# 优化方法一

12700 8核16线程，80gb内存 文件放在share memory上 写入到nvme固态

## 优化一：多进程并行计算（MPI）

相关库：`mpi4py`

优化说明：

初始版本完全串行处理所有图像帧且不区分任务分工；最终版本引入 MPI，将图像帧的处理任务在多个进程中并行执行，并通过 rank 和 size 明确分配数据处理范围，最后聚合处理结果。

初始版本代码片段1：

```
for picture in range(self.shape[0]):
    new[picture,:, :, :] = self.get_new_img(
        self.img_data_gauss[picture,:,:,:],
        self.parameter[picture],
        picture
    )
```

最终代码版本1

```
unit = self.shape[0] // size
remain = self.shape[0] % size
start = rank * unit + remain if rank != 0 else 0
end = (rank + 1) * unit + remain

seperate_img = np.empty((end - start, ...) )
for picture in range(start, end):
    seperate_img[picture-start, :, :, :] = self.get_new_img(...)

comm.Gather(seperate_img, gathered_img, root=0)
```

初始版本代码片段2(iterate 函数中的三维点分布)：

```
bi = np.zeros(self.x*self.y*self.z)
b_diff = np.zeros((self.x*self.y*self.z, 7))
for i in range(self.x):
    for j in range(self.y):
        for k in range(self.z):
            ...
```

### 最终代码版本2

```
total = self.x * self.y * self.z
unit = total // size
remain = total % size
start = rank * unit + remain if rank != 0 else 0
end = (rank + 1) * unit + remain

bi = np.zeros(end - start)
b_diff = np.zeros((end - start, 7))

for index in range(start, end):
    i = index // (self.y * self.z)
    j = (index // self.z) % self.y
    k = index % self.z
    ...
```

### 加速原理

使用 mpi4py 将计算分发到多个进程（可分布于多核心或多节点），有效利用 CPU 多核资源。

明确分工避免不同进程重复处理数据。

每个进程只创建和使用自身所需内存，降低整体内存占用。

主进程统一收集结果，便于后续统一处理和保存

### 加速结果

1200s -> 200s

## 优化二：矩阵逆计算复用

### 优化说明:

刚体变换矩阵 M 每次在像素级调用 np.linalg.inv() 会极大拖慢计算。最终版本在外部计算一次 invM 后传入内部使用。

### 初始版本代码片段

```
for i in range(...):
    M = self.rigid(q)
    interpo_pos = np.linalg.inv(M) @ [i, j, k, 1]
```

### 最终版本

```

```

```
M = self.rigid(q)
invM = np.linalg.inv(M) if np.linalg.det(M) > 1e-10 else np.linalg.pinv(M)
...
for i in prange(...):
    pos = invM @ np.array([i, j, k, 1])
```

### 加速原理

矩阵逆是一项昂贵操作，将其提至循环外只算一次，大幅减少冗余计算。

### 加速结果

200s -> 40s

## 优化三：JIT 加速与并行循环（Numba）

### 相关库：`numba`

### 优化说明：

三重 for-loop 是图像处理中的瓶颈，初始版本直接使用原生 Python 实现，执行效率极低。最终版本使用 @njit(parallel=True) 修饰函数，编译为高性能本地代码，并开启线程级并行。

### 初始版本代码片段：

```
def get_new_img(...):
    for i in range(self.shape[1]):
        for j in range(self.shape[2]):
            for k in range(self.shape[3]):
                interpo_pos = ...
                new_img[i, j, k] = ...
```

### 最终版本

```
 @njit(parallel=True)
def fast_get_new_img(resource, invM, shape1, shape2, shape3):
    new_img = np.empty((shape1, shape2, shape3))
    for i in prange(shape1):
        for j in range(shape2):
            for k in range(shape3):
                pos = invM @ np.array([i, j, k, 1.0])
                ...
```

### 加速原理

@njit: 使用 Numba 的 JIT 编译器将 Python 函数编译为本地代码。 prange: 开启多线程并行循环，加快大规模像素点处理速度。

### 加速结果

40s -> 30s

## 优化四 MPI 进程绑定策略（核心绑定）

### 涉及指令：mpirun --bind-to hwthread --map-by core

### 优化说明：

在多核 CPU 系统中，MPI 进程如果调度不当，可能在不同 CPU 核之间迁移，导致缓存频繁失效，影响性能。使用 --bind-to hwthread 或 --map-by core 等参数可以将每个 MPI 进程固定到指定的逻辑核心或物理核心上，提升数据局部性和执行效率。

### 加速结果

30s -> 20s