南开大学

Nankai University

南　　开　　大　　学

计　算　机　学　院

并行程序设计实验报告

侯嘉栋

年级：2023 级

专业：计算机科学与技术

指导教师：王刚

2025 年 3 月 30 日

摘要

该报告主要探讨矩阵与向量乘法的算法优化，包括缓存优化、OpenMP 并行化和分块算法。报告详细描述了各个算法的设计与实现，并通过性能测试对比了不同算法的效率，结果表明缓存优化和并行化策略显著提升了计算性能。

目录

一、 实验环境

笔者最初尝试在 WSL 的 Ubuntu24.04 中测试，但是由于 CPU 内核不支持等（非常深奥没完全看明白的原因），转用阿里云的服务器，以下是其配置。

```
suhipek@suhipek-BOHK-WAX9X:~$ lscpu
Architecture:                    aarch64
CPU op-mode(s):                  32-bit, 64-bit
Byte Order:                      Little Endian
CPU(s):                          1
On-line CPU(s) list:             0
Thread(s) per core:              1
Core(s) per socket:              1
Socket(s):                       1
NUMA node(s):                    1
Vendor ID:                       ARM
Model:                           0
Model name:                      Neoverse-N2
Stepping:                        r0p0
Frequency boost:                 disabled
CPU MHz:                         1223.521
CPU max MHz:                     3000.0000
CPU min MHz:                     3000.0000
BogoMIPS:                        100.00
L1d cache:                       64 KiB (1 instance)
L1i cache:                       64 KiB (1 instance)
L2 cache:                        1 MiB (1 instance)
L3 cache:                        64 MiB (1 instance)
NUMA node0 CPU(s):               0
```


二、 Github 代码仓库

这里是 Github 仓库位置欢迎 star! NKU-Parallel-Computing

三、 实验一:n*n 矩阵与向量乘积

(一) 算法设计

1. 平凡算法设计思路

平凡算法即参考实验指导书中的方法，我们优先列的计算。

2. cashe 优化算法设计思路

对其进行 chshe 优化，改为逐行访问矩阵元素：一步外层循环计算不出任何一个内积，只是向每个内积累加一个乘法结果后者的访存模式与行主存储匹配，具有很好空间局部性，令 cache 作用得以发挥。

3. 利用 OpenMP with C 并行优化 cashe 优化算法设计思路

OpenMP是一种支持共享内存并行编程的 API，适用于 C、C++ 和 Fortran，旨在简化多线程程序的开发。它通过编译器指令（如 #pragma omp parallel）、库函数和环境变量，帮助开发者轻松实现并行化，例如并行执行代码块、分配循环任务或划分代码段。OpenMP 还提供了数据共享与同步机制（如 shared、private 和 critical），以及环境变量（如 OMP_NUM_THREADS）来控制并行执行的行为。由于其易用性和跨平台特性，OpenMP 被广泛应用于高性能计算领域，如数值模拟、图像处理和机器学习等任务。

参考教程，我们可以写一个比较简单的版本。即单纯在 cache_optimized_algorithm 中加 #pragma omp parallel for 用于并行化，通过查阅资料和 GPT，我发现当前算法多个线程可能同时尝试修改同一个 result[i] 元素，导致竞争。而这里每个线程先在自己的局部变量中累加，最后再合并，减少了竞争的可能性。每个线程的 local_result 可能会更好地利用 CPU 缓存，因为线程在局部变量上连续写入，而不是频繁访问共享内存中的全局变量。

4. 分块算法

我也考虑分块算法通过多维循环划分和 OpenMP 并行指令实现了矩阵-向量乘法的并行优化。其核心思想是通过分块策略提升缓存利用率并减少线程竞争。采用 OpenMP 的策略，我也进行了类似优化

(二) 编程实现

1. 平凡算法

平凡算法

```
    for (size_t i = 0;i < n;i++)
     for(size_t j = 0;j < n;j++)
        result[i] += A[j][i] * v[j];
```

