Quantization

- Quantization is a classical lossy data compression technique
- A quantizer, in the broadest sense, is something that reduces the number of possible values that a variable has.

Original

Compressed

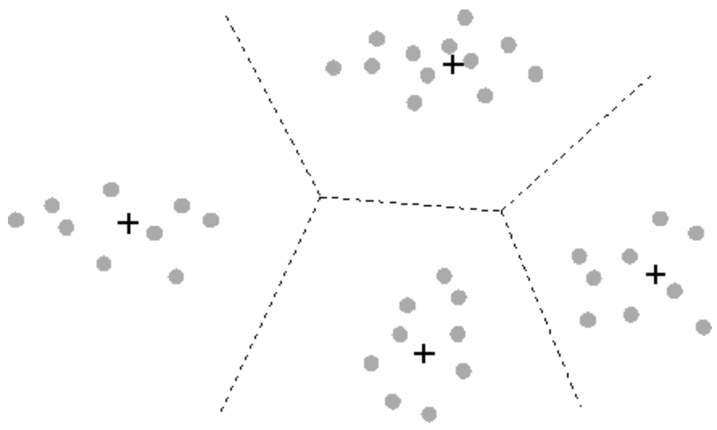
- A good example would be building a lookup table to reduce the number of colors in an image. Find the most common 256 colors, and put them in a table mapping a 24-bit RGB color value down to an 8-bit integer.

Compress dataset (1)

- Why do we need to compress the dataset?
  - Memory access times are generally the limiting factor on processing speed
  - Sheer memory capacity can be a problem for big datasets
- YouTube-8M has 1.4 billion 1024 dimensional feature vectors extracted from 560,000 hours of video using the Inception-V3 model
- While each day 720,000 hours of new video are uploaded

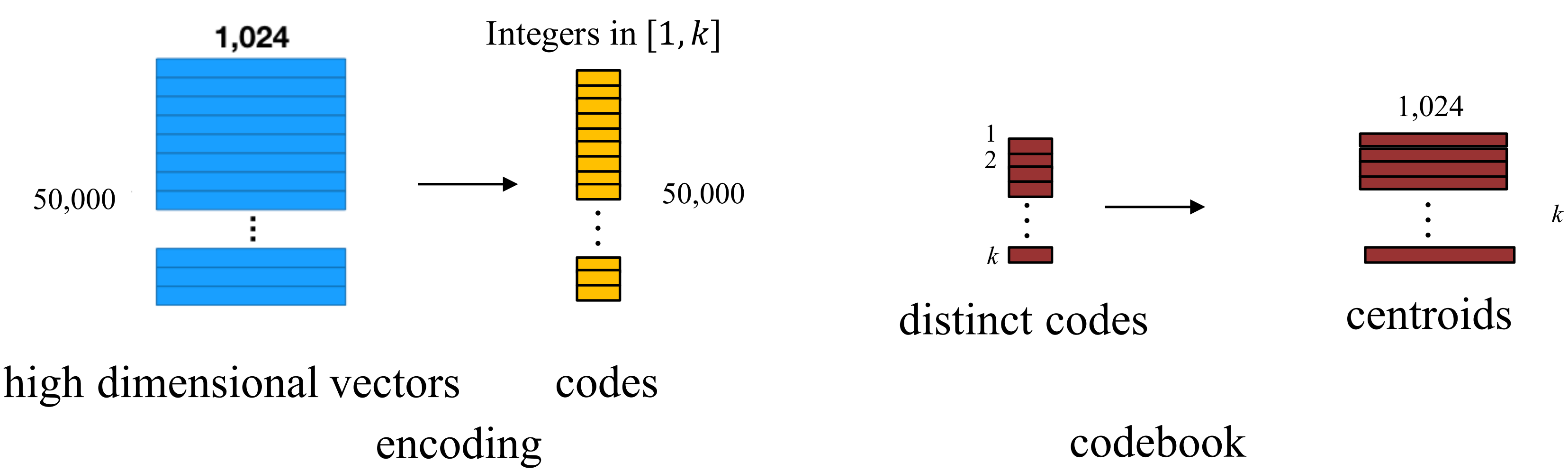
Vector Quantization

- use centroids to represent vectors in clusters
- distance(query, vector) ~ distance(query, centroid)



