南開大學
Nankai University

计算机学院

并行程序设计实验报告

SIMD 编程实验——倒排索引求交

姓名：张铭

学号：2211289

专业：计算机科学与技术

2024 年 4 月 28 日

目录

1 实验环境

本次 SIMD 实验涉及到多种 SIMD 并行指令集的使用，SIMD 并行指令集主要包括基于 ARM 处理器架构的 NEON 指令集，以及由英特尔开发的基于 x86 架构的 SSE/AVX 指令集。因此我们将在鲲鹏服务器环境下执行 NEON 指令，在 IntelDevcloud 服务器环境下执行 SSE/AVX/AVX512 指令。
github 仓库 SIMD 项目链接：https://github.com/DianGun-otto/Parallel-Project/tree/main/SIMD

2 选题描述和背景知识

本次 SIMD 实验为自主选题，与期末研究报告结合，选择倒排索引求交进行 SIMD 并行化算法求解。

2.1 倒排索引

对于一个有 U 个网页或文档（Document）的数据集，若想将其整理成一个可索引的数据集，则可以为数据集中的每篇文档选取一个文档编号（DocID），使其范围在 [1,U] 中。其中的每一篇文档，都可以看做是一组词（Term）的序列。则对于文档中出现的任意一个词，都会有一个对应的文档序列集合，该集合通常按文档编号升序排列为一个升序列表，即称为倒排列表（Posting List）。所有词项的倒排列表组合起来就构成了整个数据集的倒排索引。

2.2 倒排列表求交

在获得倒排索引列表的基础上，我们考虑倒排列表求交（List Intersection）。它指的是将多个倒排列表（即文档或记录中某个特定项的出现位置列表，比如某个关键词在不同文档中的出现位置列表）进行交集操作，以找到它们共同出现的位置或文档。

当用户提交了一个 k 个词的查询，查询词分别是 $t_1, t_2, ..., t_k$，表求交算法返回 $\cap_{1\leq i\leq k} l(t_i)$。

例如查询 “2014 NBA Final”，搜索引擎首先在索引中找到 “2014”，“NBA”，“Final” 对应的倒排列表，并按照列表长度进行排序：

$$l(2014) = (13, 16, 17, 40, 50) \tag{1}$$

$$l(NBA) = (4, 8, 11, 13, 14, 16, 17, 39, 40, 42, 50) \tag{2}$$

$$l(Final) = (1, 2, 3, 5, 9, 10, 13, 16, 18, 20, 40, 50) \tag{3}$$

求交操作返回三个倒排列表的公共元素，即：

$$l(2014) \cap l(NBA) \cap l(Final) = (13, 16, 40, 50) \tag{4}$$

2.3 位图

位图是一种数据结构，用于表示一组二进制位的集合。在位图中，每个位都只能是 0 或 1，分别表示集合中的元素是否存在或者是否被标记。使用位图来存储倒排列表时，位图的长度通常等于文档总数，每个位表示一个 DocID 是否包含在该 term 对应的倒排列表中。假定文档总数 U=10，一次搜索中的词组为 “the boy first”。我们通过以下示例来展现位运算操作实现倒排列表求交的过程。

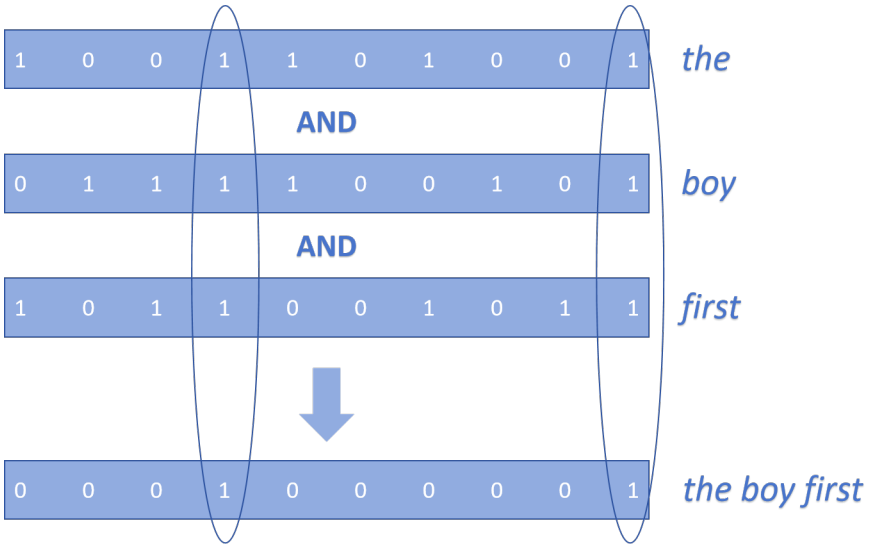


图 2.1: 位图存储的倒排列表通过 AND 位运算实现求交

3 list-wise 按表求交

3.1 算法设计

按表求交基本思想是：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。伪代码如下所示：

Algorithm 1 Set versus Set 算法

Input: l(t1),l(t2),...,l(tk), 按升序排序，|l(t1)| |l(t2)| ... |l(tk)|

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

```
for i=2 to 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
return S
```

