Bitonic Sort

刘尧力

参考资料：Wikipedia

参考实现：github

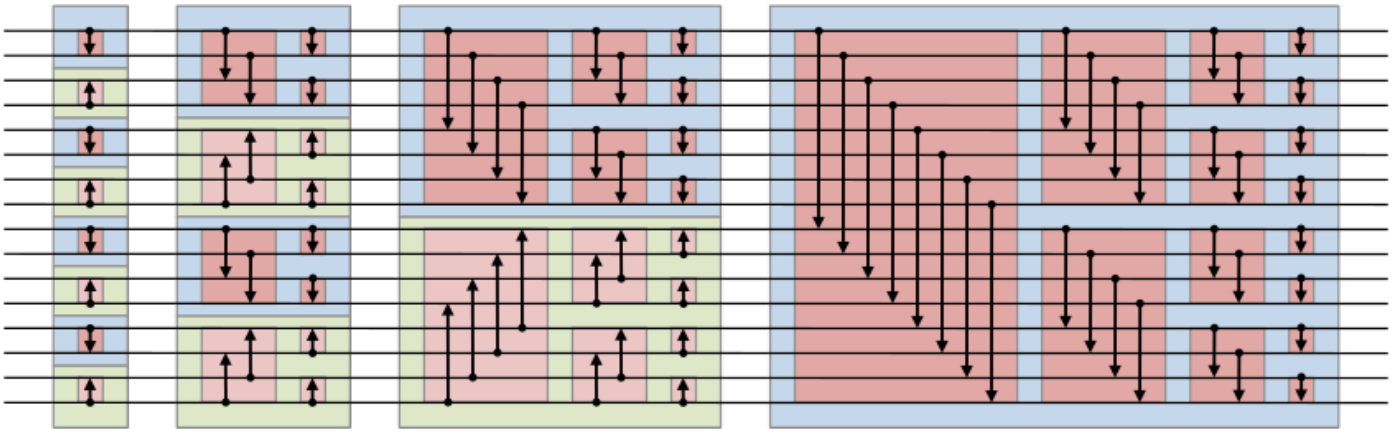
实现的加分项：不递归、不调用函数、内存高效、不需额外内存

算法描述

我们采用非递归的算法，算法首先找到对于输入N，最接近的一个2^n次幂nearest_pow_of_2

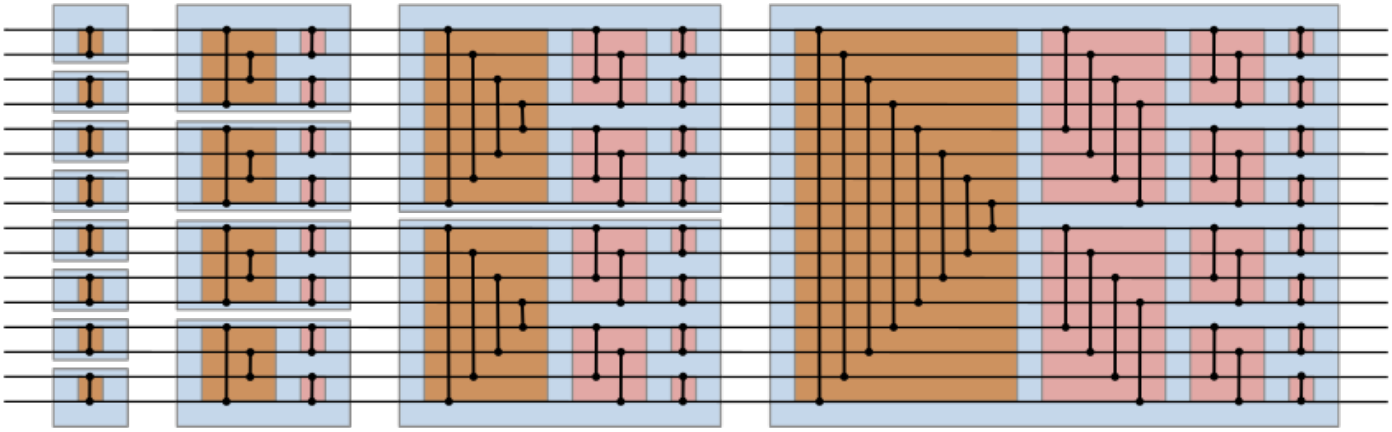
```
        int nearest_pow_of_2 = 1;
        for (; nearest_pow_of_2 < N; nearest_pow_of_2 <<= 1)
            ;
```

接下来我们实现wiki上的算法二，它由算法一衍生而来：



算法一最大的坏处在于绿色图示部分需要调转排序方向，这不利于我们实现非2^n长度的输入，而算法二可以通过超过len的就无视的小trick实现任意长度输入的处理。

> This is the most common representation of a bitonic sorting network. Unlike the previous interpretation, because the elements remain logically ordered, it's easy to extend this representation to a non-power-of-two case (where each compare-and-swap ignores any case where the larger index is out of range).



