南開大學
Nankai University

计算机学院

并行程序设计实验报告

SIMD 编程

马浩祎

学号：2213559

专业：计算机科学与技术

2024 年 4 月 27 日

目录

1 摘要

Abstract

本次实验首先设计了两个串行算法，并给出性能分析，做出了 baseline 的选择。然后进行存储格式的转换，设计位向量查询算法。在此基础上，自行设计出位向量重排序和 128 位分解的算法，可以兼具串行算法的及时停止特点和位向量的存储压缩特点。然后进一步嵌入 SSE,AVX 的数据类型和函数，真正实现 SIMD 的并行计算。同时给出层进级别的性能分析，证实这四个 SIMD 优化程序的性能有着层进的提升。最后分析了 SIMD 优化的汇编层级代码以及 Vtune 收集的具体执行数据。给出了一些 SIMD 优化算法性能优秀的原因。

关键词： SIMD，倒排索引，列表求交，并行计算

2 问题描述

2.1 背景介绍

在当代的数据库里，索引是检索数据最有效率的方式。但考虑搜索引擎的如下特点：

- 搜索引擎面对的是海量数据。像 Google，百度这样大型的商业搜索引擎索引都是亿级甚至百亿级的网页数量
- 搜索引擎使用的数据操作简单。一般而言，只需要增、删、改、查几个功能，而且数据都有特定的格式，可以针对这些应用设计出简单高效的应用程序。
- 搜索引擎面临大量的用户检索需求。这要求搜索引擎在检索程序的设计上要分秒必争，尽可能的将大运算量的工作在索引建立时完成，使检索运算尽量的少。

我们就不能去构建简单的索引了，早在 1958 年，IBM 就在一次会议上展示了一台“自动索引机器”。在当今的搜索引擎里也经常能看到它的身影，它就是——倒排索引。倒排索引又叫反向索引，它是一种逆向思维运算，是现代信息检索领域里面最有效的一种索引结构。

为满足用户需求，顺应信息时代快速获取信息的趋势，开发者们在进行搜索引擎开发时对这些信息数据进行了逆向运算，开发出“关键词——文档”形式的映射结构，实现了通过了物品属性信息对物品进行映射，可以帮助用户快速定位到目标信息，极大地降低了信息获取难度。

2.2 倒排索引求交

有了倒排索引后，搜索引擎在查询处理过程中，需要对这些索引集合进行求交操作，这个也是运算时间占比非常大的部分。假定用户提交一个 k 个词的查询，查询词分别是 $t_1, t_2, \ldots, t_k$，这 k 个关键词对应的倒排列表为 $\ell_{t_1}, \ell_{t_2}, \ldots, \ell_{t_k}$ 求交算法返回 $\cap_{1 \le i \le k} l(t_i)$。

首先求交会按照倒排列表的长度对列表进行升序排序，具体原因后面会介绍，使得：

$$|l(t_1)| \le |l(t_2)| \le \cdots \le |l(t_k)|$$

然后求交操作返回 k 个倒排列表的公共元素 $\cap_{1 \le i \le k} l(t_i)$。

2.3 SIMD 指令集

SIMD(Single Instruction Multiple Data) 即单指令流多数据流，是一种采用一个控制器来控制多个处理器，同时对一组数据（又称“数据向量”）中的每一个分别执行相同的操作从而实现空间上的并行性的技术。简单来说就是一个指令能够同时处理多个数据。

对于本问题而言，升序列表求交操作本身不是标准的单指令流多数据流模式，很难进行向量化。因此，可考虑其他利于向量化的存储方式和求交算法。例如，可考虑位图存储方式——每条链表用一个位向量表示，每个 bit 对应一个 DocID，某位为 1 表示该链表包含此 Doc、为 0 表示不包含。从而求交运算就变为两个位向量的位与运算。这种算法的优点是非常适合 SIMD 并行化，并行效率很高，下面我也会实现相关编程。

2.4 实验环境

Windows 平台下，硬件：CPU：Intel(R) Core(TM)i9-14900HX,24 核，基准速度 2.2GHz，内存容量 16GB，Cache L1: 2.1MB;L2: 32.0MB;L3: 36.0MB。软件：编程 IDE 为 CodeBlocks，编译器为 GNU GCC Compiler，实验用到的深入分析工具为 Vtune 2024.0.1，Godbolt 网站。

