ACM中国－国际并行计算挑战赛

ACM-China International Parallel Computing Challenge

目录 CONTENTS

参赛队简介

队名：七条边的凸多边形
学校：华中科技大学
成员：王绍宇，刘文卓
指导老师：石宣化

软件环境

编译器：Intel(R) oneAPI DPC++/C++ Compiler 2022

应用程序的代码结构

题目分析

给定高维空间中的n个点，要求从中选出k个作为pivot。并使用pivot，对所有点进行坐标重建。每个pivot的选取方案都对应着一个代价函数的值。试求代价函数最大/最小时的选取方案。

坐标重建：其中，每一个点重建后的坐标都是k维的，第i维的值是该点到第i个pivot的欧几里得距离。

代价函数：代价函数的定义是，重建后每两点间的切比雪夫距离之和。

事实上，题目不仅要求输出最大的方案，还要求输出代价函数前1000大与前1000小的选取方案，因此剪枝等策略是不可行的。只能采用枚举的策略。

应用程序的代码结构

算法流程

n: 点数， k: pivot 数量， d: 每个点的维度

1.枚举所有组合方案 $O(n^k)$

2.计算每个方案的代价 $O(ndk + n^2k + Mk)$

a) 重建坐标系 $O(ndk)$

b)Hotspot: 计算每两点之间切比雪夫距离之和 $O(n^2k)$

c) 更新top M $O(Mk)$

3.整理最终结果，按顺序存储top M方案

总复杂度： $O(n^k(ndk + n^2k + Mk))$

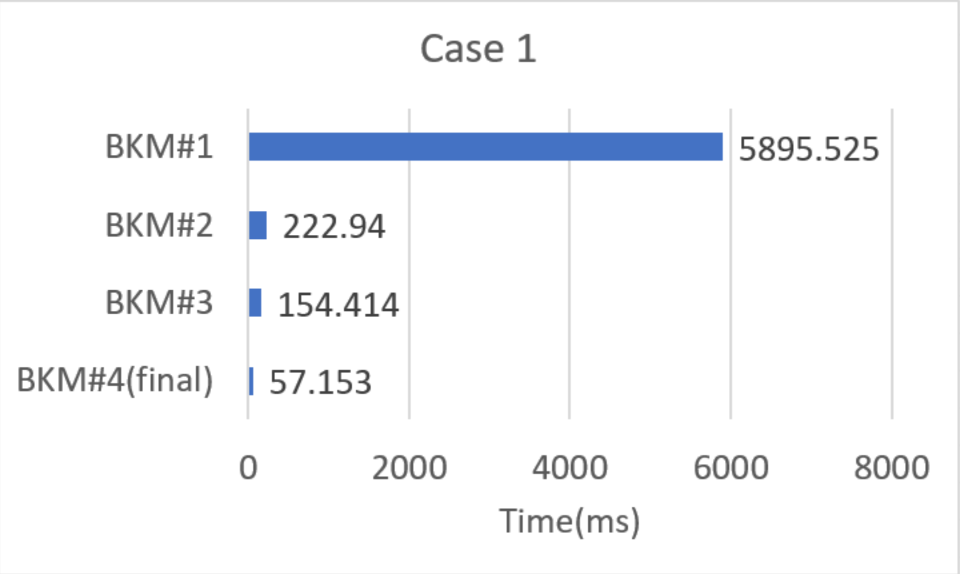
优化方法

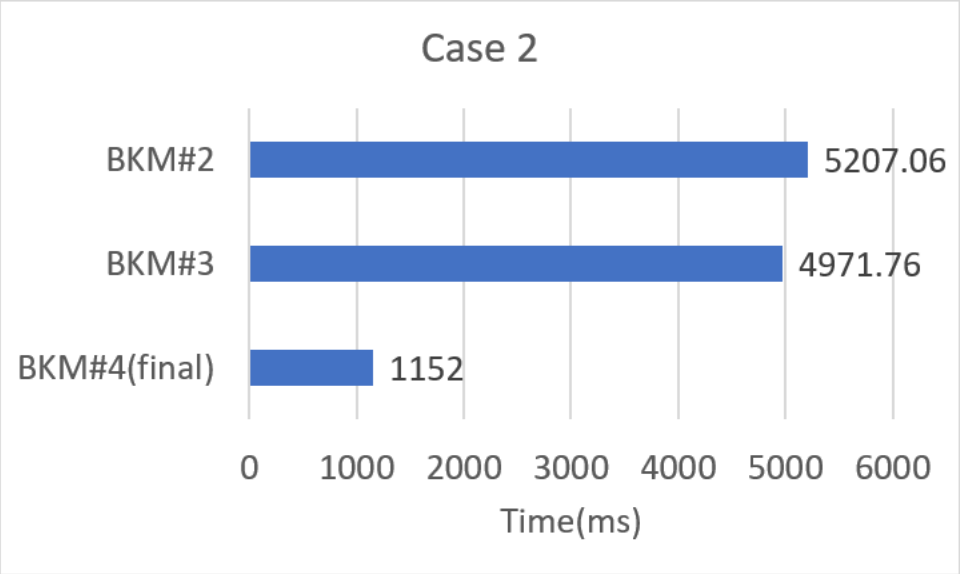
BKM 1: 算法优化
- BKM 1.1: 枚举策略优化
- BKM 1.2: 去除冗余计算
- BKM 1.3: 数据结构优化

