# Bang for the Buck: Vector Search on Cloud CPUs

Leonardo Kuffo
Centrum Wiskunde & Informatica
Amsterdam, The Netherlands
lxkr@cwi.nl

Peter Boncz
Centrum Wiskunde & Informatica
Amsterdam, The Netherlands
boncz@cwi.nl

## Abstract

Vector databases have emerged as a new type of systems that support efficient querying of high-dimensional vectors. Many of these offer their database as a service in the cloud. However, the variety of available CPUs and the lack of vector search benchmarks across CPUs make it difficult for users to choose one. In this study, we show that CPU microarchitectures available in the cloud perform significantly differently across vector search scenarios. For instance, in an IVF index on `float32` vectors, AMD's Zen4 gives almost 3x more queries per second (QPS) compared to Intel's Sapphire Rapids, but for HNSW indexes, the tables turn. However, when looking at the number of *queries per dollar (QP$)*, Graviton3 is the best option for most indexes and quantization settings, even over Graviton4 (Table 1). With this work, we hope to guide users in getting the best "bang for the buck" when deploying vector search systems.

## CCS Concepts

• **Information systems** → **Retrieval efficiency**.

## Keywords

vector similarity search, vector databases, cloud computing, cost-performance analysis, CPU microarchitectures

**ACM Reference Format:**
Leonardo Kuffo and Peter Boncz. 2025. Bang for the Buck: Vector Search on Cloud CPUs. In *21st International Workshop on Data Management on New Hardware (DaMoN '25), June 22–27, 2025, Berlin, Germany*. ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3736227.3736233

## 1 Introduction

The increasing amount of data in the shape of high-dimensional vectors (e.g., embeddings from large language models) has resulted in the surge of a new type of vector-centric systems known as Vector Databases (VecDBs) [19,29,30]. VecDBs like Weaviate, Milvus, Qdrant, and ChromaDB offer storage, management, and efficient querying of high-dimensional vectors [40,44]. The demand of users for vector systems is such that existing database systems have added vector capabilities natively (MongoDB, Redis) or via extensions (e.g., pgvector in PostgreSQL, DuckDB-VSS in DuckDB).

A core feature of VecDBs is the efficient querying of vectors through Vector Similarity Search (VSS). VSS consists of finding the vectors within a collection that are most similar to a query vector based on a distance or similarity metric. The latter is a core component of various applications, such as RAG pipelines.

**Table 1: Relative comparison of queries-per-second (QPS) and queries-per-dollar (QP$) given by AWS cloud instances on the OpenAI/1536 dataset on various vector search scenarios. ++ indicates the best, + is <20% away from the best, · is >20% but <2x from the best, lastly - and -- are >2x and >3x away from the best, resp. Each (+) is 1 and each (-) is -1 score.**

| | Microarch. | Indexes | | | | | | | | | Full Scans | | | | | | | | | Score |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | IVF | | | HNSW | | | | | | FAISS | | | | USearch | | | | | |
| | | f32 | SQ | BQ | f32 | f16 | bf | SQ | BQ | | f32 | SQ | BQ | PQ | f32 | f16 | bf | SQ | BQ | |
| Queries-per-second | Graviton3 r7g | · | -- | + | + | · | + | + | - | | · | -- | + | · | · | ++ | -- | + | + | 2 |
| | Zen3 r6a | · | · | · | + | · | + | · | - | | · | · | · | · | · | · | - | · | · | 0 |
| | Graviton4 r8g | · | -- | ++ | + | · | + | · | - | | · | -- | ++ | · | · | + | - | + | + | 3 |
| | SPR r7i | - | · | · | · | · | + | + | + | | - | · | · | + | - | · | -- | - | · | -2 |
| | Zen4 r7a | ++ | ++ | · | · | - | + | · | · | | ++ | ++ | + | · | ++ | · | ++ | ++ | ++ | **17** |
| | SPR Z r7iz | - | · | + | ++ | ++ | ++ | ++ | ++ | | - | · | · | ++ | - | · | - | - | + | 9 |
| Queries-per-dollar | Graviton3 r7g | ++ | -- | ++ | ++ | ++ | ++ | ++ | · | | + | -- | ++ | · | ++ | ++ | - | ++ | + | **17** |
| | Zen3 r6a | · | · | · | + | · | + | · | - | | + | ++ | · | · | + | · | · | · | · | 5 |
| | Graviton4 r8g | · | -- | + | + | + | ++ | + | · | | + | - | + | · | + | + | · | + | ++ | 10 |
| | SPR r7i | - | · | · | · | + | + | · | ++ | | - | · | - | ++ | - | - | - | - | · | -1 |
| | Zen4 r7a | ++ | ++ | · | · | - | · | · | · | | ++ | ++ | · | · | ++ | - | ++ | · | · | 10 |
| | SPR Z r7iz | - | - | · | · | + | · | · | + | | -- | · | - | · | - | - | -- | -- | · | -9 |

However, on a large scale, VSS poses challenges due to the large number of computations and storage needed to obtain exact answers to a query. To overcome this, applications gave up exactness as generally *approximate* answers are "good enough." The latter opened opportunities to accelerate VSS by using *approximate indexes* [16,18,25,27,37], *quantization* techniques that reduce the size of the vectors [3,13,15,20], and optimizations to the distance evaluation [14,21,45,48].

A typical business model of VecDBs is to offer their software as a service (SaaS), either in a cloud environment owned by them or by the users ("Bring Your Own Cloud"). This situation prompts the question: *Which cloud instance is the best for vector search?* While most VecDBs can be deployed on both major architectures (x86_64 and ARM), there is a lack of benchmarks comparing the performance of vector search across different architectures, let alone microarchitectures (e.g., Intel Sapphire Rapids, AMD Zens, AWS Gravitons). Qdrant, in particular, strongly advises using the latest-generation Intel processors [10]. Milvus, Chroma, and Vexless have presented benchmarks of their systems on Intel CPUs [9,26,36], while USearch [39] presents benchmarks on AWS Graviton3 (ARM).

However, far from being as trivial as choosing the CPUs with the largest caches, highest clock frequency, or specialized SIMD instructions, we uncover that the optimal choice depends on the search algorithm and quantization level used. For instance, in partition-based indexes, like IVF [18], AMD's Zen4 gives almost 3x more queries per second (QPS) than Intel's Sapphire Rapids, but the tables turn on graph indexes, like HNSW [25], in which Intel Sapphire Rapids delivers more QPS. However, when looking at the number of queries per dollar (QP$), AWS Graviton3 gives the best bang for the buck, even over its successor, Graviton4.


*DaMoN '25, Berlin, Germany*

ACM ISBN 979-8-4007-1940-0/2025/06
https://doi.org/10.1145/3736227.3736233

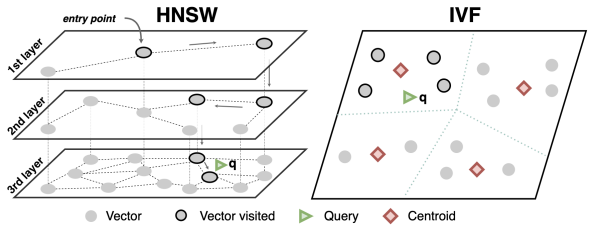



**Figure 1: Example of an HNSW and IVF index search.**

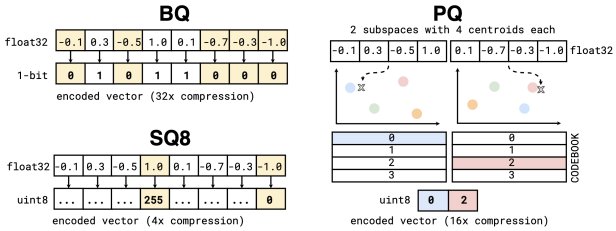



**Figure 2: Overview of different quantization techniques.**

This study aims to show which cloud CPUs give the best "bang for the buck" by experimentally evaluating their QPS and QP$ on different vector search scenarios. More importantly, we uncover that the performance across microarchitectures depends not solely on their SIMD capabilities but also on the search algorithm's data-access patterns. The latter makes the CPU cache performance (bandwidth and latency) important for vector search, as it is memory-bound in many scenarios [21,31,34,41].