2. cashe 优化算法

cashe 优化

```
    for(size_t j = 0;j < n;j++)
        for(size_t i = 0;i < n;i++)
            result[i] += A[j][i] * v[j];
}
```

3. OpenMP with C 优化

OpenMP with C 优化

```
parallel_algorithm:
    #pragma omp parallel for
    for(size_t j = 0;j < n;j++ )
        for(size_t i = 0;i < n;i++)
            result[i] += A[j][i] * v[j];
parallel_algorithm_optimized:
        Vector local_result(n, 0.0);
        #pragma omp for
        for(size_t j = 0; j < n; j++)
            for(size_t i = 0; i < n; i++)
                local_result[i] += A[j][i] * v[j];
        #pragma omp critical
        for(size_t i = 0; i < n; i++)
            result[i] += local_result[i];
```

4. 分块算法

OpenMP with C 优化

```
blocked_algorithm:
    #pragma omp parallel for
    for(size_t j = 0;j < n;j+=blockSize)
        for(size_t i = 0;i < n;i+=blockSize)
            for(size_t jj = j;jj < min(j+blockSize,n);jj++)
                for(size_t ii = i;ii < min(i+blockSize,n);ii++)
                    result[ii] += A[jj][ii] * v[jj];
blocked_algorithm_optimized:
    #pragma omp parallel
    {
        Vector local_result(n, 0.0);
        #pragma omp for schedule(dynamic)
        for(size_t j = 0; j < n; j += blockSize)
            for(size_t i = 0; i < n; i += blockSize) {
                if(j + blockSize < n)
                    __builtin_prefetch(&A[j + blockSize][0], 0, 3);
                size_t j_end = min(j + blockSize, n);
                size_t i_end = min(i + blockSize, n);
                for(size_t jj = j; jj < j_end; jj++) {
                    float vj = v[jj];
                    for(size_t ii = i; ii < i_end; ii++)
                        local_result[ii] += A[jj][ii] * vj;
                }
            }
        #pragma omp critical
        for(size_t i = 0; i < n; i++)
            result[i] += local_result[i];
    }
```

(三) 性能测试

使用chrono和 perf 验证每个算法各个效率，采用随机矩阵，单位矩阵，稀疏矩阵，Hilbert 矩阵分别对朴素算法，缓存优化算法，OpenMP 并行算法，分块算法进行测试，矩阵大小为 500, 1000, 2000,3000 4000，其中分块算法的块数分别为 16, 32, 64, 128, 256，每个测试重复 5 次，为此了测试的严谨性，我编写了一个 Baseline，用于批量测试和展示结果，将测试获得的数据生成 csv 文件。后设计脚本批量分析. 由于测的数据比较多，真实结果可见 Github 仓库。以下只给出 n=4000 的结果比较。

MatrixType	Size	Algorithm	BlockSize	ExecutionTime (ms)
随机矩阵	4000	naive	0	94.867
随机矩阵	4000	cache_optimized	0	6.76823
随机矩阵	4000	openmp_simple	0	7.17958
随机矩阵	4000	openmp_optimized	0	7.25124
随机矩阵	4000	blocked	16	15.6991
随机矩阵	4000	blocked	32	32.4256
随机矩阵	4000	blocked	64	21.4951
随机矩阵	4000	blocked	128	19.2962
随机矩阵	4000	blocked	256	22.1768
随机矩阵	4000	blocked_optimized	16	7.82895
随机矩阵	4000	blocked_optimized	32	14.7029
随机矩阵	4000	blocked_optimized	64	19.1912
随机矩阵	4000	blocked_optimized	128	14.1564
随机矩阵	4000	blocked_optimized	256	18.6779

可以明显的看到 cashe_optimized 算法和 openmp 以及 block 优化确实有作用，而且进一步我们发现对于 n=4000 的数据，块数取 128 貌似更好一点，但由于我租的服务器只有 1 个核心，所以并行的威力并未施展开，之后有条件可以考虑重新做这个实验。