BKM 2: 并行化枚举

BKM 3: 混合精度

BKM 4: SIMD优化





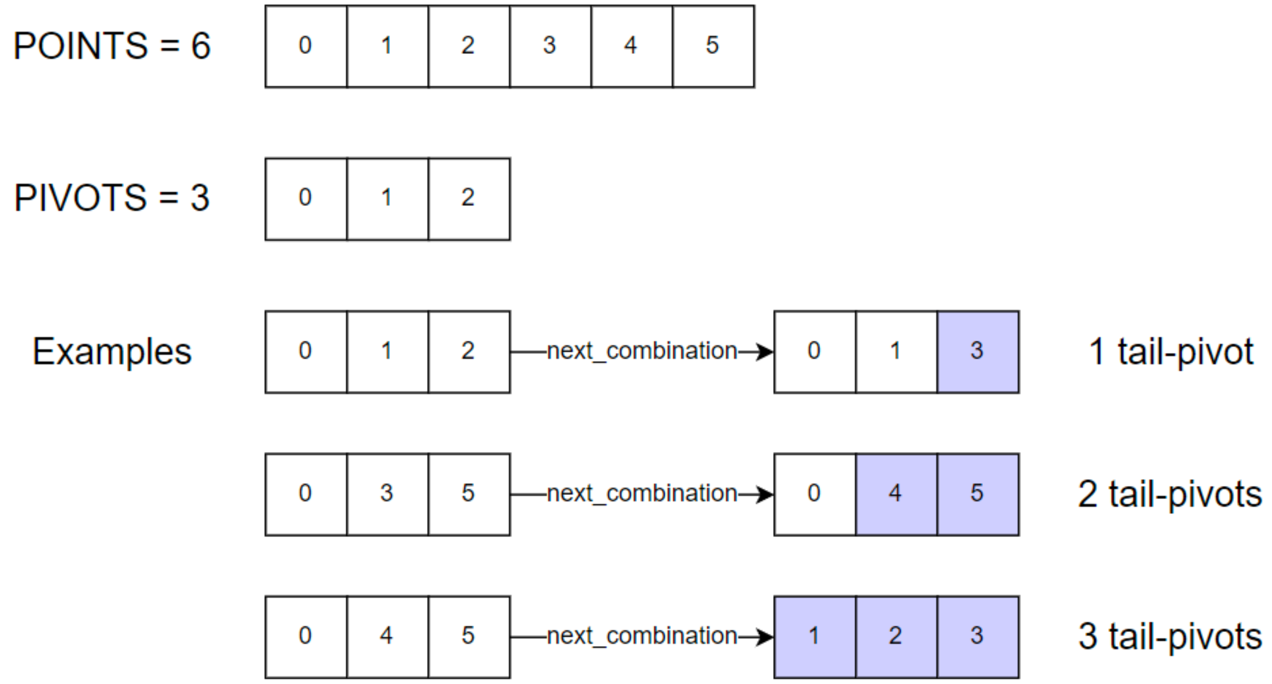
优化方法

BKM 1.1 枚举策略优化

40min → 5min

此场景下枚举的本质是求出所有的组合。考虑到递归枚举具有严重的前后依赖，不利于后续并行实现，我们采用了递推方法枚举组合。

在一个组合状态向下一个组合状态变化的过程中，只会改变数组尾部的若干个连续pivots，我们称之为tail-pivots。



而在重建坐标系与代价计算中，每个pivot的维度是彼此独立的，无需对未改变的pivot进行重复计算，只需计算发生改变的pivot所对应的维度即可。

优化方法

BKM 1.1 枚举策略优化

该策略对于重建坐标系部分可以起到优化效果。

对于重建坐标系，只需对每个点更新相对tail-pivots的坐标即可。这使得在平均意义下，重建坐标系的时间复杂度从$O(ndk)$下降到了$O(nd)$。

```
double *rebuild_coord() {
  // rebuild coordinate of each (point, pivot)
  int prev = next_comb();
  // prev: how many tail-pivots changed since last combination
  for i: 0 to POINTS
    for k: prev to PIVOTS
      re_coord[i][k] = distance(i, pivots[k])
}
```

优化方法

BKM 1.1 枚举策略优化

该策略对于求切比雪夫距离的过程可以起到优化效果。
由于当前组合方案相比上一个方案而言，更改的tail-pivots数量不确定，故需要利用max数组保存所有的前缀最大值。这使得在平均意义下，求切比雪夫距离的时间复杂度从$O(kn^2)$下降到了 $O(n^2)$。

```
// calc mx prefix
mx[PIVOTS][POINTS][POINTS]