## 2 Preliminaries

### 2.1 Approximate Vector Similarity Search

Given a collection $V$ of $n$ multi-dimensional objects $\{v_0, v_1, \cdots, v_{n-1}\}$, defined on a $D$-dimensional space, and a $D$-dimensional query $q$, VSS tries to find the subset $R \subset V$, containing the $k$ most *similar* vectors to $q$. The notion of similarity between two vectors $(v, q)$ is measured using a function $\delta(v, q)$. Usually, $\delta$ is a distance or similarity function defined in an Euclidean space. The Squared Euclidean Distance (L2) is one of the most commonly used distance metrics, and it is defined as $\delta(v, q) = \sum_{i=0}^{D}(v_i - q_i)^2$. To obtain $R$, $\delta$ must be computed for every $v \in V$, leading to a large number of calculations. However, in most vector-based applications, *approximate* answers are acceptable. This allowed VSS to scale by returning only an approximate result set $\hat{R}$, whose quality is measured by the percentage of intersection between the vectors in $R$ and $\hat{R}$ when answering the same query (*recall* metric). This tradeoff between accuracy and speed resulted in the development of approximate indexes and quantization techniques for vectors.

### 2.2 Approximate Indexes

Approximate indexes aim to build data structures that guide the search to the most suitable place of the D-dimensional space in which the query *may* find its nearest neighbours. Indexes can be categorized into three types: graph-based [25,27,32,42], partition-based [17,18,35], and hybrids [8,16]. Their common goal is only to evaluate the distance/similarity function between $q$ and a smaller set of vectors $V' \subset V$ while maintaining high recalls. The indexes that have seen the most adoption in vector systems are HNSW (Hierarchical Navigable Small Worlds) [25] and IVF (Inverted Files) [18].

**HNSW** has seen great success in achieving desirable recall in most datasets [7,49]. HNSW organizes objects into a graph where nodes represent the vectors, and edges reflect their similarity. The property of navigability and "small world" [43] is forced on the graph so that a greedy search can reach the answers to a query in logarithmic time [42]. Borrowing ideas from the skiplist data structure, HNSW organizes the nodes into different layers (see left of Figure 1). The top layer (starting point) contains "distant" nodes, and the bottom layer has all the nodes. The upper layer allows a search to quickly traverse the graph diameter with a few steps, and the lower layer allows it to traverse on a local hub of nodes.

On the other hand, **IVF** is a partition-based index that clusters the vector collection into buckets. At search time, the distance metric is first evaluated with the centroids of each bucket, and the vectors inside the nearest centroids buckets are chosen for evaluation (see right of Figure 1). The number of centroids is set in the order of $\sqrt{n}$ [11,40]. More buckets can be probed to trade off speed for more recall [11,40]. IVF works modestly well in most datasets [7,49] while scaling better than graph indexes, which have higher memory requirements and longer construction times [11,31]. A commonly used *hybrid index* consists of building an HNSW index on the IVF centroids to quickly find the most promising buckets [8].

### 2.3 Quantization Techniques

Quantization techniques aim to reduce the size of every vector. Quantization can be used on top of indexes to alleviate storage requirements. However, the more aggressive quantization, the more the recall is reduced as precision is lost. Quantization can increase query throughput, as less data must be fetched [34], and *SIMD* can operate on more values at-a-time with a single CPU instruction.

Among the most popular quantization techniques, we have binary (BQ), scalar (SQ), and product quantization (PQ) [18]. Examples of these techniques are shown in Figure 2. BQ maps every `float32` value in the vector to 0 or 1 (32x compression). In Figure 2, positive values are mapped to 1 and negative values to 0. In SQ, each `float32` value is mapped to an integer code of usually 8, 6, or 4-bit, depending on the target compression ratio. In Figure 2, each value is mapped to 8 bits by normalizing and scaling them to the [0-255] range. However, a wider family of SQ algorithms exists [3,13]. Finally, *downcasting* to `float16` or `bfloat` is a safer alternative with little information loss but only achieving a 2x compression ratio.

In PQ, the D-dimensional space is split into $M$, $\frac{D}{M}$ dimensional subspaces, and a codebook is trained for each subspace using a clustering algorithm. In Figure 2, we split vectors into two subspaces of $\frac{8}{2} = 4$ dimensions and create 4 clusters for each subspace. To encode a vector, PQ splits it into $M$ subvectors. For each subvector, PQ finds the nearest centroid on that subspace codebook, and the vector is encoded with the centroid codes. Contrary to BQ and SQ, PQ can provide variable compression ratios depending on the chosen number of subspaces and centroids per subspace. However, contrary to BQ and SQ, in PQ, the distance calculations cannot happen in the quantized domain, as each code must be first decoded. The latter affects the distance calculation latency. A wider family of PQ algorithms exists, mostly improving how the codebook is constructed to tradeoff search speed, size, and recall [11,15].

**Table 2: AWS cloud instances used (May 2025, us-east-1)**

| Microarchitecture | CPU Model | Instance Name | Price/h $USD ↓ | Freq. GHz | L1 KiB | L2 MiB | L3 MiB | SIMD ISA | SIMD Width | SIMD Capabilities | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | | | f32 | f16 | bf | i32,16,8 | popcnt |
| AWS Graviton 3 | Neoverse V1 | r7g.2x | 0.4284 | 2.6 | 64 | 1 | 32 | NEON / SVE | 128 / 256 | ✓ | ✓ | ✓ | ✓ | ✓ |
| AMD Zen 3 | EPYC 7R13 | r6a.2x | 0.4536 | 3.6 | 32 | 0.5 | 16 | AVX2 | 256 | ✓ | ✗ | ✗ | ✓ | ✗ |
| AWS Graviton 4 | Neoverse V2 | r8g.2x | 0.4713 | 2.8 | 64 | 2 | 36 | NEON / SVE2 | 128 / 128 | ✓ | ✓ | ✓ | ✓ | ✓ |
| Intel Sapphire Rapids | Platinum 8488C | r7i.2x | 0.5292 | 3.2 | 48 | 2 | 105 | AVX512 | 512 | ✓ | ✓ | ✓ | ✓ | ✓ |
| AMD Zen 4 | EPYC 9R14 | r7a.2x | 0.6086 | 3.7 | 32 | 1 | 32 | AVX512 | 512 | ✓ | ✗ | ✓ | ✓ | ✓ |
| Intel Sapphire Rapids Z | Gold 6455B | r7iz.2x | 0.7440 | 3.9 | 48 | 2 | 60 | AVX512 | 512 | ✓ | ✓ | ✓ | ✓ | ✓ |

## 2.4 Single Instruction Multiple Data (SIMD)

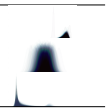
Distance calculations can be optimized in CPUs using SIMD intrinsics that process multiple values with a single CPU instruction.

**SIMD in x86_64 and ARM.** In x86_64 architectures, SIMD instructions are called Advanced Vector Extensions (AVX). The number of values AVX can process at a time depends on the SIMD register width supported by the CPU. Initially, registers of 256-bit were introduced (AVX and AVX2), further expanded to 512-bit (AVX512), which can process 16 `float32` values with one instruction. AVX512 had a rough start, reportedly down-clocking the CPU in earlier Intel microarchitectures (Sky Lake and earlier) [24]. However, this is no longer an issue in modern CPUs (Zen4, Sapphire Rapids), and AVX512 is widely used to accelerate vector search. On the other hand, ARM architectures also provide SIMD instructions through NEON and SVE. NEON was introduced first, supporting SIMD over 128-bit registers. SVE was introduced later as an improvement over NEON that, unlike traditional SIMD architectures, supports variable-size SIMD registers on its intrinsics through VLA (Variable Length Agnostic) programming. The latter alleviates technical debt as distance kernels no longer need hardware-dependent loop lengths. Graviton4 has 128-bit SVE registers, and Graviton3 has 256-bit SVE registers, with both having 128-bit NEON registers.