2.5 有关数据/性能

2.5.1 数据集

本问题研究所采用的数据集是给定好的 1000 条查询及 1756 条倒排索引列表，经过初步的数据分析，这些列表包含 DocID 最多 30000 个，最大 DocID 达 25205174，具体详见 github 仓库 src 文件夹下生成的各种数据信息。

2.5.2 性能测试方法

本问题研究的性能测试主要以时间为主，单位均为秒，profileing 部分可能会涉及其他指标，如资源插槽数等。时间测试方法如下：

```
LARGE_INTEGER freq, start, end0;
QueryPerformanceFrequency(&freq);
...Initial...
QueryPerformanceCounter(&start);
...Function...
QueryPerformanceCounter(&end0);
double elapsedSeconds = static_cast<double>(end0.QuadPart - start.QuadPart) / freq.QuadPart;
```

关于 baseline，我遵循串行最优性能原则，选择按元素求交算法的时间性能，具体见后。

2.5.3 实验正确性

由于并行优化的前提是保证结果的正确性，所以我在问题研究中的相关实验都保证结果是正确的。同时我在 github 仓库中公开了所有本问题研究过程中用到的代码项目，可下载验证结果。而正确结果的生成是由 stl 库 set_intersection 函数生成的标准结果，并保存在 github 仓库 src 文件夹下。

2.5.4 GitHub 仓库

仓库中包含实验涉及的所有程序，所有查询的信息和性能数据，以及整理出来的数据表。
https://github.com/Mhy166/parallel-programming.git

3 朴素串行算法

列表求交主要有两种方式: 按表求交(list-wise intersection); 按元素求交(element-wise-intersection)。下面我将编程实现这两种算法并给出性能分析。

3.1 按表求交

按表求交设计思想是：先使用两个表进行求交，得到中间结果再和第三条表求交，依次类推直到求交结束。这样求交的好处是，每一轮求交之后的结果都将变少，因此在接下来的求交中，计算量也将更少。下面我们看伪代码:

Algorithm 1 表求交串行算法

Input: $\boldsymbol{l}(\boldsymbol{t}_1), \boldsymbol{l}(\boldsymbol{t}_2), \ldots, \boldsymbol{l}(\boldsymbol{t}_k)$, Sorted by $|\boldsymbol{l}(\boldsymbol{t}_1)| \leq |\boldsymbol{l}(\boldsymbol{t}_2)| \leq \cdots \leq |\boldsymbol{l}(\boldsymbol{t}_k)|$
Output: $\cap_{1 \leq i \leq k} l(t_i)$

```
function FUNCTION(l(t_1), l(t_2), ..., l(t_k))
    S ← l(t_1)
    for i = 2 → k do
        for each element e ∈ S do
            found = find(e, l_i)
            if found = false then
                Delete e from S
            end if
        end for
    end for
end function
```

相对应的代码可以查看仓库，这里为节省篇幅不再赘述。

3.2 按元素求交

按元素求交算法会整体的处理所有的升序列表，每次得到全部倒排表中的一个交集元素。这类算法通常在 DAAT 查询下使用，以 Adaptive 算法为主，可以较好的应用提前停止技术。设计特点:

- 降低存储容量。在寻找文档时，由于索引列表也是升序的，故可以将小于其的 DocID 删除，直到等于或者大于。
- 及时停止。当在所有列表中寻找完某个文档之后，每条链表的剩余的未扫描文档数量也不相同，但只要其中一条链表走到尽头，这里会进行判定，则本次求交结束。
- 每次都从剩余文档数量最少的链表开始扫描，这样能够尽量缩短链表的扫描过程。

下面看本算法的伪代码:

Algorithm 2 元素求交串行算法

Input: $\boldsymbol{l}(\boldsymbol{t}_1), \boldsymbol{l}(\boldsymbol{t}_2), \ldots, \boldsymbol{l}(\boldsymbol{t}_k)$, Sorted by $|\boldsymbol{l}(\boldsymbol{t}_1)| \leq |\boldsymbol{l}(\boldsymbol{t}_2)| \leq \cdots \leq |\boldsymbol{l}(\boldsymbol{t}_k)|$