for i: 0 to POINTS
  for j: 0 to POINTS
    for k: prev to PIVOTS
      mx[i][j][k] = max(mx[i][j][k-1], fabs(re_coord[i][k] - re_coord[j][k]))
// mx[PIVOTS-1] stores Chebyshev distance of each two points
```

优化方法

BKM 1.1 枚举策略优化

在维护top M方案的过程中，使用有序数据结构维护top M和对应的方案的存储指针，在最后按序进行归并整理到结果数组中并返回，防止反复冒泡交换产生的无意义访存开销 。

```
int pivots[PIVOTS]
int smalClest_cost_pivots[1000][PIVOTS]

cost = get_cost(pivots)
cur_max_cost, index = map.max()

if cost < cur_max_cost:
  map.remove(cur_max_cost)
  copy pivots to ans_pivots[index]
  map.insert(cost, index)
```

优化方法

BKM 1.2 消除冗余计算

重建坐标系需要用到点到点的欧几里得距离，可以预先在全局计算出来，需要用到时直接进行查表，无需每次重建坐标系时都计算欧几里得距离。

```
re_coord[i][k] = LUT_euclid_dist(i, pivots[k])
```

由于切比雪夫距离具有对称性，即chebyshev(a, b) == chebyshev(b, a)，故枚举点对(a, b)时只需要枚举 a > b 的点对，最后将sum加倍。这样的枚举类似枚举方阵的左下三角。

```
for i: 0 to PIVOTS
  for j: 0 to i
    sum += chebyshev_dist(i, j)
return sum * 2.0
```