**SIMD capabilities.** SIMD instructions usually support floating-point arithmetic (double and single precision), integer arithmetic (64, 32, 16, and 8-bit), data movement, conversions, and comparisons. In addition to this, CPU vendors have added extended SIMD capabilities in different microarchitectures. For instance, Intel's Sapphire Rapids (SPR) supports arithmetic of `float16`, and both AMD's Zen4 and Intel's SPR support arithmetic of `bfloat` and a `POPCOUNT` instruction useful for 1-bit vectors. In modern ARM CPUs, SVE and NEON support arithmetic for `bfloat`, half-precision `float16`, and 1-bit vectors (`POPCOUNT`). Graviton4 supports SVE2, an extension to SVE that introduces additional intrinsics such as `MATCH` to compute the intersection between two vectors.

Table 2 shows some of the SIMD capabilities relevant to distance calculations present in modern CPUs available in the cloud. It is important to note that, in some scenarios, SIMD instructions are not directly usable in distance kernels. One example is trying to do a dot product between two 8-bit vectors in AVX2/AVX512, as the only available instruction to do so (i.e., `VPDPBUSD`) expects one vector to be `signed` and the other to be `unsigned`. Challenges also arise with sub-byte kernels. For instance, vectors quantized to 6 bits must first be *aligned* into the SIMD registers, usually with `SHIFT+OR` instructions, which impact the performance of the distance kernels.

**Table 3: Vector datasets**

| Dataset | Semantics | Size | N. Queries | Dim.↑ | Distribution |
|---|---|---|---|---|---|
| OpenAI | Text Embeddings | 999,000 | 1,000 | 1536 |  |
| arXiv | Text Embeddings | 2,253,000 | 1,000 | 768 | |
| SIFT | Image Features | 1,000,000 | 10,000 | 128 | |

**SIMD in vector libraries.** Some vector libraries, like USearch [39], leverage SIMD capabilities to provide distance kernels for quantized types in most major CPU microarchitectures. On the other hand, other libraries, like FAISS, avoid specialized kernels by decoding vectors back to the `float32` domain. The latter is called asymmetric distance calculations. This reduces technical debt and avoids complex code bases. However, the performance of such kernels is not on par with symmetric kernels that operate in the quantized domain [38]. Both USearch and FAISS prefer to use the latest SIMD ISA available in the CPU (e.g., SVE over NEON if both are available).

**x86_64 vs ARM SIMD.** A key difference between the SIMD of microarchitectures lies in their register width. However, a larger register width does not guarantee better raw performance. For instance, CPUs with NEON SIMD do not fall behind AVX512 on database workloads [2] despite having a 4x smaller register width. This is because the latency and execution throughput of the instructions used also impact performance.

## 3 Bang for the Buck

We benchmarked the end-to-end search latency of HNSW, IVF indexes, and *full scans* (i.e., without an index), with vectors quantized at different levels on five microarchitectures available in AWS. These are presented in Table 2 alongside their on-demand price in `us-east-1` at the time of doing this study. These cover the major ISAs and popular CPUs. For Intel SPR, we also present benchmarks on the *Z* series variant[1]. All machines have 64GB of DRAM, 8 vCPUs, and Ubuntu 24.04 as OS. For each experiment, we report queries-per-second (QPS) and queries-per-dollar (QP$).

For our experiments, we used FAISS (v1.9.0) [11,33] compiled to target the underlying CPU capabilities. We chose FAISS as it is the cornerstone of many vector systems. For instance, Milvus [40] and Weaviate [44] vector engines started as a fork of FAISS. OpenSearch (AWS) is directly built on top of FAISS[2]. By using FAISS, we also avoid introducing possible artifacts of vector databases (e.g., Milvus *dynamic batching* mechanism that executes queries at intervals).

[1] aws.amazon.com: r7iz instances
[2] aws.amazon.com: Amazon OpenSearch Service's VecDB capabilities explained

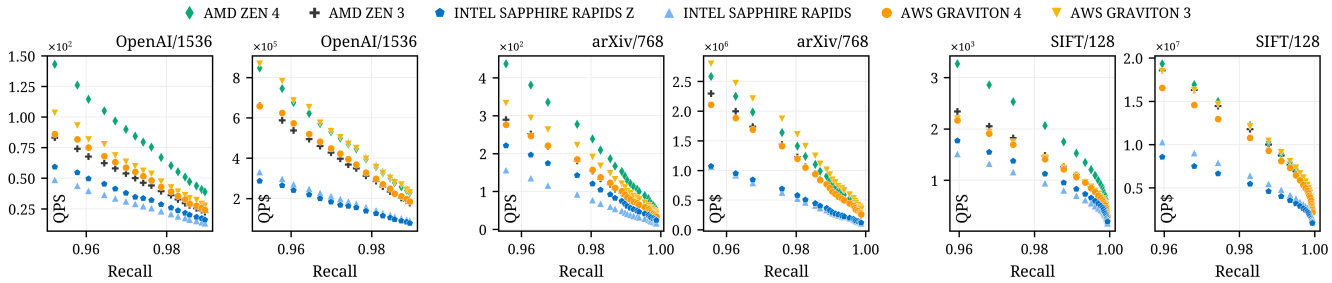



**Figure 3: QPS and QP$ of cloud instances running queries on a FAISS IVF index without quantization (float32). The Zen4 microarchitecture takes the upper hand in QPS, but Graviton3 takes a slight advantage in QP$.**

**Table 4: QPS and QP$ of cloud instances running queries on FAISS IVF indexes quantized at different levels. *QP$ values are in the order of $10^5$. Color coding is the same as in Table 1.***

| | Microarch. | float32 | | SQ8 (8-bit) | | SQ6 (6-bit) | | SQ4 (4-bit) | | BQ (1-bit) | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | QPS | /$ | QPS | /$ | QPS | /$ | QPS | /$ | QPS | /$ |
| OpenAI/1536 | Graviton3 | 27.4 | **2.3** | 9.6 | 0.8 | 5.8 | 0.5 | 6.7 | 0.6 | 504.3 | **42.4** |
| | Zen3 | 22.4 | 1.8 | 32.5 | 2.6 | 15.0 | 1.2 | 17.7 | 1.4 | 363.8 | 28.9 |
| | Graviton4 | 24.2 | 1.8 | 11.7 | 0.9 | 7.3 | 0.6 | 8.1 | 0.6 | **529.1** | 40.4 |
| | SPR | 13.3 | 0.9 | 28.9 | 2.0 | 30.9 | 2.1 | 30.9 | **2.1** | 421.1 | 28.6 |
| | Zen4 | **38.9** | **2.3** | **57.3** | **3.4** | **39.3** | **2.3** | **36.0** | **2.1** | 389.4 | 23 |
| | SPR Z | 16.1 | 0.8 | 31.0 | 1.5 | 34.7 | 1.7 | 32.1 | 1.6 | 469.3 | 22.7 |
| arXiv/768 | Graviton3 | 43.0 | **3.6** | 13.6 | 1.1 | 9.4 | 0.8 | 8.8 | 0.7 | 993.0 | 83.4 |
| | Zen3 | 32.5 | 2.6 | 45.9 | 3.6 | 24.2 | 1.9 | 23.4 | 1.9 | 847.5 | 67.3 |
| | Graviton4 | 36.2 | 2.8 | 16.7 | 1.3 | 11.8 | 0.9 | 10.7 | 0.8 | **1204.8** | **92.1** |
| | SPR | 17.6 | 1.2 | 40.1 | 2.7 | 49.5 | **3.4** | 38.0 | 2.6 | 612.4 | 41.7 |
| | Zen4 | **54.2** | 3.2 | **67.9** | **4.0** | 55.7 | 3.3 | **45.2** | **2.7** | 840.3 | 49.7 |
| | SPR Z | 25.4 | 1.2 | 46.2 | 2.2 | **61.6** | 3.0 | 45.1 | 2.2 | 684.9 | 33.1 |
| SIFT/128 | Graviton3 | 557.1 | **46.8** | 206.2 | 17.3 | 157.1 | 13.2 | 178.0 | 15.0 | 5681.8 | 476.6 |
| | Zen3 | 540.0 | 42.9 | 612.0 | 48.6 | 384.8 | 30.5 | 439.6 | 34.9 | **6250.0** | **497.5** |
| | Graviton4 | 538.2 | 41.1 | 249.9 | 19.1 | 192.0 | 14.7 | 212.6 | 16.2 | **6250.0** | 476.6 |
| | SPR | 342.1 | 23.3 | 749.6 | 51.0 | 871.1 | **59.3** | **831.3** | **56.5** | 5181.3 | 351.7 |
| | Zen4 | **745.2** | 44.1 | **886.5** | **52.4** | 849.6 | 50.2 | 814.3 | 48.2 | 5291.0 | 313.1 |
| | SPR Z | 386.0 | 18.7 | 833.3 | 40.3 | **898.5** | 43.5 | 821.0 | 39.7 | 5681.8 | 274.4 |