(四) profiling

Performance Comparison

	Algorithm	Cycles	Instructions	Ins per Cycle	Cache References	Cache Misses	Cache Miss Rate
1	Naive Algorithm	144230513	318768738	2.21	83623907	5630430	6.73%
2	Cache Optimized Algorithm	87686592	284303803	3.24	72719708	608573	0.84%
3	OpenMP Parallel Algorithm	85186330	284819281	3.34	71675539	590710	0.82%
4	Blocked Algorithm	96338297	321534361	3.34	95745601	989819	1.03%

从表中可见，cashe 优化算法确实显著的增加了 cashe 命中率，但是由于之前分析的原因，并行算法效果不明显

四、 实验二:n 个数相加

(一) 算法设计

1. unrolling 系列算法

naive_sum,**two_way_sum**,**four_way_sum**,**unrolled_sum** 思路一致，故只给 **unrolled_sum** 的代码。

- **naive_sum** 是平凡算法，直接链式累加。
- **two_way_sum** 为二路累加，实际上就是将循环每次 +1 优化为 +2，存两个 sum，分为 sum1 和 sum2,sum1 加奇数,sum2 加偶数，最后将两个 sum 相加。将循环对时间的浪费减少近似一半（不准确应该）
- **four_way_sum** 四路累加，参考前面的思路，循环步数变为 4，设置 4 个 sum, 进一步减少循环对时间的浪费。
- **unrolled_sum** 上面思路带来 unrolling 策略，这里做了 8 次展开

除此之外，还可以通过宏定义或模板递归来消除循环。但是模板递归涉及到递归深度，g++ 编译的默认深度是 900, 笔者的测试数据是从 1024 开始，故需要在编译命令时设置**-ftemplate-depth=1025**，但注意不要让递归深度过大，因为会让服务器崩掉（亲身实践过）

2. recursive_sum

也可以通过递归两两相加的方式，具体可以参考实验指导书。

(二) 编程实现

1. unrolling 系列算法

8 次循环展开

```
double unrolled_sum(const vector<double>& arr) {
    double sum = 0.0;
    int n = arr.size();
    int i = 0;
    for (; i + 7 < n; i += 8) {
        sum += arr[i];sum += arr[i + 1];sum += arr[i + 2];sum += arr[i + 3];
        sum += arr[i + 4];sum += arr[i + 5];sum += arr[i + 6];sum += arr[i +
            7];
    }
    for (; i < n; i++) {
        sum += arr[i];
    }
    return sum;
}
```

宏定义

```
#define SUM_2(arr, start) ((arr)[start] + (arr)[start+1])
```

```
#define SUM_4(arr, start) (SUM_2(arr, start) + SUM_2(arr, start+2))
\\省略~
#define SUM_256(arr, start) (SUM_8(arr, start) + SUM_8(arr, start+8))
double macro_template_sum(const vector<double>& arr) {
    double sum = 0.0;
    size_t n = arr.size();
    size_t i = 0;
    while (i + 256 <= n) {
        sum += SUM_256(arr, i);
        i += 256;
    }
\\同样省略中间部分
    while (i + 2 <= n) {
        sum += SUM_2(arr, i);
        i += 2;
    }
    if (i < n) {
        sum += arr[i];
    }
    return sum;
}
```

模板递归

```
template<size_t N>
struct FixedSizeSum {
    static double sum(const vector<double>& arr, size_t start = 0) {
        return arr[start] + FixedSizeSum<N-1>::sum(arr, start+1);
    }
};
template<>
struct FixedSizeSum<1> {
    static double sum(const vector<double>& arr, size_t start = 0) {
        return arr[start];
    }
};
template<>
struct FixedSizeSum<0> {
    static double sum(const vector<double>& arr, size_t start = 0) {
        return 0.0;
    }
};
```

2. 递归算法

递归算法

```
double recursive_sum(const vector<double>& arr, int start, int end) {
    if (end - start <= 1) {
```

```
        return start < arr.size() ? arr[start] : 0.0;
    }
    int mid = (start + end) / 2;
    return recursive_sum(arr, start, mid) + recursive_sum(arr, mid, end);
}
```