Example

- Map the 50,000-vector dataset by a vector quantizer with k centroids using k-means
- Each code is an integer ranging from 1 to k
- Codebook: a map from code to the centroid (which is a vector)



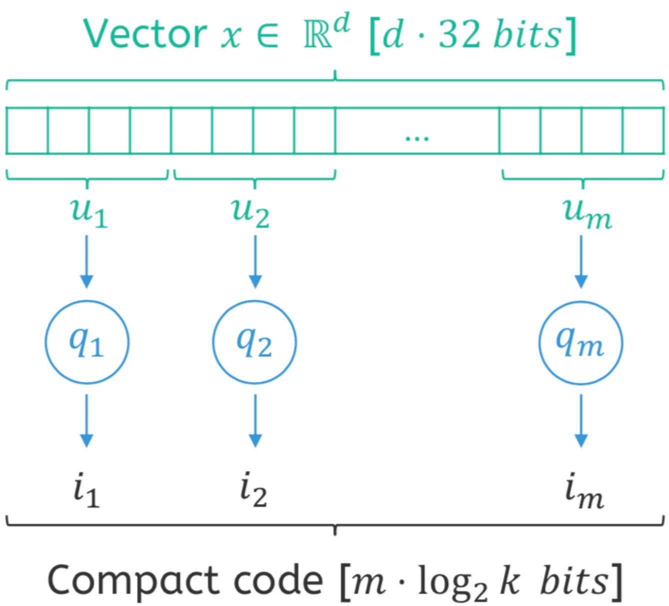
Vector Quantization

- Vector Quantizer reduces the the cardinality of the representation space
  - The memory cost of storing the centroid index is $\lceil log_2 k \rceil$ bits
  - Memory cost for whole dataset is reduced to $N \times \lceil log_2 k \rceil + k \times D \times 32$ (1 float = 32 bits)
  - In comparison, original space cost is $N \times D \times 32$

Drawback

- Needs a huge number of clusters to distinguish vectors
- A quantizer producing 64-bit codes contains k=2^{64} centroids
  - The complexity of learning the quantizer are several times k
  - Impossible to store the $D \times k$ floating point values that represent the k centroids

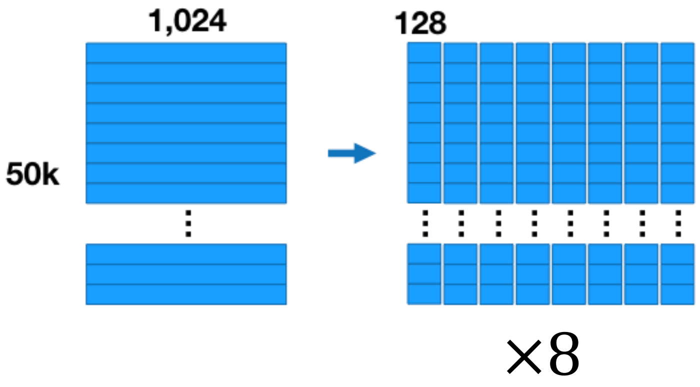
Product Quantizer (PQ)



- Split x into m sub-vectors
  Typ. $m = 8\ or\ 16$
- The input vector x is split into m distinct sub-vectors u_1 ... u_m
- Quantize each u_j with a distinct quantizer q_j. Each quantizer has k centroids.
- Each quantizer produces one log$k$ $bits$ integer

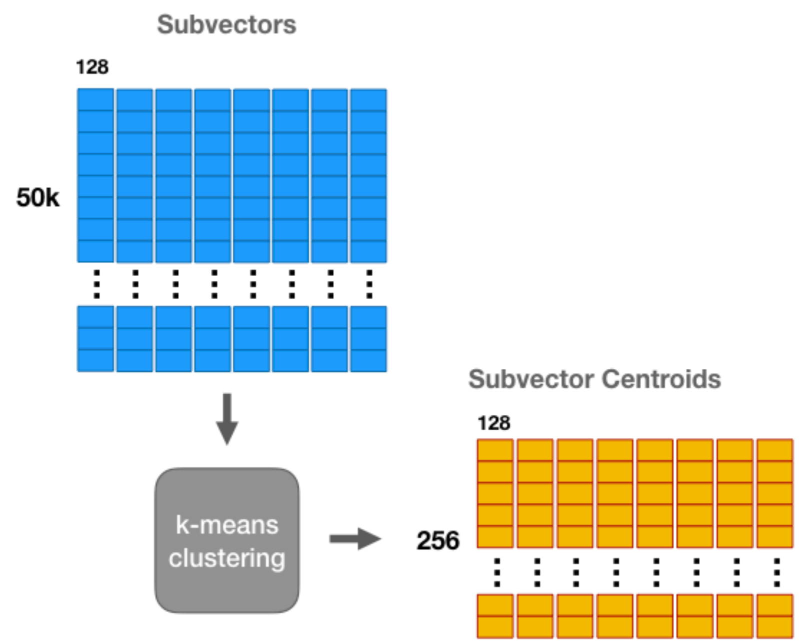
Compress dataset (2)

