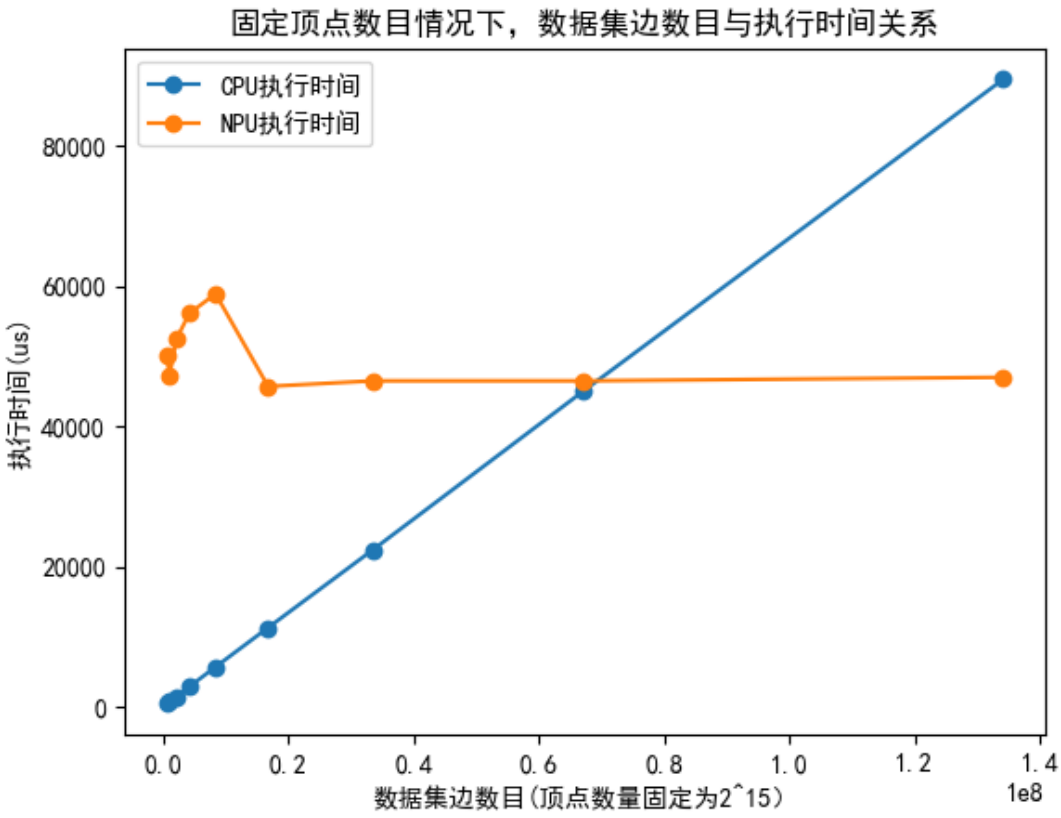
补充测试

理论上NPU在计算的时候，其处理时间与数据集规模无关，与数据稠密度有关。也就是说顶点数目确定后，NPU的执行时间就已经确定，不会随着顶点平均度数的提高而提高。但是之前测试发现，NPU执行时间还是有波动。分析了一下原因：NPU端代码在生成矩阵的时候做了一个自动去重，把非零点筛除了，所以每次生成的实际矩阵规模<=理论上矩阵的规模。在对代码修改后，重新进行了补充测试：

数据集	NPU					CPU				
	迭代轮次	总执行时间(ms)	算子耗时			总执行时间(us)				
			add算子耗时(ms)		Reducemin算子耗时(ms)					
			Add-op1(ms)	Add-op5(ms)						
自定义数据集(2^15个顶点，平均每个顶点与16条边相连)，数据集大小已达到5.7M	4	50.213	3.473	9.049	37.632	486		16	486	50.213
自定义数据集(2^15个顶点，平均每个顶点与32条边相连)，数据集大小已达到12M	3	47.259	4.33	8.499	34.385	775		32	775	47.259
自定义数据集(2^15个顶点，平均每个顶点与64条边相连)，数据集大小已达到23M	3	52.396	5.006	9.85	37.492	1484		64	1484	52.396
自定义数据集(2^15个顶点，平均每个顶点与128条边相连)，数据集大小已达到46M	3	56.129	5.555	10.943	39.579	2902		128	2902	56.129
自定义数据集(2^15个顶点，平均每个顶点与256条边相连)，数据集大小已达到91M	3	58.916	5.931	11.693	41.251	5730		256	5730	58.916
自定义数据集(2^15个顶点，平均每个顶点与512条边相连)，数据集大小已达到181M	2	45.718	6.168	8.079	31.435	11321		512	11321	45.718
自定义数据集(2^15个顶点，平均每个顶点与1024条边相连)，数据集大小已达到362M	2	46.509	6.325	8.248	31.899	22449		1024	22449	46.509
自定义数据集(2^15个顶点，平均每个顶点与4096条边相连)，数据集大小已达到1.5G	2	（数据传输时间过	6.444	8.417	32.118	89509		4096	89509	47.013



补充测试证明了，当数据集规模在NPU能力范围之内时，NPU的处理时间与数据集稠密度无关，CPU的执行时间与数据集稠密度呈现正相关。理论上随着数据集稠密度的提升，CPU端的执行时间会与NPU端的执行时间相交：

- NPU的内存资源有限，实验表明可以处理的数据集规模最大约为2^15。在此规模下，平均度数最大为2 ^15。根据模型分析，CPU的理论执行时间为718292us。NPU的理论执行时间为51.02ms。此时NPU的执行时间确实快于CPU的执行时间

