Bolt: Accelerated Data Mining with Fast Vector Compression

Davis Blalock
MIT CSAIL
Cambridge, Ma, USA
dblalock@mit.edu

John Guttag
MIT CSAIL
Cambridge, Ma, USA
guttag@mit.edu

Massachusetts Institute of Technology

CSAIL

Problem

- Computing on large datasets is costly in terms of both compute **time** and storage **space**
- Goal: **reduce these costs** for datasets that consist of (or can be converted to) dense vectors
  - E.g., image descriptors, not-too-sparse feature vectors, embeddings
- Approach: **compress** the vectors and operate directly on compressed representations

Roadmap

- Problem and Assumptions
- Background and Related Work
- Algorithm
- Theoretical Results
- Experimental Results
- Conclusion

Scenario / Assumptions

- We have a collection of **"data" vectors** $\mathcal{X}$
  - Vectors may be inserted/deleted at any time
- We receive **"query" vectors** $\mathbf{q}$
  - Want dot products / distances between $\mathbf{q}$ and each vector in $\mathcal{X}$
- We have a **training set** and a training phase before any queries are received
- Vectors are all the same length
- Computing on **CPU** (GPU future work)

Problem Statement Setup

Let $\mathbf{q} \in \mathbb{R}^J$ be a *query* vector and let $\mathcal{X} = \{\mathbf{x}_1, \ldots, \mathbf{x}_N\}, \mathbf{x}_i \in \mathbb{R}^J$ be a collection of *database* vectors. Further let $d : \mathbb{R}^J \times \mathbb{R}^J \to \mathbb{R}$ be a distance or similarity function that can be written as:

$$d(\mathbf{q}, \mathbf{x}) = f\Big(\sum_{j=1}^{J} \delta(q_j, x_j)\Big) \tag{1}$$

where $f : \mathbb{R} \to \mathbb{R}$, $\delta : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$.

Problem Statement

Construct three functions $g : \mathbb{R}^J \rightarrow \mathcal{G}$, $h : \mathbb{R}^J \rightarrow \mathcal{H}$, and $\hat{d} : \mathcal{G} \times \mathcal{H} \rightarrow \mathbb{R}$ such that for a given approximation loss $\mathcal{L}$,

$$\mathcal{L} = E_{\mathbf{q},\mathbf{x}}[(d(\mathbf{q}, \mathbf{x}) - \hat{d}(g(\mathbf{q}), h(\mathbf{x})))^2] \tag{2}$$

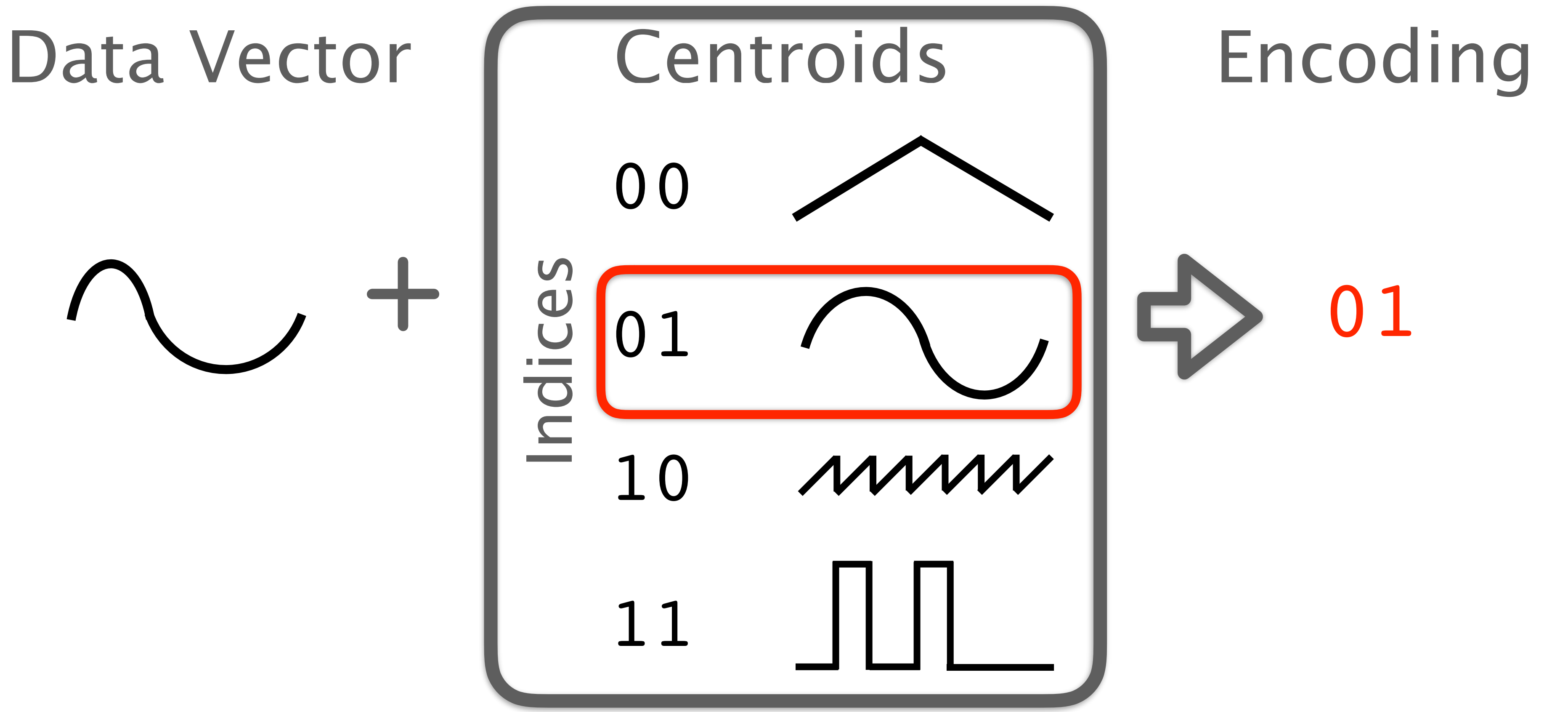
the computation time T,

$$T = T_g + T_h + T_d \tag{3}$$

is minimized, where T_g is the time to encode received queries $\mathbf{q}$ using g, T_h the time to encode $\mathcal{X}$ using h, and T_d the time to compute the approximate distances between the encoded queries and encoded database vectors.