3.2 串行算法的实现与优化

3.2.1 顺序查找求交

最基础的倒排索引求交算法使用在 find 过程中使用顺序查找策略，暴力从头至尾遍历 l 中的所有元素，并返回 found 结果

```
bool find(const vector<uint32_t>list,int element){
for(int i=0;i<list.size();i++)
```

```
    if(list[i]==element)
        return true;
return false;

```

3.2.2 二分查找求交

考虑到顺序查找的效率较低，我们在 find 过程中使用二分策略进行查找，从而将单次查找的时间复杂度由线性级降低到了对数级，大大提高了程序效率。

```
bool binarySearch(const vector<uint32_t> list, int element) {
    int left = 0;
    int right = list.size() - 1;
    while (left <= right) {
        int mid = left + (right - left) / 2;
        if (list[mid] == element) return true;
        else if (list[mid] < element) left = mid + 1;
        else right = mid - 1;
    }
    return false;
}
```

3.2.3 双指针查找求交

此外，在 find 过程中我们还使用了双指针法（Two Pointers）进行查找。由于每条列表都是有序的，因此我们可以在 l 上建立一个指针索引 index，每次查找元素 e 后更新指针索引，这样做可以减少查找下个元素时比较的次数。

```
bool find(vector<uint32_t>list,int element,int &index){
    while(index<list.size()-1&&list[index]<element)
        index++;
    if(list[index]==element)
        return true;
    return false;
}
```

3.2.4 多链路式求交

将多条列表分为两组，分别进行组内求交，最后对两组得到的结果再求交

```
for(int i=2;i<Lists.size()/2;i++){
    vector<uint32_t> l0=Lists[i];
```

```
    vector<uint32_t> l1=Lists[i+1];

    for(int j=0;j< Lists[0].size();j++){
        int element= Lists[0][j];
        bool found=binarySearch(l0,element);
        if(!found)
             Lists[0].erase(Lists[0].begin()+j--);
    }

    for(int j=0;j< Lists[1].size();j++){
        int element=Lists[1][j];
        bool found=binarySearch(l1,element);
        if(!found)
            Lists[1].erase(Lists[1].begin()+j--);
    }
}
//汇总结果
for(int j=0;j<Lists[0].size();j++){
        int element=Lists[0][j];
        bool found=binarySearch(Lists[1],element);
        if(!found)
            Lists[0].erase(Lists[0].begin()+j--);
    }
```

3.3 并行算法的实现

3.3.1 位图优化策略

按数组存储文档编号的串行算法一次性无法处理多组数据，运算效率较低，因此，我们考虑以向量的形式一次同时处理多组数据。由于上述求交策略本身不是标准的单指令流多数据流模式，也很难直接进行向量化。因此考虑按位图存储的优化策略。

我们的位图存储方式为: 使用 32 位无符号整形数组存储，每一个 int 型数据的 32 个 bit 的 0/1 代表一个 DocID 是否包含在该 term 对应的倒排列表中。

```
for(vector<uint32_t> arr : Lists){
    vector<uint32_t> l(size,0);
    for(int num : arr){
        unsigned int bitIndex = (num)/(sizeof(uint32_t)*8);
        int offset = num%(sizeof(uint32_t)*8);
        l[bitIndex] |= (1 << offset);
    }
    bitmap.push_back(l);
```

```
}

```

对于 list-wise 算法，不难发现，通过两个列表向量之间的按位与运算，可以取得与指令集并行类似的并行效果，因此，将这一位图基础算法归纳为并行优化算法之一。具体实现代码如下：

```
vector<uint32_t> vector<uint32_t> list_wise_bitmap(vector<vector<uint32_t>>bitLists){
    for(int i=1;i<bitLists.size();i++)
        for(int j=0;j<bitLists[0].size();j++)
            bitLists[0][j]&=bitLists[i][j];

    return bitLists[0];
}
```