Output: $\cap_{1 \leq i \leq k} l(t_i)$

function FUNCTION($\boldsymbol{l}(\boldsymbol{t}_1), \boldsymbol{l}(\boldsymbol{t}_2), \ldots, \boldsymbol{l}(\boldsymbol{t}_k)$)
  $S = \emptyset$
  while No empty list **do**
    Reorder the lists by increasing number of undetected elements.
    $e = first\_unfound\_element(\boldsymbol{l}(\boldsymbol{t}_1))$
    $s \leftarrow 2$
    repeat
      $found = find(e, \boldsymbol{l}(\boldsymbol{t}_s))$
      $s \leftarrow s + 1$
    until $s = k$ *or* $found = false$
    if $s = k$ *and* $found = true$ **then**
      $S.add(e)$
    end if
  end while
end function

接下来看一下代码：

```
vector<list<uint32_t>> lists(queryIndices.size());//保存本次查询需要的列表的副本

while (!lists[0].empty()) {
    uint32_t tmp = lists[0].front(); lists[0].pop_front();
    uint32_t cnt = 1;
    bool flag = false;
    for (uint32_t i = 1; i < lists.size(); i++) {
        while (true) {
            if (lists[i].empty()) {
                flag = true;
                break;
            }
            if (lists[i].front() < tmp) lists[i].pop_front();
            else if (lists[i].front() == tmp) {
                lists[i].pop_front();
                cnt++;
            }
            else break;
        }
        if (flag) break;
    }
```

```
22      if (cnt == lists.size()) res.insert(tmp);
23      sort(lists.begin(), lists.end(), cmp);
24  }
```

3.3 串行性能简析

这里分析两串行算法性能时观察的规模是列表平均长度，下面给出规模上有代表性的结果表，即表 1。这里给出查询 ID 是为了证明所取查询的真实性。同时给出 1000 条查询整体的性能对比折线图，如图 3.1。

查询 ID	列表平均长度	按表求交时间/s	按元素求交时间/s
648	17	0.000008	0.000002
217	595	0.000221	0.000036
268	1103	0.000660	0.000072
52	5813	0.000565	0.000303
504	10049	0.123752	0.000647
940	15003	0.000415	0.000293
403	20269	0.154836	0.000833
312	25096	4.895270	0.002965
39	30000	6.423090	0.003568
732	30000	2.431860	0.005586