Unfortunately, when using SQ, `float16`, or `bfloat`, FAISS first decodes vectors back to the `float32` domain to perform the distance calculations. To make up for that, we add USearch (v.2.16.9) [39] to our benchmarks. USearch is a vector engine focused on getting the best performance on different CPUs with symmetric kernels for various distance metrics and quantized types on HNSW indexes and full scans. USearch is currently used by ClickHouse and DuckDB on their VSS extensions.

For our analysis, we have chosen three datasets that exhibit different dimensionalities, presented in Table 3. These datasets are commonly used to evaluate VSS techniques [7,49]. From these collections, one represents vectors from image data (SIFT/128), and two represent vector embeddings from text (arXiv/768, OpenAI/1536).

**Indexes hyperparameters.** FAISS IVF: Number of centroids: $4 * \sqrt{n}$, buckets visited (for recall tuning): from 2 to 512, training points: all. Quantized with 8, 6, 4, and 1-bit. FAISS FULL SCAN: Quantized with 8, 6, 4, 1-bit and PQ (`n_bits`: 8, `subspaces m`: $\frac{D}{4}$). USearch HNSW: `m`: 16, `ef_construction`: 128, `ef_search` (for recall tuning): from 2 to 512. Quantized with `float16`, `bfloat`, 8, and 1-bit. USearch FULL SCAN: Quantized with `float16`, `bfloat`, 8, and 1-bit. In all settings, we ran queries individually (no multi-threading) with $k = 10$, using L2 and Hamming (on BQ) as distance metrics.

**Table 5: QPS and QP$ of cloud instances running queries on USearch HNSW quantized at different levels. *QP$ values are in the order of $10^6$. Color coding is the same as in Table 1.***

| | Microarch. | float32 | | float16 | | bfloat | | SQ8 (8-bit) | | BQ (1-bit) | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | QPS | /$ | QPS | /$ | QPS | /$ | QPS | /$ | QPS | /$ |
| OpenAI/1536 | Graviton3 | 192.0 | **1.6** | 247.3 | **2.1** | 196.3 | **1.6** | 345.3 | **2.9** | 639.0 | 5.4 |
| | Zen3 | 194.2 | 1.5 | 197.1 | 1.6 | 191.7 | 1.5 | 265.0 | 2.1 | 462.5 | 3.7 |
| | Graviton4 | 183.6 | 1.4 | 241.4 | 1.8 | 204.0 | **1.6** | 310.9 | 2.4 | 620.9 | 4.7 |
| | SPR | 182.0 | 1.2 | 270.2 | 1.8 | 198.6 | 1.4 | 331.2 | 2.3 | 1140.4 | **7.8** |
| | Zen4 | 165.6 | 1.0 | 168.0 | 1.0 | 194.9 | 1.2 | 279.5 | 1.7 | 838.8 | 5.0 |
| | SPR Z | **228.7** | 1.1 | **359.3** | 1.7 | **235.9** | 1.1 | **389.3** | 1.9 | **1313.9** | 6.4 |
| arXiv/768 | Graviton3 | 212.6 | 1.8 | 279.8 | 2.4 | 226.9 | **1.9** | 327.9 | 2.8 | 554.9 | 4.7 |
| | Zen3 | 242.4 | **1.9** | 237.9 | 1.9 | 229.8 | 1.8 | 319.7 | 2.5 | 585.0 | 4.6 |
| | Graviton4 | 224.0 | 1.7 | 276.5 | 2.1 | 227.3 | 1.7 | 317.0 | 2.4 | 664.1 | 5.1 |
| | SPR | 229.3 | 1.6 | 382.4 | **2.6** | 238.4 | 1.6 | **450.3** | **3.1** | 965.3 | **6.6** |
| | Zen4 | 212.3 | 1.3 | 205.0 | 1.2 | 244.8 | 1.4 | 307.8 | 1.8 | 718.4 | 4.2 |
| | SPR Z | **297.7** | 1.4 | **472.7** | 2.3 | **254.7** | 1.2 | 432.7 | 2.1 | **965.9** | 4.7 |
| SIFT/128 | Graviton3 | 626.7 | 5.3 | 1299.2 | 10.9 | 479.4 | 4.0 | 870.9 | 7.3 | 1527.2 | 12.8 |
| | Zen3 | 693.4 | 5.5 | 517.5 | 4.1 | 505.7 | 4.0 | 812.5 | 6.4 | 1879.7 | 14.9 |
| | Graviton4 | 555.0 | 4.2 | 1583.5 | 12.1 | 437.5 | 3.3 | 1024.4 | 7.8 | 1600.8 | 12.2 |
| | SPR | 905.8 | **6.2** | 2337.0 | **15.9** | 918.7 | **6.2** | 1338.3 | **9.1** | 2907.0 | **19.8** |
| | Zen4 | 704.7 | 4.2 | 616.4 | 3.6 | 773.5 | 4.6 | 901.5 | 5.3 | 2136.8 | 12.6 |
| | SPR Z | **959.7** | 4.6 | **2866.2** | 13.9 | **1071.6** | 5.2 | **1542.3** | 7.5 | **3510.0** | 17.0 |

## 3.1 IVF

Figure 3 shows the performance of the microarchitectures on an IVF index without quantization at different recall levels. In this setting, Zen4 is the clear winner across all datasets in QPS, giving 3x the performance of both Intels. However, in QP$, Graviton3 ties with Zen4. Zen3 and Graviton4 follow closely in QP$, and their gap closes as dimensionality decreases. Another observation is that the targeted recall does not affect the relative performance of microarchitectures.

Table 4 shows the performance of each microarchitecture with different quantization levels at the highest possible recall. In SQ vectors, Zen4 and Intel perform well. However, Zen4 takes the upper hand in higher dimensionalities. The Gravitons heavily underperform in SQ vectors (up to 3x and 6x less QP$ and QPS, resp.). This is because the asymmetric distance calculation in FAISS requires going from the quantized domain to the `float32` domain. The latter can be done with SIMD instructions (e.g., `HI/LO-UNPACK` on 8-bit and `SHIFT+OR` on 6-bit vectors). However, FAISS currently uses scalar code to do this in ARM CPUs, contrary to the fully SIMDized kernels used in AVX2/AVX512. Nevertheless, the Gravitons shine on 1-bit vectors as FAISS *does* implement a specialized kernel for BQ in NEON. The latter shows the importance of SIMD kernels.