Background: K-Means Quantization

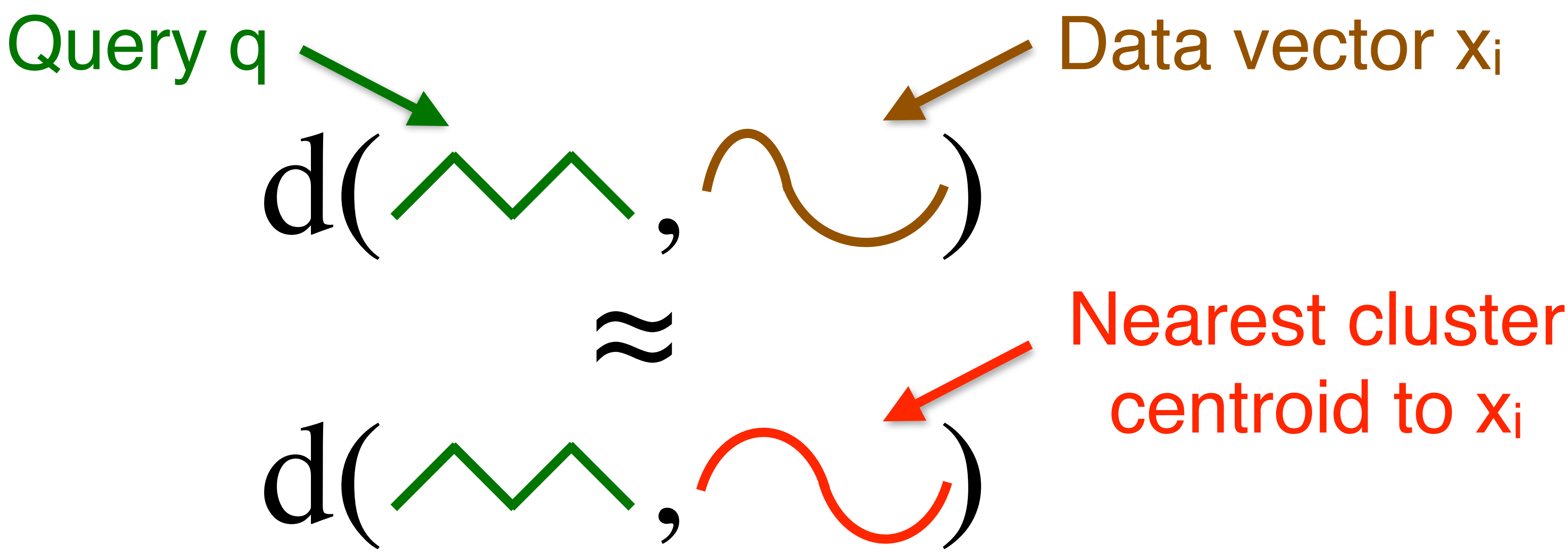
- Simple way to quantize a vector: k-means
- Encoding is index of nearest centroid