- Apply PQ in our problem
- Settings: m=8, k=256
- Chop up the vectors into 8 sub-vectors, each of length 128
  - This divides our dataset into 8 matrices that are [50K x 128] each



Compress dataset (3)

- Run k-means clustering separately on each of these 8 matrices with k = 256
- Get 256×8 centroids
- Each centroid has 128 dimensions

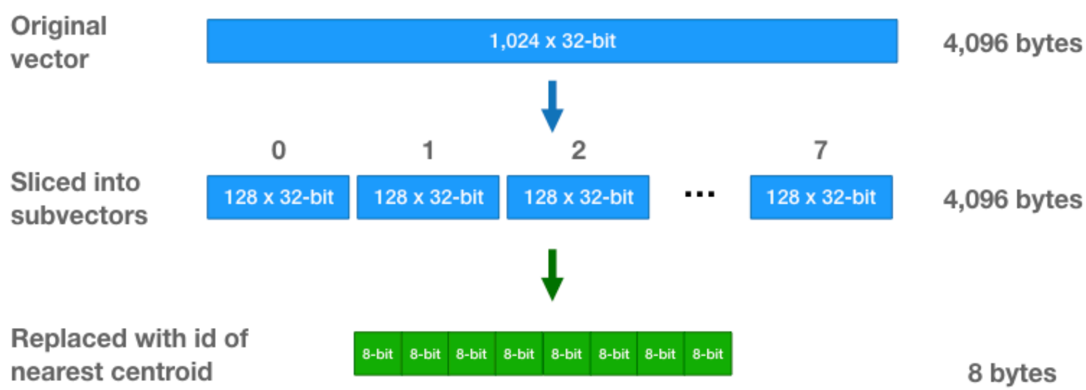


Compress dataset (4)

- Centroids are like "prototypes"
  - Represent the most commonly occurring patterns in the dataset sub-vectors
- Use these centroids to compress our vector dataset
  - Replace each sub-region of a vector with the closest matching centroid
  - New vectors are different from the original, but hopefully still close

Compress dataset (5)

- For each sub-vector, we find the closest centroid, and store the *id* of that centroid
- Each vector will be replaced by a sequence of 8 centroid ids



512× smaller space

Example PQ codes

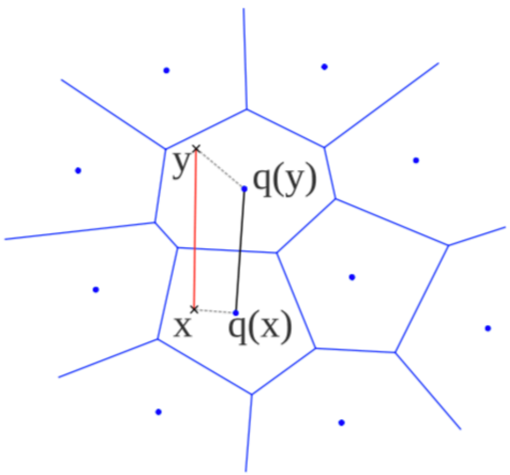
0							7
01	03	02	05	06	09	04	08
3f	11	21	00	01	f2	12	11
04	0c	0e	1a	f1	0f	a9	17
f6	ff	f6	f0	23	0b	b6	2f
37	1a	21	00	32	8b	e9	03
f5	fc	ff	f1	46	33	cf	2c
			⋮				

Product Quantizer (PQ)

