# Parallelizing Similarity Search

Exploring the Parallelization of Inverted File Indexes

Akash Nayar, Dhruva Byrapatna

December 2025

## 1 Summary

We parallelized the training, building, and searching of inverted vector file indexes (IVF indexes), a data structure used to store and index vector embeddings. We explored multiple avenues of optimization, such as cache-friendly/vectorized distance calculations and different modes of parallelization using SIMD intrinsics and the OpenMP framework. We tested our implementations on the GHC and PSC Bridges machines, demonstrating that a combination of SIMD vectorization and OpenMP parallelization can achieve a total speedup of up to 310× on train, 51× on build, and 43× on query.

## 2 Background

A key component in the rise of large language models (LLMs) has been the rapid development of similarity search indexes. Key technologies such as Retrieval-Augmented Generation (RAG) and recommendation systems rely on efficient and accurate retrieval of similar high-dimensional vectors to a given query. For instance, with RAG, large language models can index and incorporate new information from document stores that was not available in the model's training data. Before a RAG system is deployed, each document in the document store is encoded into a vector embedding—a list of floating point features which describes the document's contents and semantics—and stored in a vector database. During runtime, a query is similarly encoded into an embedding and the vector database is scanned to find entries similar to the query. The corresponding results are appended to the query which is then fed into the LLM.

The process seems relatively simple on the surface, but the inner mechanics of efficiently and accurately performing similarity search over a high-dimensional space are significantly more complicated. A naive implementation of document storage and retrieval might, after converting a query to a vector, compare each document vector in the database to the query to find the most similar vectors. This, of course, would result in extremely slow query processing for RAG and recommendation systems with large document stores, making the system prohibitively expensive. Optimizations are evidently needed to make these systems usable in a production environment.

An inverted vector file index (IVF) is a common structure used to speed up similarity search in such settings. The core idea is to avoid comparing a query against every vector in the dataset. Instead, the dataset is partitioned into $k$ clusters using the $k$-means algorithm. Each cluster has a centroid and the IVF index stores, as its "inverted lists", the vectors that belong to each cluster.

At query time, we first compare the query vector to all $n_{\text{list}}$ (greater than 0; a user-controlled parameter) centroids and select the $n_{\text{probe}}$ ($1 \leq n_{\text{probe}} \leq n_{\text{list}}$; also user-controlled) closest ones. Lower settings of $n_{\text{probe}}$ will compromise accuracy for speed. Only the vectors stored in the corresponding $n_{\text{probe}}$ clusters are then compared against the query to identify the most similar items. This strategy of searching only in the nearest clusters dramatically reduces the number of distance computations needed while retaining high accuracy, and is used in major similarity search libraries and applications like Facebook AI Similarity Search (Faiss).

We initially implemented a serial version of the $k$-means algorithm for training and building an IVF, as well as a serial similarity search routine. The IVF data structure represents a significant improvement over a naive similarity search ($k$-NN), but the initial serial implementation still has a lot of places where parallelization can lead to additional gains. While such indexes can technically work on any distance metric, we stuck with just the squared $L_2$ norm (squared Euclidian distance) for simplicity.

Overall, the many squared $L_2$ distance calculations performed at query-time are very computationally intensive and have the ability to be parallelized. In the serial implementation, each query is processed one-by-one, and is compared to each candidate vector sequentially. This workflow is very amenable to parallelism—there are repeated computations with high spatial locality, and we explored using SIMD, OpenMP, and other restructuring methods to improve performance.

The squared $L_2$ norm of a $d$-dimensional vector $\mathbf{v}$ is calculated as follows:

$$||\mathbf{v}||_2^2 = \sum_{i=1}^{d} v_i^2 \tag{1}$$

The squared distance between a query $\mathbf{q}$ and a candidate $\mathbf{c}$ is simply $||\mathbf{q}-\mathbf{c}||_2^2$. Since this just requires a subtraction followed by squaring (and subsequent accumulation), it lends itself to vecotrization.

Thus, a SIMD-ized squared $L_2$ distance kernel should already lead to nontrivial speedups.

With respect to the training and building phase, one of the key areas where parallelization helps is in the $k$-means convergence loop where centroids are learned (train; using $k$-means++) and points are assigned to their corresponding centroid (build). In the serial implementation, each vector is compared to each centroid in sequence, and the vector is then assigned to the centroid list with which it has the highest similarity. These comparisons can be done independently and therefore there exist significant opportunities for data-parallelization.

With respect to the search phase, another key area where parallelization helps is when there are multiple queries. In the serial implementation of search, each query is processed sequentially-it is independently compared to each centroid, and then independently compared to each vector in the chosen lists. Because all of these comparisons are independent read-only operations, there exist significant opportunities to parallelize this using OpenMP, assigning each query to a separate thread for distance computation work. Query vectors can also be compared to candidates in parallel, allowing us to explore multiple methods of parallelization.

# 3 Approach

## 3.1 Approach Outline and Serial Implementation

Our implementation was written entirely in C++. While we researched the design of IVF and $k$-means algorithms, we did not rely on any reference or starter code to implement these. We began with a clean, fully serial baseline implementation so that correctness and performance differences were easy to measure. To evaluate correctness, we exposed Python bindings for all major operations—training, building, and searching—and integrated our implementation into a custom Python test suite. This allowed us to directly compare our outputs against Scikit-Learn's K-Means and a brute-force $k$-NN implementation, which greatly sped up debugging and validation. Performance testing was conducted primarily on the GHC Linux machines, with additional large-scale runs performed on PSC Bridges.

We first implemented a traditional $k$-means training loop following the standard convergence procedure: initialize centroids using $k$-means++, then iteratively assign each vector to its closest centroid and recompute centroid positions, stopping when centroids converge (no more updates are made). This was adapted closely from the algorithmic description on Wikipedia but rewritten from scratch for our use-case in C++. After convergence, the IVF "build" step simply inserts each vector into the inverted list corresponding to its nearest centroid. Building is therefore straightforward once the cluster assignments are known.

The search operation follows the classic IVF workflow. For each query, we compute its distances to all $n_{\text{list}}$ centroids, select the $n_{\text{probe}}$ closest ones, and then compute squared $L_2$ distances between the query and all vectors in those selected lists (known as "candidates").

## 3.2 Distance Computations

Because distance calculations dominate runtime in both training and search, we implemented a vectorized $L_2$ routine using SIMD. Our best-performing version processes elements in blocks of 8 float values using 256-bit registers (`__m256`), with a short scalar tail loop to handle the case where the dimensionality is not divisible by 8. This SIMD routine produced substantial reductions in distance-computation latency with minimal code complexity, and in practice outperformed all other distance-related optimizations we attempted. Some of these are discussed below.

We experimented with cache-friendly distance calculation optimizations. Our first attempt (called "Cache" in the results) aimed to calculate the squared $L_2$ norm between a query vector and its candidates in 64 byte blocks. Our idea was that loading one chunk of the query vector into the L1 cache at a time and using it against all candidates in an inverted list would improve performance by not repeatedly streaming the entire query through cache for each candidate. We expected this to be especially helpful in higher-dimensional settings, and while we saw some performance gains through this method, were suspicious that a naive storage layout (with vectors stored sequentially, one-after-another) was holding us back.

Our second iteration ("CacheV2") similarly aimed to operate over the same "slice" of the query over multiple candidates. However, we now were restructuring the storage layout such that the first "chunk" (of 16 elements) of each query was stored first, followed by the next 16, etc. We could similarly avoid streaming the query through cache for each candidate, and also enable a more linear access pattern over the candidates, possibly enabling more effective prefetching. This second approach was massively effective in greatly reducing the number of L1 and L3 cache misses over our initial attempt (results discussed more in Section 4.4).

We also experimented with OpenMP parallelization within the distance kernel (essentially parallelizing over dimensions of the query), but the results were incredibly poor due to extremely high overhead and will not be discussed further in this report.

### 3.3 Query Parallelization

During search, the bulk of the work lies in computing query–candidate distances. These operations are entirely read-only and independent, making them a good fit for coarse-grained parallelism. Our best-performing design maps each query to a separate OpenMP thread. This query-per-thread strategy works well because queries do not interact with each other and each query involves enough computation to amortize any OpenMP overhead. We achieved strong results by using the default static scheduler for OpenMP, and employed static scheduling in all other instances where OpenMP parallelization was tested, as all of these instances had similar workloads for each parallel section. We also found that combining this query-level parallelization with our SIMD-ized distance kernel led to extremely strong speedups discussed in Section 4.

### 3.4 Candidate Parallelization

We similarly experimented with candidate-level parallelization. We now processed queries sequentially, assigning threads to subsets of vectors within a single inverted list. Each thread would compute the $L_2$ distance between a query vector and a subset of candidate vectors in the inverted list. The set up for this was similar to query parallelization, but results were not as strong; this is discussed further in the results section, but in short we believe that significantly higher OpenMP overhead led to worse runtime and speedup when employing this strategy.

### 3.5 Training Parallelization

In the $k$-means assignment step, distance comparisons between each point and each centroid are independent. We parallelized this by assigning different subsets of points to different OpenMP threads. Each thread accumulates its own temporary membership counts and centroid sums, which are safely merged at the end of each iteration. This reduces synchronization overhead while aligning naturally with the algorithm's dataflow. Because each vector's assignment can be computed independently, $k$-means training achieved strong scaling and required no major algorithmic restructuring beyond reorganizing accumulation into per-thread buffers.