Davis Blalock

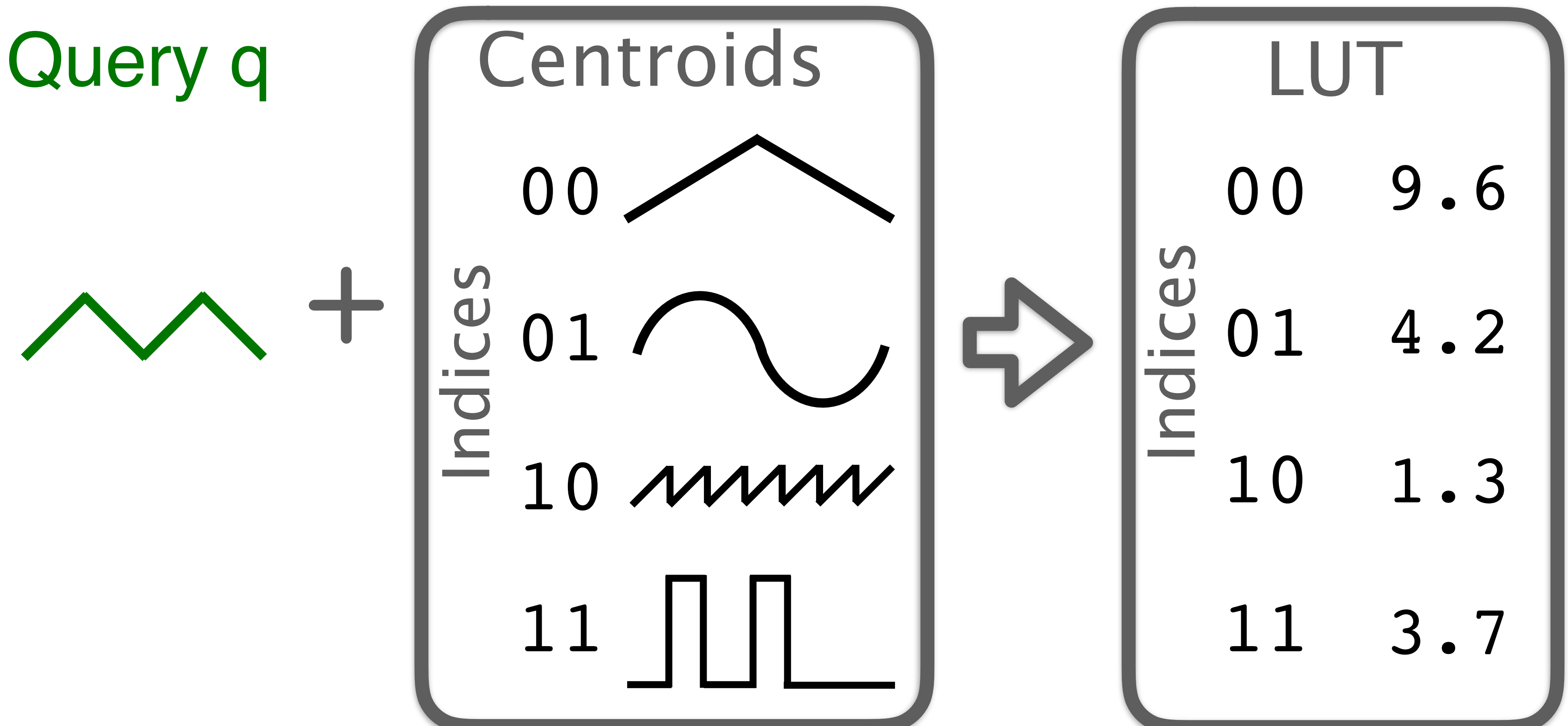
Background: K-Means Quantization

- Can approximate distance to vector with distance to its nearest centroid



Background: K-Means Quantization

- K centroids ⇒ only K possible distances
  - Precompute them all, then use a look-up table (LUT)
  - E.g., distance to encoding `01 = LUT[01] = 4.2`



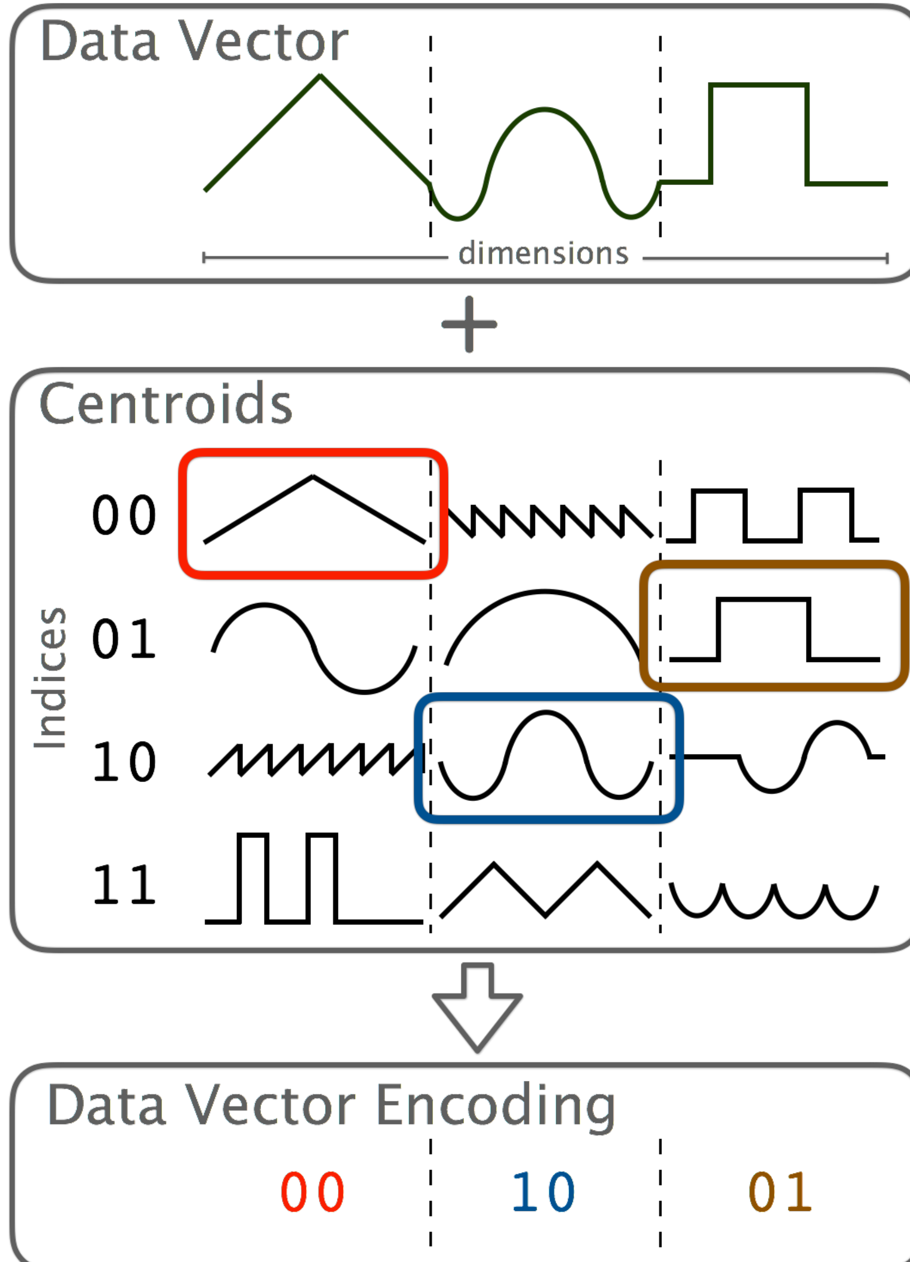
Background: Product Quantization

- Only K centroids is a problem
  - Small K ⇒ can't capture distribution or distinguish distances
  - Large K ⇒ huge encoding cost

- Solution: treat the vector as M separate *subvectors* and encode each separately
  - K^M possible encodings

