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：

```
def reslicing(self,name):
        '''
        用刚体变换参数将图片进行重采样对齐\n
        Resampling and aligning the image with the rigid-body transform parameter\n
        output: 对齐后的图片(Aligned image)
        '''
        self.interpolator = itk.BSplineInterpolateImageFunction.New(self.img_gauss)  # B样条插值器(B-spline interpolation)

        print("开始重采样对齐(注意，需要很长时间，建议先去干点别的事)\nreslicing")
        new = np.ndarray(self.shape)
        for picture in range(self.shape[0]):
            new[picture,:, :, :] = self.get_new_img(self.img_data_gauss[picture,:, :, :], self.parameter[picture],picture)
            print(f"进度{picture*100/self.shape[0]}%", end="\r")
        self.save_img(new, name)
        return new
```

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, ) + tuple(self.shape[1:]))
for picture in range(start, end):
    seperate_img[picture - start, :, :, :] = self.get_new_img(
        self.img_data_gauss[picture,:,:,:],
```

```
            self.parameter[picture],
            picture
        )

if rank == 0:
    gathered_img = np.empty((unit * size, ) + tuple(self.shape[1:]))
    comm.Gather(seperate_img[remain:], gathered_img, root=0)
    self.save_img(gathered_img, name)
else:
    comm.Gather(seperate_img, None, root=0)
python
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):

```
def iterate(self, reference,resource, q,pic_num):
        '''
        高斯牛顿迭代(gauss-newton iterate)\n
        输入参数(parameter):\n
        resource:原始图像(raw image)\n
        reference:参考图像(reference image)\n
        q:旋转平移参数(rotation and translation parameters)\n
        输出(output):\n
        q:更新后的旋转平移参数(new rotation and translation parameters)\n
        bi:残差(residual)\n
        b_diff:偏导矩阵(derivative matrix)
        '''

        interpolator = self.interpolator  # B样条插值(B-spline
interpolation)
        step = self.step  # 选点间隔(point interval)
        bi = np.zeros(self.x*self.y*self.z)    # 残差 (residual)
        # 偏导矩阵 (derivative matrix)
        b_diff = np.zeros(((self.x*self.y*self.z), 7))
        index = 0
        for i in range(self.x):
            for j in range(self.y):
                for k in range(self.z):
                    n_i = i*step  # 对应位置的转换,获得点在原图的位置(get the
position of the point in the original image)
                    n_j = j*step
                    n_k = k*step
```

```
                    M = self.rigid(q)
                    M[3,3]=0.0
                    if np.linalg.det(M)==0:
                        interpo_pos = np.linalg.pinv(M)@[n_i, n_j, n_k, 1]
                    else:
                        interpo_pos =np.linalg.inv(M)@[n_i, n_j, n_k, 1]
                    point = itk.Point[itk.D, 4]()
                    if 0 <= interpo_pos[0] < self.shape[1] and 0 <=
interpo_pos[1] < self.shape[2] and 0 <= interpo_pos[2] < self.shape[3]:  #
判断是否在范围内
                        point[0] = pic_num
                        point[1] = interpo_pos[0]
                        point[2] = interpo_pos[1]
                        point[3] = interpo_pos[2]

                    else:
                        point[1] = n_i
                        point[2] = n_j
                        point[3] = n_k
                        point[0] = pic_num

                    bi[index] = (interpolator.Evaluate(point)-
reference[n_i][n_j][n_k])**2
                    tem=interpolator.Evaluate(point)-reference[n_i][n_j]
[n_k]
