
  }
}
```

显式使用 AVX

```
root@autodl-container-3e134ab2e1-4706bdba:~/asm-learning/x86/lab2# objdump -d build/v5_c_SIMD_mul.o | grep vfmadd
 195:   c4 e2 f5 a8 04 d0       vfmadd213pd (%rax,%rdx,8),%ymm1,%ymm0
 226:   c4 c2 d5 98 4c 19 f8    vfmadd132pd -0x8(%r9,%rbx,1),%ymm5,%ymm1
 261:   c4 c2 d9 98 04 f1       vfmadd132pd (%r9,%rsi,8),%xmm4,%xmm0
 282:   c4 e2 e1 99 14 d1       vfmadd132sd (%rcx,%rdx,8),%xmm3,%xmm2
 2fd:   c4 c2 e9 a9 03          vfmadd213sd (%r11),%xmm2,%xmm0
 321:   c4 e2 e9 a9 07          vfmadd213sd (%rdi),%xmm2,%xmm0
 340:   c4 e2 e9 a9 07          vfmadd213sd (%rdi),%xmm2,%xmm0
 35d:   c4 e2 c9 99 54 31 18    vfmadd132sd 0x18(%rcx,%rsi,1),%xmm6,%xmm2
```

通过 objdump 确认使用了 AVX

2.7 使用 AVX512 优化

有些 CPU 可以使用 AVX512 实现更高程度的向量化，因此可以将 5 中的 256 位的内联函数改为对应的 512 位。需加入 `-mavx512f -mavx512dq` 编译选项。

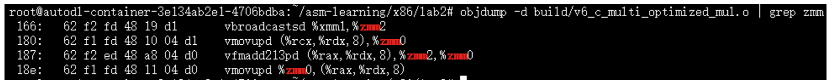
此版本和 5 的区别仅在于最内层循环的函数有所改变：

```c
for (int i = i_b; i < i_e; ++i) {
  const double *restrict rowA = A->data + i * A->cols;
  double *restrict rowC = C->data + i * C->cols;
  for (int k = k_b; k < k_e; ++k) {
    const double val = rowA[k];
    const double *restrict rowB = B->data + k * B->cols;
    int j = j_b;

    __m512d val_vec = _mm512_set1_pd(val);
    for (; j <= j_e - 8; j += 8) {
      __m512d b_vec = _mm512_loadu_pd(rowB + j);
      __m512d c_vec = _mm512_loadu_pd(rowC + j);
      c_vec = _mm512_fmadd_pd(val_vec, b_vec, c_vec);
      _mm512_storeu_pd(rowC + j, c_vec);
    }

    for (; j < j_e; ++j)
      rowC[j] += val * rowB[j];
  }
}
```

显式使用 AVX512



通过 objdump 确认使用了 AVX512

3 效果分析

实验一共在 5 台基于不同硬件的计算机中进行测试。以下为五台计算机CPU的型号：

序号	CPU型号
1	AMD Ryzen 9 8940HX
2	13th Gen Intel(R) Core(TM) i9-13900H
3	11th Gen Intel(R) Core(TM) i5-1135G7 2.40GHz
4	12th Gen Intel(R) Core(TM) i9-12900H 2.50Ghz
5	Intel(R) Xeon(R) Platinum 8470Q

以下为运行结果：

序号	NumPy	v1	v2	v3	v4	v5	v6
1	12358	2767098	40564	5512	3959	1533	1763
2	135141	~ 3600000	90091	16824	20477	12396	不支持
3	20115	~ 3600000	42991	16812	12785	7166	6501
4	12476	1895516	37861	7441	4235	2186	不支持
5	18135	3232503	66255	11103	5335	2436	2584

相对 v2 的加速比：

序号	v3	v4	v5	v6
1	7.36	10.25	26.45	23.01
2	5.35	4.40	7.27	不支持
3	2.56	3.36	6.00	6.61
4	5.09	8.94	17.31	不支持
5	5.97	12.42	27.20	25.64

由结果可见，这些写法的速度基本上呈现递增的趋势。当然，也能注意到一些反常现象比如在 2 号上分块多线程比不分块要慢，这可能是 `block_size` 大小过大导致的。另外，还能够发现，对于一些比较新的 CPU，AVX512比256反而更慢一些，这可能是因为 AVX512 指令集的设计没有跟上硬件的迭代更新。

4 备注

本实验代码仓库已经上传到github上，在x86/lab2目录下。