3.3.2 NEON/AVX/SSE

处理器指令集中的向量化操作是利用 SIMD（Single Instruction, Multiple Data）指令执行多个数据元素的并行计算。这种操作能够在单个指令周期内同时处理多个数据，从而提高了计算效率和性能。SIMD 指令执行多个数据元素的向量化并行计算主要包括数据加载 (load)、向量化指令执行 (Vectorized Instruction Execution) 以及数据存储 (Store)。对于 list-wise 的位图并行优化，我们考虑两列表求交时多个 int 型整数同时按位与操作，实现向量化。以 NEON 指令集为例，部分代码如下：

```
    for (int j = 0; j + 3 < bitSize; j += 4) {
        uint32x4_t intersec_vec = vld1q_u32(&intersection[j]);
        uint32x4_t bitList_vec = vld1q_u32(&bitLists[i][j]);
        intersec_vec = vandq_u32(intersec_vec, bitList_vec);
        vst1q_u32(&intersection[j], intersec_vec);
    }
```

4 element-wise 按元素求交

4.1 算法设计

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（Document at a time）查询下使用，DAAT 的基本思路就是在各个列表中寻找当前文档，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，则本次求交结束。

4.2 串行算法的实现

4.2.1 基础求交策略

各个文档集合按大小由小到大排序，假设每个 term 所包含的文档集合内部是无序的，根据提前停止的技术，以 Adaptive 算法为主，可以得到按元素求交的基础算法如下：

```
vector<int> element_wise(vector<vector<uint32_t>>vec){//按元素求交
vector<int> result;
    bool found = false;
    for(int i = 0;i < vec[0].size();i++){
        found = false;
        for(int j = 1;j < vec.size();j++){
            found = find(vec[j],vec[0][i]);
            if(!found) break;
            if(found && j == vec.size() - 1) result.push_back(vec[0][i]);
        }
    }
    return result;
}
```

4.2.2 双指针查找求交

与 list-wise 类似，将单一指针更换为多指针。考虑每个 term 所包含的文档集合是有序的，我们引入 int 型数组 point，修改 find() 函数得到 element-wise 的串行优化算法。每一轮中，在每一个文档集合寻找完毕后，记录此时的数组索引，下一轮的寻找则从此开始，从而大大减少了寻找和比较的次数。

4.2.3 二分查找求交

与 list-wise 类似，将基础算法中的 find() 函数替换为 binarySearch()，可以得到 element-wise 的另一个串行优化算法。

4.3 并行算法的实现

按数组存储文档编号的串行算法一次性无法处理多组数据，运算效率较低，因此，我们考虑以向量的形式一次同时处理多组数据。由于上述求交策略本身不是标准的单指令流多数据流模式，也很难直接进行向量化。由此，我们引入按位图存储的优化策略。具体优化方式为，每一个 32 位 int 型数据存储 32 个 bits，每个 bits 的 0/1 代表某一个文档编号是否存在在这一 term 的查询结果中。

4.3.1 位图优化策略

按元素求交的位图优化策略包括两个过程，第一步是对位图化的倒排列表进行转置（transpose）操作，转置的目的是，由对每两个位图化的列表并行求交改为对每一个文档编号进行并行判断。转置完成后，对其中每一行数据进行判断：如果其每一个 bit 位均为 1（可以转化为判断这个 int 型整数是否等于某个值），则求交结果包含该文档编号。由此，我们实现了对这一整行数据的并行化处理。转置后的具体代码如图所示。

```
vector<uint32_t> element_wise_bitmap(vector<uint32_t>bitLists_transpose,int test_number){
    int num = bitLists_transpose.size();
    vector<uint32_t> intersection(num/(sizeof(uint32_t)*8)+1);
```

```
    for(int i = 0;i < bitLists_transpose.size();i ++){
        if(bitLists_transpose[i] == test_number){
            int offset = i%(sizeof(uint32_t)*8);
            intersection[i/(sizeof(uint32_t)*8)] |= (1 << offset);
        }
    }
    return intersection;
}
```

5 对比分析

我们采取读文件的方式，在 ExpQuery.txt 中读取每一行的查询记录，再在 ExpIndex 中读取每条查询记录对应 term 下标的数组，从而获得倒排列表的二维数组以进行求交操作。由此，我们初步将问题规模设置为查询记录的数量 n($1 \leq n \leq 1000$)，测试不同问题规模下串行算法/并行算法的性能、不同算法/编程策略对性能的影响等。

5.1 串行算法中不同策略的对比

5.1.1 list-wise

对于 element-wise 的串行算法，我们测得以下数据：

Queries(n)	Basic	Binary	Pointer	Multi	Queries(n)	Basic	Binary	Pointer	Multi
10	5140.96	1902.01	1906.55	1876.78	200	131666	46493.7	46434.9	45428.7
20	12755.7	4804.78	4775.42	4731.51	300	196049	68898.2	68878.9	67842
30	20921.8	7923.09	7873.99	7868.8	400	263739	92862.7	92842.6	91458
40	28302.2	10684.2	10684.5	10587.6	500	323614	113959	113958	111911
50	36754.7	13819.8	13743.8	13628.9	600	389343	137207	137088	134738
60	41329.8	15560.2	15426.7	15333.7	700	456464	161838	161810	157686
70	47319.6	17864.9	17797.2	17545.7	800	524293	186112	186085	180853
80	52192.2	19581.5	19482.3	19183.5	900	592358	210316	210341	204480
90	59474.5	22380.1	22314.9	21935.6	1000	663716	235759	235746	229873
100	67070.3	25272.5	24882.7	24603.8					