3. 浮点数运算次序的不同对结果的影响

我们生成具有特定模式的测试数据以显示浮点数舍入差异

浮点数运算次序数据生成

```
vector<double> generate_fp_test_data(int n) {
    vector<double> arr(n);
    for (int i = 0; i < n; i++) {
        if (i % 3 == 0) {
            arr[i] = 1e15 + static_cast<double>(i);   // 非常大的值
        } else if (i % 3 == 1) {
            arr[i] = 1e-15 * static_cast<double>(i); // 非常小的值
        } else {
            arr[i] = static_cast<double>(i);         // 普通值
        }
    }
    return arr;
}
```

浮点数在计算机中是以有限的精度表示的，因此在进行加法、乘法等运算时，可能会出现舍入误差。这种误差在多次运算中会累积，导致最终结果的偏差。同时浮点数的加法是非结合的，这意味着不同的运算顺序会导致不同的结果。例如，先加小数再加大数，可能会导致小数的影响被忽略，从而产生较大的误差。正向累加和反向累加的结果可能会因为运算顺序的不同而有所不同。当参与运算的数值范围差异较大时（例如，一个非常大的数和一个非常小的数），小数的影响可能会被忽略，导致结果不准确。特别是在累加时，较小的数可能会因为精度限制而无法对结果产生影响。笔者测试从前往后相加，从后往前相加，二路相加，以及排序后相加的结果，计算这些结果之间的最大差异并对比启用快速浮点数模式 (-ffast-math) 后的差异。

```
标准浮点数模式结果:
浮点数累加顺序对结果的影响测试
------------------------------------------------------------
向量大小        正向累加        反向累加        成对累加        排序后累加      最大差异
------------------------------------------------------------
    1024    3.42000000e+17  3.42000000e+17  3.42000000e+17  3.42000000e+17 8.96000000e+02
    2048    6.83000000e+17  6.83000000e+17  6.83000000e+17  6.83000000e+17 3.32800000e+03
    4096    1.36600000e+18  1.36600000e+18  1.36600000e+18  1.36600000e+18 1.48480000e+04
    8192    2.73100000e+18  2.73100000e+18  2.73100000e+18  2.73100000e+18 5.63200000e+04
   16384    5.46200000e+18  5.46200000e+18  5.46200000e+18  5.46200000e+18 2.27328000e+05
------------------------------------------------------------
注意：差异反映了浮点数舍入误差在不同累加顺序下的影响。
      成对累加和排序后累加通常能提供更稳定和准确的结果。

启用快速浮点数模式结果(-ffast-math):
浮点数累加顺序对结果的影响测试
------------------------------------------------------------
向量大小        正向累加        反向累加        成对累加        排序后累加      最大差异
------------------------------------------------------------
    1024    3.42000000e+17  3.42000000e+17  3.42000000e+17  3.42000000e+17 1.28000000e+02
    2048    6.83000000e+17  6.83000000e+17  6.83000000e+17  6.83000000e+17 2.56000000e+02
    4096    1.36600000e+18  1.36600000e+18  1.36600000e+18  1.36600000e+18 5.12000000e+02
    8192    2.73100000e+18  2.73100000e+18  2.73100000e+18  2.73100000e+18 1.02400000e+03
   16384    5.46200000e+18  5.46200000e+18  5.46200000e+18  5.46200000e+18 2.04800000e+03
------------------------------------------------------------
```

图 1: 浮点数运算次序的不同对结果的影响

最大差异指的是所有算法中差异最大的，根据2我们可以明显的看到在数据量级到达 1017 次方左右，存在 102 左右的误差，而快速浮点数模式可以缓解一些误差。

图 2: 不同运算次序不同算法之间的差异比较

不同运算次序不同算法之间存在不一样的差异，不过由于量级很大，很难分辨哪个更接近真实值

(三) 性能测试

我们同样探索不同编译优化对程序性能的影响，计算所用的时间和对应的加速比，上述过程可以通过笔者编写的 baseline 实现 从8中可以看出，在 n 比较小时，unrolling 的次数效果并不

