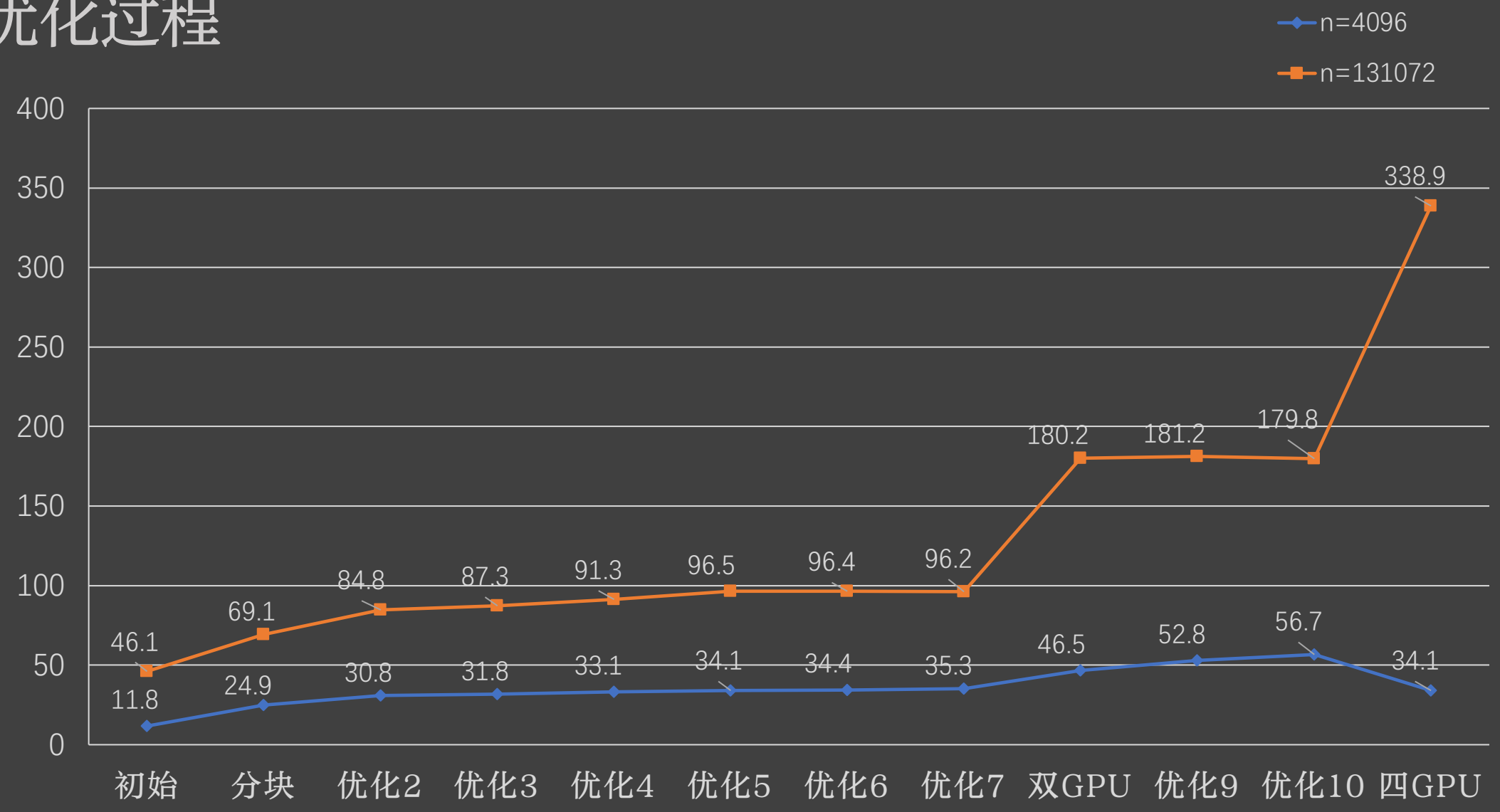
CUDA N-body优化

By 刘畅

优化过程


n=4096
n=131072
400
350
300
250
200
150
100
50
0
46.1
69.1
84.8
87.3
91.3
96.5
96.4
96.2
180.2
181.2
179.8
338.9
11.8
24.9
30.8
31.8
33.1
34.1
34.4
35.3
46.5
52.8
56.7
34.1
初始
分块
优化2
优化3
优化4
优化5
优化6
优化7
双GPU
优化9
优化10
四GPU