                    derivative = interpolator.EvaluateDerivative(point)
                    diff_x = derivative[1]
                    diff_y = derivative[2]
                    diff_z = derivative[3]
                    (a,b,c)=self.shape[1:]
                    b_diff[index][1:4] = diff_x*tem, diff_y*tem, diff_z*tem
                    b_diff[index][4] = (diff_y*(-0.5*self.affine[0][0]*a*
(sin(q[4])*sin(q[6]) - sin(q[5])*cos(q[4])*cos(q[6]))/self.affine[1][1] +
self.affine[0][0]*(sin(q[4])*sin(q[6]) -
sin(q[5])*cos(q[4])*cos(q[6]))/self.affine[1][1] - 0.5*b*(-
sin(q[4])*cos(q[6]) - sin(q[5])*sin(q[6])*cos(q[4])) - sin(q[4])*cos(q[6])
- sin(q[5])*sin(q[6])*cos(q[4]) - 0.5*self.affine[2]
[2]*c*cos(q[4])*cos(q[5])/self.affine[1][1] + self.affine[2]
[2]*cos(q[4])*cos(q[5])/self.affine[1][1]) + \
                        diff_z*(-0.5*self.affine[0][0]*a*
(sin(q[4])*sin(q[5])*cos(q[6]) + sin(q[6])*cos(q[4]))/self.affine[2][2] +
self.affine[0][0]*(sin(q[4])*sin(q[5])*cos(q[6]) +
sin(q[6])*cos(q[4]))/self.affine[2][2] - 0.5*self.affine[1][1]*b*(
                            sin(q[4])*sin(q[5])*sin(q[6]) -
cos(q[4])*cos(q[6]))/self.affine[2][2] + self.affine[1][1]*
(sin(q[4])*sin(q[5])*sin(q[6]) - cos(q[4])*cos(q[6]))/self.affine[2][2] +
0.5*c*sin(q[4])*cos(q[5]) - sin(q[4])*cos(q[5])))*tem

                    b_diff[index][5] = (diff_x*(0.5*a*sin(q[5])*cos(q[6]) -
sin(q[5])*cos(q[6]) + 0.5*self.affine[1]
[1]*b*sin(q[5])*sin(q[6])/self.affine[0][0] - self.affine[1]
[1]*sin(q[5])*sin(q[6])/self.affine[0][0] - 0.5*self.affine[2]
[2]*c*cos(q[5])/self.affine[0][0] + self.affine[2]
[2]*cos(q[5])/self.affine[0][0]) +\
```

```
                        diff_y*(0.5*self.affine[0]
[0]*a*sin(q[4])*cos(q[5])*cos(q[6])/self.affine[1][1] - self.affine[0]
[0]*sin(q[4])*cos(q[5])*cos(q[6])/self.affine[1][1] +
0.5*b*sin(q[4])*sin(q[6])*cos(q[5]) - sin(q[4])*sin(q[6])*cos(q[5]) +
0.5*self.affine[2][2]*c*sin(q[4])*sin(q[5])/self.affine[1][1] -
self.affine[2][2]*sin(q[4])*sin(q[5])/self.affine[1][1]) +\
                        diff_z*(0.5*self.affine[0]
[0]*a*cos(q[4])*cos(q[5])*cos(q[6])/self.affine[2][2] - self.affine[0]
[0]*cos(q[4])*cos(q[5])*cos(q[6])/self.affine[2][2] + 0.5*self.affine[1]
[1]*b*sin(
                            q[6])*cos(q[4])*cos(q[5])/self.affine[2][2] -
self.affine[1][1]*sin(q[6])*cos(q[4])*cos(q[5])/self.affine[2][2] +
0.5*c*sin(q[5])*cos(q[4]) - sin(q[5])*cos(q[4])))*tem

                    b_diff[index][6] = (diff_x*(0.5*a*sin(q[6])*cos(q[5]) -
sin(q[6])*cos(q[5]) - 0.5*self.affine[1]
[1]*b*cos(q[5])*cos(q[6])/self.affine[0][0] + self.affine[1]
[1]*cos(q[5])*cos(q[6])/self.affine[0][0]) +\
                        diff_y*(-0.5*self.affine[0][0]*a*
(sin(q[4])*sin(q[5])*sin(q[6]) - cos(q[4])*cos(q[6]))/self.affine[1][1] +
self.affine[0][0]*(sin(q[4])*sin(q[5])*sin(q[6]) -
cos(q[4])*cos(q[6]))/self.affine[1][1] - 0.5*b*(-
sin(q[4])*sin(q[5])*cos(q[6]) - sin(q[6])*cos(q[4])) -
sin(q[4])*sin(q[5])*cos(q[6]) - sin(q[6])*cos(q[4])) +\
                        diff_z*(-0.5*self.affine[0][0]*a*
(sin(q[4])*cos(q[6]) + sin(q[5])*sin(q[6])*cos(q[4]))/self.affine[2][2] +
self.affine[0][0]*(sin(q[4])*cos(q[6]) +
sin(q[5])*sin(q[6])*cos(q[4]))/self.affine[2]
                                [2] - 0.5*self.affine[1][1]*b*
(sin(q[4])*sin(q[6]) - sin(q[5])*cos(q[4])*cos(q[6]))/self.affine[2][2] +
self.affine[1][1]*(sin(q[4])*sin(q[6]) -
sin(q[5])*cos(q[4])*cos(q[6]))/self.affine[2][2]))*tem