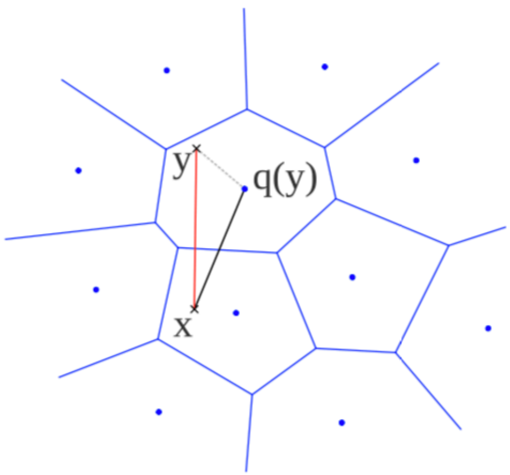
- A reproduction value of the product quantizer is identified by an element of the product index set $I = I_1 \times \cdots \times I_m$
- The codebook: $C = C_1 \times \cdots \times C_m$
- A centroid of C is the concatenation of centroids of m subquantizers
- Assuming each subquantizer has $k*$ centroids, the total number of centroids in C is $k = (k^*)^m$
- The learning complexity is m times the complexity of performing k-means clustering with $k*$ centroids of dimension $D*$

Two cases of distance compute

- Symmetric: $d(x, y) \approx d(q(x), q(y))$
- Asymmetric: $d(x, y) \approx d(x, q(y))$



symmetric case



asymmetric case

Nearest Neighbor Search

	Subspace 1	…	Subspace M
1st centroid	0.45	…	1.24
…	…	…	…
k-th centroid	0.88	…	0.82

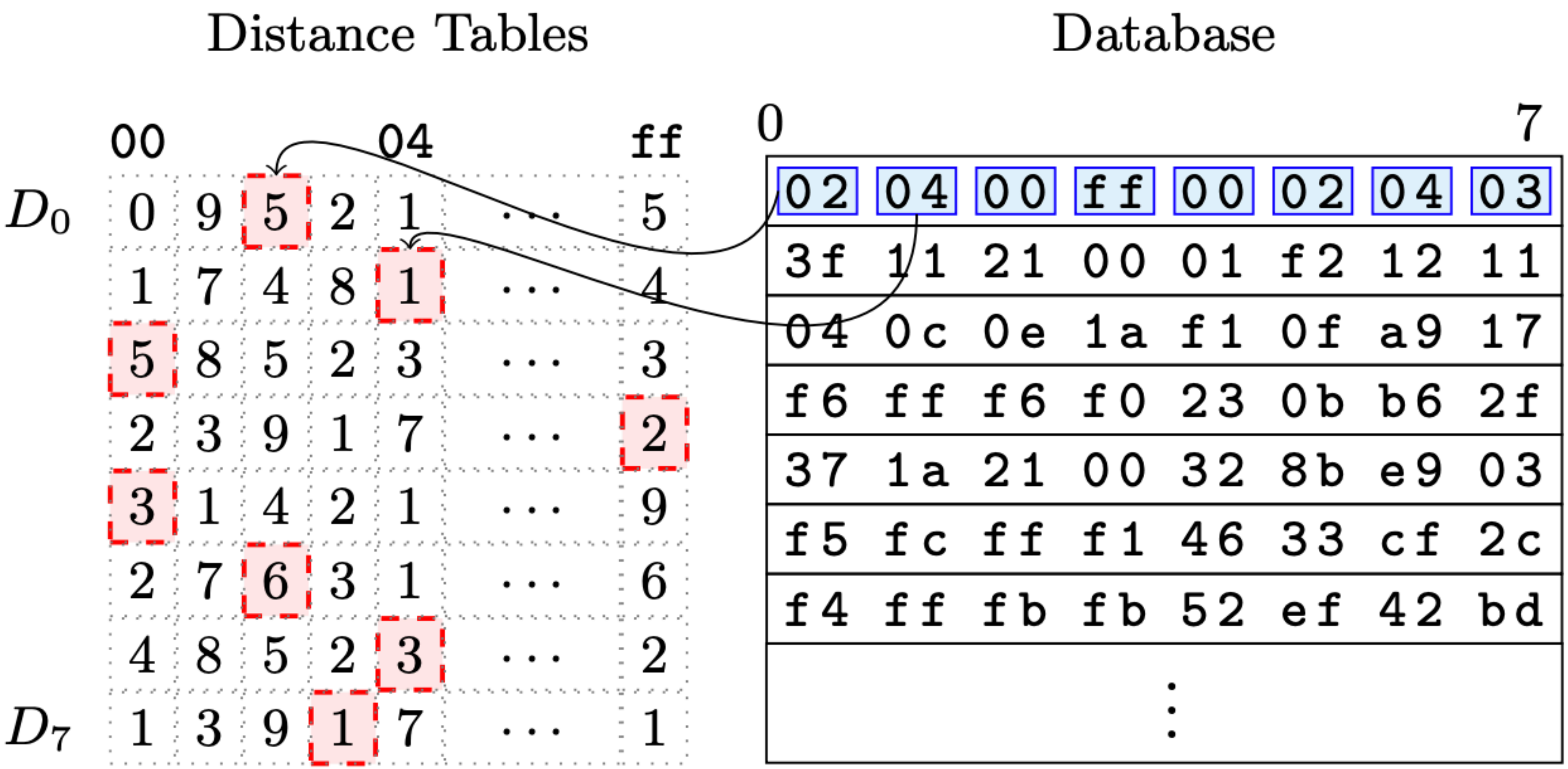
Step 1: given a query, build a distance lookup table (only 256 x 8 = 2048 entries)