利用Shared Memory分块计算

» 分块大小等于block大小

```
__global__ void
bodyUpdateVelocity(float3 *pos, float3 *vel, float dt, int n) {
        int bodyId = blockIdx.x * blockDim.x + threadIdx.x;
        if (bodyId >= n) return;
        extern __shared__ float3 posShared[];

        float3 acc = { 0.0f, 0.0f, 0.0f };
        float3 p = pos[bodyId];

        for (int i = 0; i < gridDim.x; i++) {
                int id = i * blockDim.x + threadIdx.x;
                __syncthreads();
                posShared[threadIdx.x] = pos[id];
                __syncthreads();
                bodyGatherAccelTiled(p, acc);
        }
        vel[bodyId] += acc * dt;
}
```

1

12→25 $(n = 2^{12})$

46→69 $(n = 2^{17})$

2

25→31

69→85

循环展开

- 大概还是由于降低了循环中分支的频率
- #pragma unroll自动预处理

```
__forceinline__ __device__
void bodyGatherAccelTiled(float3 pos, float3 &acc) {
        extern __shared__ float3 posShared[];
#pragma unroll 32
        for (int i = 0; i < blockDim.x; i += 2) {
                bodyGetAccelPairwise(pos, posShared[i + 1], acc);
                bodyGetAccelPairwise(pos, posShared[i], acc);
        }
}
```

- 再尝试手动展开一层，便于调整shared mem访问顺序（微微作用）

3

34.1→34.4

96.5→96.4

CUDA PTX内联汇编优化

» 乘加指令：$D = A \times B + C$

比较明显的模式编译器能够自动识别，不过有例外：

```
distSqr = dx * dx + dy * dy + dz * dz + SOFTENING
```

```
__forceinline__ __device__
float dot(const float3 &a, const float3 &b) {
        const float SOFTENING = 1e-9f;
        float res = SOFTENING;
        asm("fma.rn.f32 %0, %1, %2, %3;" :
                "=f"(res) : "f"(a.x), "f"(b.x), "f"(res));
        asm("fma.rn.f32 %0, %1, %2, %3;" :
                "=f"(res) : "f"(a.y), "f"(b.y), "f"(res));
        asm("fma.rn.f32 %0, %1, %2, %3;" :
                "=f"(res) : "f"(a.z), "f"(b.z), "f"(res));
        return res;
}
```

» 效果极其有限

4

35→46

96→180

双GPU，双倍快乐

» 两张Tesla K80可用，然而

```
+-----------------------------------------+------------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|=========================================+========================+======================|
|   0  Tesla K80           Off  | 00000000:09:00.0 Off |                  Off |
| N/A   38C    P8    27W / 149W |     11MiB / 12206MiB |      0%      Default |
+-----------------------------------------+------------------------+----------------------+
|   1  Tesla K80           Off  | 00000000:0A:00.0 Off |                  Off |
| N/A   32C    P8    28W / 149W |     11MiB / 12206MiB |      0%      Default |
+-----------------------------------------+------------------------+----------------------+
|   2  Tesla K80           Off  | 00000000:86:00.0 Off |                  Off |
| N/A   59C    P0    57W / 149W |    582MiB / 12206MiB |     30%      Default |
+-----------------------------------------+------------------------+----------------------+
|   3  Tesla K80           Off  | 00000000:87:00.0 Off |                  Off |
| N/A   43C    P0    70W / 149W |    321MiB / 12206MiB |      0%      Default |
+-----------------------------------------+------------------------+----------------------+
```

» 原来一张卡上有两个

NVIDIA Tesla K80

GK210 x2	2496 x2	208 x2	48 x2	12 GB x2	GDDR5	384 bit x2
GRAPHICS PROCESSOR	CORES	TMUS	ROPS	MEMORY SIZE	MEMORY TYPE	BUS WIDTH