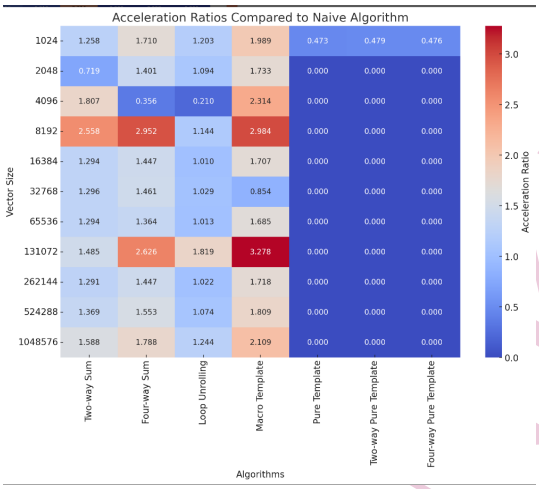

图 3: 不同算法对平凡算法的加速比热力图

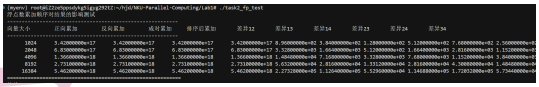

图 4: 不同算法的运行时间

明显，随着 n 的增大，unroll 的层数越多，优化效果越好，也就是说并非 unroll 层数越多越好，而是根据数据规模选择适当的 unroll 层数。宏定义展开也是同理，在起初 n 并不是很大的时候，运行时间很长，而当 n 非常大时，他的优势便很明显的凸显出来。由于递归深度的原因，未能验证模板递归，不过我猜想应该结果也呈现类似趋势

五、 不同编译选项对程序运行时间影响

我选择 task2 的 naive_sum 来进行探索，在compiler exploer中通过 x86-64 gcc 14.2 编译器，探索 O0,O1,O2,O3 优化对程序的影响。

naive_sum

```
1 double naive_sum(const vector<double>& arr) {
2     double sum = 0.0;
3     int n = arr.size();
4     for (int i = 0; i < n; i++) {
```

```
5           sum += arr[i];
6       }
7       return sum;
8   }
```