表 1: 四种 list-wise 串行算法的性能对比

我们发现，相比于基础串行算法而言，二分法/指针法/多链法均有较好的优化效果，并且对于算法的优化程度大致相同。具体而言，二分法针对一个有序数组进行二分查找，优化了查找方法；指针法通过设置指针，记录上一次查找时的结束位置，避免了大量重复的比较和查找操作；而多路链式算法将多条列表分为两组，分别进行组内求交，最后对两组得到的结果再求交，减少了循环的次数，并且在求交查找时使用二分查找，进一步提高效率。

5.1.2 element-wise

对于 element-wise 的串行算法，我们测得以下数据：

我们发现，相比于基础串行算法而言，二分法/指针法均有较好的优化效果，并且对于算法的优化程度大致相同，优化的方式如同 list-wise，优化的效果同样也是显而易见的。

Queries(n)	Basic(ms)	Binary(ms)	Pointer(ms)	Queries(n)	Basic(ms)	Binary(ms)	Pointer(ms)
10	4424.49	1227.8	1221.44	200	115354	29939.3	29726.9
20	10932.9	3028.48	3005.51	300	171931	44542.6	44240.9
30	17812.6	4931.72	4907.33	400	231345	60045.4	59666.2
40	24102.8	6672.18	6638.77	500	282590	73342.4	72776
50	31387.3	8691.8	8656.84	600	340761	88226	87505.2
60	35277.1	9757.87	9716.3	700	399495	103398	102807
70	40393.2	11180.6	11126.1	800	459087	119016	118151
80	44694.3	12395.5	12314.7	900	518611	134387	133591
90	50979.5	14139.4	14085.4	1000	580229	151196	149937
100	61562.4	16030.9	15971.4				

表 2: 三种 element-wise 串行算法的的性能对比

5.1.3 list-wise VS element-wise

以串行算法的指针优化策略为例，查询数量 n 设定为 $100 \leq n \leq 1000$，我们以折线图的形式对比 list-wise 和 element-wise 的性能表现：

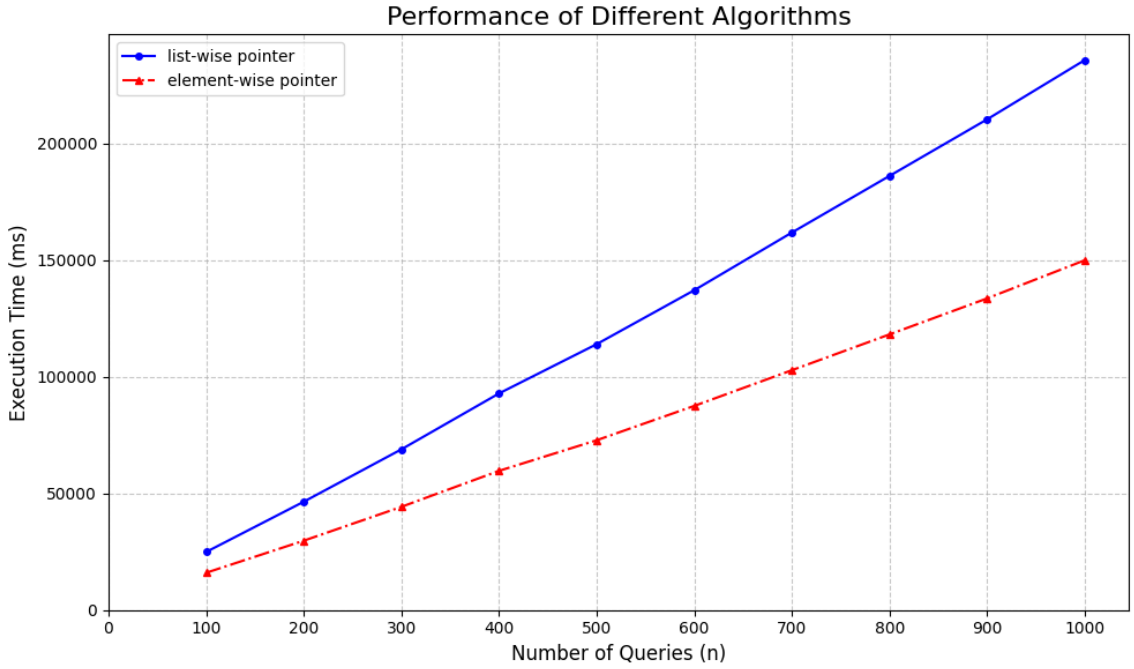


图 5.2: list-wise pointer 与 element-wise pointer 的性能对比

总体上来看，对于串行算法，element-wise 的性能优化效果要好于 list-wise，加速比约为 **1.5**。原因之一是，element-wise 运用了提前停止技术，在按元素查找的过程中，只要在某个倒排列表中未找到该元素，这一循环提前终止。因此，element-wise 算法的时间复杂度会小于 list-wise。