### 3.6 Optimal Implementation and Other Notes

Our final, most optimal implementation combines the SIMD-ized distance kernel calculations with OpenMP query parallelization in the search function and OpenMP parallelization during $k$-means iterations, giving us a total speedup of 32-310× over our serial implementation.

We are fairly confident we have found a near optimal solution—we experimented with a couple other combinations of methods, like combining query and candidate parallelization strategies with OpenMP, and trying to combine cache-friendly and SIMD-ized distance computations, but nothing could beat the simpler solution. We tested our implementations on the GHC and PSC machines and saw strong speedups in line with what we anticipated. In addition, we explored all avenues of parallelization discussed in our proposal except for ISPC-based distance vectorization, which we ultimately decided against. Our custom SIMD implementation achieved excellent performance with far lower complexity, and adding ISPC would have required intrusive changes to our build system with limited expected benefit. We chose instead to focus our efforts on additional areas of parallelization, like in the $k$-means convergence loop, which gave strong results. A full breakdown of the results from our various approaches is discussed in the next section.

## 4 Results

To evaluate the performance of our implementation, we measured the execution time (in seconds) of each of the three main operations: index training ($k$-means), index build (adding vectors into the appropriate inverted lists), and similarity search. This allows us to derive speedup values by comparing the execution time of a specific index on $n > 1$ threads to its single-threaded performance. For the GHC machines, we vary $n$ from 1 to 8 in powers of 2. For the PSC machines, we do the same up to 128. The legends of the following plots contain shorthand notations for the index names. They can be interpreted as follows: "SIMD" denotes a vectorized distance kernel while the lack thereof denotes a scalar implementation. "Cache"/"CacheV2" denotes our cache-optimized implementation. "QPara" refers to an implementation where we parallelize over multiple queries while "CPara" refers to when we instead parallelize over candidates within an IVF cluster. For example, "SIMDQPara" refers to our vectorized, query-parallel implementation.

As a note, the results for a single-threaded run of "CPara" and "QPara" can be interpreted as the performance for a fully serial version of the IVF code (with OpenMP disabled), as this is the equivalent of a non-parallelized non-SIMD-ized build, train, and query.

This section will cover our key results, where we select a subset of our implementations to analyze. All speedup graphs contain a dotted reference line at 1×, denoting "no speedup". Following each batch of graphs is a table denoting the maximum speedups observed on either the GHC and PSC machines for train, build, and query. This is calculated by finding the smallest recorded time for each operation. These numbers are compared to the fastest single-threaded performance between

CPara and QPara (since both have approximately the same performance on one thread). Our full results on all attempted implementations can be found in Appendix A.

## 4.1 Experimental Setup

We benchmarked our implementations on 5 datasets:

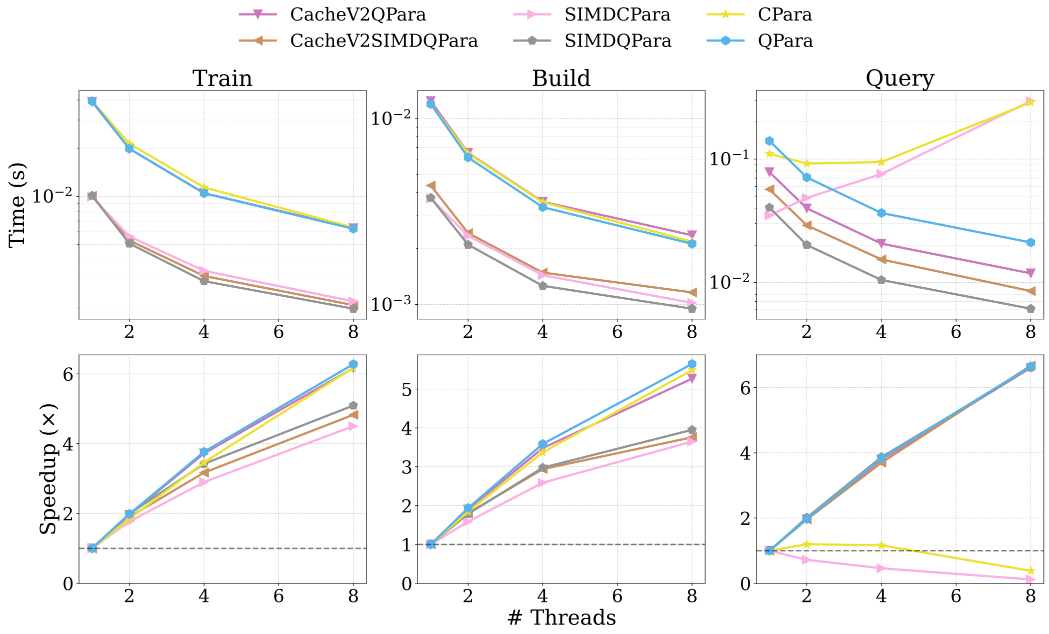
| **Dataset Name** | $d$ | **Dataset Size** ($n_b$) | **Train Size** ($n_t$) | **# Queries** ($n_q$) | $n_{\text{probe}}$ | $n_{\text{list}}$ |
|---|---|---|---|---|---|---|
| Easy | 128 | 10,000 | 1,000 | 256 | 5 | 10 |
| Medium | 512 | 10,000 | 1,000 | 256 | 5 | 10 |
| Hard | 1024 | 100,000 | 1,000 | 256 | 25 | 50 |
| Extreme | 1024 | 1,000,000 | 10,000 | 256 | 25 | 50 |
| GIST | 960 | 1,000,000 | 10,000 | 256 | 5 | 50 |

Table 1: Dataset Parameters

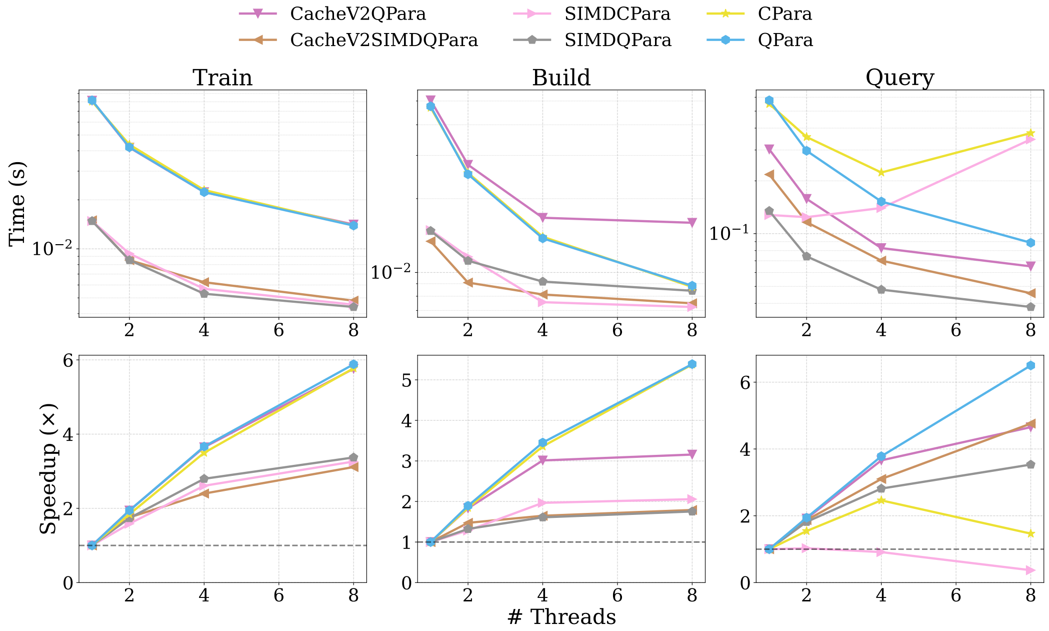
To generate datasets “Easy”, “Medium”, “Hard”, and “Extreme” we used the Faiss library’s `SyntheticDataset` generator (which yields random, yet challenging data for this specific purpose). On the PSC machines, we also benchmarked on the GIST1M dataset, which contains 960-dimensional image descriptors. This dataset was too large to load onto the GHC machines. As a note, the easy and medium datasets do not necessarily reflect real world scenarios-they are smaller than the vast majority of document stores and other related vector databases. While we felt their inclusion would be useful to understand how our system operates in low-load situations, tests on the remaining 4 datasets are far more representative of how our parallelized IVF implementation would behave in real-world applications.