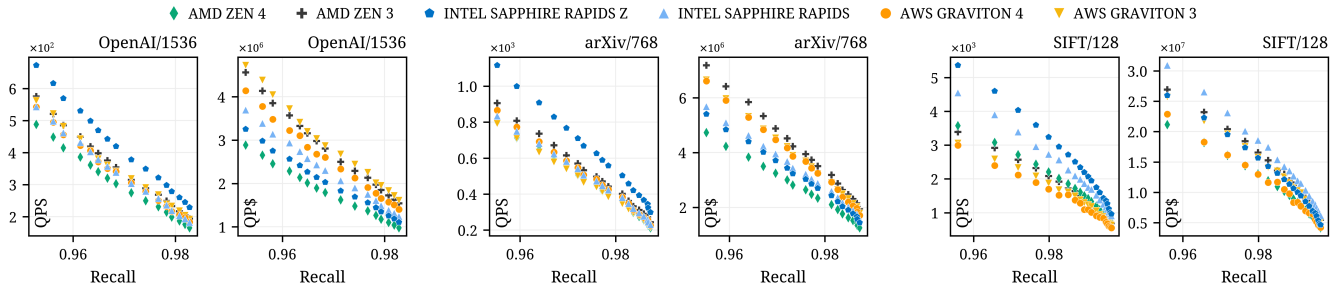



**Figure 4: QPS and QP$ of cloud instances running queries on a flat HNSW index (float32). Intel Z takes the upper hand in QPS, but Zen3 and Graviton3 give higher QP$ on high-dimensional vectors. Here, contrary to IVF indexes, Zen4 underperforms.**

**Table 6: QPS and QP$ of cloud instances running full scan queries. Intel performs the best in PQ vectors. The Gravitons take the upper hand in float16 and 1-bit vectors. In all the other settings, Zen4 is the overall winner. The Gravitons perform well in USearch SQ but not in FAISS SQ due to the absence of SIMD. *QP$ is in the order of $10^4$. Color coding is the same as Table 1.***

| | | FAISS FULL SCAN | | | | | | | | | | | | USearch FULL SCAN | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Microarch. | float32 | | SQ8 (8-bit) | | SQ6 (6-bit) | | SQ4 (4-bit) | | BQ (1-bit) | | PQ | | float32 | | float16 | | bfloat | | SQ8 (8-bit) | | BQ (1-bit) | |
| | | QPS | /$ | QPS | /$ | QPS | /$ | QPS | /$ | QPS | /$ | QPS | /$ | QPS | /$ | QPS | /$ | QPS | /$ | QPS | /$ | QPS | /$ |
| OpenAI/1536 | Graviton3 | 3.9 | 3.2 | 1.2 | 1.0 | 0.8 | 0.7 | 0.9 | 0.8 | 82.5 | **69.4** | 4.1 | 3.5 | 3.7 | **3.1** | **6.8** | **5.8** | 2.9 | 2.5 | 12.8 | **10.8** | 37.5 | 31.5 |
| | Zen3 | 3.6 | 2.8 | 4.4 | **3.5** | 2.1 | 1.6 | 2.5 | 2.0 | 62.7 | 49.8 | 4.2 | 3.4 | 3.3 | 2.6 | 4.0 | 3.2 | 3.9 | 3.1 | 7.6 | 6.1 | 28.9 | 22.9 |
| | Graviton4 | 3.6 | 2.7 | 1.5 | 1.2 | 1.0 | 0.8 | 1.1 | 0.8 | **90.3** | 69 | 4.3 | 3.3 | 3.3 | 2.5 | 6.2 | 4.7 | 3.6 | 2.8 | 13.9 | 10.6 | 46.2 | **35.3** |
| | SPR | 1.9 | 1.3 | 4.0 | 2.7 | 4.6 | **3.1** | 4.7 | **3.2** | 49.2 | 33.5 | 6.6 | **4.5** | 1.9 | 1.3 | 3.7 | 2.5 | 2.9 | 2.0 | 6.1 | 4.1 | 34.0 | 23.1 |
| | Zen4 | **5.6** | **3.3** | **6.0** | **3.5** | 4.7 | 2.8 | **4.9** | 2.9 | 77.5 | 45.8 | 4.0 | 2.4 | **5.2** | **3.1** | 4.3 | 2.6 | **8.8** | **5.2** | **14.3** | 8.4 | **46.7** | 27.6 |
| | SPR Z | 2.3 | 1.1 | 4.2 | 2.0 | **4.8** | 2.3 | **4.9** | 2.4 | 57.1 | 27.6 | **7.0** | 3.4 | 2.3 | 1.1 | 4.6 | 2.2 | 3.1 | 1.5 | 6.8 | 3.3 | 40.8 | 19.7 |
| arXiv/768 | Graviton3 | 3.4 | 2.8 | 1.1 | 0.9 | 0.7 | 0.6 | 0.8 | 0.7 | **71.4** | **60** | 4.2 | 3.5 | 3.4 | **2.9** | **6.0** | **5.0** | 3.3 | 2.8 | 9.7 | 8.2 | 20.5 | 17.3 |
| | Zen3 | 3.0 | 2.4 | 3.8 | 3.0 | 1.8 | 1.5 | 2.2 | 1.7 | 48.7 | 38.6 | 4.5 | 3.6 | 2.7 | 2.2 | 3.3 | 2.6 | 3.2 | 2.5 | 5.8 | 4.6 | 19.0 | 15.1 |
| | Graviton4 | 3.3 | 2.5 | 1.3 | 1.0 | 0.9 | 0.7 | 1.0 | 0.7 | 66.6 | 50.9 | 4.4 | 3.3 | 2.9 | 2.2 | 5.7 | 4.4 | 3.0 | 2.3 | **11.7** | **8.9** | **23.7** | **18.1** |
| | SPR | 1.5 | 1.1 | 3.5 | 2.4 | 4.0 | **2.7** | 3.8 | **2.6** | 37.1 | 25.2 | 6.3 | **4.3** | 1.7 | 1.1 | 3.2 | 2.1 | 2.5 | 1.7 | 4.9 | 3.4 | 19.1 | 13.0 |
| | Zen4 | **4.9** | **2.9** | **6.1** | **3.6** | 4.4 | 2.6 | 4.3 | **2.6** | 56.6 | 33.5 | 4.0 | 2.4 | **4.3** | 2.5 | 3.6 | 2.1 | **6.5** | **3.9** | 9.0 | 5.3 | 21.6 | 12.8 |
| | SPR Z | 2.3 | 1.1 | 3.9 | 1.9 | **4.8** | 2.3 | **4.5** | 2.2 | 40.3 | 19.5 | **7.2** | 3.5 | 2.0 | 1.0 | 3.7 | 1.8 | 2.7 | 1.3 | 5.4 | 2.6 | 21.3 | 10.3 |
| SIFT/128 | Graviton3 | 42.3 | 35.5 | 13.8 | 11.6 | 9.0 | 7.6 | 10.3 | 8.6 | 1032.0 | 867.3 | 96.7 | 81.2 | 27.4 | **23.0** | 35.9 | 30.2 | 26.1 | 21.9 | 47.0 | 39.5 | 50.1 | 42.1 |
| | Zen3 | 39.6 | 31.4 | 44.5 | 35.3 | 23.0 | 18.2 | 26.3 | 20.8 | 1246.9 | **989.4** | 105.5 | **83.7** | 20.3 | 16.1 | 25.5 | 20.2 | 24.1 | 19.1 | 38.4 | 30.5 | 54.5 | 43.2 |
| | Graviton4 | 44.9 | 34.3 | 17.0 | 13.0 | 11.2 | 8.5 | 12.5 | 9.5 | 1081.1 | 826.2 | 108.6 | 83.0 | 28.2 | 21.5 | **40.4** | **30.9** | 24.9 | 19.0 | **55.6** | **42.5** | **59.2** | **45.2** |
| | SPR | 22.8 | 15.5 | 48.4 | 32.9 | 54.1 | **36.8** | 53.8 | **36.6** | 1121.1 | 762.4 | 107.3 | 73.0 | 17.0 | 11.6 | 26.1 | 17.8 | 25.8 | 17.5 | 39.8 | 27.1 | 55.7 | 37.9 |
| | Zen4 | **66.7** | **39.5** | **76.1** | **45.0** | 55.4 | 32.8 | 51.8 | 30.6 | 1344.1 | 794.9 | 108.8 | 64.4 | **34.9** | 20.6 | 25.8 | 15.3 | **38.4** | **22.7** | 42.3 | 25.0 | 52.0 | 30.7 |
| | SPR Z | 26.9 | 13.0 | 54.6 | 26.4 | **59.8** | 29.0 | **56.8** | 27.5 | **1396.6** | 675.5 | **112.8** | 54.6 | 19.0 | 9.2 | 28.9 | 14.0 | 29.0 | 14.0 | 44.0 | 21.3 | 58.7 | 28.4 |

## 3.2 HNSW

Figure 4 shows the performance of the different microarchitectures on an HNSW index without quantization. In this setting, Intel *Z* offers the highest QPS. However, it never gives the highest QP$. For QP$, Zen3 and Graviton3 perform the best in high-dimensional vectors. Contrary to IVF indexes, Zen4 performs the worst. Another observation is that, like in IVF indexes, the targeted recall does not affect the relative performance of microarchitectures.

