

## Supplementary Material

### A *i*QAN Algorithm

---

**Algorithm 2:** *i*QAN: Intra-Query Parallel ANNS

---

**Input:** graph  $G$ , starting point  $P$ , query  $Q$ , queue capacity  $L$ , number of workers  $T$

**Output:**  $K$  nearest neighbors of  $Q$

```

1431
1432
1433
1434
1435
1436
1437
1438
1439
1440
1441
1442
1443
1444
1445
1446
1447
1448
1449
1450
1451
1452
1453
1454
1455
1456
1457
1458
1459
1460
1461
1462
1463
1464
1465
1466
1467
1468
1469
1470
1471
1472
1473
1474
1475
1476
1477
1478
1479
1480
1481
1482
1483
1484
1485
1486
1487
1488
1489
1490
1491
1492
1493
1494
1495
1496
1497
1498
1499
1500
1501
1502
1503
1504
1505
1506
1507
1508
1509
1510
1511
1512
1513
1514
1515
1516
1517
1518
1519
1520
1521
1522
1523
1524
1525
1526
1527
1528
1529
1530
1531
1532
1533
1534
1535
1536
1537
1538
1539
1540
1541
1542
1543
1544
1545
1546
1547
1548
1549
1550
1551
1552
1553
1554
1555
1556
1557
1558
1559
1560
1561
1562
1563
1564
1565
1566
1567
1568
1569
1570
1571
1572
1573
1574
1575
1576
1577
1578
1579
1580
1581
1582
1583
1584
1585
1586
1587
1588
1589
1590
1591
1592
1593
1594
1595
1596
1597
1598
1599
1600
1601
1602
1603
1604
1605
1606
1607
1608
1609
1610
1611
1612
1613
1614
1615
1616
1617
1618
1619
1620
1621
1622
1623
1624
1625
1626
1627
1628
1629
1630
1631
1632
1633
1634
1635
1636
1637
1638
1639
1640
1641
1642
1643
1644
1645
1646
1647
1648
1649
1650
1651
1652
1653
1654
1655
1656
1657
1658
1659
1660
1661
1662
1663
1664
1665
1666
1667
1668
1669
1670
1671
1672
1673
1674
1675
1676
1677
1678
1679
1680
1681
1682
1683
1684
1685
1686
1687
1688
1689
1690
1691
1692
1693
1694
1695
1696
1697
1698
1699
1700
1701
1702
1703
1704
1705
1706
1707
1708
1709
1710
1711
1712
1713
1714
1715
1716
1717
1718
1719
1720
1721
1722
1723
1724
1725
1726
1727
1728
1729
1730
1731
1732
1733
1734
1735
1736
1737
1738
1739
1740
1741
1742
1743
1744
1745
1746
1747
1748
1749
1750
1751
1752
1753
1754
1755
1756
1757
1758
1759
1760
1761
1762
1763
1764
1765
1766
1767
1768
1769
1770
1771
1772
1773
1774
1775
1776
1777
1778
1779
1780
1781
1782
1783
1784
1785
1786
1787
1788
1789
1790
1791
1792
1793
1794
1795
1796
1797
1798
1799
1800
1801
1802
1803
1804
1805
1806
1807
1808
1809
1810
1811
1812
1813
1814
1815
1816
1817
1818
1819
1820
1821
1822
1823
1824
1825
1826
1827
1828
1829
1830
1831
1832
1833
1834
1835
1836
1837
1838
1839
1840
1841
1842
1843
1844
1845
1846
1847
1848
1849
1850
1851
1852
1853
1854
1855
1856
1857
1858
1859
1860
1861
1862
1863
1864
1865
1866
1867
1868
1869
1870
1871
1872
1873
1874
1875
1876
1877
1878
1879
1880
1881
1882
1883
1884
1885
1886
1887
1888
1889
1890
1891
1892
1893
1894
1895
1896
1897
1898
1899
1900
1901
1902
1903
1904
1905
1906
1907
1908
1909
1910
1911
1912
1913
1914
1915
1916
1917
1918
1919
1920
1921
1922
1923
1924
1925
1926
1927
1928
1929
1930
1931
1932
1933
1934
1935
1936
1937
1938
1939
1940
1941
1942
1943
1944
1945
1946
1947
1948
1949
1950
1951
1952
1953
1954
1955
1956
1957
1958
1959
1960
1961
1962
1963
1964
1965
1966
1967
1968
1969
1970
1971
1972
1973
1974
1975
1976
1977
1978
1979
1980
1981
1982
1983
1984
1985
1986
1987
1988
1989
1990
1991
1992
1993
1994
1995
1996
1997
1998
1999
2000

```

Algorithm 2 describes the overall algorithm of *i*QAN. At the beginning of each global step, the global queue evenly divides its unchecked candidates among all workers. After that, each worker performs a local best-first search based on its own local queue (Line 10 to Line 24). In a local search step, a worker expands its best-unchecked candidate and updates its private queue accordingly. A worker continues expansion until the *checker* indicates a sync or it has no unchecked candidates left locally. In the path-wise parallelism, the visiting map is shared by all workers to indicate if a vertex has been visited. Since multiple threads may access the shared visiting map concurrently, locking or lock-free algorithms are required to ensure a vertex is visited only once. Fortunately, the ANNS algorithm remains correct even if a vertex is calculated multiple times because the local candidates are

guaranteed to be merged back to the global priority queue. Therefore, multiple threads do not need to exclusively update the visiting map, which is then called a *loosely synchronized visiting map* and can reduce communication overhead. It is implemented using a bit vector instead of a byte array for faster access.