5.2 并行算法中不同策略的对比

5.2.1 list-wise-bitmap VS list-wise-NEON VS element-wise-bitmap

对于 list-wise 和 element-wise 的并行算法，我们绘制以下图表：

Queries	ListBitmap	ListNEON	ElementBitmap	Queries	ListBitmap	ListNEON	ElementBitmap
10	56.456	48.002	1674.91	200	992.022	902.022	32485.3
20	83.31	74.53	2710.47	300	1499.41	1321.27	50550.7
30	119.521	106.549	4153.62	400	1973.81	1757.65	69823.2
40	157.887	141.111	5547.02	500	2545.65	2279.39	84935.2
50	203.682	181.37	7221.95	600	3062.62	2728.65	101492
60	267.643	239.357	9048.54	700	3576.13	3188.59	115877
70	340.298	303.789	10551	800	4146.95	3778.16	136748
80	386.661	345.495	12607.3	900	4701.21	4111.66	146731
90	436.826	383.413	14257.9	1000	5000.21	4464.77	162897
100	480.386	427.252	15725.7				

表 3: list-wise bitmap、list-wise NEON 和 element-wise bitmap 的性能对比

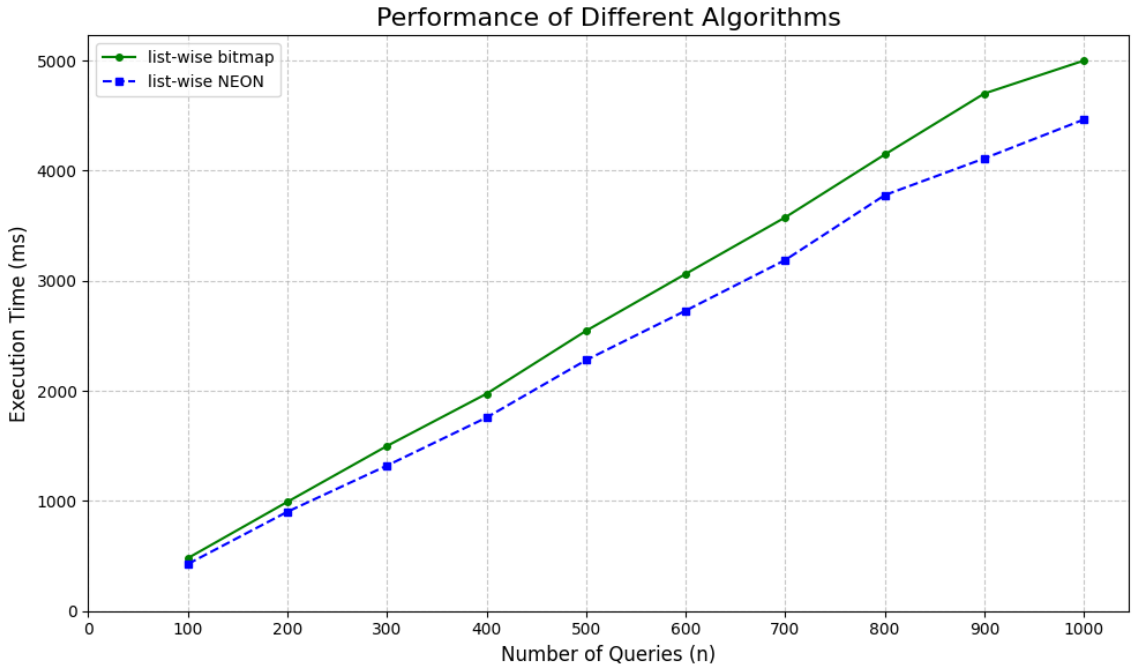


图 5.3: list-wise bitmap 与 list-wise NEON 的性能对比

不难发现，list-wise 的两种并行策略均对性能的提升有显著的效果 (这类讨论在 5.3 中呈现)，其中，位图存储的形式以牺牲空间为代价，大大缩短了程序运行时间。而 NEON 指令集发挥了 SIMD 向量化并行的优势，在此基础上进一步优化了性能。