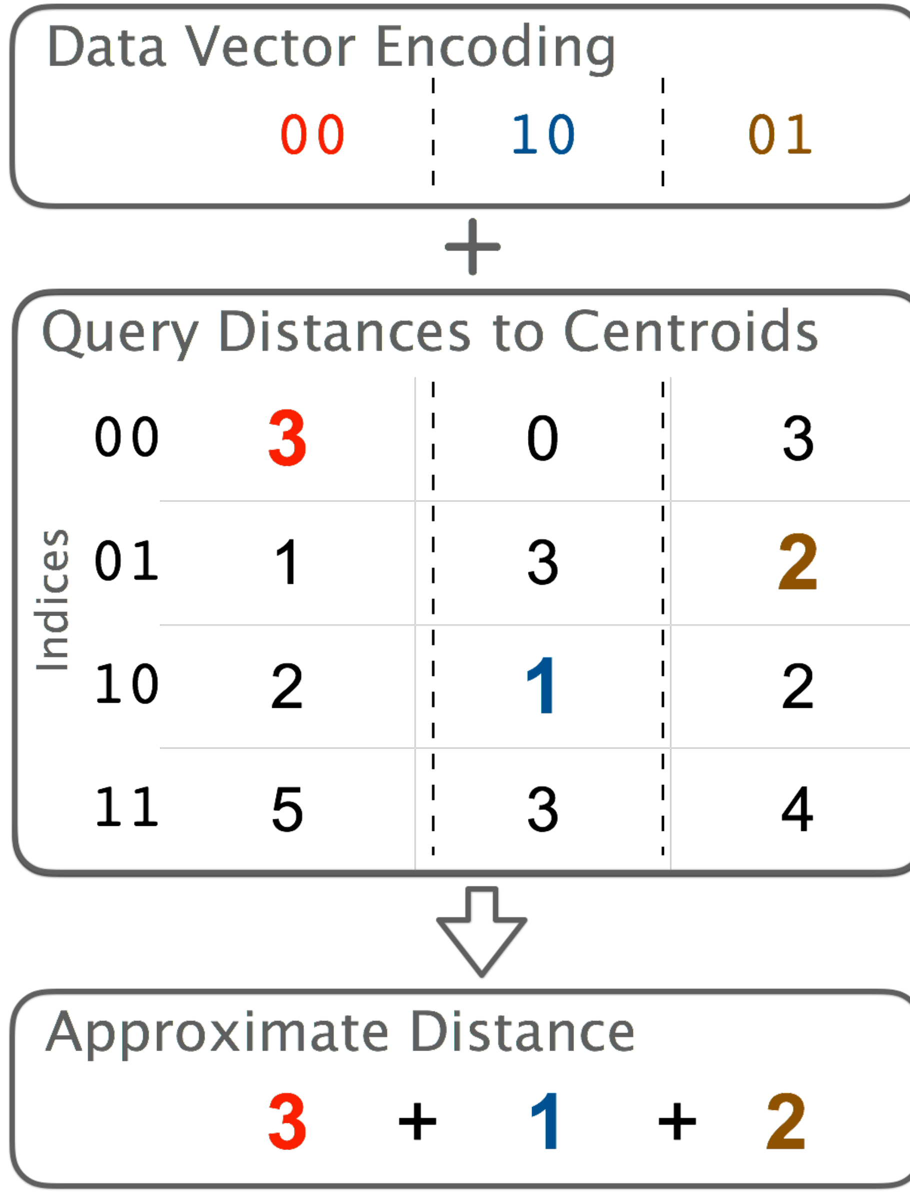
Product Quantization Data Encoding

- Learn K-means centroids separately in each subspace
- Run K-means quantization in each subspace
- Concatenate encodings



Product Quantization Distances

- For each query, compute lookup table for each subspace (each column on the right)
- Sum distances from each subspace



Product Quantization Complexity

- Encoding x_i
  - K-means quantize each and concatenate encodings
  - **O(KD)** time, K = # of centroids, D = dimensionality
- Encoding q
  - Compute M look-up tables (LUTs) for the M subspaces
  - **O(KD)** time
- Approximating $d(q, x_i)$
  - Perform 1 lookup in each of the M LUTs
  - **O(M)** time (and M << D)

Bolt

- Similar to PQ, but 10+ times faster
  - While retaining compatibility with published extensions/complementary techniques

- Idea 1: Use more subvectors, fewer centroids
- Idea 2: Approximate look-up tables

- Secret sauce is using both together…

Bolt: M bigger, K smaller

- Encoding times are O(KD) — no M!
- Everyone uses K=256 so each subspace is 1byte
- Use K = 16 instead; each subspace is 4 bits
- No loss in capacity (in theory):
  - 8B PQ code → K = 256, M = 8
    - $256^8 = 2^{64}$ possible encodings
  - 8B Bolt code → K = 16, M = 16
    - $16^{16} = 2^{64}$ possible encodings

Bolt: M bigger, K smaller

- On its own, setting K = 16 is strictly worse for distance computations
  - Less ability to exploit structure
  - Slower
- But when combined with Idea 2…

Bolt: Approximate LUTs

- Do lookup tables really need 32bit precision?
- Quantize to 8 bits, [0,255]