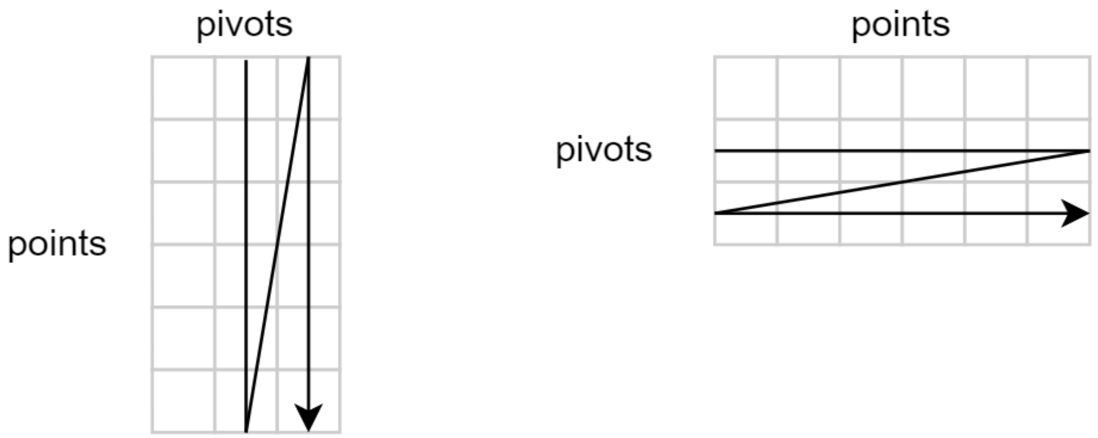
优化方法

BKM 1.3 数据结构优化

整个算法以pivot作为计算的依据，可以考虑以pivot为主序进行存储，提高re_coord, mx的访存局部性。

```
for k, i
  re_coord[k][i] = LUT_euclid_dist(i, pivots[k])
for k, i, j
  mx[k][i][j] =
    max(mx[k-1][i][j],
        fabs(re_coord[k][i] - re_coord[k][i]))
```