表 1: 串行算法性能对比表（秒）

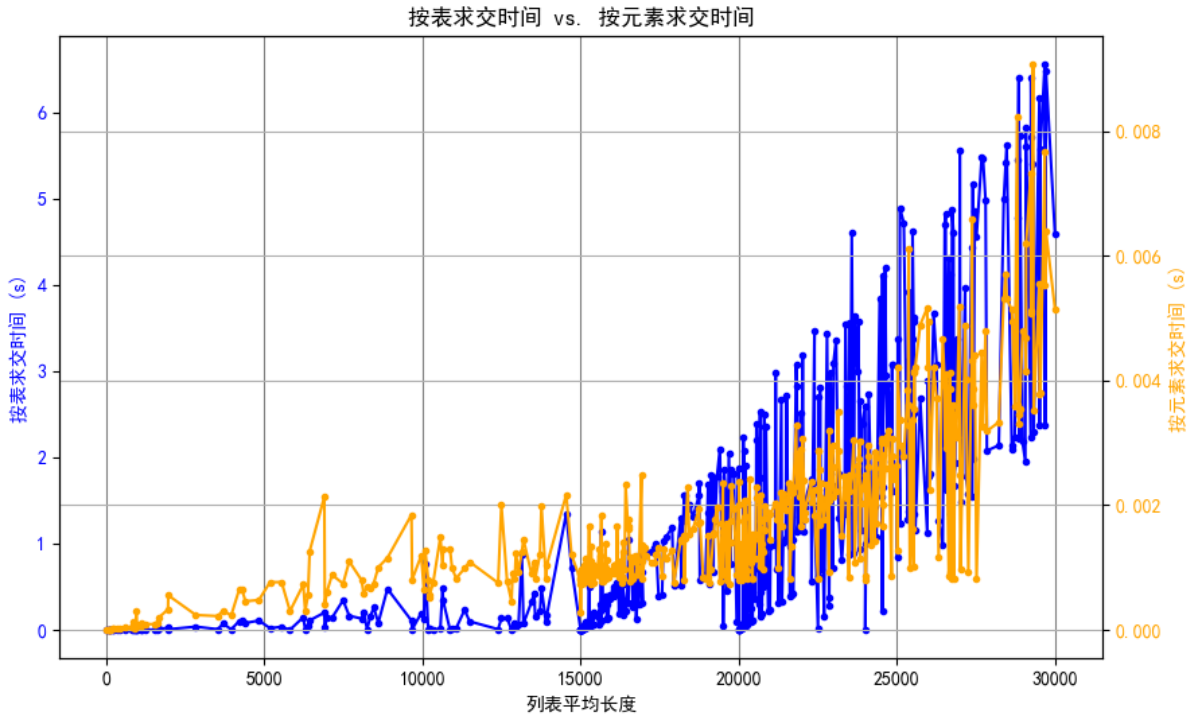

图 3.1: 串行算法性能对比图

通过图中数据可看出，两个算法都会随着规模的增大而更加耗时，但是按元素求交算法增长过程始终不会超过 0.01s，按表求交则一直以秒为单位增长，最终能达到 6s 左右。分析如下：

- 按表求交算法将结果集置为 L1 列表，虽然可以每次循环减少结果集的元素个数，但是显然每次循环，它都要耗费列表规模级别的时间。
- 按元素求交算法用到了索引列表间有序，和列表内数据有序的特征，及时筛除掉所有不可能成为结果元素的列表元素，同时加上了空表特判，及时结束，这在对于每个列表的数据大致范围区间间隔较大时的筛查起到了很好的效果！
- 由于数量级差异，为了体现性能趋势，这里图中是双纵轴。值得注意的是，两串行算法的趋势在所选用的列表平均长度上趋势近乎是一样的，这有力证明了这个规模选取的有效性。

根据串行最优性能原则，今后的 baseline 将会选取按元素求交算法的性能数据。

4 SIMD 优化算法

4.1 位向量化

同问题描述中所介绍，下面我们直接介绍位向量化的算法：

算法较简单，但是需要提前将存储格式转换成 bitset，具体转换见仓库源文件吧。而对于设计思想来说，bitset 数据结构提供了简单的按位与运算符，故直接对每个 bit 列表进行运算即可。这里我们直接介绍查询处理的代码：

```
1   for (auto idx : queryIndices) {
2       if (flag) {
3           res = bitmaps[idx];
4           flag = false;
5       }
6       else {
7           res &= bitmaps[idx];
8       }
9   }
```

然而，由于我之前分析过数据，得出最大 DocID 为 25205174，也就是说位向量 bitset 的大小会高达 25205200，然而索引长度最多 30000，意味着这些位当中最多只有 30000 位是 1，可见空间利用效率非常低，因此接下来我进行了一种优化方式，性能分析会在之后进行解释。

4.2 128/256 位--排序最小位向量优化法

这一步为 SSE/AVX 优化打下了很好的铺垫。而且这个方法是我自己思考后，自行设计的算法，该算法设计特点如下：

- 改变排序方式，选择具有最小的最大 DocId 的列表作为 L1 位向量，其后的列表的 DocID 不管多大，都可以不必考虑，只需保存 L1 位向量的大小的位数即可。如在查询 1 里，可以将位向量从 2500 万优化到 62 万位左右。

- 存储格式按照 128 位为单位去存放，这样还可方便未来的 SSE 优化。但是需要额外保证总体位向量的位数也应该扩展到 128 位的倍数。
- 存储格式按照 256 位为单位去存放，这样还可方便未来的 AVX 优化。但是需要额外保证总体位向量的位数也应该扩展到 256 位的倍数。