## 4.2 GHC Results

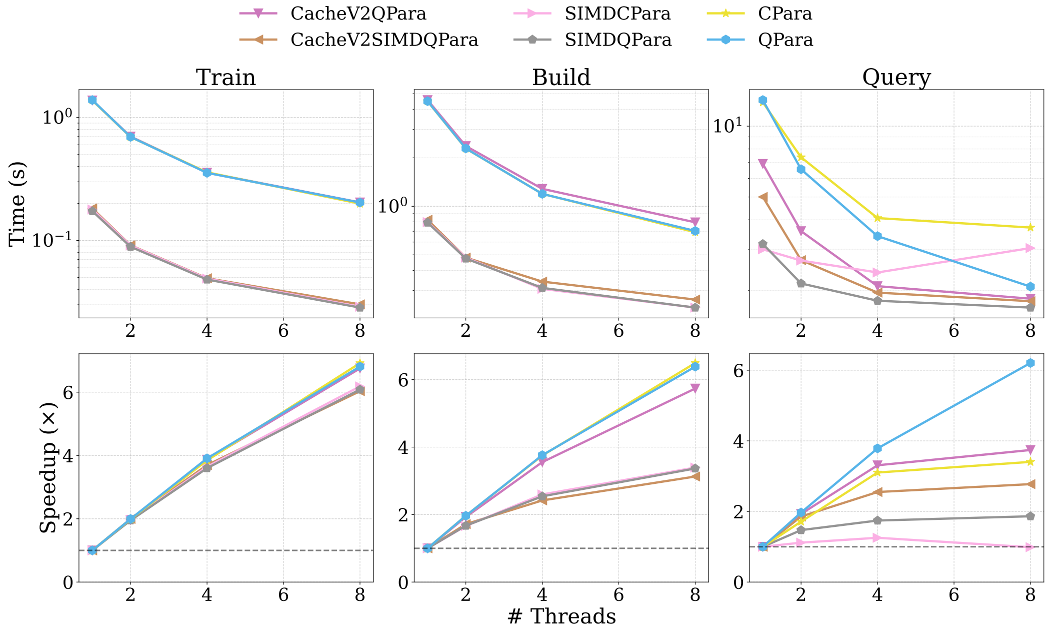
### Easy:



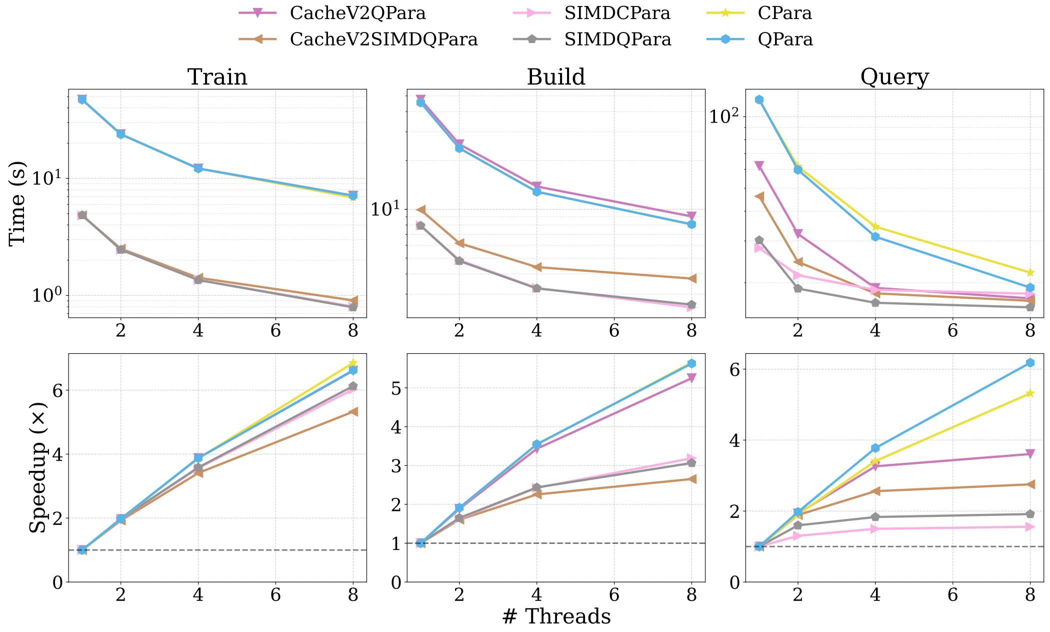
### Medium:



**Hard:**



**Extreme:**

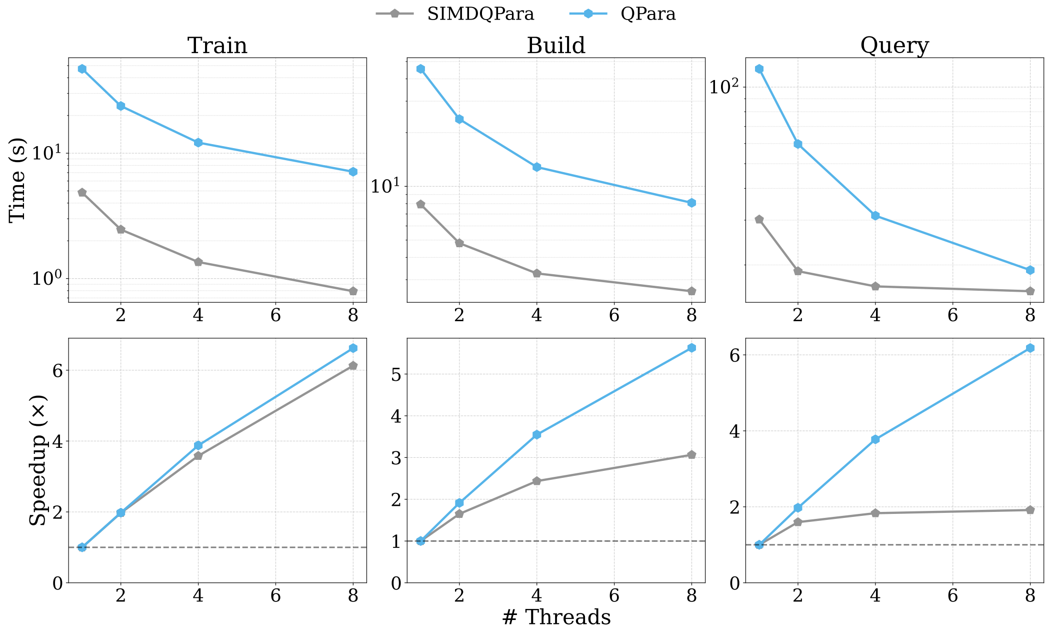


| **Dataset** | **Metric** | **Base Index** | **Best Index** | **# Threads** | **Max Speedup (×)** |
|---|---|---|---|---|---|
| Easy | Train | QPara | SIMDQPara | 8 | 19.85 |
| | Build | CPara | SIMDQPara | 8 | 12.61 |
| | Query | CPara | SIMDQPara | 8 | 18.01 |
| Medium | Train | CPara | SIMDQPara | 8 | 18.28 |
| | Build | CPara | SIMDCPara | 8 | 6.49 |
| | Query | CPara | SIMDQPara | 8 | 14.40 |
| Hard | Train | CPara | SIMDQPara | 8 | 48.42 |
| | Build | CPara | SIMDCPara | 8 | 19.06 |
| | Query | CPara | SIMDQPara | 8 | 7.45 |
| Extreme | Train | CPara | SIMDQPara | 8 | 59.47 |
| | Build | CPara | SIMDCPara | 8 | 18.28 |
| | Query | CPara | SIMDQPara | 8 | 7.44 |

Table 2: Maximum Speedups on GHC Machines

We will first discuss general trends in runtime and speedup by workload, and then discuss which parallelization methods result in the best speedup overall. Overall, when looking at speedups by thread count, we see roughly similar trends across workloads. In general, doubling thread count leads to close to 2× speedup in the training phase, while in the build and query phases we see some speedup by doubling thread counts, but it is more variable depending on the type of parallelization strategy employed. This general trend holds for all except the Easy dataset, where the speedups are much more variable depending on thread count. This is likely because the Easy test is so non-computationally intensive that the additional OpenMP setup overhead outweighs the benefits of parallelizing computations across threads. This variability persists somewhat in the Medium dataset, but largely disappears when we get to the Hard and Extreme datasets, which as discussed above are far more representative of an actual IVF workload.

We will now discuss our best implementation's performance, the factors contributing to it, and the reasons why some of our other implementations fall short of this mark. Overall, our SIMDQPara implementation (the SIMD-ized distance kernel combined with OpenMP based query parallelization) performs the best across all datasets. An 8-threaded version shows approximately 7.5× over a serial IVF implementation on production level datasets. Interestingly, however, the SIMDQPara implementation does not scale well as its serial counterpart, as shown in the graphs below of runs done on the Extreme dataset on the GHC machines.



A query-parallel implementation alone achieves roughly 6× speedup at 8 threads, while SIMDQPara struggles to reach even 2× under the same conditions. After investigating this discrepancy, we are strongly convinced that the bottleneck is memory bandwidth rather than any issue in the compute pipeline. The SIMD-accelerated distance kernel performs extremely well in isolation—even on single-core runs—indicating that the vectorized arithmetic is not the limiting factor. We also avoid divergent behavior inside SIMD lanes by explicitly falling back to a short serial tail for any remainder elements, so lane divergence is not a plausible explanation. Likewise, neither lack of available parallelism nor excessive synchronization appears to be responsible: the pure query-parallel (QPara) implementation scales nicely, and SIMD operations synchronize only within a single core, never across cores.

Taken together, this leads us to conclude that SIMDQPara saturates memory bandwidth well before saturating computational throughput (i.e., it is memory-bound). Supporting this is the fact that the speedup slowdown becomes more pronounced as dataset size grows. Likely what we are seeing is that as each query touches more and more candidates, threads begin to issue memory loads faster than the memory subsystem can serve them, and performance stalls. In this regime, adding SIMD makes the demand for memory higher by making computation more efficient. Meanwhile, QPara alone performs better because each thread still enjoys favorable locality over its assigned candidates, keeping it closer to the cache hierarchy and reducing contention. The combined SIMD + QPara design therefore enters a memory-bandwidth-bound regime where additional computational parallelism provides diminishing returns.