6 points, 3pivots



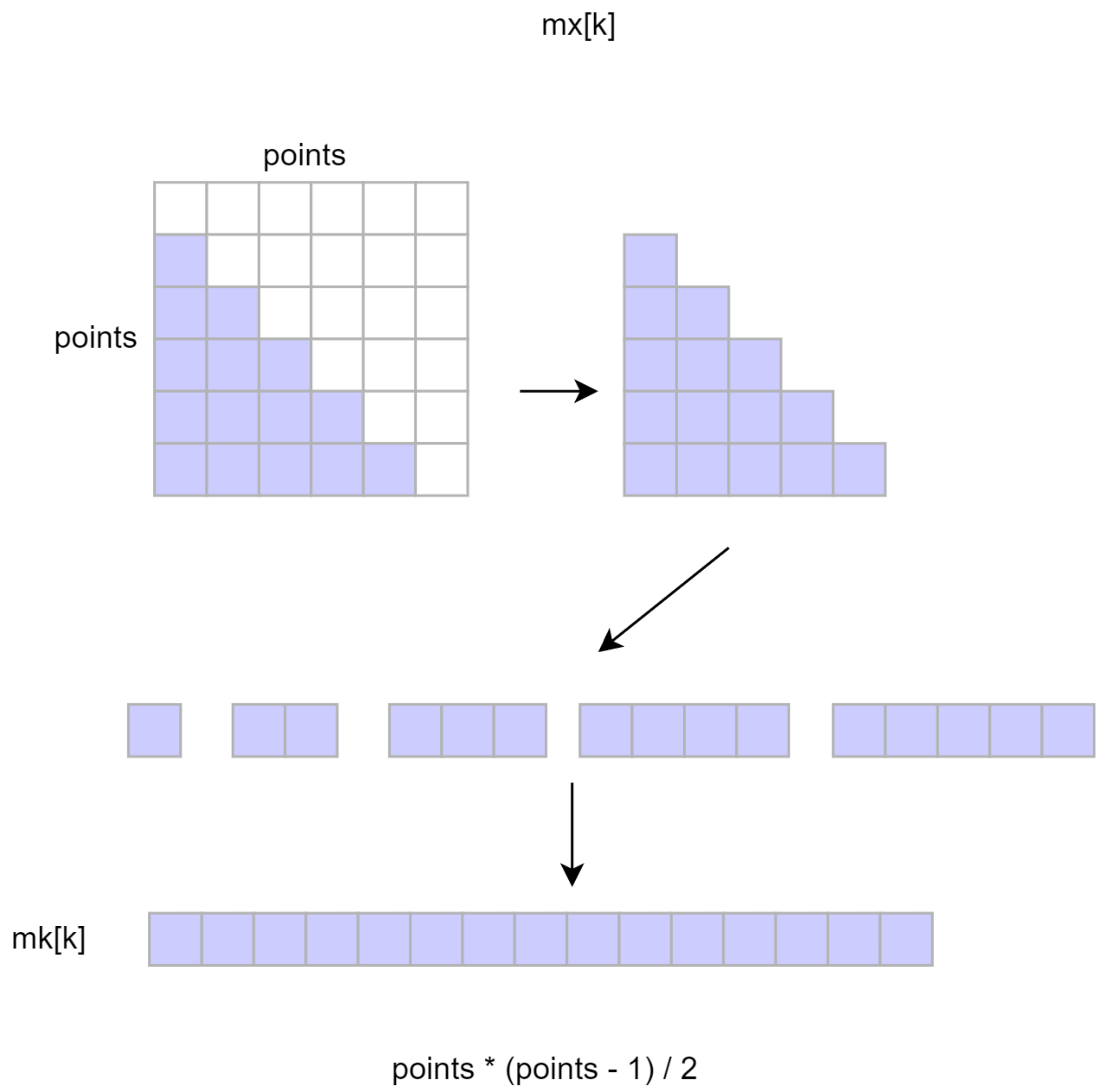
rebuilt_coord

优化方法

BKM 1.3 数据结构优化

在去除切比雪夫距离的冗余计算后，只需要进行类似方阵左下三角的求和，而右上三角完全未被使用。

可以考虑压缩右上三角，只存储左下三角。



优化方法

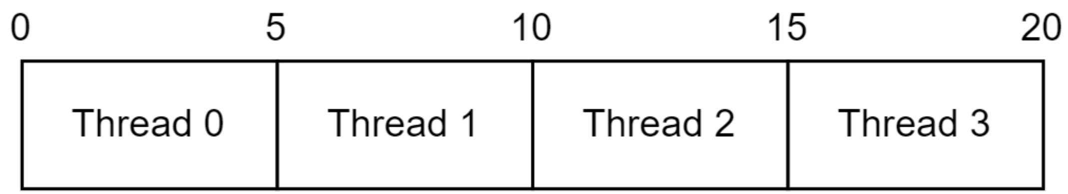
BKM 2 并行化枚举

5min → 5.2s

为了充分利用多核处理器，考虑将枚举任务分发到不同的线程进行计算

- 每个线程枚举不同的组合，实现不重不漏
  - 假设共有m个组合，t个线程，那么每个线程的工作量应当是 m / t
  - 因此，第i个线程应当从排名是 i * (m / t) 的组合开始，连续枚举 m / t 个组合
- 所有线程枚举完成后，需要合并所有线程维护的 local top n，得到 gloabl top n
  - 使用单线程的多路归并算法。
- 该并行策略非常简单，使用 MPI 简单修改后便可拓展到多机运行
  - 由于后续优化中，发现MPI初始化代价过大，因此放弃使用MPI.

N=6, Pivots=3, Threads=4



优化方法

BKM 3 混合精度

5.2s →4.9s

- 考虑通过把double型改为float型来提高吞吐量。
- 但是float精度过低，无法通过正确性检查。
- 因此采用混合精度的思路：对于精度要求较高的累加变量，使用double型，其他部分使用float型。取得速度和精度之间的平衡。