接下来简要看一下 128 位排序优化的存储和计算部分的代码：

```
sort(queryIndices.begin(), queryIndices.end(), compareBySize);//按照最大 DocID 排升序
vector<vector<bitset<128>>> rebit(queryIndices.size());
uint32_t id = queryIndices[0];
for (uint32_t i = 0; i < arrays[id][arrays[id].size() - 1]; i += 128) {
    bitset<128> tmp;
    for (uint32_t j = 0; j < 128; j++) {
        tmp[j] = bitmaps[queryIndices[0]][j + i];
    }
    rebit[0].push_back(tmp);
}
for (uint32_t i = 1; i < queryIndices.size(); i++) {
     for (uint32_t k = 0; k < arrays[id][arrays[id].size() - 1]; k += 128) {
         //存储同上，保持最大大小是排序后第一个位向量的最大 DocID
     }
}
//计算，结果直接就存在第一个位向量的 128 位子向量里面
for (uint32_t i = 1; i < queryIndices.size(); i++) {
    for (uint32_t j = 0; j < rebit[i].size(); j++) {
        rebit[0][j] &= rebit[i][j];
    }
}
```

4.3 SSE+AVX 优化

4.3.1 SSE-128 优化

接下来我将 128 位--排序最小位向量优化法当中的存储位图 128 位的数据类型改成 SSE 支持的 128 位向量即可。这里介绍所用到的 SSE 指令及其对 SIMD 加速的影响：

- $\_mm\_and\_si128$：这个函数执行两个 128 位整数寄存器之间的按位与操作。这种按位与操作可以同时处理多个数据元素。这种并行计算的能力是 SIMD 加速的核心，它允许一次性处理多个数据，从而提高了计算效率。
- $\_mm\_store\_si128$：这个函数将一个 128 位整数寄存器的内容存储到内存中的指定位置。在 SIMD 加速中，经常需要将计算结果存储到内存中以便后续使用。这个函数提供了一种高效的方式来处理大量数据。此外，存储操作也可以减少数据在寄存器和内存之间的频繁移动，从而提高了计算效率。

- $\_mm\_load\_si128$：这个函数从内存中加载一个 128 位整数到一个寄存器中。在 SIMD 加速中，加载操作通常用于将数据准备好，以便后续的并行计算。这个函数允许在单个指令中获取多个数据元素，从而提高了加载数据的效率。加载操作还可以减少数据在内存和寄存器之间的传输时间，从而减少了内存访问造成的延迟。

下面给出 SSE 优化的关键部分，即存取数据，计算处理的代码：

```
vector<vector<__m128i>> rebit(queryIndices.size());
//存入数据（第一个大向量的，其他类似）
for (uint32_t i = 0; i < arrays[id][arrays[id].size() - 1]; i += 128) {
    bitset<128> tmp;
    for (uint32_t j = 0; j < 128; j++) {
        tmp[j] = bitmaps[queryIndices[0]][j + i];
    }
    const uint32_t* bitmapPtr = reinterpret_cast<const uint32_t*>(&tmp);
    __m128i bitmap = _mm_load_si128((__m128i*)bitmapPtr);
    rebit[0].push_back(bitmap);
}
//计算按位与（这是循环体，循环部分省略）。
rebit[0][j] = _mm_and_si128(rebit[0][j], rebit[i][j]);
//取回数据
for (uint32_t i = 0; i < rebit[0].size(); i++) {
    bitset<128>tmp;
    _mm_store_si128((__m128i*) & tmp, rebit[0][i]);
    result.push_back(tmp);
}
```

4.3.2 AVX-256 优化

类似的，将数据类型改成 AVX 支持的 256 位向量即可。这里简要介绍所用到的 AVX 指令：

- $\_mm256\_load\_si256$：从内存中加载一个 256 位整数到一个 256 位整数寄存器中，可以进一步提高加载数据的效率和吞吐量。
- $\_mm256\_and\_si256$：执行两个 256 位整数寄存器之间的按位与操作。AVX 函数能够同时处理更多的数据元素，进一步提高了并行计算的效率。
- $\_mm256\_store\_si256$：将一个 256 位整数寄存器的内容存储到内存中的指定位置。通过一次性存储 256 位整数，减少了存储操作的开销，并提高了存储数据的效率。