LUT

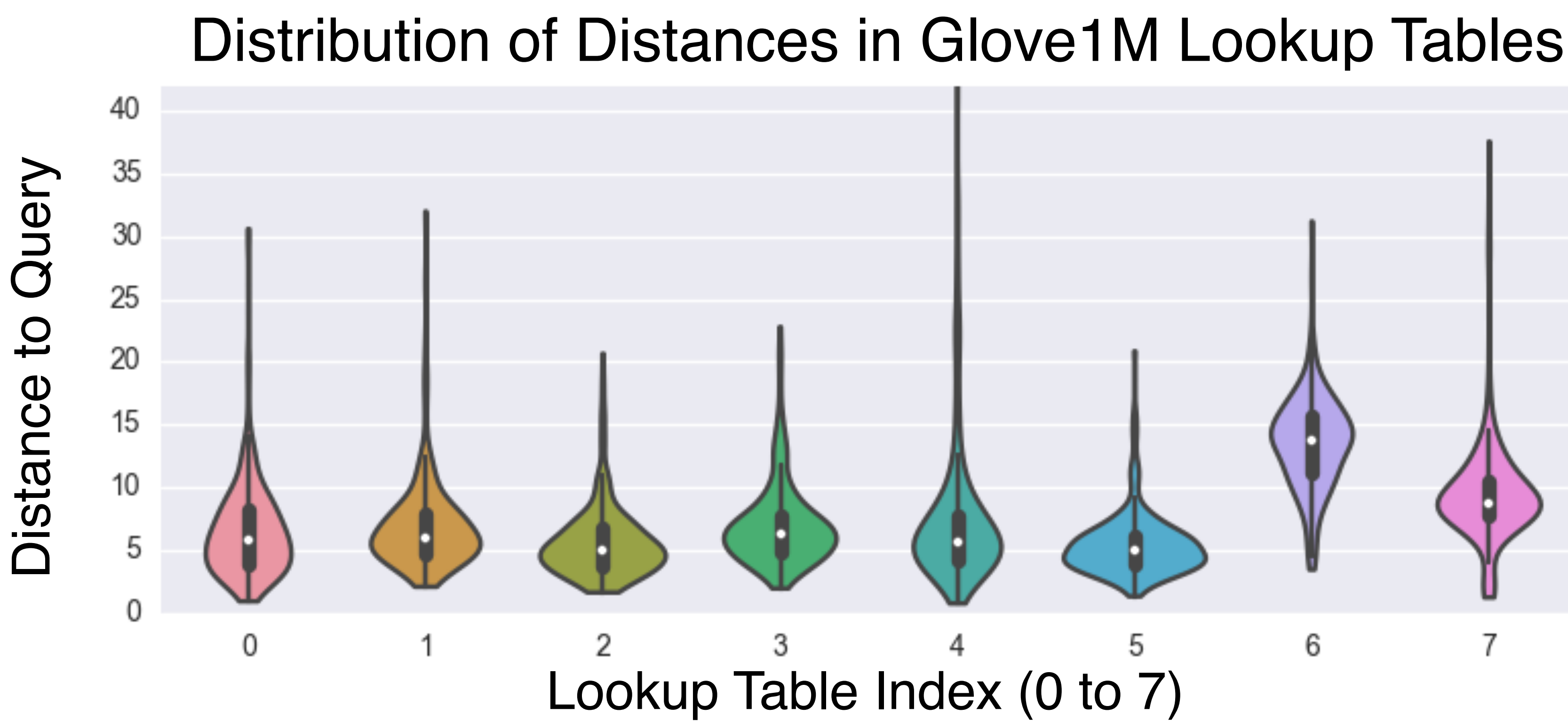
Indices	
00	9.6128…
01	4.2729…
10	1.3371…
11	3.7045…

LUT

Indices	
00	217
01	96
10	30
11	83

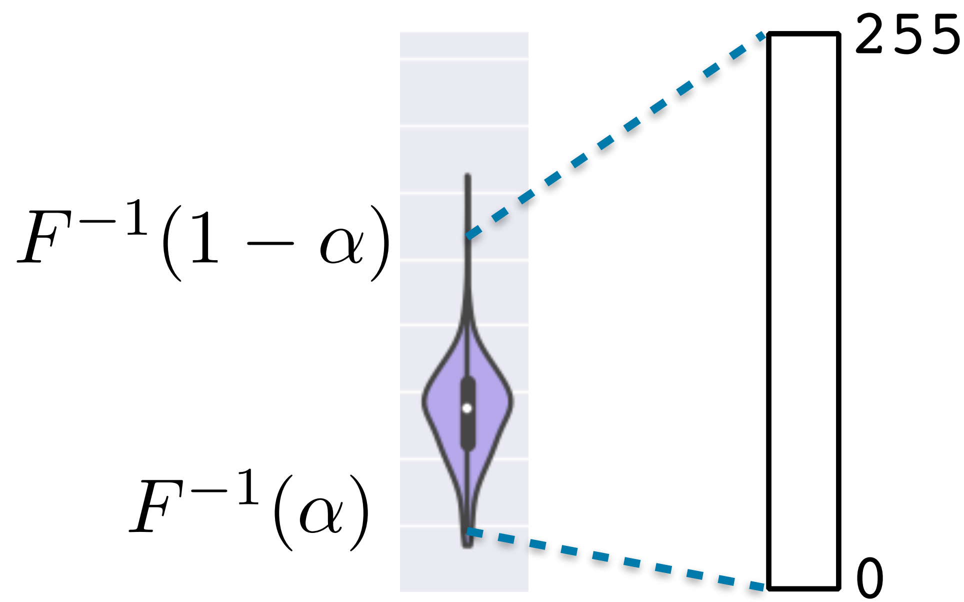
Bolt: Approximate LUTs

- But directly quantizing fails miserably…



Bolt: Approximate LUTs

- Solution: Learn a quantization function for each LUT at training time
- Find quantile α that minimizes quantization error when distances in $[\alpha, 1\text{-}\alpha]$ are linearly scaled to [0, 255]
- Cheap to evaluate, so just try α in {0, .005, ..., .05} and take the best



Bolt: Secret Sauce

- Using ideas 1 and 2, we have 16 centroids per LUT, and 1B per LUT entry
- 16B LUTs, 4 bit indices into it