优化方法

BKM 4 SIMD 优化

4.9s → 1.1s

profile显示workload类型是compute bound。因此考虑通过使用SIMD指令集来提高吞吐量

正常部分：直接使用SIMD进行处理

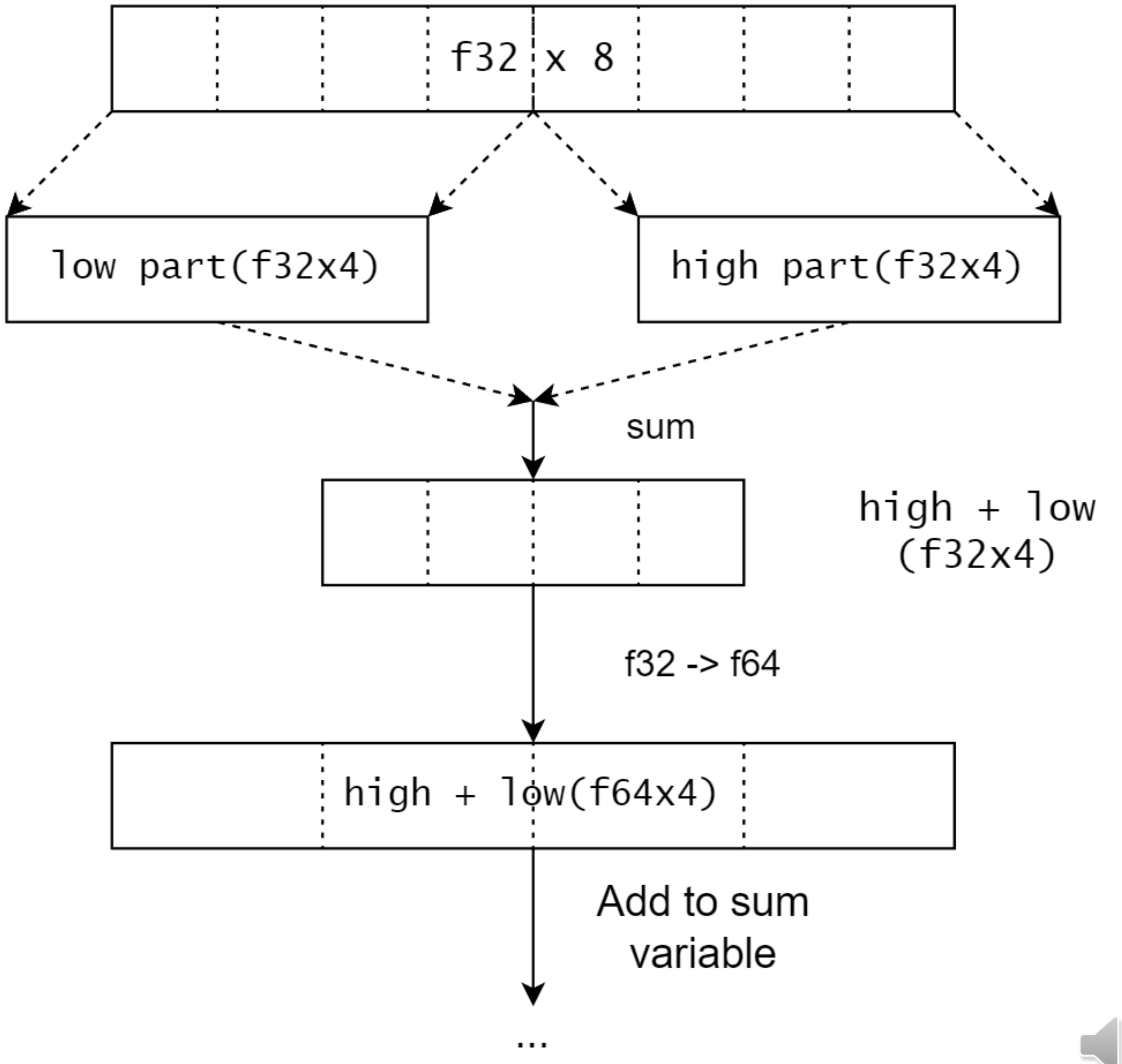
```
for (j = 0; j ≤ i - 8; j += 8) {
    __m256 current_f32×8 = abs_ps(_mm256_sub_ps(re_coord_k_i_f32×8, _mm256_loadu_ps(&
    rebuilt_coord[last * npoints + j])));
    __m256 mx_k_1_j_f32×8 = _mm256_loadu_ps(&mx[(last - 1) * points_pairs + idx_cnt + j]);
    __m256 max_value_f32×8 = _mm256_max_ps(current_f32×8, mx_k_1_j_f32×8);
    __m128 high_part = _mm256_extractf128_ps(max_value_f32×8, 1);
    __m128 low_part = _mm256_extractf128_ps(max_value_f32×8, 0);
    __m128 high_p_low = _mm_add_ps(high_part, low_part);
    sum_buffer_f64×4 = _mm256_add_pd(sum_buffer_f64×4, _mm256_cvtps_pd(high_p_low));
}
```