同样，下面给出 AVX 优化的存取数据，计算处理的代码：

```
vector<vector<__m256i>> rebit(queryIndices.size());
//存入数据（第一个大向量的，其他类似）
for (uint32_t i = 0; i < arrays[id][arrays[id].size() - 1]; i += 256) {
```

```
        bitset<256> tmp;
            ...do the same as SSE
        __m256i bitmap = _mm256_load_si256((reinterpret_cast<const __m256i*>(bitmapPtr)));
        rebit[0].push_back(bitmap);
    }
    //计算按位与（这是循环体，循环部分省略）。
    rebit[0][j] = _mm256_and_si256(rebit[0][j], rebit[i][j]);
    //取回数据
    for (uint32_t i = 0; i < rebit[0].size(); i++) {
        bitset<256>tmp;
        _mm256_store_si256(reinterpret_cast<__m256i*>(&tmp), rebit[0][i]);
        result.push_back(tmp);
    }
```

4.4 SIMD 优化性能分析

接下来我们要分析串行 baseline 和上面四种层进优化的性能。以说明我的优化思路可以达到更好的效率。如表 2，与表 1 不同，表 1 保留了仅 6 位小数，这里由于数据精度较高，我多保留了 1 位。所选用的查询样例也和表 1 的查询样例一样，方便检查真实性。

列表平均长度	按元素求交时间	位向量化	128 排序优化	128 位 SSE 并行优化	256 位 AVX 并行优化
17	0.0000021	0.0013464	0.0028101	0.0011007	0.0004376
595	0.0000362	0.0013898	0.0020883	0.0011107	0.0006047
1103	0.0000721	0.0011305	0.0030368	0.0020187	0.0007267
5813	0.0003027	0.0011902	0.0019833	0.0012230	0.0006195
10049	0.0006470	0.0014469	0.0028836	0.0013194	0.0005962
15003	0.0002930	0.0012131	0.0010344	0.0005815	0.0003393
20269	0.0008331	0.0026441	0.0000914	0.0000461	0.0000246
25096	0.0029649	0.0010250	0.0002780	0.0001290	0.0000763
30000	0.0035677	0.0013276	0.0000501	0.0000374	0.0000333
30000	0.0055859	0.0029690	0.0011053	0.0002918	0.0001654

表 2: SIMD 并行优化整体性能表（秒）

接下来，与准确科学的表格不同，我们看更为直观的总体折线图 4.2 和分规模性能对比图 4.3。(有必要说明：图 4.2 用 python 绘制，由于曲线较陡，为提高辨识度和观感，采用 rolling 平滑窗口，窗口大小为 5)