(ff, 11, 04, … … … … ... , 00)

256th centroid, 0.88 1st centroid, 1.24

Step 2: scan all PQ code, calculate the distances using lookup table, and return the top-k results

Nearest Neighbor Search

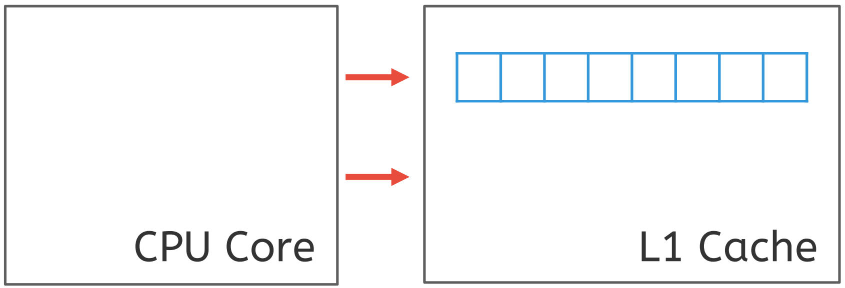


Nearest Neighbor Search

- Then, for each database vector, we use those centroid ids to lookup the partial distances in the table, and sum those up.
  - Database vector $v = \{cid_1, cid_2, \dots cid_M\}$
  - $Dist(v, query) = \sum_{i=1}^{M} dist_{cid_i}^{i}$
- M additions instead of D subtractions, D multiplications and $D - 1$ additions
- Scan the whole database to find the nearest neighbors

PQ Fast Scan

PQ Scan and Cache Accesses



Lookup tables in L1 Cache

PQ Scan

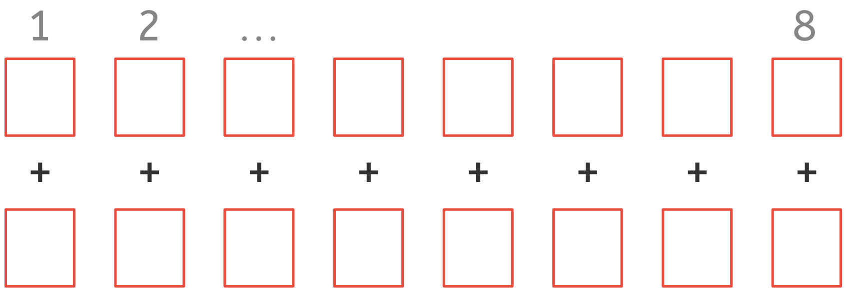
2 Concurrent accesses

4 Cycles latency

- Each $dist$ computation requires
  - $m = 8$ table lookups ($D_j[i_j]$)
  - $m - 1 = 7$ additions
- Lookup tables $D_1 \dots D_8$ are stored in L1 cache (fastest cache)
- Cache accesses are still costly
- **Bottleneck:** Cache accesses