Table 5 shows each microarchitecture's performance with different quantization levels at the highest possible recall. Intel *Z* gives the highest QPS in nearly all settings. However, things change when looking at QP$. In `float16` vectors, the Intels and Graviton3 shine, and the Zens struggle due to the lack of `float16` SIMD. On `bfloat`, the Gravitons are the winners for high-dimensional vectors, with Zen3 following closely, despite the latter not having direct support for `bfloat`. Finally, in quantized vectors, the Gravitons give the highest QP$ at higher dimensionalities, and the Intels on vectors of lower dimensionality. It is important to note that in HNSW, the gap between architectures is less evident (fewer red and yellow cells).

## 3.3 Full Scans

Table 6 shows the performance of each microarchitecture when doing full scans. For `float32` vectors, Zen4 and Graviton3 give the best QP$, while the Intels fall short. Contrary to HNSW, the Gravitons always take the lead on `float16` instead of Intel, and Zen4 takes the upper hand on `bfloat`. In 8-bit vectors, USearch and FAISS differ: the Gravitons take the lead in USearch thanks to the symmetric kernels, whereas in FAISS, Zen4 and Intel perform well across SQ settings, with Gravitons heavily underperforming again (lack of SIMD decoding). Finally, on 1-bit and PQ vectors, the Gravitons and Intel are the best performers, respectively.

## 3.4 Why the differences across microarchs?

While having symmetric kernels with specialized SIMD *does* make a difference in performance under certain settings, it is not a rule of thumb for which microarchitecture will perform better. For instance, SPR does not win in `float16`+full scan. The performance differences arise due to two factors: the data-access patterns of the search algorithm (since vector search is mostly data-access bound [21,34, 41]) and the efficiency of the distance kernel.

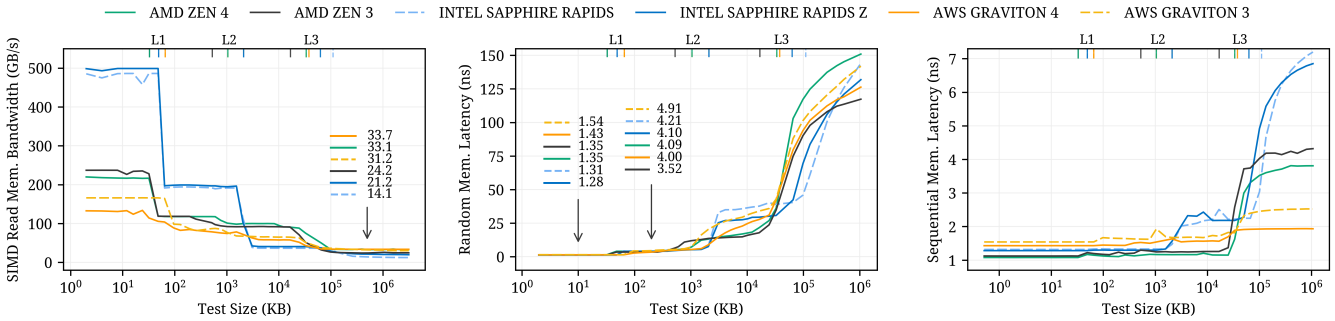



**Figure 5: SIMD read bandwidth and access latency (random and sequential) of cache and DRAM in cloud CPUs.**

**Data-access patterns of indexes.** On the one hand, IVF indexes (and full scans) sequentially access large chunks of vectors that likely exceed the size of the L2 cache or L3 in full scans. As a result, microarchitectures with higher latencies and lower read bandwidths at L3/DRAM are at a disadvantage. Our memory latency benchmarks, presented in Figure 5, show that when data is bigger than L3, the latency of sequential access (plot on the right) on SPR is almost 2x higher than the Zens and almost 4x higher than the Gravitons. Furthermore, the memory bandwidth (plot on the left) of SPR at L3 and DRAM is the lowest. These data-access-related capabilities are reflected in our vector search experiments, in which the performance gap between SPR and Zen4 closes in IVF and full scans as vectors are quantized at smaller bit-widths (see in Table 4 how SPR becomes green from `float32` to SQ4).

On the other hand, searches on an HNSW index access fewer vectors, which results in less pressure on the caches. Moreover, the vectors in the upper layer of the HNSW index are cached efficiently as the entry point of the index is always the same (see left of Figure 1). Hereby, microarchitectures like SPR take advantage, especially with quantized vectors, thanks to having bigger L2/L3 (see Table 2), higher read bandwidth at L1/L2, and a higher SIMD `LOAD` throughput. These observations align with our HNSW experiments, where both SPRs have the highest QPS in all scenarios. PQ also benefits from architectures like SPR, as the codebooks can be cached and accessed more efficiently.

**Efficiency of the distance kernel.** The raw performance of distance kernels depends on the execution throughput and latency of the SIMD instructions used. We define *execution throughput* as the maximum number of bits an instruction can process per CPU cycle across the microarchitecture [5]. Both SPR and Zen4 have a floating-point (FP) execution throughput of 1024 bits per CPU cycle [12]. On the other hand, the Gravitons have an FP execution throughput of 512 bits, which is half of Zen4 and SPR. However, the 1024 bits of Zen4 are arranged as four specialized execution ports of 256 bits each, two for `FP-FMA` (fused multiply-add) and two for `FP-ADD`. Effectively achieving a maximum throughput of 512 bits per cycle for `FP-FMA`. On the contrary, all four ports (512 bits) in Gravitons can handle both FP operations. Thus, depending on the distance kernel, the Gravitons can be on par with Zen4 in FP instructions throughput. However, as data must be first *loaded* into the registers, the throughput of `LOAD` instructions also plays a key role in the efficiency of the kernels. In fact, SPR's main advantage on SIMD is that it can serve 2x512 bits loads per cycle from L1, while Zen4 and Graviton3 can only load 2x256 bits [5,12] and Graviton4 3x128 bits [6]. This effect is visible in the left plot of Figure 5, with SPR having twice as much bandwidth than the Zens when data fits in L1. However, this advantage rapidly degrades as soon as data spills to L3. Finally, regarding latencies, all architectures are close by, with latencies of 3-4 cycles for FP instructions [1,5,6,12,22,23]. However, at smaller register widths, the instructions must be called more times to process the same amount of values. The latter gives an advantage to wider register widths due to the instructions call latency.

**Data size.** To further investigate the effect of data size, we ran full scans on random collections of `float32` vectors of different sizes and dimensionalities. We found that when the collection fits in the L2 cache, SPR achieves 10% more performance than Zen4, with Zen3 and the Gravitons underperforming. However, as soon as data spills to L3, the performance of SPR degrades, and Zen4 takes the lead. SPR performance further degrades when data spills to DRAM, delivering 30% less performance than Zen3 and Graviton4 and 2x less performance than Zen4.