算法二在算法一的基础上提前做一次交叉划分，由于调转后的双调队列还是双调的，我们可以在调转的基础上，不必受到方向约束的进行排序

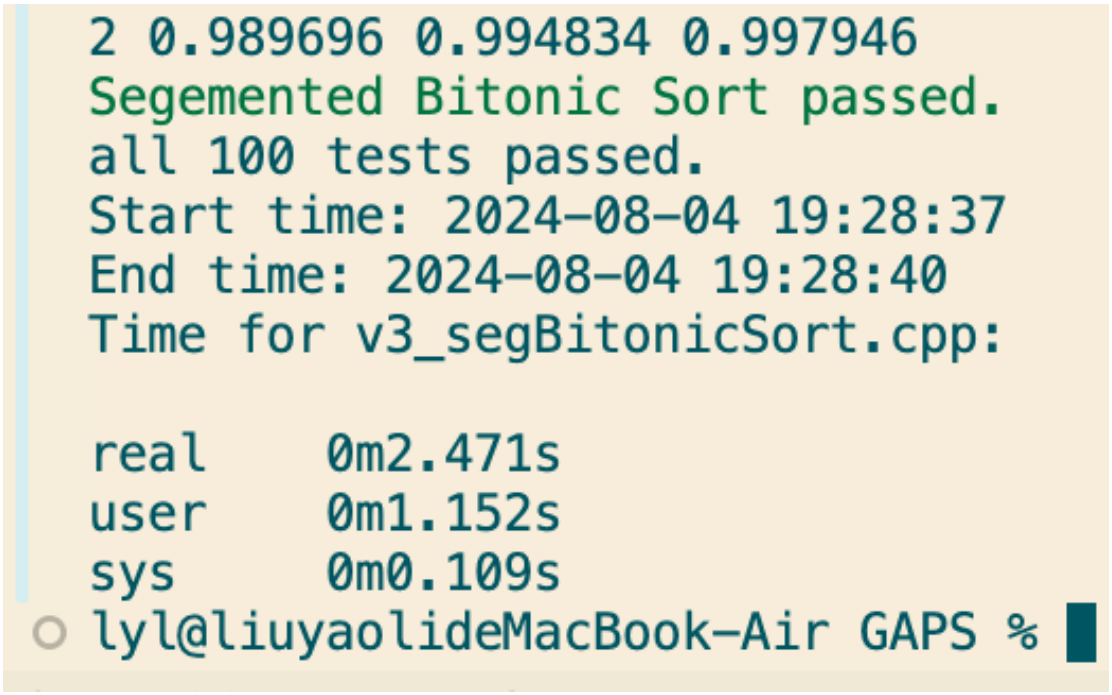
```
        for (int outer_step = 2; outer_step <= nearest_pow_of_2; outer_step<<=1)
        {
            for (int n = 0; n < nearest_pow_of_2; n += outer_step)
            {
                float *p = &mem[n];
                int limit = N - n;
              // 进行图中的调转操作
                for (int z = 0; z < outer_step >> 1; z++)
                {
                    int a_index = z;
                    int b_index = outer_step - z - 1;
                    if (a_index >= limit)
                        break;
                    float A = p[a_index];
                    float B = (b_index >= limit) ? INFINITY : p[b_index];
                    if (A > B)
                    {
                        p[a_index] = B;
                        if (b_index < limit)
                            p[b_index] = A;
                    }
                }
              // 不受方向约束的排序
                for(int inner_step = outer_step>>1; inner_step>1; inner_step>>=1)
                {
                    for (int m = 0; m < outer_step; m += inner_step)
                    {
                        for (int n = 0; n < inner_step >> 1; n++)
                        {
                            int a_index = m + n;
                            int b_index = m + (inner_step >> 1) + n;
                            if (a_index >= limit)
                                break;
                            float A = p[a_index];
                          // 不合法下标的b_idx可以直接视作inf，如果我们之前没有调转，inf将在从大到小的排序中参与排序，导致算法失效
                            float B = (b_index >= limit) ? INFINITY : p[b_index];
                            if (A > B)
                            {
                                p[a_index] = B;
                              // 只有当下标合法时才处理
                                if (b_index < limit)
                                    p[b_index] = A;
                            }
                        }
                    }
                }
            }
        }
```

测试

我们通过自动化测试程序进行测试，test.cpp将分别对三个版本的实现生成100个长度为10000，分段为20的浮点数序列，并分别调用我们实现的segmentBitconSort和内置的sort函数进行排序，最后检查结果是否一致，同时记录测试时间

test.sh是一键脚本

测试结果如下



性能分析

复杂度为$n \log n$

非递归的实现相对于递归版本节约了递归调用本身的时间，无函数调用版本进一步节约了调用时间

我注意到互联网上有直接通过不断记录当前组别中的mid，并围绕mid进行比较的实现，这类实现相对于我利用trick来忽略大于len的idx，会更加节省时间