5.2.2 list-wise-SSE: 对齐指令与非对齐指令间性能对比

我们在 IntelDevcloud 服务器上分别测试收集对齐和不对齐的数据，如表4 所示，我们发现使用对齐和不对齐指令的性能差异并不大。基于这一实验现象，我们推测原因与 list-wise 算法的特性有关，由于我们对列表进行了预处理，将其长度填补至 4 的倍数，因此使用对齐或不对齐指令时，单次求交的存取次数的差值为 1，又因为数据集中每个 query 的待求交列表数量较少（2 5 条），因此整体上程序性能趋于一致，并未产生较大差异。

Queries(n)	SSE-aligned	SSE-unaligned	Queries(n)	SSE-aligned	SSE-unaligned
10	76.848	81.474	200	4306.97	4291.96
20	184.786	179.263	300	6154.57	6125.37
30	343.541	324.456	400	8641.19	8571.51
40	527.872	506.251	500	11789.5	11739.6
50	771.263	754.489	600	15526.2	15506.3
60	1097.37	1077.47	700	19805.6	19893.5
70	1488.79	1503.65	800	24709.5	24845.6
80	1953.53	1967.51	900	30243.4	30431.4
90	2472.81	2488.65	1000	36269.5	36495.2
100	3051.09	3068.21			

表 4: list-wise-SSE 使用对齐指令与非对齐指令间性能对比

5.3 串行算法与并行算法的对比

5.3.1 list-wise

我们以串行算法中的性能优化较为明显的 list-wise-pointer 策略以及并行算法中的 NEON 策略进行对比分析，得到的加速比如图所示：

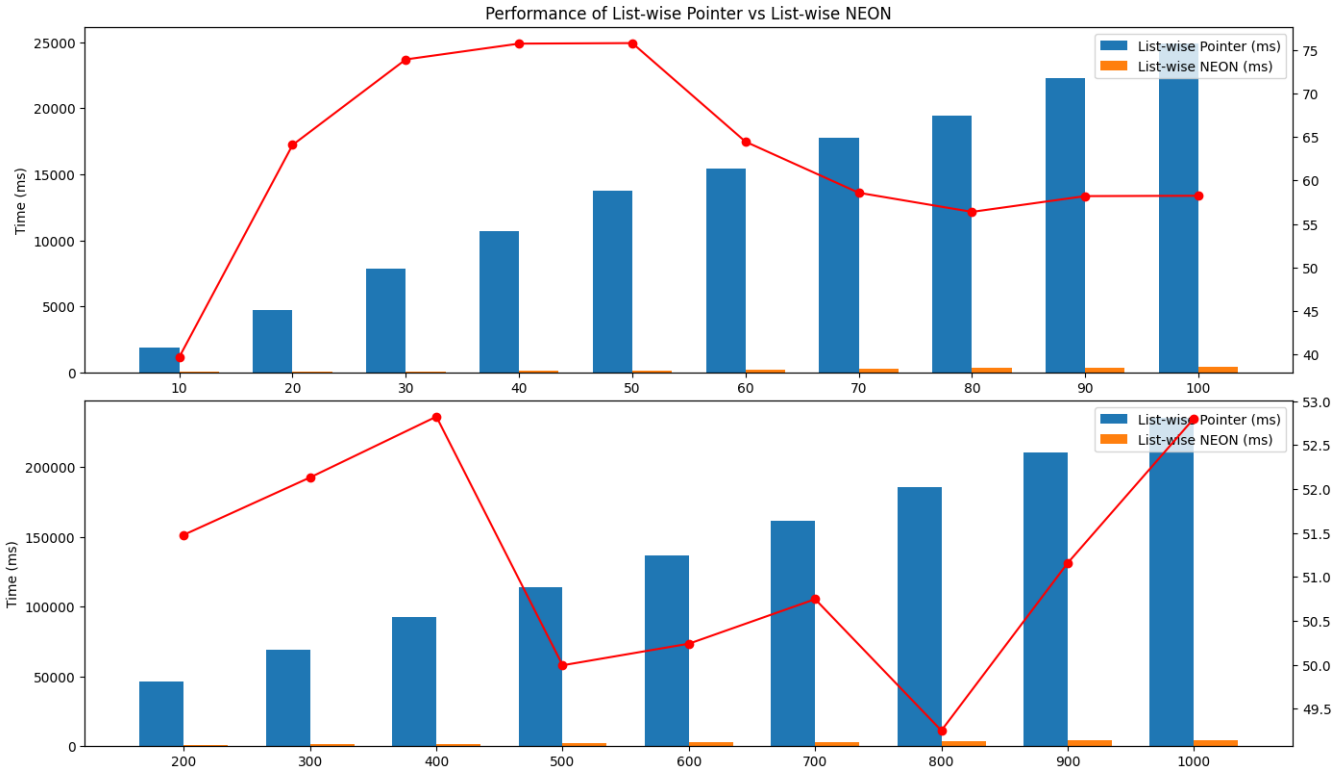


图 5.4: list-wise pointer 与 list-wise NEON 的性能对比以及加速比

可以得到，在查询记录的数量 $1 \leq n \leq 100$ 时，加速比在 **40~75** 之间浮动，变化幅度较大，而随着查询数量的增加，即 $100 \leq n \leq 1000$ 时，加速比稳定在 **50** 上下。因此，可以粗略估计本次实验以 list-wise 算法得到的并行化的加速比为 **50**。