**Graviton4 vs Graviton3.** In many of our experiments, Graviton3 performed better than Graviton4, both in QP$ and QPS (see `float32` in Table 4). The main reason behind this is that Graviton3 has double the size of SVE registers (256 bits) compared to Graviton4 (128 bits). While both architectures have the same FP execution throughput, the total latency cost to process the same amount of data is higher in Graviton4 due to the smaller register. The latter becomes more critical in distance kernels used in vector search, where a dependency chain exists as distances are accumulated on the same SIMD lanes. Furthermore, Graviton3 has a higher execution throughput of `LOAD`, capable of loading 2x256 bits in one CPU cycle [5], while Graviton4 only 3x128 bits [6]. Further experiments on full scans on our random collections of `float32` vectors reveal that SVE is 37% faster than NEON on Graviton3 (due to doubling the register size). In contrast, in Graviton4, NEON and SVE perform the same since the register size, execution throughput, and latencies are the same. When comparing NEON to NEON, Graviton4 is faster than Graviton3 by 10%. However, when switching to SVE, the tables turn, and Graviton3 is 31% faster than Graviton4. These findings differ from most benchmarks, in which Graviton4 always delivers more performance. However, we believe these observations have been under the radar since most benchmarks use NEON, as SVE has not yet been widely adopted.

## 4 Bang for the Buck: Takeaways

In Table 1, we rank every microarchitecture on five tiers based on their QPS and QP$ on the OpenAI/1536 dataset: ( ++ ) for the best one, ( + ) for the ones with a performance at most 20% away from the best, ( - ) when they provide 2x less, and ( -- ) when they provide 3x less QP$ than the best option, and ( · ) for options within the middle ground. Finally, we present an aggregated score by giving 1 point for every (+) and -1 point for every (-).

Graviton3 gives the best "bang for the buck," even over its successor, Graviton4. Graviton3 is only pushed back in our scoring system due to the lack of symmetric kernels in FAISS. Note that Graviton3 excels in the following areas: a variety of SIMD capabilities, high read throughput, and low sequential memory latency at L3/DRAM. More importantly, it is cheap. Zen4 is still a solid option for vector search on IVF indexes and full scans, especially in `float32` and `bfloat` vectors. Zen3 has few negative points thanks to its low price and low latencies for L2/L3/DRAM access. However, it does not excel in any setting. Finally, the SPRs have the lowest score in QP$ and do not excel in any setting. Despite SPR Z having the best QPS score for HNSW, its price brings down its QP$ score.

## 5 Discussion

The need for Vector Databases is widely disputed among the community [46], with some foreseeing them merging with existing database systems. While the future of VecDBs as standalone systems remains uncertain, vector workloads are here to stay. We believe our insights are important as the decoupling of storage and compute that the cloud provides makes it possible to switch microarchitectures easily depending on the search algorithm to be used. Furthermore, providing the right microarchitecture can incur huge savings, especially in serverless vector search services [28,36], where every millisecond counts toward billing.

In this study, we have focused on the three most common use cases in vector search: HNSW, IVF, and full scans, at different quantization levels. However, there exists a wider variety of indexes (e.g., hybrids), quantization techniques (e.g., residual quantizers [11]), and distance metrics (e.g., cosine similarity, inner product). Therefore, we encourage users to perform data-driven benchmarks to uncover the best microarchitecture for their use case.

It is important to acknowledge that the microarchitectures presented in this study will be outdated in the future. For instance, although not yet available in AWS, Intel has already released Emerald Rapids, the successor to Sapphire Rapids. Similarly, AMD has already released Zen5, the successor to Zen4. Nevertheless, the insights presented here should motivate researchers in the vector search community to not only strive for lower theoretical complexities but also for better data-access patterns and storage designs of newly developed algorithms. The latter is critical for the performance of vector search on a large scale, where it is heavily data-access bound [4,13,21,31,34]. Finally, we encourage researchers *always* to use SIMD-optimized implementations, as newly developed algorithms that may shine in their scalar implementation can fail to be on par with SIMD-optimized approaches [21,47].

## 6 Conclusions and Future Work

In this paper, we have shed light on how to get the best "bang for the buck" when using vector search solutions in the cloud. We showed that the performance of vector search does not solely depend on the SIMD capabilities of the underlying CPU, as the data-access patterns of the search algorithm and the size of the vectors also play an important role. We have shown that these differences in CPUs and search algorithms are enough to get ≈3x more queries per dollar (QP$) if the microarchitecture is carefully chosen. Overall, in AWS, Graviton3 (r7g) and Zen4 (r7a) CPUs give you a good "bang for the buck" in the majority of vector search scenarios.

In future work, different types of vector-centric workloads can be benchmarked (e.g., interleaving updates and queries, batch queries with multithreading, index construction). Also, a similar study with different GPU microarchitectures could bring valuable insights. Furthermore, a study comparing the QP$ yielded by different VecDBs would be a next step to understand further the tradeoff between storage designs and distance kernels implementations. Ultimately, the current landscape of vector research needs more established benchmarks, such as TPC-H or TPC-DS in conventional databases.

## References


[1] Andreas Abel and Jan Reineke. 2019. uops. info: Characterizing latency, throughput, and port usage of instructions on intel microarchitectures. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. 673–686.

[2] Azim Afroozeh and Peter Boncz. 2023. The FastLanes Compression Layout: Decoding > 100 Billion Integers per Second with Scalar Code. *Proc. VLDB Endow.* 16, 9 (jul 2023), 2132–2144. doi:10.14778/3598581.3598587

[3] Cecilia Aguerrebere, Ishwar Bhati, Mark Hildebrand, Mariano Tepper, and Ted Willke. 2023. Similarity search in the blink of an eye with compressed indices. *arXiv preprint arXiv:2304.04759* (2023).

[4] Cecilia Aguerrebere, Mark Hildebrand, Ishwar Singh Bhati, Theodore Willke, and Mariano Tepper. 2024. Locally-Adaptive Quantization for Streaming Vector Search. *arXiv preprint arXiv:2402.02044* (2024).

[5] Arm Limited. 2022. *Arm Neoverse V1 Software Optimization Guide.* Version 6.0.

[6] Arm Limited. 2022. *Arm Neoverse V2 Software Optimization Guide.* Version 3.0.

[7] Martin Aumüller, Erik Bernhardsson, and Alexander Faithfull. 2020. ANN-Benchmarks: A benchmarking tool for approximate nearest neighbor algorithms. *Information Systems* 87 (2020), 101374.

[8] Qi Chen, Bing Zhao, Haidong Wang, Mingqin Li, Chuanjie Liu, Zengzhong Li, Mao Yang, and Jingdong Wang. 2021. Spann: Highly-efficient billion-scale approximate nearest neighborhood search. *Advances in Neural Information Processing Systems* 34 (2021), 5199–5212.

[9] Chroma. 2024. Single-Node Chroma: Performance and Limitations. https://docs.trychroma.com/production/administration/performance.

[10] Kumar Shivendu David Myriel. 2024. Intel's New CPU Powers Faster Vector Search. https://qdrant.tech/blog/qdrant-cpu-intel-benchmark/.

[11] Matthijs Douze, Alexandr Guzhva, Chengqi Deng, Jeff Johnson, Gergely Szilvasy, Pierre-Emmanuel Mazaré, Maria Lomeli, Lucas Hosseini, and Hervé Jégou. 2024. The faiss library. *arXiv preprint arXiv:2401.08281* (2024).

[12] Agner Fog. 2016. The microarchitecture of Intel, AMD and VIA CPUs: An optimization guide for assembly programmers and compiler makers. *Software optimization resources* (2016).

[13] Jianyang Gao, Yutong Gou, Yuexuan Xu, Yongyi Yang, Cheng Long, and Raymond Chi-Wing Wong. 2024. Practical and Asymptotically Optimal Quantization of High-Dimensional Vectors in Euclidean Space for Approximate Nearest Neighbor Search. *arXiv preprint arXiv:2409.09913* (2024).

[14] Jianyang Gao and Cheng Long. 2023. High-dimensional approximate nearest neighbor search: with reliable and efficient distance comparison operations. *Proceedings of the ACM on Management of Data* 1, 2 (2023), 1–27.