- Magic: Hardware vectorization!
  - 16B LUT fits in a SIMD register, not L1 cache
  - Can do 32 lookups in parallel by byte shuffling

Bolt: Vectorization

```
dists = zeros(32)

codes = 32xM block of X encodings

LUTS = Mx16 lookup tables for query

for i = 1 to 32:  // PQ

    code = codes[i]

    for m = 1 to M:

      dists[i] += LUTS[m][code[m]]

for m = 1 to M:  // Bolt

    dists += LUTS[m, codes[:, m]] // O(1)
```

Theoretical Guarantees

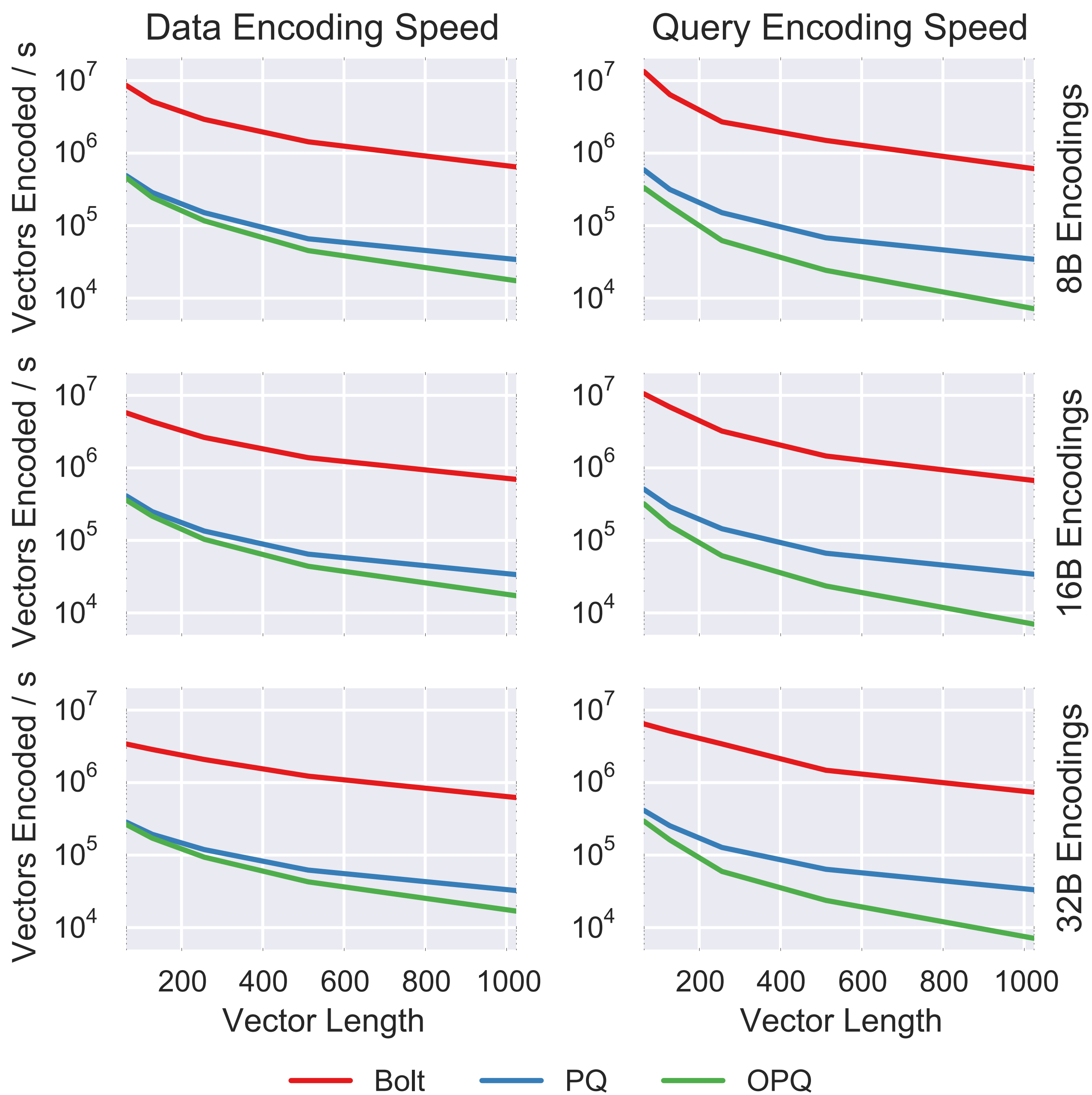
- Error in distance and dot product computations bounded by PQ quantization error + LUT quantization error
- If distribution of LUT entries is any Gaussian, Subgaussian, Laplace, or Exponential, quantization error has $O(\exp(-\varepsilon))$ concentration bound
- Overall probability of error ε in distances using PQ-style approach is $O(\exp(-\varepsilon^2))$

Experiments

- Metrics:
  - # of vectors compressed / sec
  - # of distances computed / sec
  - Recall@R for nearest neighbors
  - Correlation of approx. distances with true distances
- Comparisons:
  - PQ, OPQ, BLAS/raw floats, binary embedding
  - Bolt without quantization