但是实际中无法忽略掉NPU端的数据传输、数据转换的开销。实验中当数据集的平局、平均度数提高，NPU端的数据处理开销也成倍提高。当平均度数达到2^13时，数据处理开销的时间超出了10min（数据开销大的原因应该是数据集文件的大小太大，造成处理困难）。

此外为了验证"NPU处理时间与数据集稠密度无关"的猜想，实验数据集采用了极端的稠密度，实际数据集中数据稠密度很低（之前的汇报有证明）。后面还证明了，即使采用多种重排序测试，得到的稠密子图的稠密度比原始图还要低。

补充：前面说"**NPU的执行时间与数据集的规模有关，与数据集的稠密度无关**"其实并不严谨。对于图算法来说，数据的稠密度会影响迭代的次数，只不过对于BFS算法来说，数据集越稠密，所需要的迭代次数越少。而实验中迭代次数一直保持最低的迭代次数，所以实验结果中NPU的执行时间几乎不变。更严谨的说法是："**NPU的单次迭代执行时间与数据集的规模有关，与数据集的稠密度无关**"

pagerank测试

数据集	NPU				CPU
	迭代轮次	总执行时间(ms)	MatMul-op1(ms)	MatMul-op5(ms)	总执行时间(us)
自定义数据集(2^15个顶点, 平均每个顶点与16条边相连)，数据集大小已达到5.7M	10	51.848	4.304	47.257	9917
自定义数据集(2^15个顶点, 平均每个顶点与32条边相连)，数据集大小已达到12M	10	61.884	5.135	56.449	20054
自定义数据集(2^15个顶点, 平均每个顶点与64条边相连)，数据集大小已达到23M	11	78.233	5.989	71.895	37100
自定义数据集(2^15个顶点, 平均每个顶点与128条边相连)，数据集大小已达到46M	11	85.29	6.536	78.402	302448
自定义数据集(2^15个顶点, 平均每个顶点与256条边相连)，数据集大小已达到91M	11	89.2	6.839	81.959	149807
自定义数据集(2^15个顶点, 平均每个顶点与512条边相连)，数据集大小已达到181M	12	99.264	7.07	91.835	308957

不同图的连通性差别很大，一开始没有考虑这一因素，导致测出的pagerank算法性能波动很大。为了使实验结果更加严谨，对于同一规模的数据集要连续测试10次性能。

todo……

图重排序测试

之前仅采用"**顶点度数**"一项标准，进行子图划分。发现排序后子图的稠密度并没有显著提升。下面是采用新的图重排序的算法的测试结果。后面又尝试了基于图划分和基于图聚类的方法来提取一个稠密子图。

1. 算法步骤：

1. 读取原始数据集文件，并创建一个有向图对象。
2. 为了去除重复的边，使用一个集合来存储边的信息。
3. 遍历原始数据集文件的每一行，将起始点和终点解析为整数，并将边的信息添加到集合中。
4. 将集合中的边添加到图对象中。
5. 打印原始图数据集的信息，包括顶点数、边数和平均度数。
6. 根据给定的方法进行稠密子图划分：
    - 如果方法是'partition'（图划分方式）：
        - 使用弱连通组件算法找到原始图中的所有连通分量。
        - 计算每个连通分量的平均度数。
        - 按照平均度数从高到低对连通分量进行排序。
        - 选择最稠密的连通分量，使得总顶点数大于等于目标顶点数。
        - 构建最终的稠密子图，包含选定的连通分量中的顶点及其之间的边。
    - 如果方法是'clustering'（图聚类方式）：
        - 将有向图转换为无向图。
        - 使用图聚类算法计算每个节点的聚类系数。
        - 根据聚类系数对节点进行排序。
        - 选择排名靠前的节点作为稠密子图的顶点。
        - 构建最终的稠密子图，包含选定的顶点及其之间的边。
    - 如果方法选择无效，则打印错误消息并返回。
7. 打印稠密子图数据集的信息，包括顶点数、边数和平均度数。
8. 将稠密子图输出到文件中。

2. 算法原理：

该算法的目标是从给定的图数据集中提取出一个稠密子图。根据给定的方法选择，算法有两种不同的实现方式。

1. 图划分方式（method='partition'）：
    - 首先，使用弱连通组件算法找到原始图中的所有连通分量。
    - 然后，计算每个连通分量的平均度数，通过计算每个连通分量的边数与顶点数之比的两倍来得到平均度数。
    - 接下来，按照平均度数从高到低对连通分量进行排序。
    - 选择最稠密的连通分量，并确保选定的连通分量的总顶点数小于等于目标顶点数。
    - 最后，构建稠密子图，包含选定连通分量中的顶点及其之间的边。
2. 图聚类方式（method='clustering'）：
    - 首先，将有向图转换为无向图，以便应用图聚类算法。
    - 然后，使用图聚类算法计算每个节点的聚类系数，聚类系数表示节点在其邻居节点中形成三角形的概率。
    - 接下来，根据聚类系数对节点进行排序，按照聚类系数从高到低进行排序。
    - 选择排名靠前的节点作为稠密子图的顶点。
    - 最后，构建稠密子图，包含选定的顶点及其之间的边。

无论使用哪种方式，算法最后会输出稠密子图的信息，并将稠密子图保存到指定的输出文件中。

3. 实验结论

异构计算的方式下，子图规模过小的话，矩阵运算的性能增益不足以弥补，信息传输的开销。所以子图划分的规模是根据NPU上限定的40000。但是实验发现，两种方式划分得到的稠密子图，其平均度数均小于原始图的平均度数，没有明显增益。