5.3.2 element-wise

与 list-wise 算法不同的是，在串行算法中表现优异的 element-wise 较难实现并行化，由于其独有的中断策略，在考虑将他并行化时首先得将倒排列表矩阵转置 (transpose)，再对转置后的二维数组的每一行作判断 (如若所有的 bit 为均为 1，则将这一行对应的文档编号加入最终交集的结果)。然而，因为以位图方式存储的倒排列表转置过程过于复杂，加之对每一行 bit 位作判断的过程计算过多，因此，element-wise 在并行过程中的效果不尽人意，并行后的程序运行时间反而长于串行算法。数据如下:

Queries(n)	ElementPointer	ElementBitmap	Queries(n)	ElementPointer	ElementBitmap
10	1221.44	1674.91	200	29726.9	32485.3
20	3005.51	2710.47	300	44240.9	50550.7
30	4907.33	4153.62	400	59666.2	69823.2
40	6638.77	5547.02	500	72776	84935.2
50	8656.84	7221.95	600	87505.2	101492
60	9716.3	9048.54	700	102807	115877
70	11126.1	10551	800	118151	136748
80	12314.7	12607.3	900	133591	146731
90	14085.4	14257.9	1000	149937	162897
100	15971.4	15725.7			

表 5: element-wise pointer 与 element-wise bitmap 的性能对比

5.4 不同 SIMD 指令集（SSE、AVX、AVX-512）间对比

以 list-wise 算法为例本次 SIMD 实验中，我们在 IntelDevcloud 服务器上分别使用了三种 SIMD 指令集进行位图向量化操作，三种指令集分别使用 m128i、m256i 和 m512i 三种类型进行相量操作，收集到的数据如表6所示，通过数据我们发现虽然 AVX 和 AVX-512 的使用了更宽位数的向量进行操作，但实际的运行效率相较于 SSE 基本一致。基于这一实验现象，我们初步推测原因可能为以下两点：

- **g++ 编译器无法充分优化使用了不同 SIMD 指令集的代码**
- **程序的性能受到内存带宽与 cache 的限制**

Queries(n)	SSE	AVX	AVX512	Queries(n)	SSE	AVX	AVX512
10	81.474	61.438	79.318	200	4291.96	4140.37	4208.61
20	179.263	154.354	181.616	300	6125.37	5931.9	6044.53
30	324.456	287.938	328.283	400	8571.51	8317.58	8484.8
40	506.251	468.28	511.623	500	11739.6	11416.8	11676.9
50	754.489	706.838	750.541	600	15506.3	15152.1	15456.1
60	1077.47	1026.25	1072.82	700	19893.5	19462	19853.2
70	1503.65	1427.89	1477.21	800	24845.6	24473.7	24901.1
80	1967.51	1879.18	1927.51	900	30431.4	29993.8	30590.3
90	2488.65	2393.36	2443.66	1000	36495.2	36025.2	36839.3
100	3068.21	2962.62	3031.77				