优化方法

BKM 4 SIMD 优化

正常部分：直接使用SIMD进行处理

完成计算后，将f32x8的寄存器拆分为两部分，进行求和后，再转换为double，累加到求和变量上

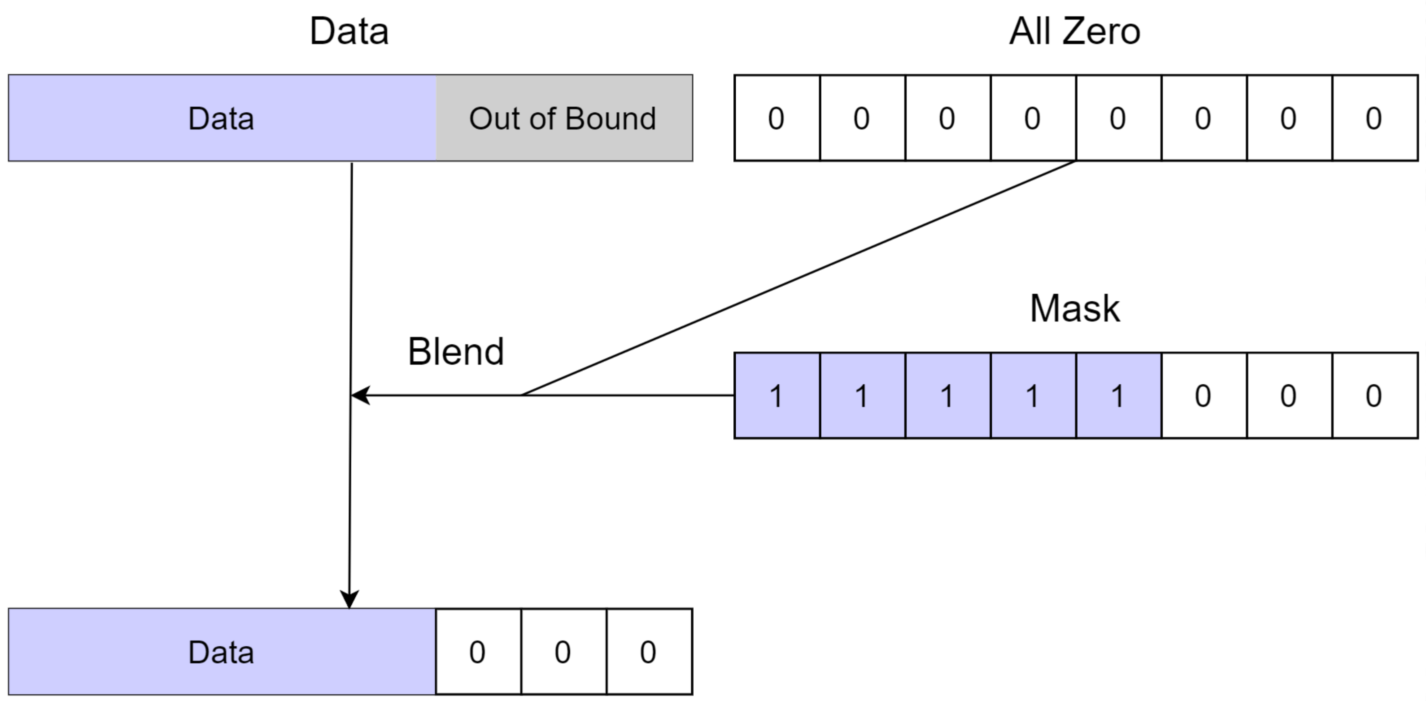


优化方法

BKM 4 SIMD 优化

边界部分：我们使用了 SIMD 的 mask 和 blend 指令。将超出边界的值抛弃，不纳入计算。

通过blend指令，将寄存器中超出循环边界的部分置零，避免影响求和。



优化方法

Tricks

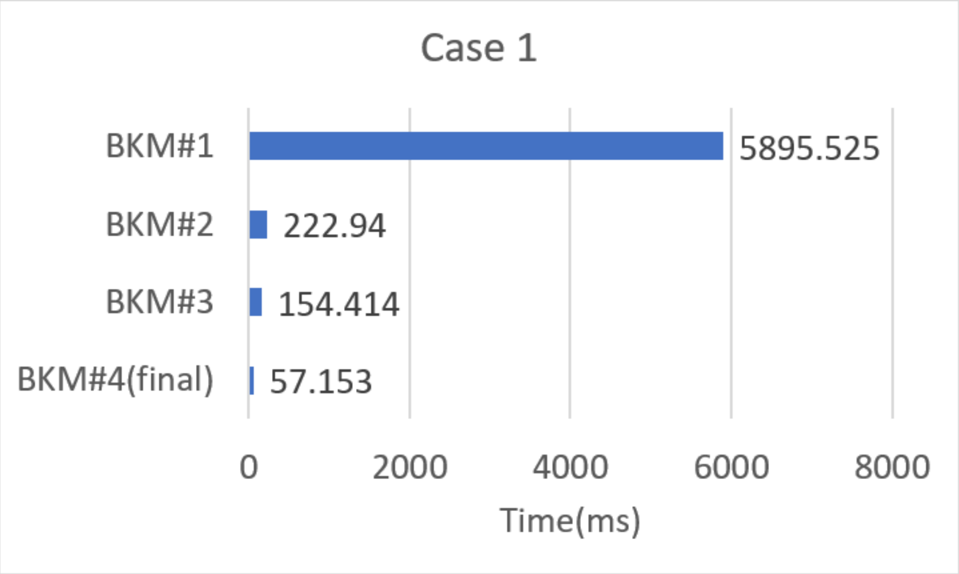
- 绑核：将线程绑定到固定的核心上，避免调度器切换造成性能损失
- 负载均衡：将任务的进一步切分，粒度更细，使各个线程的负载更加均衡