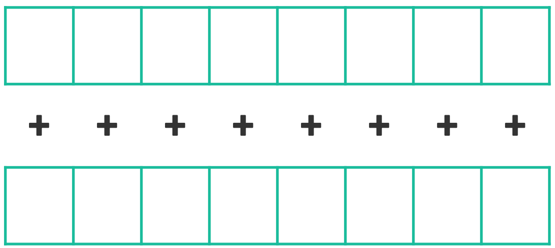
SIMD primer

Scalar (regular) add



8 instructions

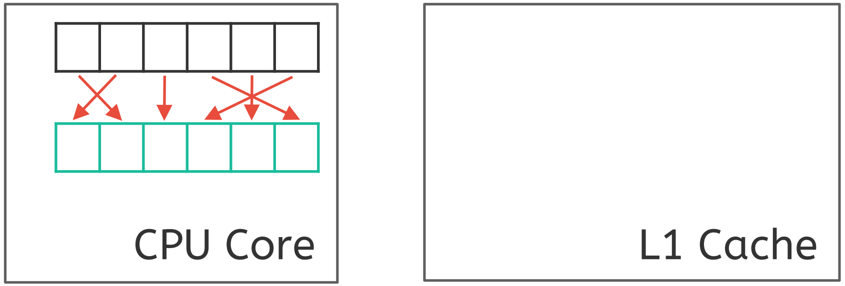
SIMD add



1 SIMD instruction

- **S**ingle **I**nstruction **M**ultiple **D**ata
- Process **multiple data elements** at once
- SIMD computing unit in **each core**
- Used for **high-performance** (e.g. linear algebra)

PQ Fast Scan Key Idea



Lookup tables in SIMD registers

PQ Fast Scan

16

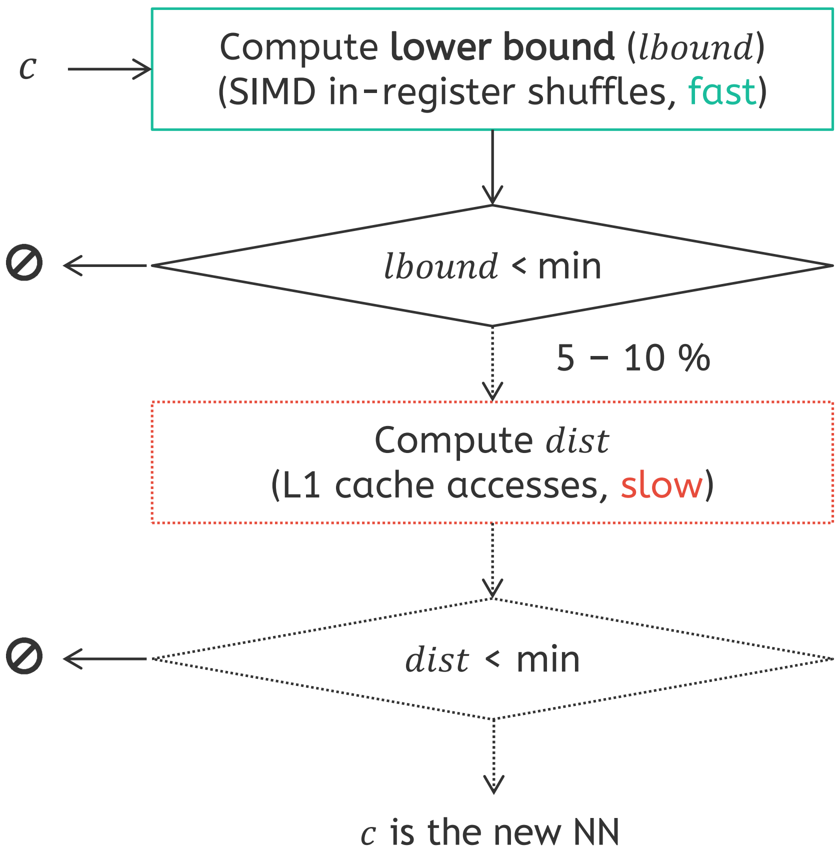
Concurrent accesses

1

Cycles latency

- **Key Idea:** Replace cache accesses by SIMD in register shuffles
- Bonus: Use SIMD additions to further increase performance
- **Challenge:** Lookup tables do not fit SIMD registers
  - Lookup table : 256 x 32 bits
  - SIMD register : 16 x 8 bits