表 6: SSE、AVX、AVX-512 指令集间性能对比

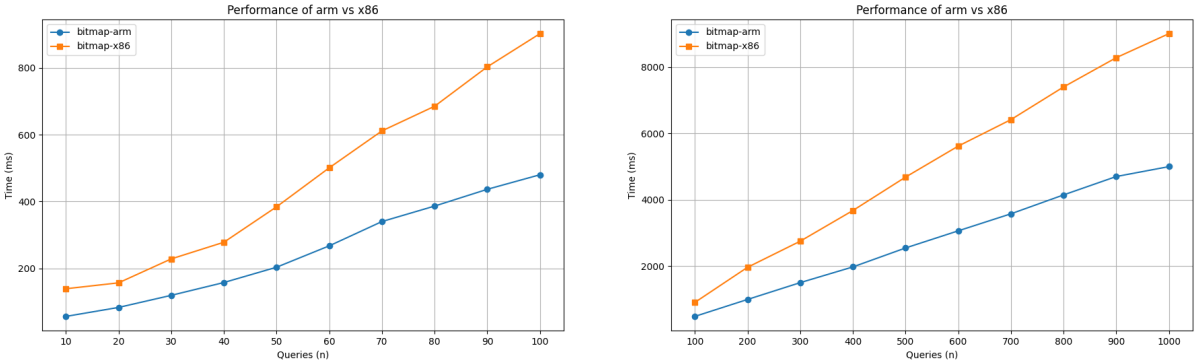
5.5 不同平台（ARM 和 x86）间对比

以 list-wise 算法为例

我们选取 bitmap 位图存储算法，分别在华为鲲鹏服务器和 IntelDevcloud 服务器上进行实验. 所得实验数据如表7所示

Queries(n)	bitmap-arm	bitmap-x86	Queries(n)	bitmap-arm	bitmap-x86
10	56.456	139.241	200	992.022	1962.13
20	83.311	156.893	300	1499.41	2749.44
30	119.521	228.533	400	1973.81	3674.94
40	157.887	278.418	500	2545.65	4683.69
50	203.682	384.451	600	3062.62	5618.93
60	267.643	501.179	700	3576.13	6417.43
70	340.298	611.586	800	4146.95	7403.97
80	386.661	685.376	900	4701.21	8287.06
90	436.826	803.284	1000	5000.21	9012.09
100	480.386	902.677			

表 7: arm 与 x86 平台间对比



(a) n=10:100

(b) n=100:1000

图 5.5: arm 平台与 x86 平台间对比

根据折线图，我们可以看出 arm 平台与 x86 平台的运行时间均随问题规模程线性增长，二者的比值基本趋于稳定。考虑到 cache 方面的影响，我们初步分析比值大小与 cache 容量之比存在比例关系；另一方面，由于位图算法使用到了按位与操作，因此程序效率还和 ARM 和 x86 架构的按位与指令的设计有关。

6 Perf-Profiling

基于串并行算法的对比分析，我们在华为鲲鹏服务器上使用 perf 对 list-wise-pointer、list-wise-bitmap 和 list-wise-neon 三种算法进行性能剖析，问题规模设定为 **[10,50,100,500,1000]**，测量 cache miss 率与 IPC，数据如下表所示：

通过表格可以发现，在 cache 表现上来看，随着查询数量的变化，基于 NEON 进行 SIMD 向量化的 list-wise NEON 的 cache miss 率始终为最低，表明 SIMD 向量化对于 cache 有比较好的优化效果。

查询数量	cache miss 率 (%)				
	10	50	100	500	1000
list-wise point	0.66	0.6	0.5	0.55	0.5
list-wise bitmap	0.58	0.54	0.56	0.57	0.56
list-wise NEON	0.44	0.41	0.43	0.44	0.43

查询数量	IPC			指令数/10^8			周期数/10^8		
	pointer	bitmap	NEON	pointer	bitmap	NEON	pointer	bitmap	NEON
10	2.47	2.23	2.13	123.958	4.452	3.193	50.151	1.993	1.51
50	2.5	2.25	2.14	906.044	19.225	14.333	362.328	8.539	6.705
100	2.49	2.34	2.16	1636.24	43.446	31.583	656.073	18.528	14.653
500	2.49	2.25	2.14	7437.24	222.657	160.5	2983.91	98.905	74.84
1000	2.5	2.25	2.14	15301.2	446,223	322.979	6129.12	198.726	150.768

表 8: list-wise pointer、bitmap 和 NEON 的部分性能指标对比

同时，从 IPC、指令数 (instructions) 和周期数 (cycles) 上来看，得出相同数量的查询结果，pointer 算法的三个指标均为最高，表明该算法的 CPU 负载程度最高；bitmap 和 NEON 算法相较于 pointer 算法，指令数和周期数有大幅度的减少，表明位图存储以及 SIMD 并行优化可以大大降低程序运行的复杂度，提高运行效率和程序性能。值得一提的是，NEON 算法在程序性能的提升上表现最优异，相比 pointer 算法而言，其指令数和周期数的下降程度大约为 50 倍，也能侧面印证上文所得到的加速比。

7 组内分工

本次基于 SIMD 的倒排索引求交编程实验由张高和张铭同学合作开展，具体分工如下：

- 张铭同学主要负责：基于 list-wise 思路实现串行算法优化和并行算法求解并在此基础上进行改进；基于 list-wise 思路实现 NEON/SSE/AVX 指令集架构下的 SIMD 并行编程，并在此基础上进行改进。
- 张高同学主要负责：处理 ExpQuery 以及 ExpIndex 相关数据集，从中读取实验所需的倒排列表，完成准备工作；基于 element-wise 思路实现串行算法优化和并行算法求解并在此基础上进行改进，使用 perf 进行程序性能剖析。
- 共同完成的任务：基于并行课程设计的 SIMD 编程实验内容和要求，指定完成计划和设计方案并且参与报告的撰写工作，其中包括：测定并汇总实验数据、绘制图表、分析得出结论并撰写相应报告。