                    b_diff[index][0] = 1
                    index += 1
        return bi, b_diff
```

最终代码版本2

```
    def iterate(self, reference,resource, q,pic_num):
        '''
        高斯牛顿迭代(gauss-newton iterate)\n
        输入参数(parameter):\n
        resource:原始图像(raw image)\n
        reference:参考图像(reference image)\n
        q:旋转平移参数(rotation and translation parameters)\n
        输出(output):\n
        q:更新后的旋转平移参数(new rotation and translation parameters)\n
        bi:残差(residual)\n
        b_diff:偏导矩阵(derivative matrix)
        '''
```

```
        interpolator = self.interpolator  # B样条插值(B-spline interpolation)
        step = self.step  # 选点间隔(point interval)


        total = self.x * self.y * self.z
        unit = total // size  # 每个进程处理的图像数量
        remain = total % size  # 剩余的图像数量

        if rank == 0:
            start = 0
        else:
            start = rank * unit + remain

        end = (rank + 1) * unit + remain

        bi = np.zeros(end - start)  # 残差 (residual)
        b_diff = np.zeros((end - start, 7))  # 偏导矩阵 (derivative matrix)

        M = self.rigid(q)
        M[3,3]=0.0
        if np.linalg.det(M) <= 1e-10:
            M = np.linalg.pinv(M)
        else:
            M = np.linalg.inv(M)

        # 处理每个进程分配的图像
        for index in range(start, end):
            i = index // (self.y * self.z)
            j = (index // self.z) % self.y
            k = index % self.z

            n_i = int(i*step * self.affine[1, 1]) if int(i*step * self.affine[1, 1]) else self.shape[1] - 1
            n_j = int(j*step * self.affine[2, 2]) if int(j*step * self.affine[2, 2]) else self.shape[2] - 1
            n_k = int(k*step * self.affine[3, 3]) if int(k*step * self.affine[3, 3]) else self.shape[3] - 1

            interpo_pos = M @ [n_i, n_j, n_k, 1]  # 变换后的坐标(transformed coordinate)

            point = itk.Point[itk.D, 4]()
            if 0 <= interpo_pos[0] < self.shape[1] and 0 <= interpo_pos[1] < self.shape[2] and 0 <= interpo_pos[2] < self.shape[3]:  # 判断是否在范围内
                point[0] = pic_num
                point[1] = interpo_pos[0]
                point[2] = interpo_pos[1]
                point[3] = interpo_pos[2]
            else:
                point[1] = n_i
                point[2] = n_j
                point[3] = n_k
```

```
                point[0] = pic_num
            # bi[index - start] = (interpolator.Evaluate(point)-
reference[n_i][n_j][n_k])**2
            bi[index - start] = self.use_inter_evaluate(point,
interpolator, reference, n_i, n_j, n_k)
            # tem=interpolator.Evaluate(point)-reference[n_i][n_j][n_k]
            # derivative = interpolator.EvaluateDerivative(point)
            derivative = self.use_inter_evaluate_derivative(point,
interpolator)
            diff_x = derivative[1]
            diff_y = derivative[2]
            diff_z = derivative[3]
            (a,b,c)=self.shape[1:]
            b_diff[index - start][1:4] = diff_x, diff_y, diff_z
            # b_diff[index - start][1:4] = derivative[1:4]
            b_diff[index - start][4] = (diff_y*(-0.5*self.affine[0][0]*a*
(sin(q[4])*sin(q[6]) - sin(q[5])*cos(q[4])*cos(q[6]))/self.affine[1][1] +
self.affine[0][0]*(sin(q[4])*sin(q[6]) -
sin(q[5])*cos(q[4])*cos(q[6]))/self.affine[1][1] - 0.5*b*(-
sin(q[4])*cos(q[6]) - sin(q[5])*sin(q[6])*cos(q[4])) - sin(q[4])*cos(q[6])
- sin(q[5])*sin(q[6])*cos(q[4]) - 0.5*self.affine[2]