A small experiment on the GHC machines examining the average query latency as a function of the number of threads seems to corroborate our hypothesis. On the Extreme dataset, we observe the following:

| # Threads | Avg. Query Latency (ms) |
|---|---|
| 1 | 117.328 |
| 2 | 147.625 |
| 4 | 255.523 |
| 8 | 490.316 |

Table 3: Extreme SIMDQPara Latency

| # Threads | Avg. Query Latency (ms) |
|---|---|
| 1 | 461.051 |
| 2 | 465.602 |
| 4 | 485.664 |
| 8 | 589.266 |

Table 4: Extreme QPara Latency

For SIMDQPara, by the time we reach 4 threads, we notice that our average query latency starts scaling linearly with the number of threads. This implies that the threads start fighting with each over memory access, and are thus simply slowing each other down. For QPara, however, we do not notice the same query latency degredation. Since each query is inherently slower due to a slack of vectorization, threads spend more time with each byte of data, allowing them to compete less for memory access. To further corroborate our hypothesis, Tables 5 and 6 show us that the same phenomenon does not occur on our Easy dataset, where the working set is far smaller (10000 * 128 * 4B = 4.88 MiB) and fits entirely within L3 cache (12 MiB).

| # Threads | Avg. Query Latency ($\mu$s) |
|---|---|
| 1 | 157.439 |
| 2 | 159.046 |
| 4 | 164.431 |
| 8 | 184.618 |

Table 5: Easy SIMDQPara Latency

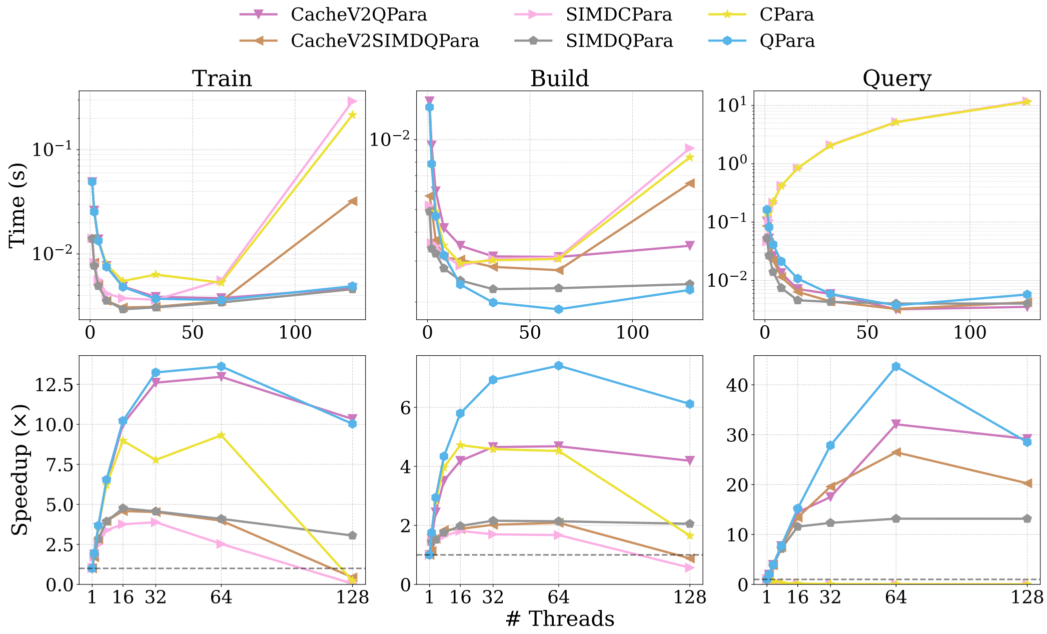
| # Threads | Avg. Query Latency ($\mu$s) |
|---|---|
| 1 | 545.731 |
| 2 | 547.969 |
| 4 | 558.816 |
| 8 | 624.413 |

Table 6: Easy QPara Latency

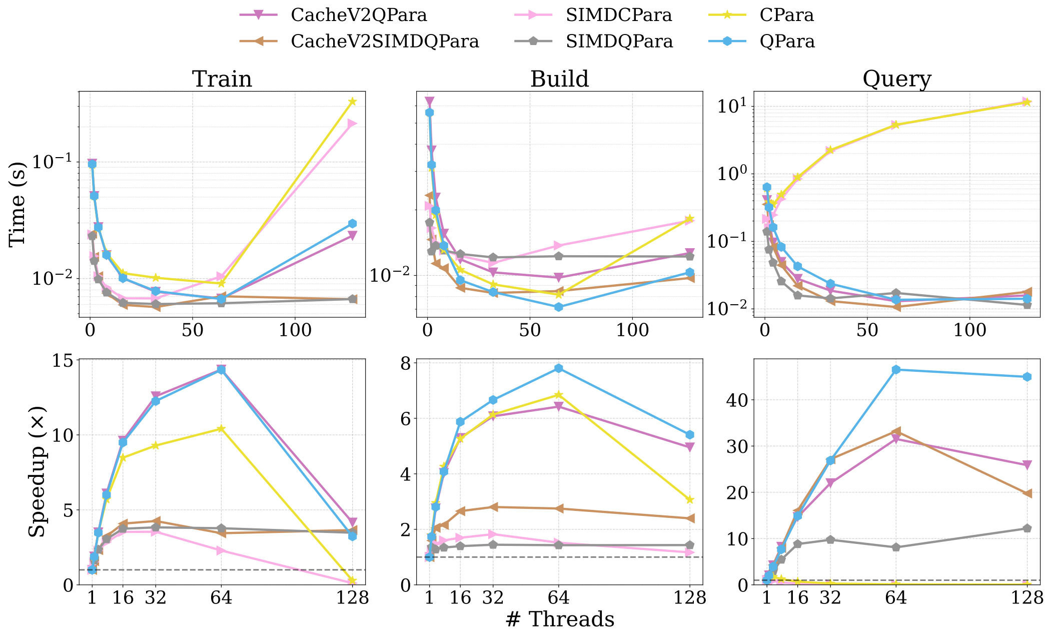
Furthermore, we also notice subpar scaling of the candidate-parallel technique across workloads. We largely expected candidate-parallelism to perform worse than query parallelism due to higher OpenMP overhead. Consider the Hard dataset, for example. We execute 256 queries and for each query, based on $n_{\text{probe}}$ and $n_{\text{list}}$, 50,000 database vectors are compared to the query. With query parallelism, we call `#pragma omp for` once at the beginning to map threads to their corresponding queries. However, with candidate parallelism, we assign each candidate within each of the $n_{\text{probe}}$ (25) explored clusters to a different core when processing a query, for a total of $256 \times 25 = 6400$ instances of thread spawning. The additional OpenMP overhead from this causes our candidate parallelism to perform worse than query parallelism—in certain instances we see 8 threaded candidate parallel instances run longer than 4 threaded instances.

## 4.3 PSC Results

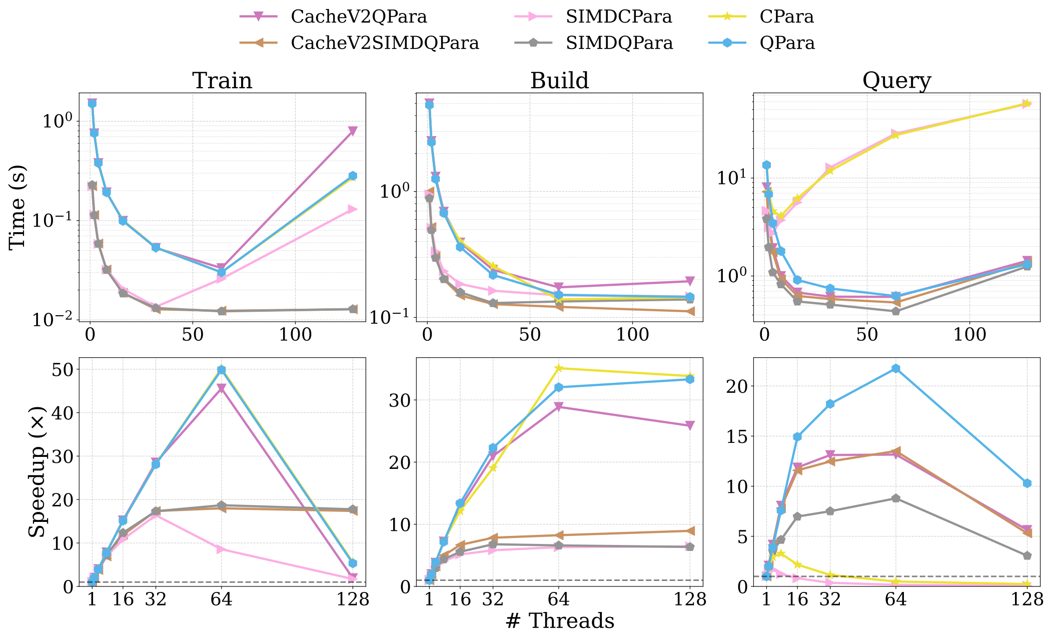
### Easy:



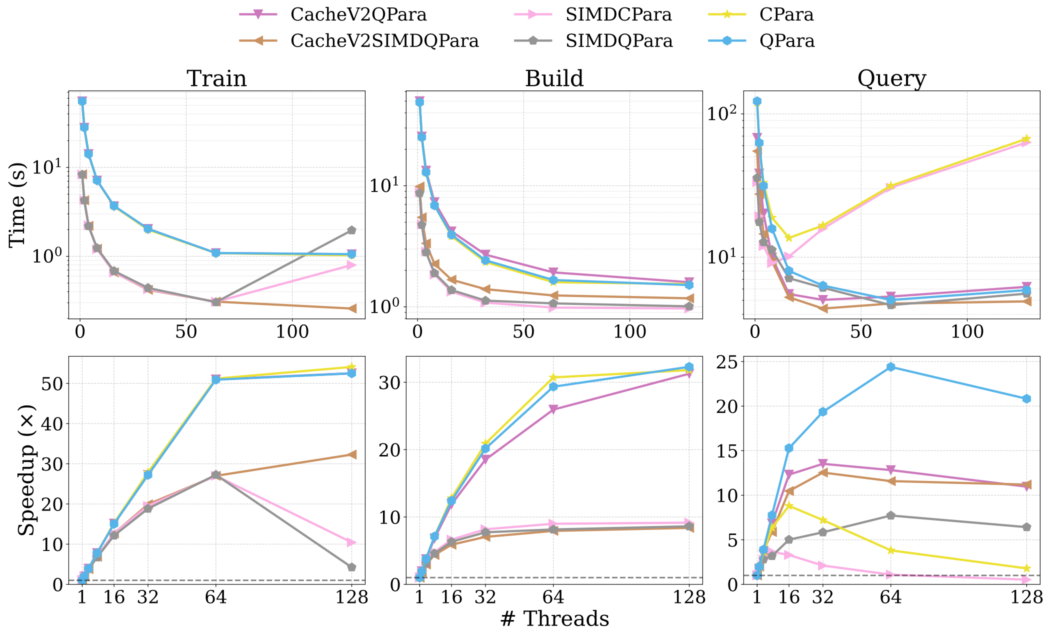
### Medium:



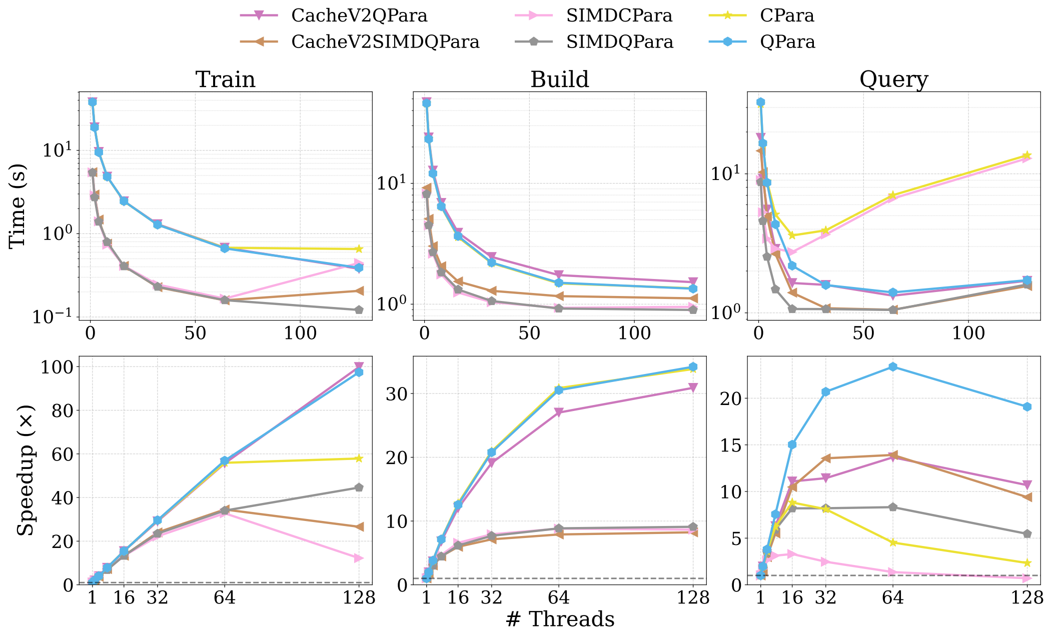
**Hard:**



**Extreme:**



**GIST:**



| Dataset | Metric | Base Index | Best Index | # Threads | Max Speedup (×) |
|---|---|---|---|---|---|
| Easy | Train | QPara | SIMDQPara | 16 | 16.71 |
| | Build | QPara | QPara | 64 | 7.41 |
| | Query | CPara | CacheV2QPara | 64 | 43.80 |
| Medium | Train | CPara | CacheV2SIMDQPara | 32 | 16.55 |
| | Build | CPara | QPara | 64 | 7.80 |
| | Query | CPara | CacheV2SIMDQPara | 64 | 57.48 |
| Hard | Train | QPara | SIMDQPara | 64 | 123.02 |
| | Build | QPara | CacheV2SIMDQPara | 128 | 43.33 |
| | Query | CPara | SIMDQPara | 64 | 30.95 |
| Extreme | Train | QPara | CacheV2SIMDQPara | 128 | 214.53 |
| | Build | QPara | SIMDCPara | 128 | 50.45 |
| | Query | CPara | CacheV2SIMDQPara | 32 | 27.39 |
| GIST | Train | CPara | SIMDQPara | 128 | 311.69 |
| | Build | CPara | SIMDQPara | 128 | 51.16 |
| | Query | CPara | SIMDQPara | 64 | 30.29 |

Table 7: Maximum Speedups on PSC Machines

As with our discussion of results on the GHC machines, we will first discuss general trends in runtime and speedup by workload, and then discuss which parallelization methods result in the best speedup overall. Like with the GHC machines, with the exception of the "Easy" dataset we see similar trends per workload, although the specific trends themselves differ. In general, doubling thread count leads to speedups up to a certain point, at which for many parallelization methods the OpenMP overhead begins to dominate runtime and decreases total speedup. For the medium, hard, extreme, and GIST datasets, most parallelization methods see the most benefit at around 64 threads-at 128 threads, we begin to see slowdowns on all datasets due to overhead. Interestingly, on these 4 datasets we see speedups for the candidate parallelization methods peak at 16 threads during the search phase, whereas for all other methods query speedup peaks at 64 threads. As with the GHC machines, speedups in the easy dataset vary widely due to high overhead compared to computational load.

We see similar relative performance and trends between methods on the PSC machines in comparison to the GHC machines. First of all, the SIMDQPara in general shows the best performance compared to the serial version, as can be seen in Figure 1:

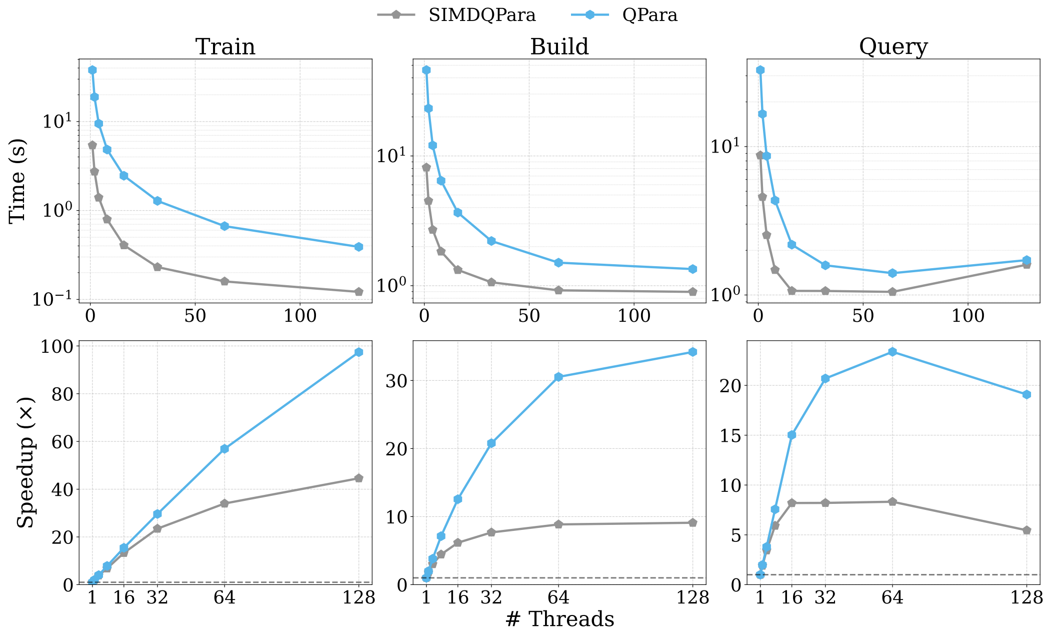


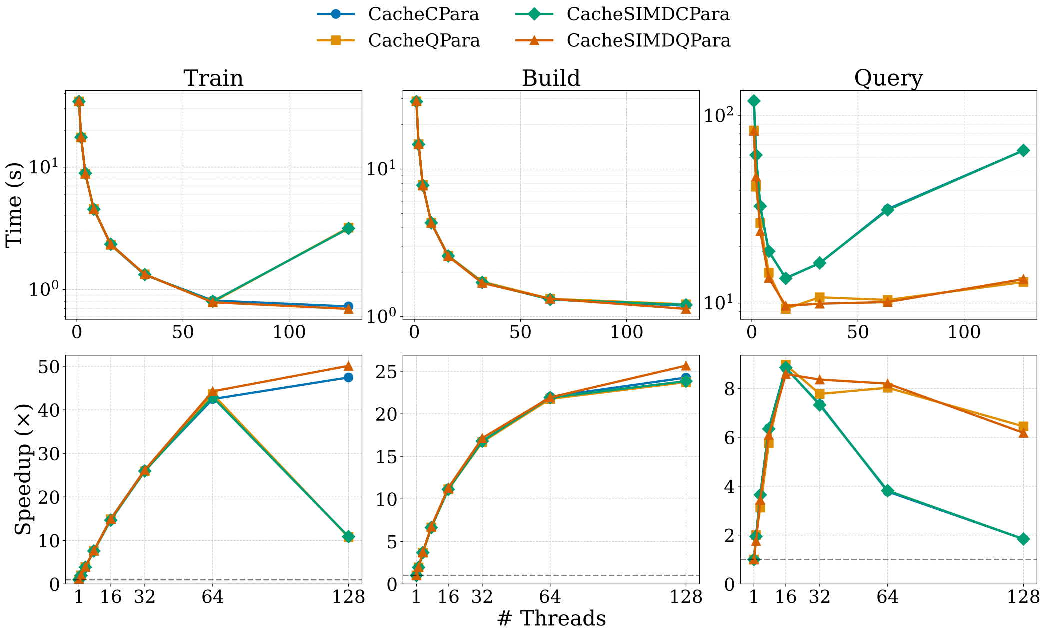
Figure 1: SIMDQPara on PSC

We see that similar to the GHC machine, speedup with this method is not linear with respect to the number of threads, likely due to becoming memory bound at higher thread counts as discussed above. Similarly, we see that candidate parallelization performs worse than query parallelization, again likely due to high OpenMP overhead. Lastly, it is interesting to note that for the easy and medium datasets, speedups and runtimes vs thread count plots are far more variable on the PSC machines than on the GHC machines. This is likely due to the ballooning cost of OpenMP synchronization/parallelism as we expand past the 8 thread maximum on the GHC machines. However, overall, as with the GHC machines the SIMD distance and query parallelization combination works the best overall in reducing runtime, and trends are similar across workloads and parallelization types.

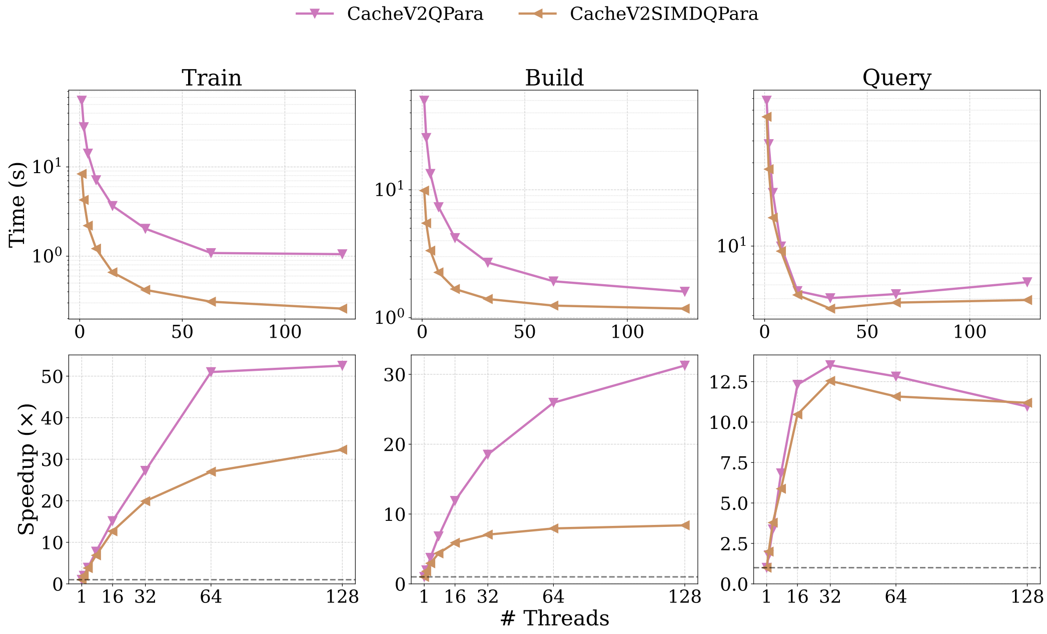
As a note, all of the previous experiments done on the GHC and Bridges machines were done on CPUs. In theory, we could have explored a GPU-based implementation of inverted vector file indexes-some aspects of the workflows, such as comparing queries to different candidates, could have benefited from the embarrassingly parallel structure of GPU's. However, we decided to focus on OpenMP and SIMD based parallelization as we deemed these parallelization methods both more interesting, more challenging to implement, and more generally applicable across a wide range of devices. GPU parallelization of IVF's, however, would make for a very interesting future project.

## 4.4 Cache Friendliness

As discussed in Section 3.2, we explored two main methods to enable a more cache-friendly distance computation. The first method, titled "Cache", did not meet our expectations, as can be seen in the following results from the PSC machines:



The results on train and build are decent (except for the SIMD-ized versions plummeting on train at 128 threads. The query speedups, however, are the most interesting. Thread-scaling appears to be normal until 16 threads, after which it absolutely plummets (especially for the candidate-parallel implementations). The results for CacheV2 are below:



While these plots show the expected trends, the actual cache miss results are very interesting. On a new dataset with 1000 points of 2048 dimensions each, we get the following results:

| **Index Name** | **L1 Miss** ($10^9$) | **L2 Miss** ($10^9$) | **L3 Miss** ($10^6$) | **Total Time (s)** |
|---|---|---|---|---|
| QPara | 8.26 | 11.00 | 8.84 | 111.004 |
| CacheQPara | 19.94 | 32.13 | 27.92 | 46.486 |
| CacheV2QPara | 8.32 | 11.07 | 8.99 | 53.316 |

Table 8: Cache Miss and Time Elapsed Results (1 Thread)

These results were very surprising and went against our initial expectations. It is possible that the staggered nature of the distance computation (where it only operates on a 16-element slices of the query-candidate pair and thus prohibits the compiler from possibly unrolling the distance computation loop) outweighed the locality benefits gained—it is also possible the original data structure was more amenable to a SIMD-ized distance calculation. Of note, tests on the GHC machines, PSC machines, and on our local machines all yielded different results, further compounding the intractability of this problem. Ultimately, because our SIMDQPara implementation achieved better speedups than any of these, we decided not to pursue this further, but it would be an interesting avenue to explore for a potential project extension.

# 5 Work Distribution

Both members of the group contributed equally to the success of this project. On various occasions we met up together in person to brainstorm solutions and pair-program. As such, the total credit for this project should be attributed equally (50%-50%) to the both of us. The work performed by each member can be summarized as follows:

**Drew Byrapatna:**

1. Initial implementation of IVF build
2. Initial implementation of IVF search
3. SIMD distance kernel
4. Cache-friendly distance computation
5. Test suite (for correctness)
6. Benchmarking suite

**Akash Nayar:**

1. Initial $k$-means implementation (IVF train)
2. Query and candidate parallelization
3. SIMD distance kernel
4. Cache-friendly distance computation
5. Running experiments
6. Visualization scripts