PQ Fast Scan Overview



- Compute small tables S_1 … S_8 that fit SIMD registers
- Use S_1 … S_8 to compute lower bounds on distances
- Lower bounds are used to prune *dist* computations
- **By design, same results as PQ Scan**

Small Tables Construction

Each small table S_j is built from the corresponding D_j table:

D_j

↓ 256 x 32 bits (floats)

Code grouping / Minimum tables

↓ 16 x 32 bits (floats)

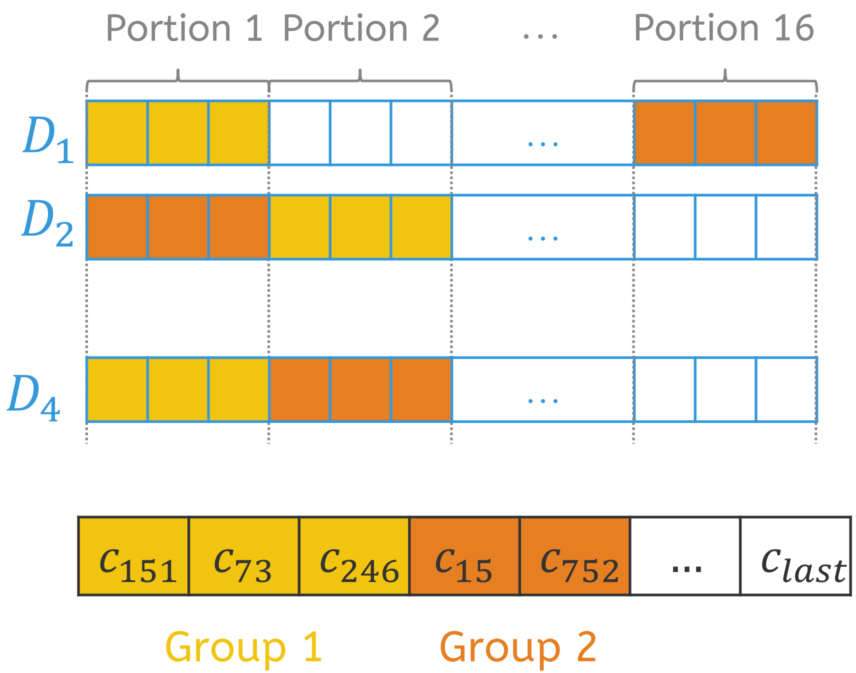
Quantization of floats

↓ 16 x 8 bits (int)

S_j

Code grouping

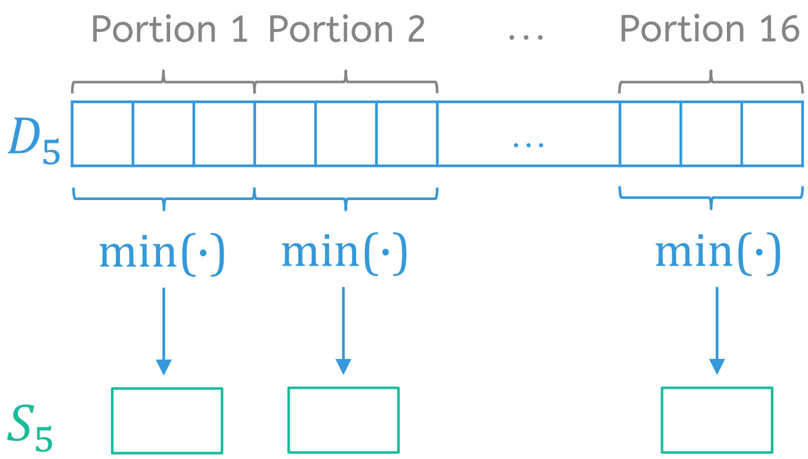
256 floats ➞ 16 floats
Used for $D_1 \dots D_4$



- Split D_j (256 floats) into 16 **portions** of 16 floats each
- **Group** inverted lists
- Load portions in SIMD registers (small table) to scan a group
- More tables ➞ smaller groups
- Too small groups detrimental for performance