» 先试试二路

4

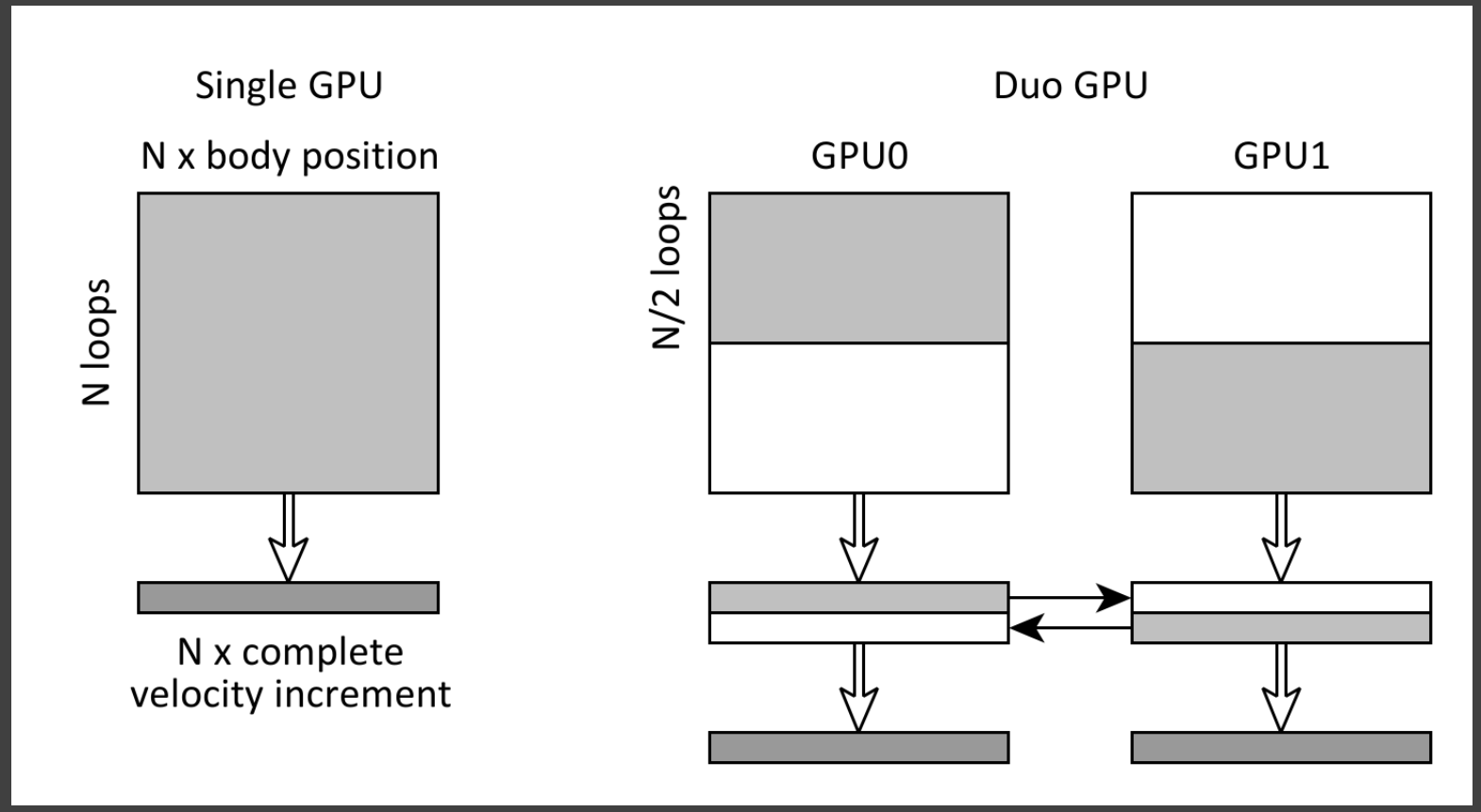
35→46

96→180

双GPU，双倍快乐

» 如何划分n^2对加速度的计算?

» GPU数据怎样共享/传输/同步?