# 6 References

1. Faiss (Douze et al. 2025)
2. Inverted Files (Bruch et al. 2024)
3. $k$-means (Hamerly and Elkan 2002)

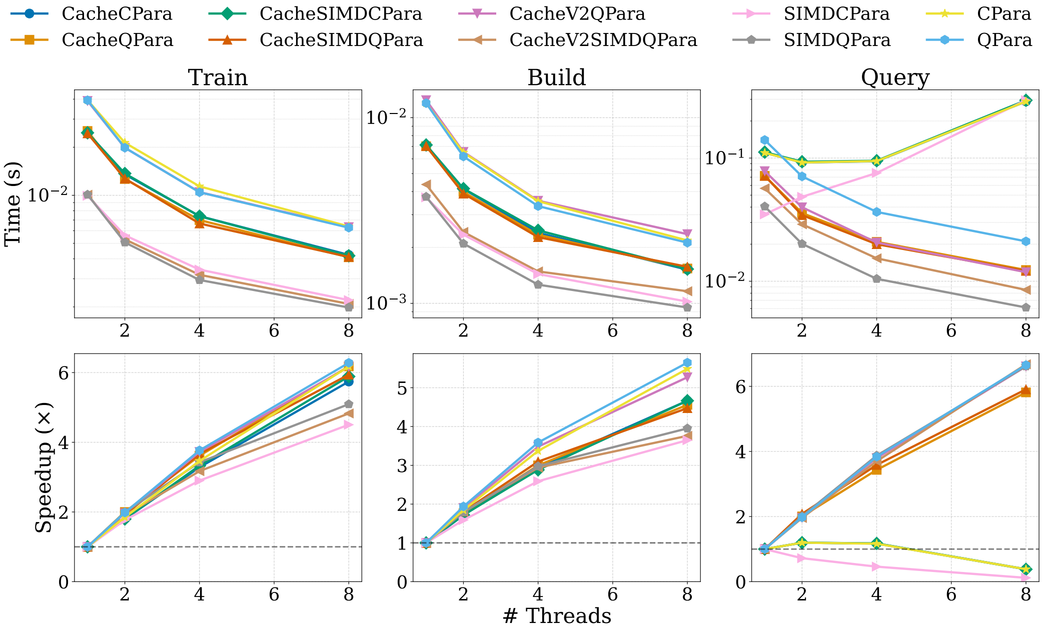
# Appendix

## A All Results

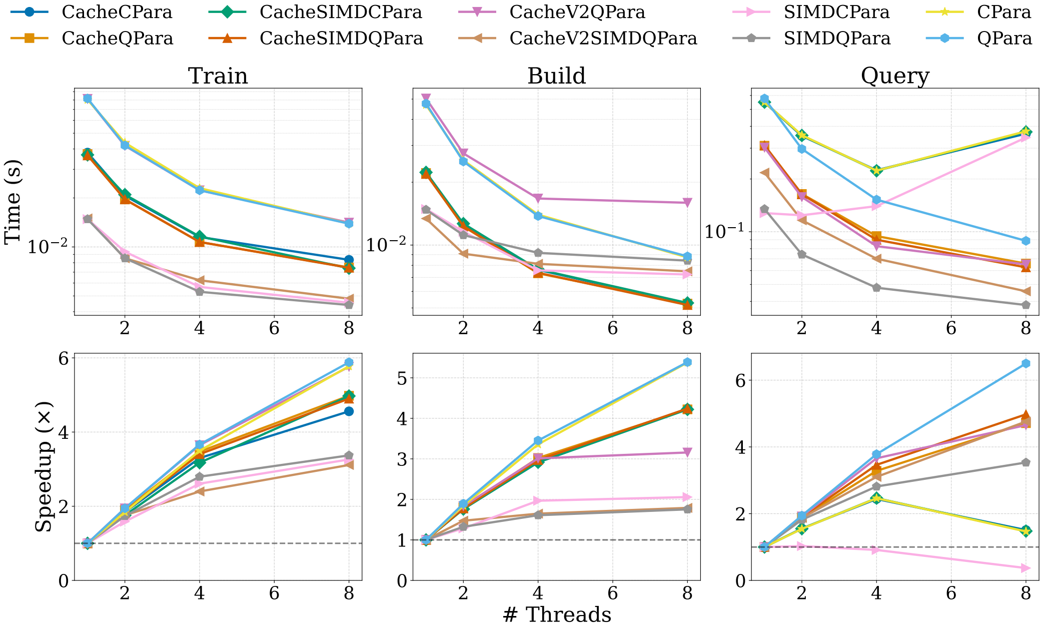
### A.1 All GHC Results

All raw GHC data can be found on our GitHub.