Encoding Speed

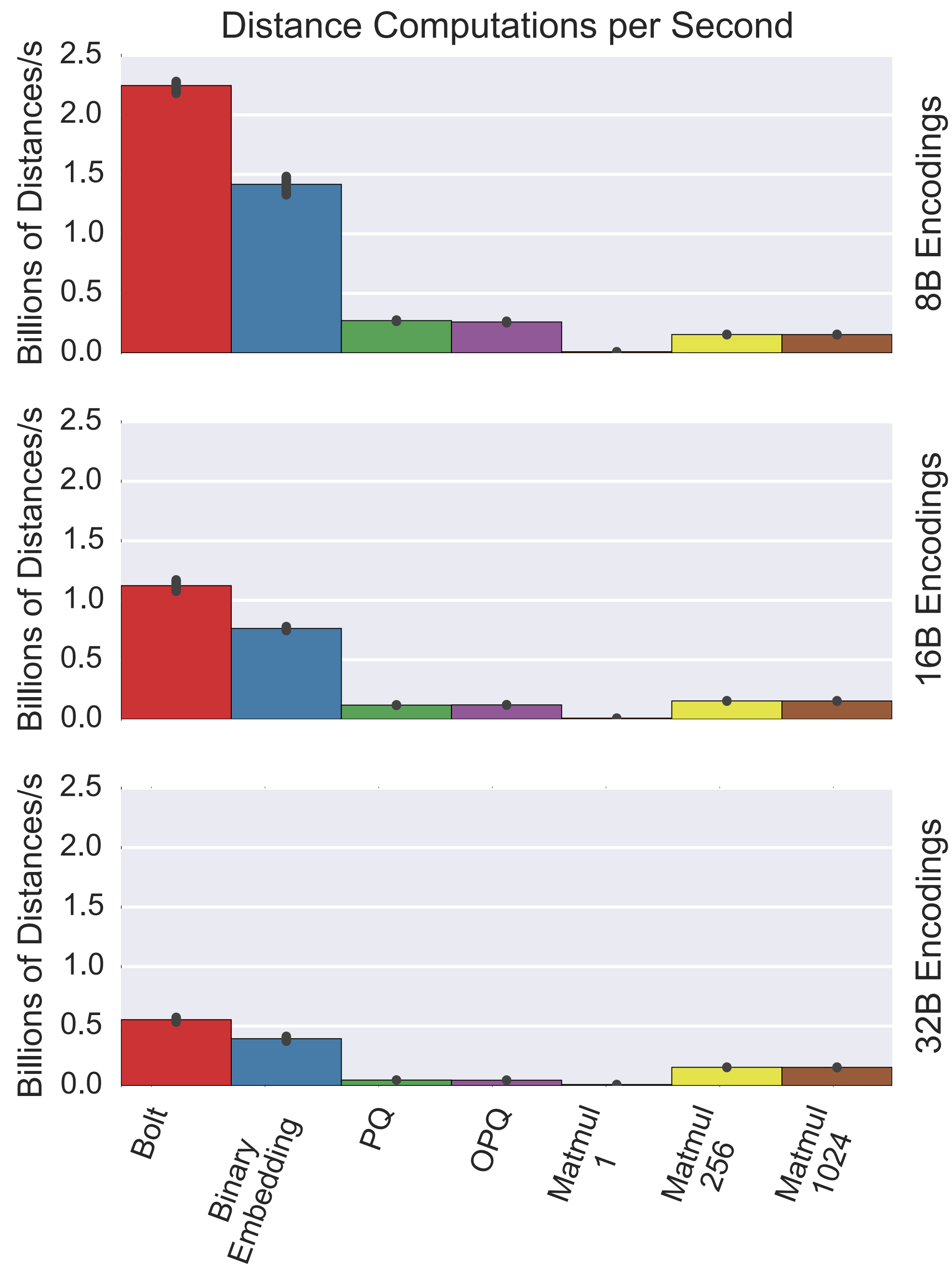
- Over 3GB/s for 8B codes
- Throughput inversely proportional to code length