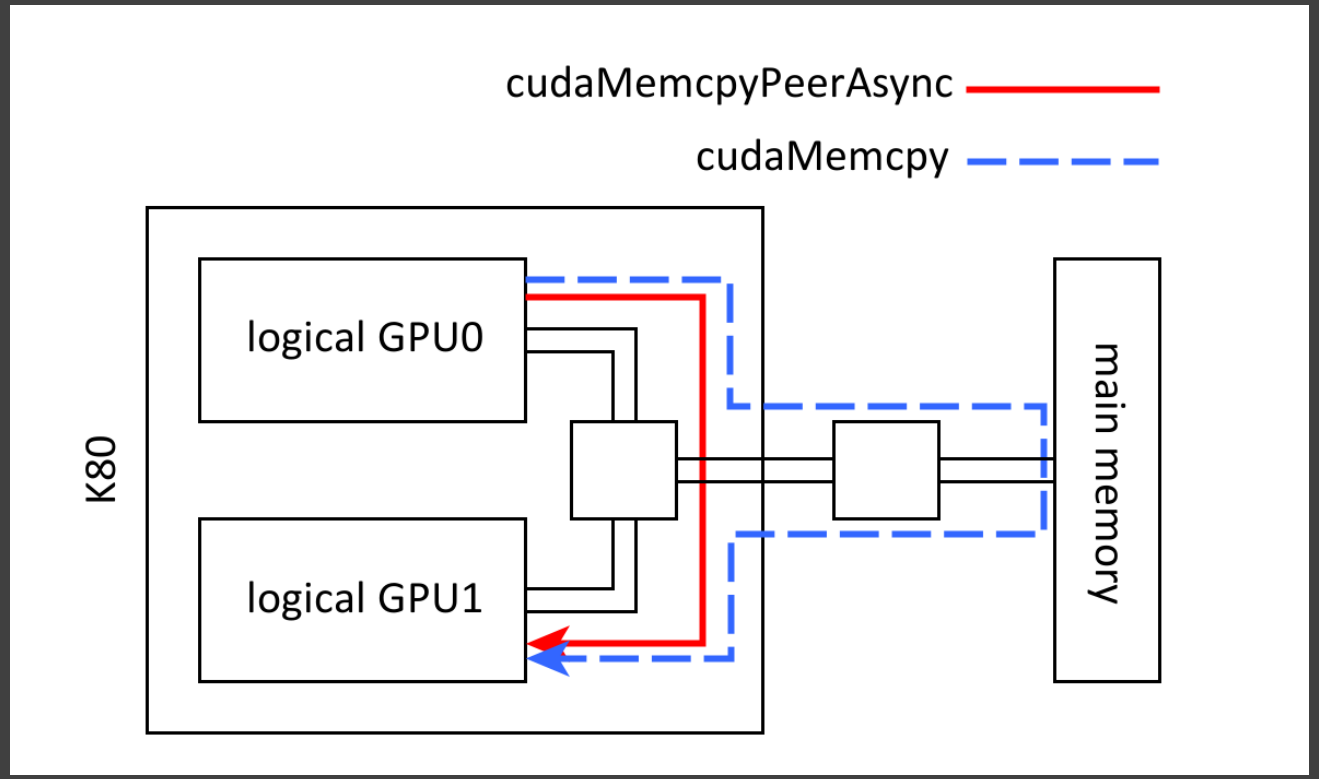
优化数据传输

5

46→53

180→181

» 直接走内部总线



» 需要peer access支持

继续利用peer access

6

53→57