- 全局常量：将用到的常量改为 const 型全局变量，可以从编译器的优化中受益
- 合并循环：对最后一个pivot求mx数组时，一并统计求和，使得最后一个mx数组无须存入内存

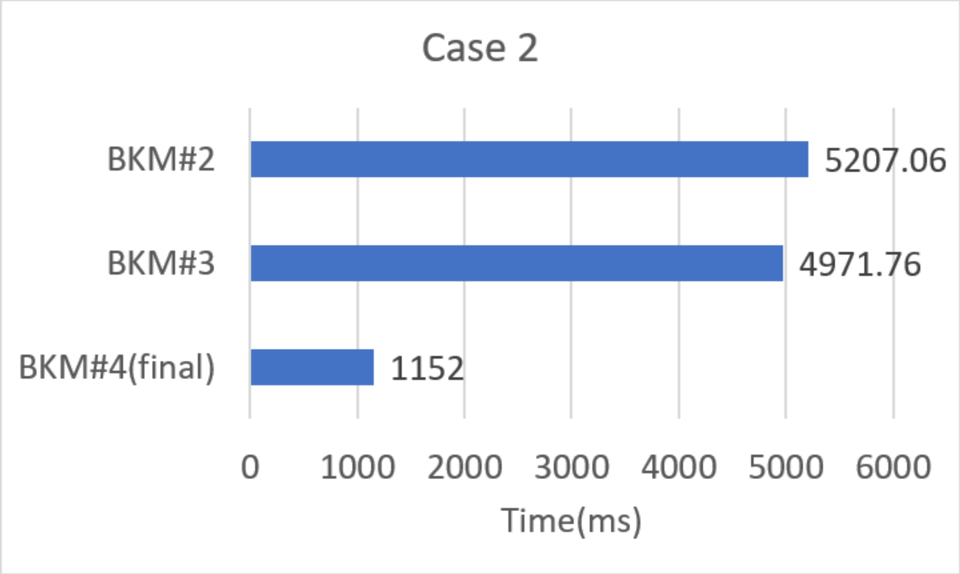
程序运行结果

最终版本的代码在两个数据上的运行时间分别是 57.163 ms 和 1152.774 ms

使用icx编译器在O3优化下测得baseline运行时间分别为为39s和40min。因此最终的加速比分别为687倍和2083倍。

我们的算法优化使得时间复杂度减少了一个 k，且大数据中的k=5，小数据中的k=2。因此我们的算法在大数据中加速比更高。





感谢指导

THANKS FOR GUIDANCE