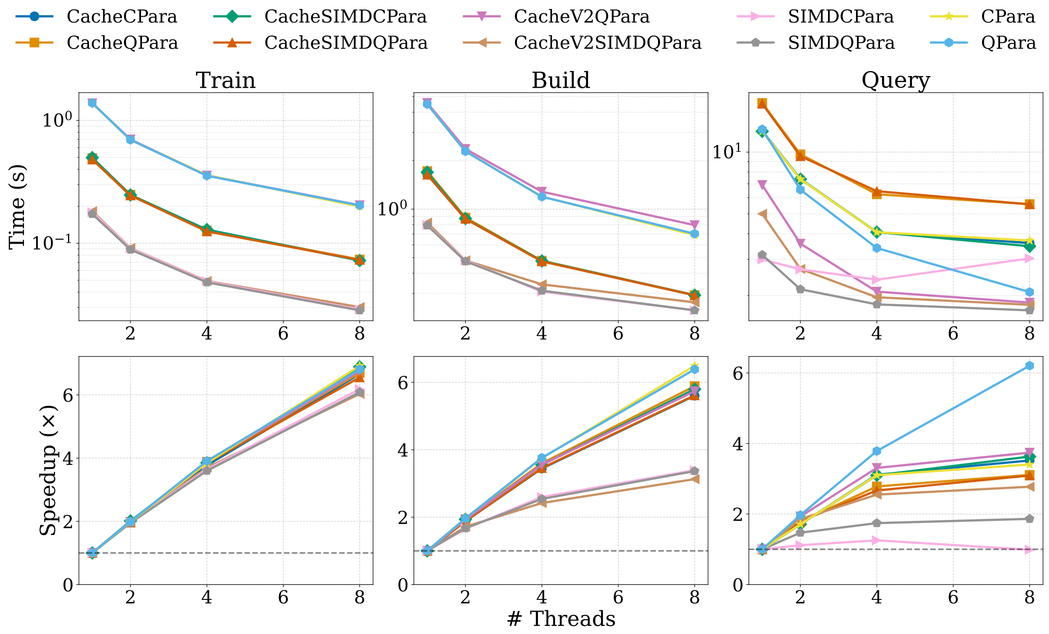
**Easy:**



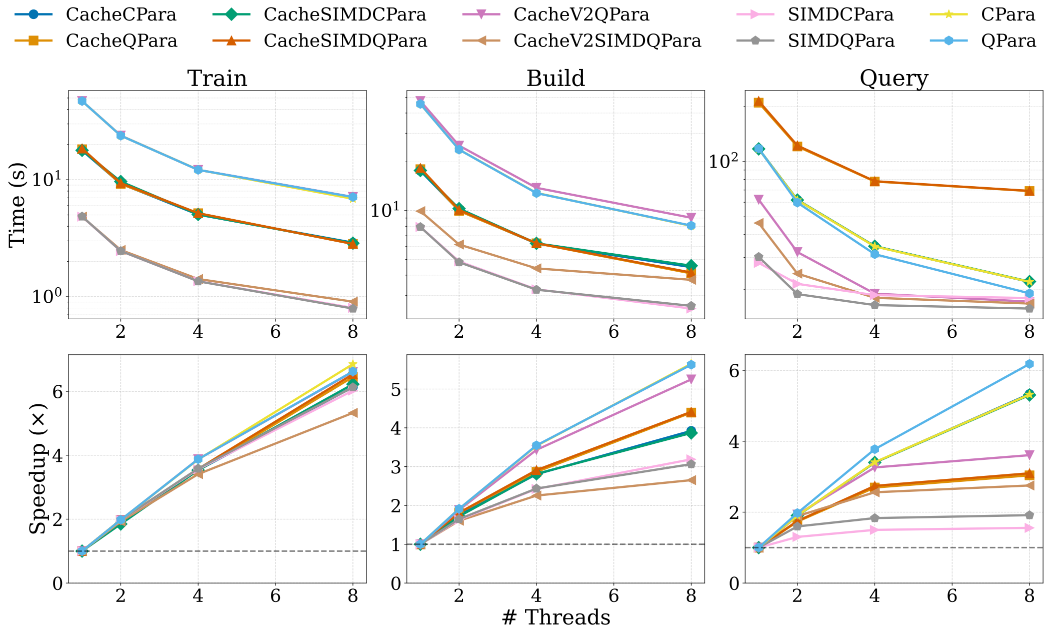
**Medium:**



**Hard:**



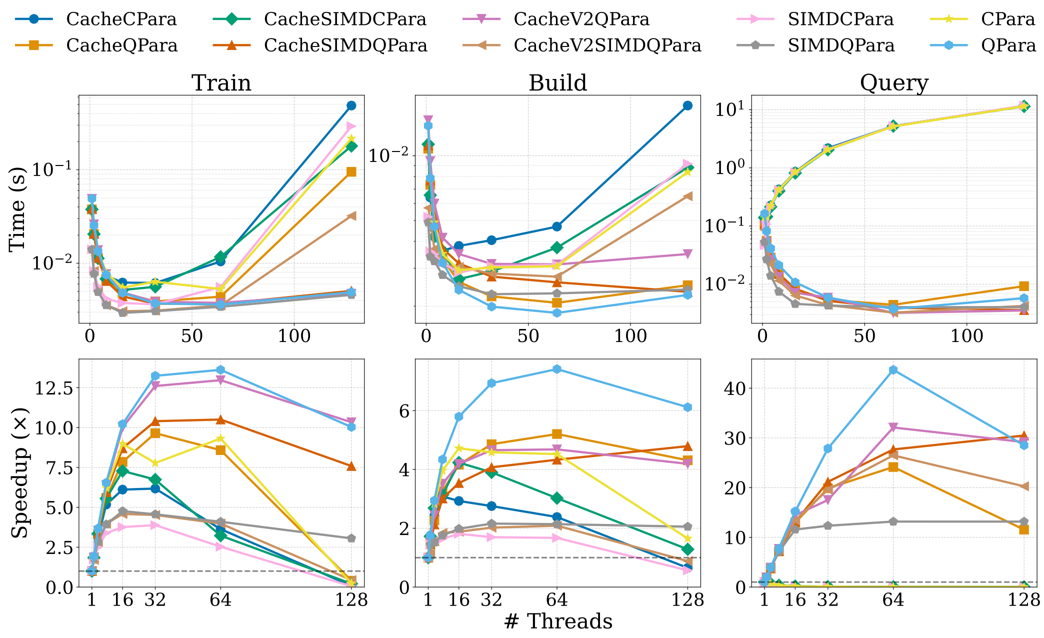
**Extreme:**



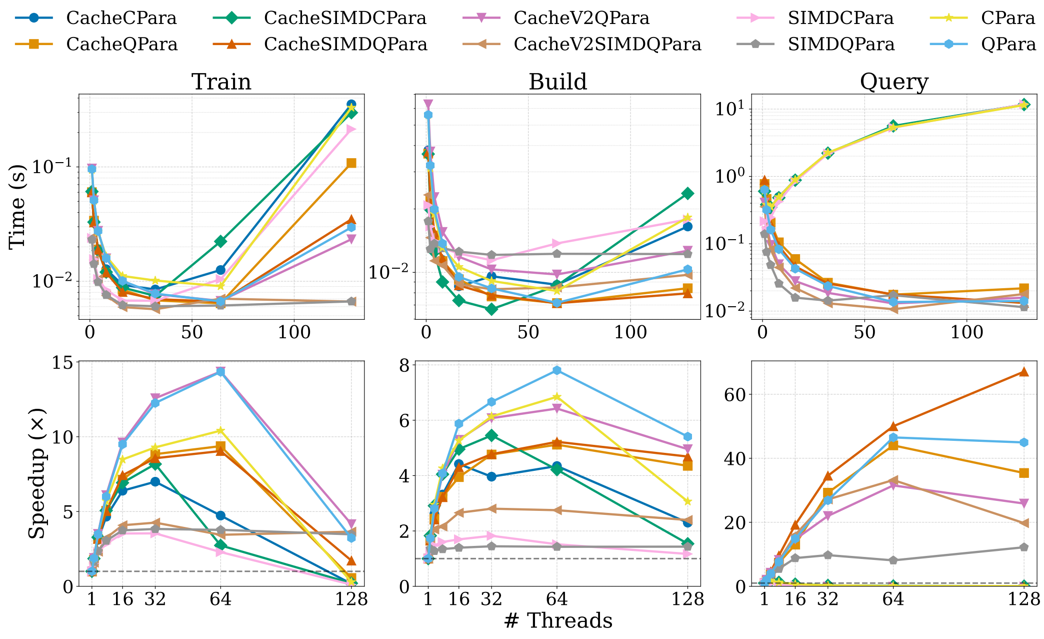
## A.2 All PSC Results

All raw PSC data can be found on our GitHub.

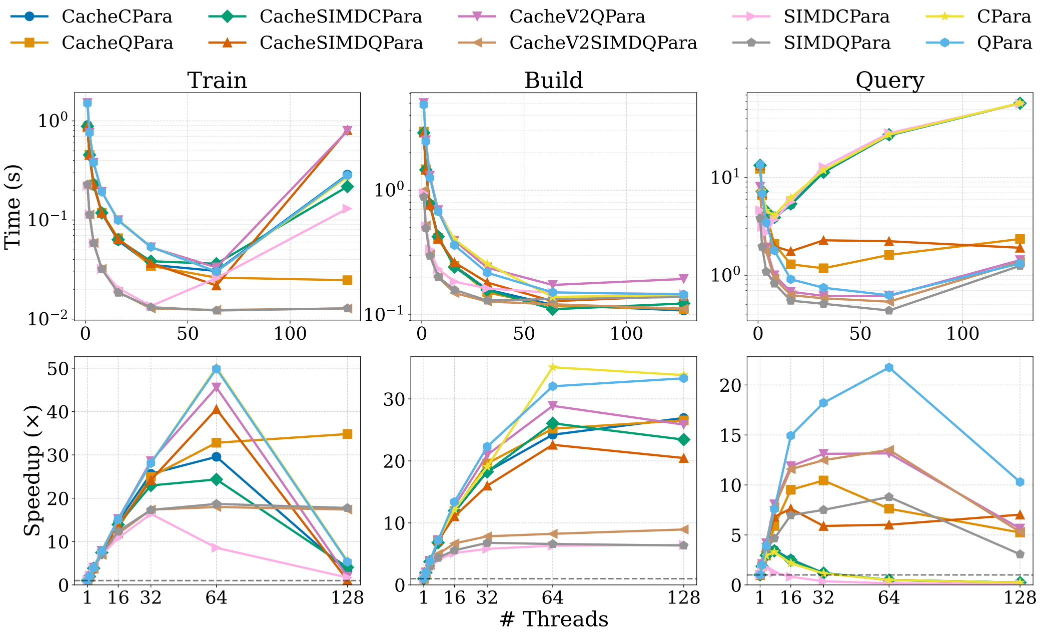
**Easy:**



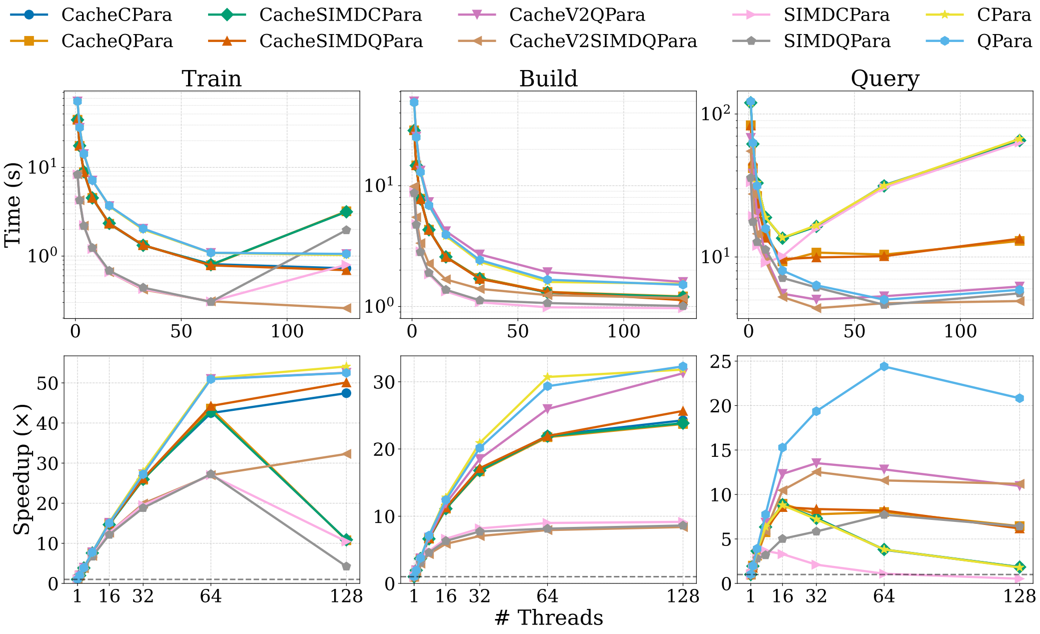
**Medium:**



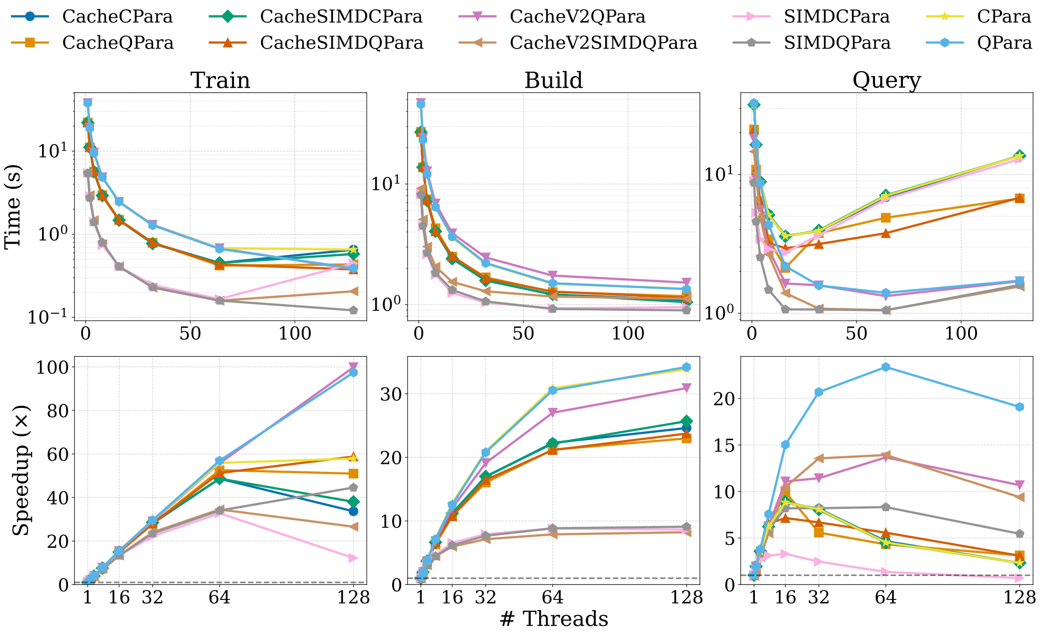
**Hard:**



**Extreme:**



**GIST:**


CacheCPara
CacheQPara
CacheSIMDCPara
CacheSIMDQPara
CacheV2QPara
CacheV2SIMDQPara
SIMDCPara
SIMDQPara
CPara
QPara
Train
Build
Query
Time (s)
$10^{1}$
$10^{0}$
$10^{-1}$
0
50
100
Speedup (×)
100
80
60
40
20
0
30
20
10
15
5
1
16
32
64
128
# Threads