[2]*c*cos(q[4])*cos(q[5])/self.affine[1][1] + self.affine[2]
[2]*cos(q[4])*cos(q[5])/self.affine[1][1]) + \
                diff_z*(-0.5*self.affine[0][0]*a*
(sin(q[4])*sin(q[5])*cos(q[6]) + sin(q[6])*cos(q[4]))/self.affine[2][2] +
self.affine[0][0]*(sin(q[4])*sin(q[5])*cos(q[6]) +
sin(q[6])*cos(q[4]))/self.affine[2][2] - 0.5*self.affine[1][1]*b*(
                    sin(q[4])*sin(q[5])*sin(q[6]) -
cos(q[4])*cos(q[6]))/self.affine[2][2] + self.affine[1][1]*
(sin(q[4])*sin(q[5])*sin(q[6]) - cos(q[4])*cos(q[6]))/self.affine[2][2] +
0.5*c*sin(q[4])*cos(q[5]) - sin(q[4])*cos(q[5]))) #*tem
            b_diff[index - start][5] = (diff_x*(0.5*a*sin(q[5])*cos(q[6]) -
sin(q[5])*cos(q[6]) + 0.5*self.affine[1]
[1]*b*sin(q[5])*sin(q[6])/self.affine[0][0] - self.affine[1]
[1]*sin(q[5])*sin(q[6])/self.affine[0][0] - 0.5*self.affine[2]
[2]*c*cos(q[5])/self.affine[0][0] + self.affine[2]
[2]*cos(q[5])/self.affine[0][0]) +\
                diff_y*(0.5*self.affine[0]
[0]*a*sin(q[4])*cos(q[5])*cos(q[6])/self.affine[1][1] - self.affine[0]
[0]*sin(q[4])*cos(q[5])*cos(q[6])/self.affine[1][1] +
0.5*b*sin(q[4])*sin(q[6])*cos(q[5]) - sin(q[4])*sin(q[6])*cos(q[5]) +
0.5*self.affine[2][2]*c*sin(q[4])*sin(q[5])/self.affine[1][1] -
self.affine[2][2]*sin(q[4])*sin(q[5])/self.affine[1][1]) +\
                diff_z*(0.5*self.affine[0]
[0]*a*cos(q[4])*cos(q[5])*cos(q[6])/self.affine[2][2] - self.affine[0]
[0]*cos(q[4])*cos(q[5])*cos(q[6])/self.affine[2][2] + 0.5*self.affine[1]
[1]*b*sin(
                    q[6])*cos(q[4])*cos(q[5])/self.affine[2][2] -
self.affine[1][1]*sin(q[6])*cos(q[4])*cos(q[5])/self.affine[2][2] +
0.5*c*sin(q[5])*cos(q[4]) - sin(q[5])*cos(q[4])))#*tem
            b_diff[index - start][6] = (diff_x*(0.5*a*sin(q[6])*cos(q[5]) -
sin(q[6])*cos(q[5]) - 0.5*self.affine[1]
[1]*b*cos(q[5])*cos(q[6])/self.affine[0][0] + self.affine[1]
[1]*cos(q[5])*cos(q[6])/self.affine[0][0]) +\
```

```
                diff_y*(-0.5*self.affine[0][0]*a*
(sin(q[4])*sin(q[5])*sin(q[6]) - cos(q[4])*cos(q[6]))/self.affine[1][1] +
self.affine[0][0]*(sin(q[4])*sin(q[5])*sin(q[6]) -
cos(q[4])*cos(q[6]))/self.affine[1][1] - 0.5*b*(-
sin(q[4])*sin(q[5])*cos(q[6]) - sin(q[6])*cos(q[4])) -
sin(q[4])*sin(q[5])*cos(q[6]) - sin(q[6])*cos(q[4])) +\
                diff_z*(-0.5*self.affine[0][0]*a*(sin(q[4])*cos(q[6]) +
sin(q[5])*sin(q[6])*cos(q[4]))/self.affine[2][2] + self.affine[0][0]*
(sin(q[4])*cos(q[6]) + sin(q[5])*sin(q[6])*cos(q[4]))/self.affine[2][2] -
0.5*self.affine[1][1]*b*(sin(q[4])*sin(q[6]) -
sin(q[5])*cos(q[4])*cos(q[6]))/self.affine[2][2] + self.affine[1][1]*
(sin(q[4])*sin(q[6]) - sin(q[5])*cos(q[4])*cos(q[6]))/self.affine[2]
[2]))#*tem
            b_diff[index - start][0] = 1

            # print('tem:',tem)
            # break

        # 汇总所有进程处理的结果
        gathered_bi = np.zeros(unit * size)  # 创建一个空数组用于存储汇总结果
        gathered_b_diff = np.zeros((unit * size, 7))  # 创建一个空数组用于存储
汇总结果

        if rank == 0:
            comm.Gather(bi[remain:], gathered_bi, root=0)  # 将每个进程处理的
结果汇总到主进程
            comm.Gather(b_diff[remain:], gathered_b_diff, root=0)  # 将每个
进程处理的结果汇总到主进程
        else:
            comm.Gather(bi, gathered_bi, root=0)  # 将每个进程处理的结果汇总到
主进程
            comm.Gather(b_diff, gathered_b_diff, root=0)  # 将每个进程处理的结
果汇总到主进程
        return gathered_bi, gathered_b_diff
```

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

```python
 @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