181→180

» cudaDeviceEnablePeerAccess

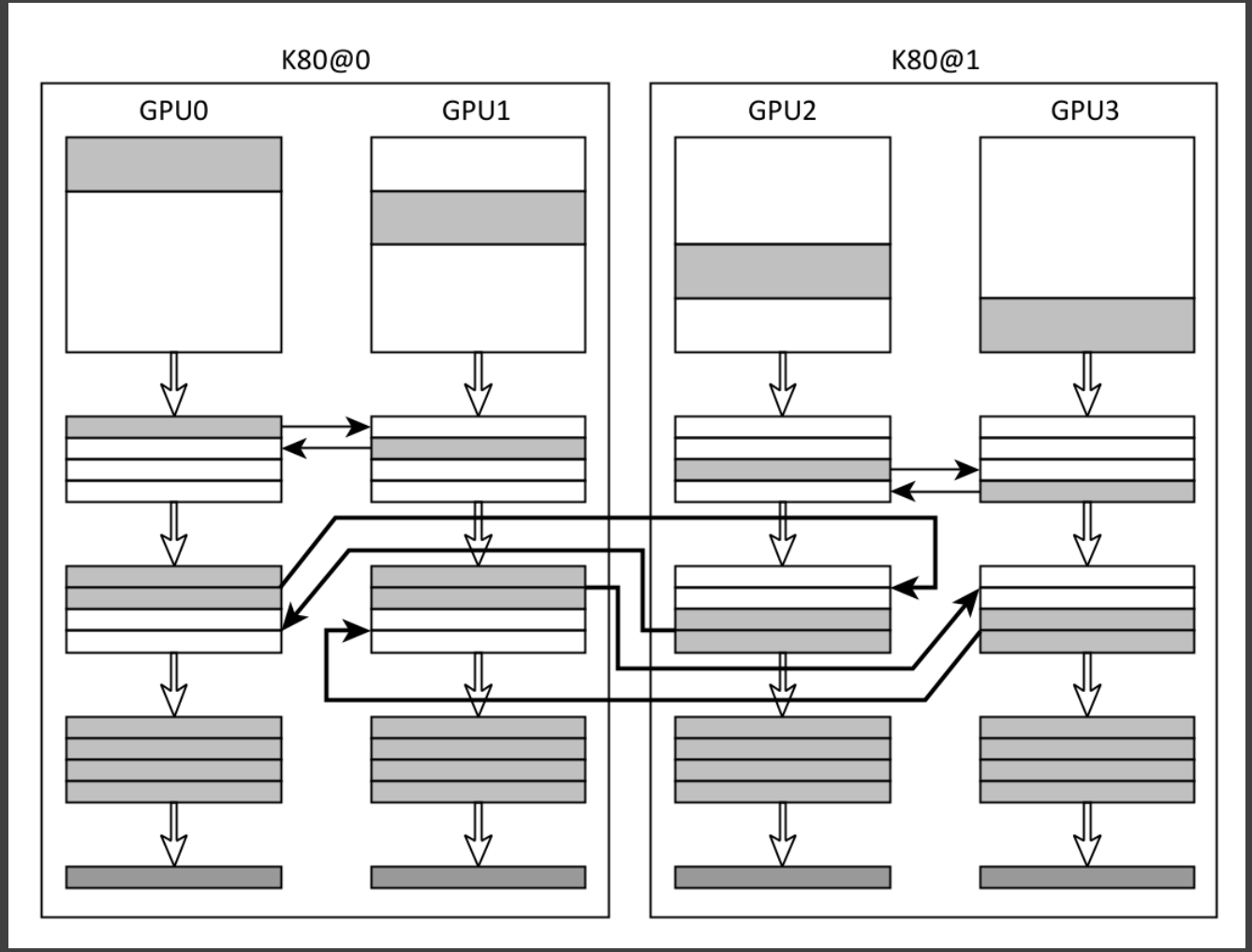
- 这样两个GPU可以直接访问对方的内存地址空间
- 直接累加对方的速度增量到自己算出来的部分
- 不稳定