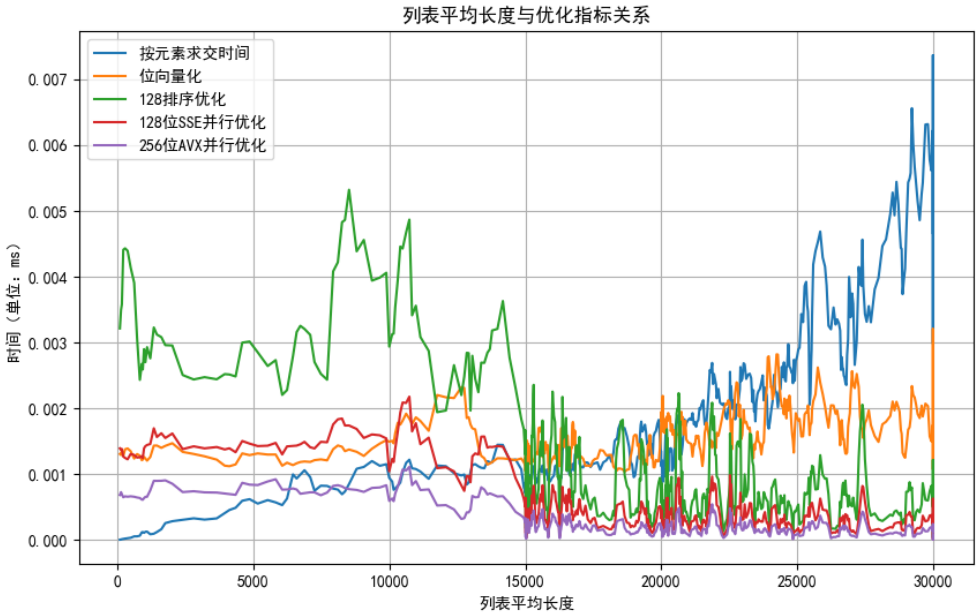

图 4.2: SIMD 并行优化——整体性能折线图

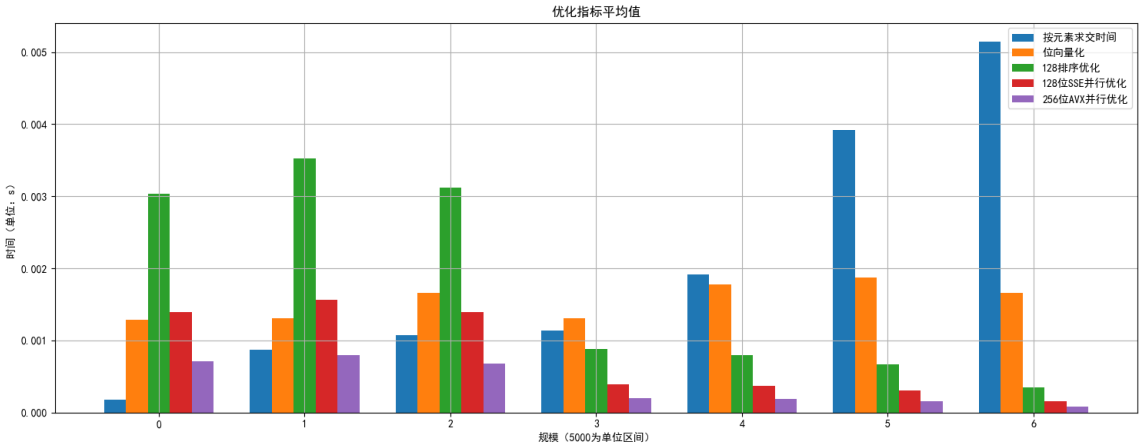

图 4.3: SIMD 并行优化——规模平均性能对比图

观察数据，我们可以进行如下分析和得出结论：

- 在规模较小的时候，列表的长度要比 DocID 决定的位向量长度要小很多，此时按元素求交算法可以很快遍历完，并终止查询。因此，图表数据显示规模较小时按元素求交算法性能会优于并行算法。
- 当规模逐渐增大，并行算法优势便体现出来，而且呈现逐级优化的性能。
  - 位向量算法作为转换存储方式的首次尝试，具有空间的压缩优势，和时间上位运算的快速效率。其长度固定不变，可以解释其曲线走势相对平稳。
  - 128 位排序优化算法，这是我结合按元素求交算法的及时停止技术和位向量算法的快速存储运算技术所设计的优化算法。可以看到在较大规模下，其具有比位向量算法优秀的时间性能，

甚至接近 SSE 优化算法，这印证了我所设计算法的高效性。但也需要注意的是，规模较小时该算法性能将不如串行算法。

- SSE+AVX 优化，作为更深层次的优化算法，也是真正手动使用了 SIMD 并行策略的算法。我们可以看到，相对于前面的算法，SSE 具有更好的性能，而且受规模影响较小。与 128 位 SSE 比较，AVX 的 256 位向量存储则表现出更优秀的性能，始终不会超过 0.001s。
- 加速比则和规模有直接关系，SSE 对于串行加速比可以最大达到 20 倍，规模较小时则无法起到加速效果。但是相对于 128 排序优化，（因为二者之间只有数据类型不同，可以非常好的体现控制变量的思想和 SIMD 指令集的效力），还是有 1 至 2 倍的加速比，AVX 则有 3 至 4 倍的加速比，这证明了通过使用 SIMD，我实现了并行加速优化的策略。

5 基于工具的 Profiling 分析

5.1 Godbolt 分析

Godblot 是一个线上生成汇编代码的网站，可以直观地看出我所写的高级语言代码对应的汇编代码，以获得深入的并行程序理解。这里以 128 位优化和 SSE 优化进行底层汇编代码的对比。由于代码过长，这里我挑选出最有对比价值的查询处理部分的代码，它可以体现存储，指令，运算多方面的差异！如图 5.4。

```
576         call    std::vector<std::vector<std::bitset<1
577         mov     rdx, rax
578         mov     eax, DWORD PTR [rbp-60]
579         mov     rsi, rax
580         mov     rdi, rdx
581         call    std::vector<std::bitset<128ul>, std::
582         mov     rsi, rbx
583         mov     rdi, rax
584         call    std::bitset<128ul>::operator&=(std::b
585         add     DWORD PTR [rbp-60], 1
586 .L57:
```