Minimum tables

256 floats ➞ 16 floats
Used for $D_5 \ldots D_8$

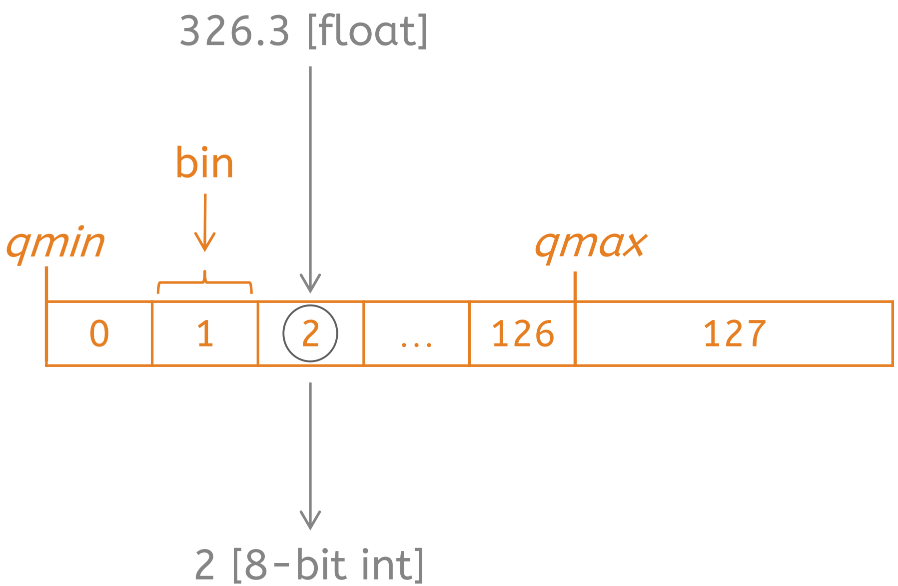


1. Split D_j (256 floats) into 16 **portions** of 16 floats each
2. Take the minimum of each portion
   16-element minimum table

- Table loaded only once in SIMD registers

Quantization of Distances

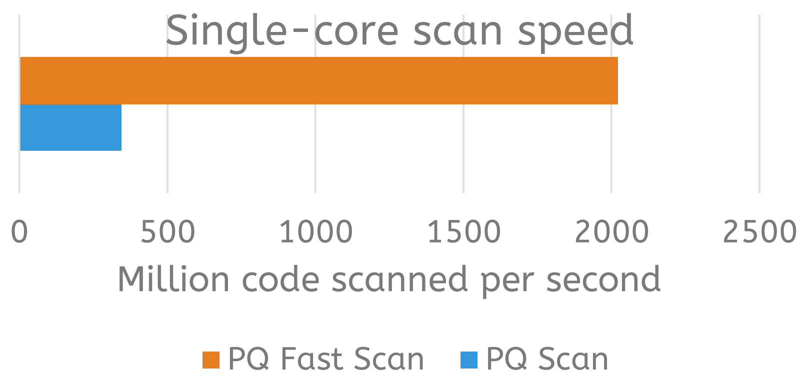
16 floats ➞ 16 8-bit ints



- **Scalar** quantizer
  Not a vector quantizer
- **Signed** 8-bit int
  SIMD limitation
  Positive range: 0-127
- **Saturated** quantization
  Saturated adds

Evaluation: Global Performance

Scan speed



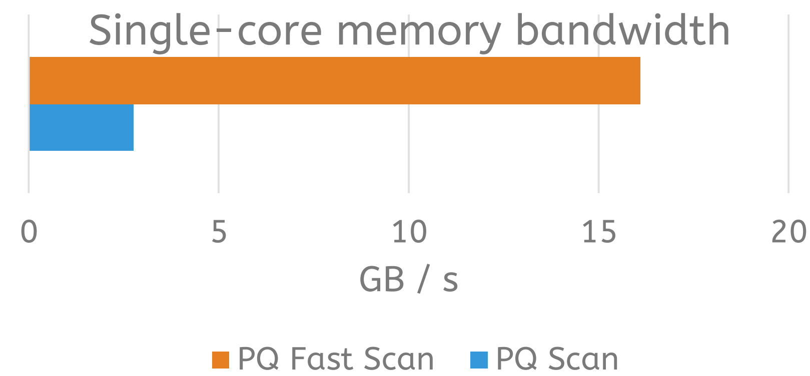
Typical Speedup

vs. PQ Scan

4-6x

Higher scan speed

Memory bandwidth



Typical Mem. bandwith

Single-core

12-16 GB/s

Memory-bandwith bound on multicore CPUs