双双GPU，四倍快乐

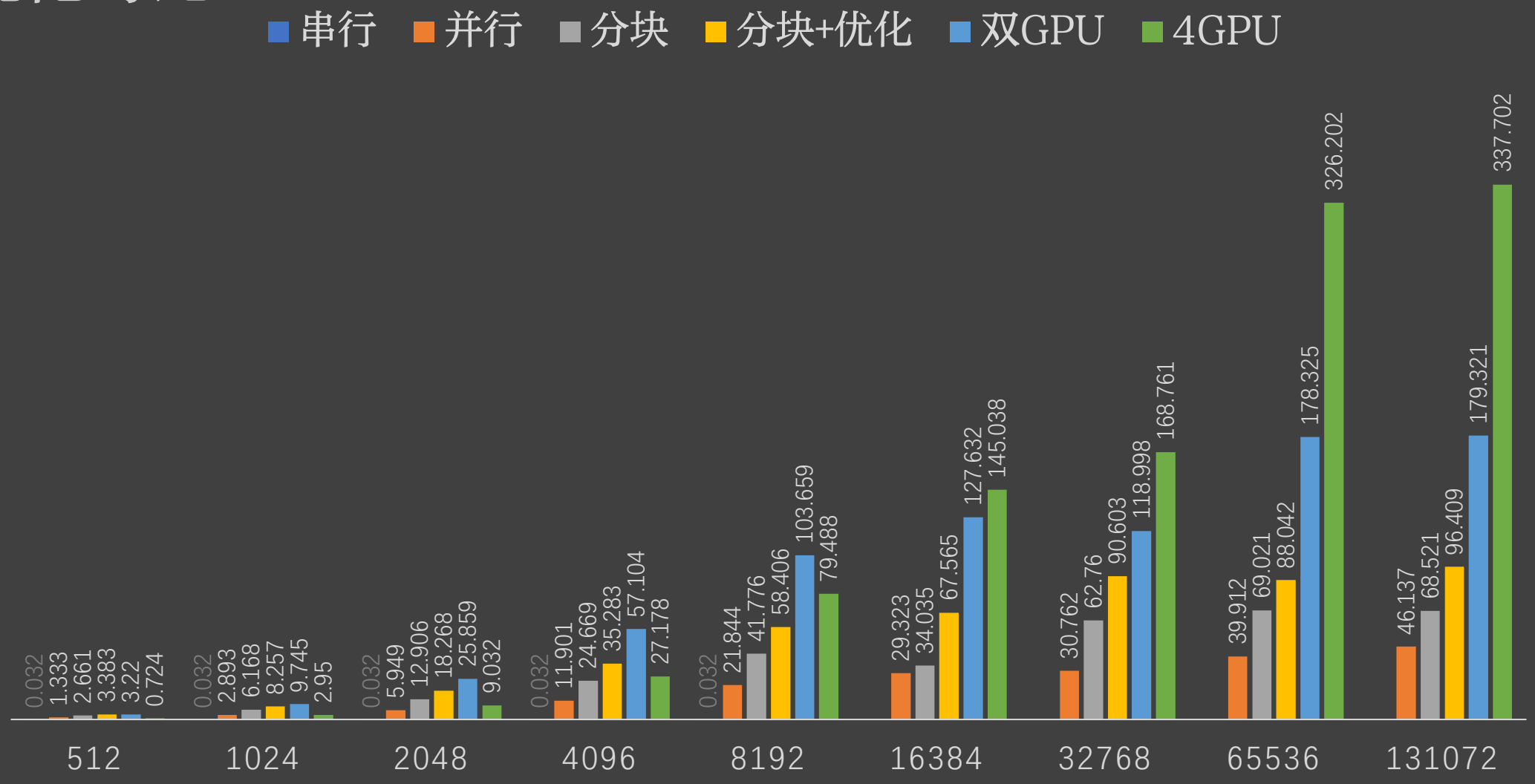
7

57→34

180→339



优化对比


串行
并行
分块
分块+优化
双GPU
4GPU
0.032
1.333
2.661
3.383
3.22
0.724
0.032
2.893
6.168
8.257
9.745
2.95
0.032
5.949
12.906
18.268
25.859
9.032
0.032
11.901
24.669
35.283
57.104
27.178
0.032
21.844
41.776
58.406
103.659
79.488
29.323
34.035
67.565
127.632
145.038
30.762
62.76
90.603
118.998
168.761
39.912
69.021
88.042
178.325
326.202
46.137
68.521
96.409
179.321
337.702
512
1024
2048
4096
8192
16384
32768
65536
131072

其他方法

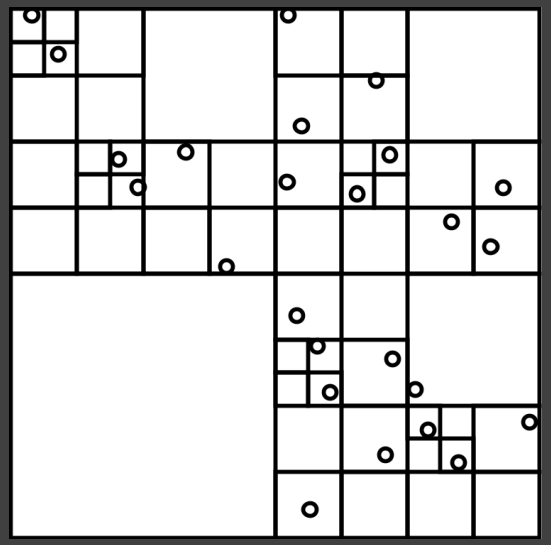
2.基于空间划分和远距离近似的快速算法（硬核）

» 基于以下近似假设

当多个body聚集在很小一团时，其对很远之外的一个body的作用可近似为单个整体

» 因此，可以将body按空间划分

对于离得远的那些body，就不必逐一计算



» 代表性算法

- Barnes-Hut $O(n\log n)$
- Fast Multipole Method $O(n)$

» 每次计算前都要并行构建八叉树 ←KD树和BVH应该也可以
或者动态调整结构

» 结点需要保存质量、质心、包围盒（球）信息

» 当结点内的多个body的包围盒相对被计算body的立体角小于某个阈值，可近似为单个质点

1.多thread对一body

» 小规模数据较快

» 大规模数据吞吐率提升有限

可能是由于大量原子操作导致的串行化