(a) 128 位优化汇编代码

```
596         call    std::vector<long long __vector(2),
597         movdqa  xmm0, XMMWORD PTR [rax]
598         movaps  XMMWORD PTR [rbp-144], xmm0
599         movdqa  xmm2, XMMWORD PTR [rbp-2480]
600         movaps  XMMWORD PTR [rbp-160], xmm2
601         movdqa  xmm1, XMMWORD PTR [rbp-144]
602         movdqa  xmm0, XMMWORD PTR [rbp-160]
603         pand    xmm0, xmm1
604         movaps  XMMWORD PTR [rbp-2480], xmm0
```

(b) SSE 优化汇编代码

图 5.4: 不同算法的汇编代码

通过研究汇编代码，我可以得出如下结论：

- 两图中 call 都是在调用一个存储函数，可以看到 128 位优化的存储是 bitset<128ul>,SSE 优化则是 long long ___vector, 指针的类型也有所差异，这体现了汇编级别存储的差异。也体现出 SIMD 的向量寄存器的使用。
- 二者指令有所不同。128 位优化的指令比较常见，如 mov，add；但是 SSE 优化的指令则是专属的，用 movdqa 来存储，movaps 来对齐，这很好的体现了 SIMD 中专属指令的使用。
- 对查询按位与运算的处理不同。可以看到 128 位优化采用调用 bitset 专属函数来进行按位与操作，SSE 优化则是用专属指令 pand 来对 xmm0 和 xmm1 两个向量器进行运算。显然 128 位的需要额外考虑调用的时间耗费，而 SSE 则直接进行 SIMD 的专属指令的快速运算。这应该是 SSE 比前者性能要好的一方面原因。

5.2 Vtune 分析

为了探究程序运行时的细节，这里利用 Vtune 工具对 SSE 优化进行简要分析。这里主要选取如下指标：

- CPI：平均每条指令的周期数，一般情况下越低代表执行效率越高。
- Instructions：执行指令数，一般情况下越低代表效率越高。
- L1 Hit：L1 缓存的命中次数，可以有效表示程序访问 cache 的次数。
- CPU Time：CPU 执行时间，越低越好。
- TopDown.Slots：指的是处理器资源的插槽，能一定程度上表示程序利用 CPU 的效率，越高，代表利用越充分。

指标	按元素求交算法	SSE 优化
CPI	0.354	0.226
Instructions	115200000	43200000
L1 Hit	3000009	30000090
CPU Time	0.01	0.008
TopDown.Slots	10000003	200000300

本例选取的是查询 ID 为 10 的查询。可以看到，优化过后 CPI 有一定的降低，这里 CPI 小于 1 应该是因为现代处理器中 pipeline 的存在和超标量无依赖优化，提升了吞吐率。指令数也有所缩减，这是因为 SSE 能在单个指令下处理更多的数据。缓存命中数和资源插槽也增加了，我推测是 SSE 的优化更能充分利用寄存器等内置资源，而且存储格式的优化压缩也适合 cache 的命中。最后 CPU 时间减少是毋庸置疑的，前述 3 条分析可以解释性能的增强。

6 总结

本次实验是倒排索引问题的 SIMD 优化实验，也作为期末研究的一个子实验，主要有以下收获：

- 实现了两个串行算法，并给出性能分析，做出了今后的 baseline 的选择。
- 进行存储格式的转换，实现位向量查询算法，证实其在大规模下的优良性能。
- 进行技术融合，自行设计出重排序 128 位存储算法，可以兼具串行算法的及时停止特点和位向量的存储压缩特点。并证实大规模下性能的进一步增强。
- 嵌入 SSE,AVX 的数据类型和函数，真正实现 SIMD 的并行计算。分析了加速比，证实 SSE，AVX 的性能可以在各种规模下都有着更优秀的表现！
- 进行 SIMD 优化的汇编层级代码分析以及 Vtune 具体执行数据的分析。给出了一些证实优化算法性能优秀的原因。

总之，这次实验帮我打好了未来子问题研究和期末研究的基础，比如多线程中也可应用 SIMD 技术，同时增强了我的并行编程能力，加强了我对 SIMD 指令集的理解。