[15] Ruiqi Guo, Philip Sun, Erik Lindgren, Quan Geng, David Simcha, Felix Chern, and Sanjiv Kumar. 2020. Accelerating large-scale inference with anisotropic vector quantization. In *International Conference on Machine Learning*. PMLR, 3887–3896.

[16] Masajiro Iwasaki and Daisuke Miyazaki. 2018. Optimization of indexing based on k-nearest neighbor graph for proximity search in high-dimensional data. *arXiv preprint arXiv:1810.07355* (2018).

[17] Omid Jafari, Preeti Maurya, Parth Nagarkar, Khandker Mushfiqul Islam, and Chidambaram Crushev. 2021. A survey on locality sensitive hashing algorithms

and their applications. *arXiv preprint arXiv:2102.08942* (2021).
[18] Herve Jegou, Matthijs Douze, and Cordelia Schmid. 2010. Product quantization for nearest neighbor search. *IEEE transactions on pattern analysis and machine intelligence* 33, 1 (2010), 117–128.
[19] Zhi Jing, Yongye Su, Yikun Han, Bo Yuan, Haiyun Xu, Chunjiang Liu, Kehai Chen, and Min Zhang. 2024. When large language models meet vector databases: A survey. *arXiv preprint arXiv:2402.01763* (2024).
[20] Anthony Ko, Iman Keivanloo, Vihan Lakshman, and Eric Schkufza. 2021. Low-precision quantization for efficient nearest neighbor search. *arXiv preprint arXiv:2110.08919* (2021).
[21] Leonardo Kuffo, Elena Krippner, and Peter Boncz. 2025. PDX: A Data Layout for Vector Similarity Search. *Proc. ACM Manag. Data* (2025).
[22] Chester Lam. 2023. Sapphire Rapids: Golden Cove Hits Servers. https://chipsandcheese.com/p/a-peek-at-sapphire-rapids.
[23] Chester Lam. 2024. Arm's Neoverse V2, in AWS's Graviton 4. https://chipsandcheese.com/p/arms-neoverse-v2-in-awss-graviton-4.
[24] Daniel Lemire. 2018. AVX-512: when and how to use these new instructions. https://lemire.me/blog/2018/09/07/avx-512-when-and-how-to-use-these-new-instructions/.
[25] Yu A Malkov and Dmitry A Yashunin. 2018. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. *IEEE transactions on pattern analysis and machine intelligence* 42, 4 (2018), 824–836.
[26] zilliz Milvus. 2020. Milvus performance on AVX-512 vs. AVX2. https://milvus.io/blog/2020-11-10-milvus-performance-AVX-512-vs-AVX2.md.
[27] Javier Vargas Munoz, Marcos A Gonçalves, Zanoni Dias, and Ricardo da S Torres. 2019. Hierarchical clustering-based graphs for large scale approximate nearest neighbor search. *Pattern Recognition* 96 (2019), 106970.
[28] Joe Oakley and Hakan Ferhatosmanoglu. 2025. SQUASH: Serverless and Distributed Quantization-based Attributed Vector Similarity Search. *arXiv preprint arXiv:2502.01528* (2025).
[29] James Jie Pan, Jianguo Wang, and Guoliang Li. 2023. Survey of vector database management systems. *arXiv preprint arXiv:2310.14021* (2023).
[30] James Jie Pan, Jianguo Wang, and Guoliang Li. 2024. Vector Database Management Techniques and Systems. In *Companion of the 2024 International Conference on Management of Data*. 597–604.
[31] Yannis Papakonstantinou, Alan Li, Ruiqi Guo, Sanjiv Kumar, and Phil Sun. 2024. *ScaNN for AlloyDB*. Technical Report. Google Cloud. https://services.google.com/fh/files/misc/scann_for_alloydb_whitepaper.pdf Whitepaper.
[32] Liana Patel, Peter Kraft, Carlos Guestrin, and Matei Zaharia. 2024. Acorn: Performant and predicate-agnostic search over vector embeddings and structured data. *Proceedings of the ACM on Management of Data* 2, 3 (2024), 1–27.
[33] Meta Research. 2024. *Faiss: A library for efficient similarity search and clustering of dense vectors*. https://github.com/facebookresearch/faiss
[34] Viktor Sanca and Anastasia Ailamaki. 2024. Efficient Data Access Paths for Mixed Vector-Relational Search. In *Proceedings of the 20th International Workshop on Data Management on New Hardware*. 1–9.
[35] Spotify. 2017. *ANNOY by Spotify*. https://github.com/spotify/annoy
[36] Yongye Su, Yinqi Sun, Minjia Zhang, and Jianguo Wang. 2024. Vexless: A Serverless Vector Data Management System Using Cloud Functions. *Proceedings of the ACM on Management of Data* 2, 3 (2024), 1–26.
[37] Philip Sun, David Simcha, Dave Dopson, Ruiqi Guo, and Sanjiv Kumar. 2023. SOAR: improved indexing for approximate nearest neighbor search. *Advances in Neural Information Processing Systems* 36 (2023), 3189–3204.
[38] Ash Vardanian. 2023. *SimSimd: Up to 200x Faster Dot Products & Similarity Metrics*. https://github.com/ashvardanian/SimSIMD
[39] Ash Vardanian. 2023. *USearch by Unum Cloud*. doi:10.5281/zenodo.7949416
[40] Jianguo Wang, Xiaomeng Yi, Rentong Guo, Hai Jin, Peng Xu, Shengjun Li, Xiangyu Wang, Xiangzhou Guo, Chengming Li, Xiaohai Xu, et al. 2021. Milvus: A purpose-built vector data management system. In *Proceedings of the 2021 International Conference on Management of Data*. 2614–2627.
[41] Mengzhao Wang, Haotian Wu, Xiangyu Ke, Yunjun Gao, Yifan Zhu, and Wenchao Zhou. 2025. Accelerating Graph Indexing for ANNS on Modern CPUs. *arXiv preprint arXiv:2502.18113* (2025).
[42] Mengzhao Wang, Xiaoliang Xu, Qiang Yue, and Yuxiang Wang. 2021. A comprehensive survey and experimental comparison of graph-based approximate nearest neighbor search. *arXiv preprint arXiv:2101.12631* (2021).
[43] Duncan J Watts and Steven H Strogatz. 1998. Collective dynamics of 'small-world'networks. *nature* 393, 6684 (1998), 440–442.
[44] Weaviate. 2019. *Weaviate*. https://github.com/weaviate/weaviate
[45] Jiuqi Wei, Xiaodong Lee, Zhenyu Liao, Themis Palpanas, and Botao Peng. 2025. Subspace Collision: An Efficient and Accurate Framework for High-dimensional Approximate Nearest Neighbor Search. *Proceedings of the ACM on Management of Data* 3, 1 (2025), 1–29.
[46] Jasper Xian, Tommaso Teofili, Ronak Pradeep, and Jimmy Lin. 2024. Vector search with OpenAI embeddings: Lucene is all you need. In *Proceedings of the 17th ACM International Conference on Web Search and Data Mining*. 1090–1093.
[47] Qian Xu, Juan Yang, Feng Zhang, Junda Pan, Kang Chen, Youren Shen, Amelie Chi Zhou, and Xiaoyong Du. 2025. Tribase: A Vector Data Query Engine for Reliable and Lossless Pruning Compression using Triangle Inequalities. *Proceedings of the ACM on Management of Data* 3, 1 (2025), 1–28.
[48] Mingyu Yang, Wentao Li, Jiabao Jin, Xiaoyao Zhong, Xiangyu Wang, Zhitao Shen, Wei Jia, and Wei Wang. 2024. Effective and General Distance Computation for Approximate Nearest Neighbor Search. *arXiv preprint arXiv:2404.16322* (2024).
[49] Xianzhi Zeng, Zhuoyan Wu, Xinjing Hu, Xuanhua Shi, Shixuan Sun, and Shuhao Zhang. 2024. CANDY: A Benchmark for Continuous Approximate Nearest Neighbor Search with Dynamic Data Ingestion. *arXiv preprint arXiv:2406.19651* (2024).