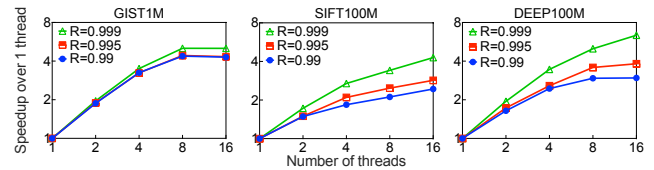
### B Details of Datasets

**Table 2.** Characterization of datasets. Dim. denotes the dimension of a feature vector, #base denotes the number of points, and #queries denotes the number of queries.

Dataset	Dim.	#base	#queries
SIFT1M	128	1M	10K
GIST1M	960	1M	1K
DEEP10M	96	10M	10K
SIFT100M	128	100M	10K
DEEP100M	96	100M	10K
BIGANN	128	1B	10K
DEEP1B	96	1B	10K

Table 2 summarizes the details of the datasets we use in the evaluation.

### C Latency Speedups on Skylake



**Figure 18.** Speedups of *i*QAN over the sequential baseline on Skylake.

Figure 18 shows that *i*QAN on Skylake achieves average 4.4× and 5.2× speedups over the sequential baseline with 8- and 16-thread at recall 0.999, respectively.

### D Comparison with a GPU Implementation.

**Table 3.** Latency comparison of *i*QAN and Faiss-GPU on five datasets. Lt. means *Latency*. OOM means *out of memory*. Faiss-GPU’s index format is IVFFlat. *i*QAN uses 32 threads.

Datasets	Faiss-GPU w/ IVFFlat		<i>i</i> QAN-32T on KNL	
	R@100	Lt. (ms.)	R@100	Lt. (ms.)
SIFT1M	0.52	0.87	<b>0.91</b>	<b>0.61</b>
GIST1M	0.36	7.25	<b>0.90</b>	<b>1.21</b>
DEEP10M	0.62	5.79	<b>0.90</b>	<b>0.96</b>
SIFT100M	OOM	OOM	<b>0.90</b>	<b>2.00</b>
DEEP100M	OOM	OOM	<b>0.90</b>	<b>1.91</b>

We also compare *i*QAN with a GPU-based large-scale ANNS algorithm [30] in the Faiss library [1]. The GPU experiments are conducted on an NVIDIA Tesla P100 with CUDA 10.2. Faiss is set to have one query in every batch because we focus on reducing the online query latency to meet stringent latency requirements. Table 3 shows the latency comparison results on five datasets. *i*QAN uses 32 threads on KNL. For

the SIFT100M and DEEP100M, Faiss-GPU complains of out-of-memory errors. For other datasets, *iQAN* outperforms Faiss-GPU with 1.4× to 6.0× speedup and much better recall, which indicates that *iQAN* can effectively achieve faster latency on CPUs than GPU-based algorithms.

### E Effects on memory bandwidth.

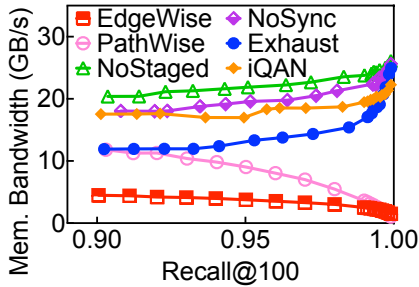


Figure 19. Memory bandwidth on KNL using 32 threads.

Figure 19 shows that *iQAN* achieves a much higher memory bandwidth utilization than *EdgeWise*. For example, the memory bandwidth increases from 1.6 GB/s to 21.1 GB/s at recall 0.999. This bandwidth improvement comes from *iQAN*'s design to expose more parallelism during neighbor expansion while carefully eliminating synchronization

overhead and redundant computations. To see how each optimization affects the memory bandwidth utilization, we also include the results of intermediate configurations. *PathWise* overall achieves better memory bandwidth utilization than *EdgeWise*. This is because it exposes more parallelism during the neighbor expansion process. However, as the recall target increases, the memory bandwidth of *PathWise* declines gradually towards *EdgeWise*. This is because *PathWise* does not apply redundancy-aware synchronization and still merges local candidates to the global priority queue in every iteration. As a result, the synchronization overhead becomes a dominant factor and limits the memory bandwidth utilization when the search is under the high recall region. *NoStaged*, *NoSync*, *Exhaust*, and *iQAN* all improve the memory bandwidth utilization considerably because they all reduce frequent synchronizations. *NoStaged* and *NoSync* have slightly higher memory bandwidth utilization than *iQAN* because *NoStaged* and *NoSync* either use the maximum expansion width during the entire search process or perform no synchronization across worker threads, which leads to more (redundant) data loads during neighbor expansion. Therefore, *iQAN* results in shorter query latency than *NoStaged* and *NoSync* even though it requires slightly less memory bandwidth utilization.