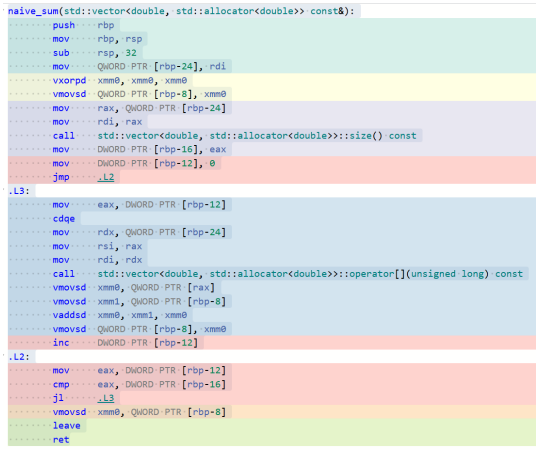

图 5: O0 优化结果

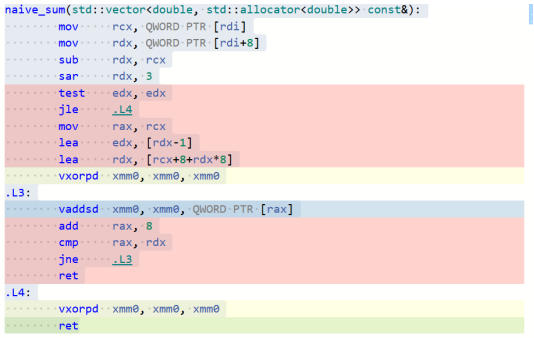

图 6: O1 优化结果

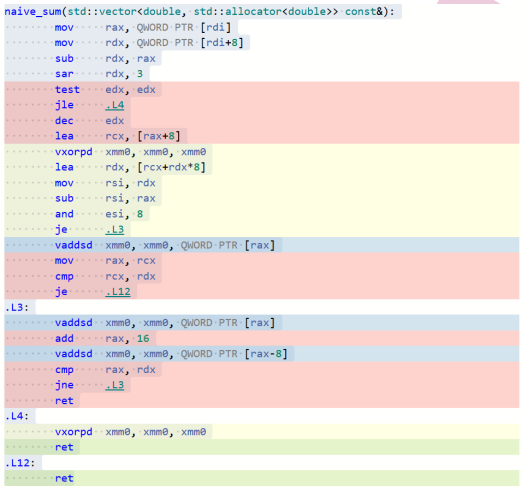

图 7: O2 优化结果

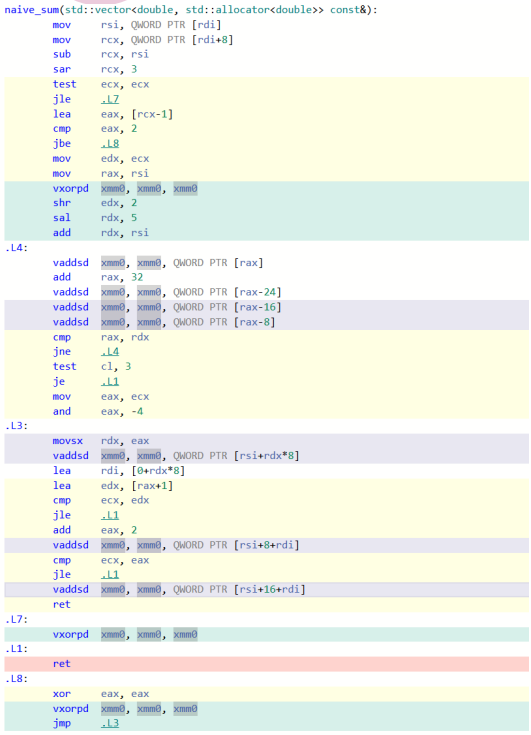

图 8: O3 优化结果

首先分析 O0, 在函数入口与栈帧设置时其建立栈帧并分配 32 字节的栈空间，未进行栈空间优化，保留所有临时变量和中间结果，也未使用寄存器复用，所有操作均依赖栈内存。在循环时显式调用 std::vector::size() 函数，而非直接计算 end - start，其也未内联 size() 方法，导致额外的函数调用开销。循环过程中未进行循环展开或预处理，直接使用显式循环结构，每次循环迭代都调用

operator[](unsigned long) 进行间接内存访问，导致大量间接跳转和函数调用开销并且每次累加都需要从栈中读取临时变量，写回后又存入栈，导致内存访问开销显著增加。

再看 O1, 在函数入口与栈帧设置时,直接通过指针计算元素个数 n,避免了调用 std::vector::size() 的函数开销，从而减少函数调用的间接跳转和栈操作，提升速度。在初始化与循环准备时，使用 lea 指令高效计算最后一个元素的地址，避免循环中重复计算。在循环体中通过指针 rax 直接遍历数组，避免索引 i 的计算（如 arr[i]）。

再看 O2, 相比于 O1,O2 采用了循环展开，每次循环处理两个元素（[rax] 和 [rax-8]），减少循环迭代次数，从而减少循环开销，连续访问内存地址，利用 CPU 预取机制，减少内存延迟。

最后看 O3, 在 O2 的基础上，O3 首先对 $n <= 3$ 的情况单独处理，避免循环展开的冗余。O3 使用更加激进的 Unroll 策略, 每轮循环处理 4 个 double 元素，循环次数仅为 n/4。

六、 总结

这次实验，我全面的了解了 cache 优化，unroll 技术，浮点数的舍入误差以及底层汇编代码实现等等，惊奇的发现，底层优化采用的技术正是这几个关键的技术，可见其重要性。更培养我设计实验探索问题的能力。

参考文献