Davis Blalock

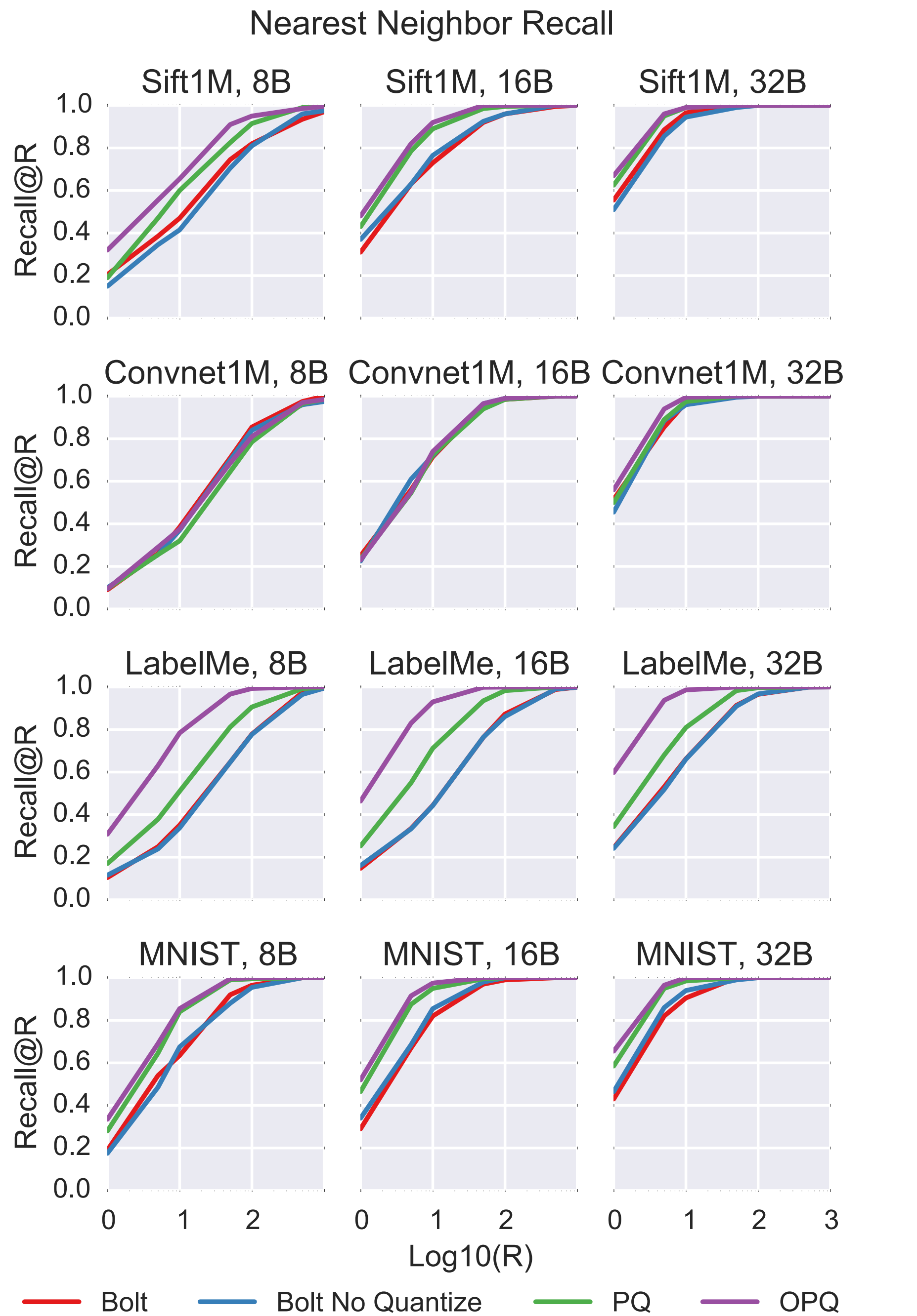
Distance Computation Speed

- Faster than even hardware `popcnt` used by binary embedding
- Matmul{#} refers to batching queries
- 10x faster than PQ/OPQ, 100x more than raw floats (Matmul1)



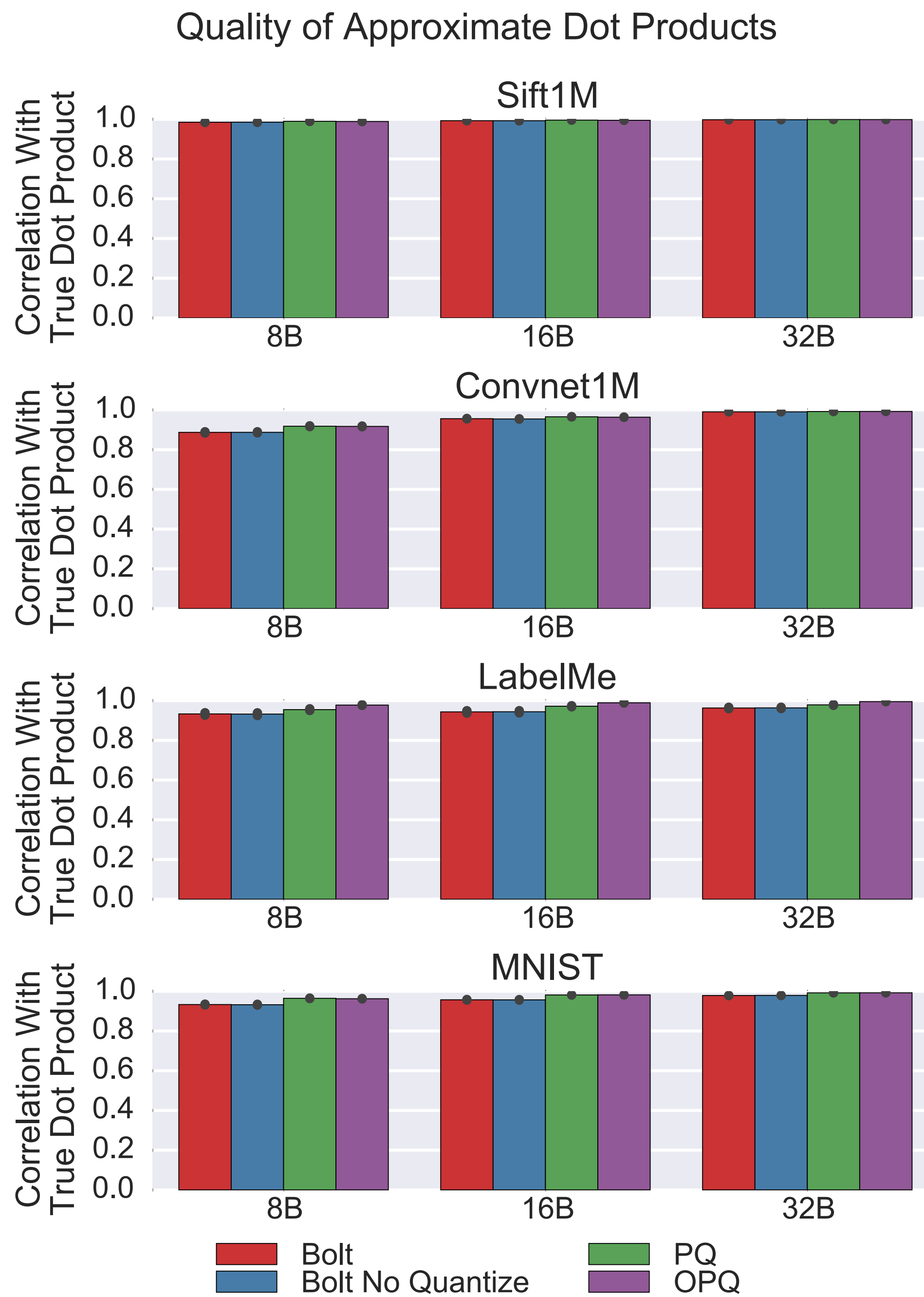
Nearest Neighbor Recall

- Have to retrieve more points than other algorithms for a given level of compression
- Our LUT quantization causes no loss in accuracy



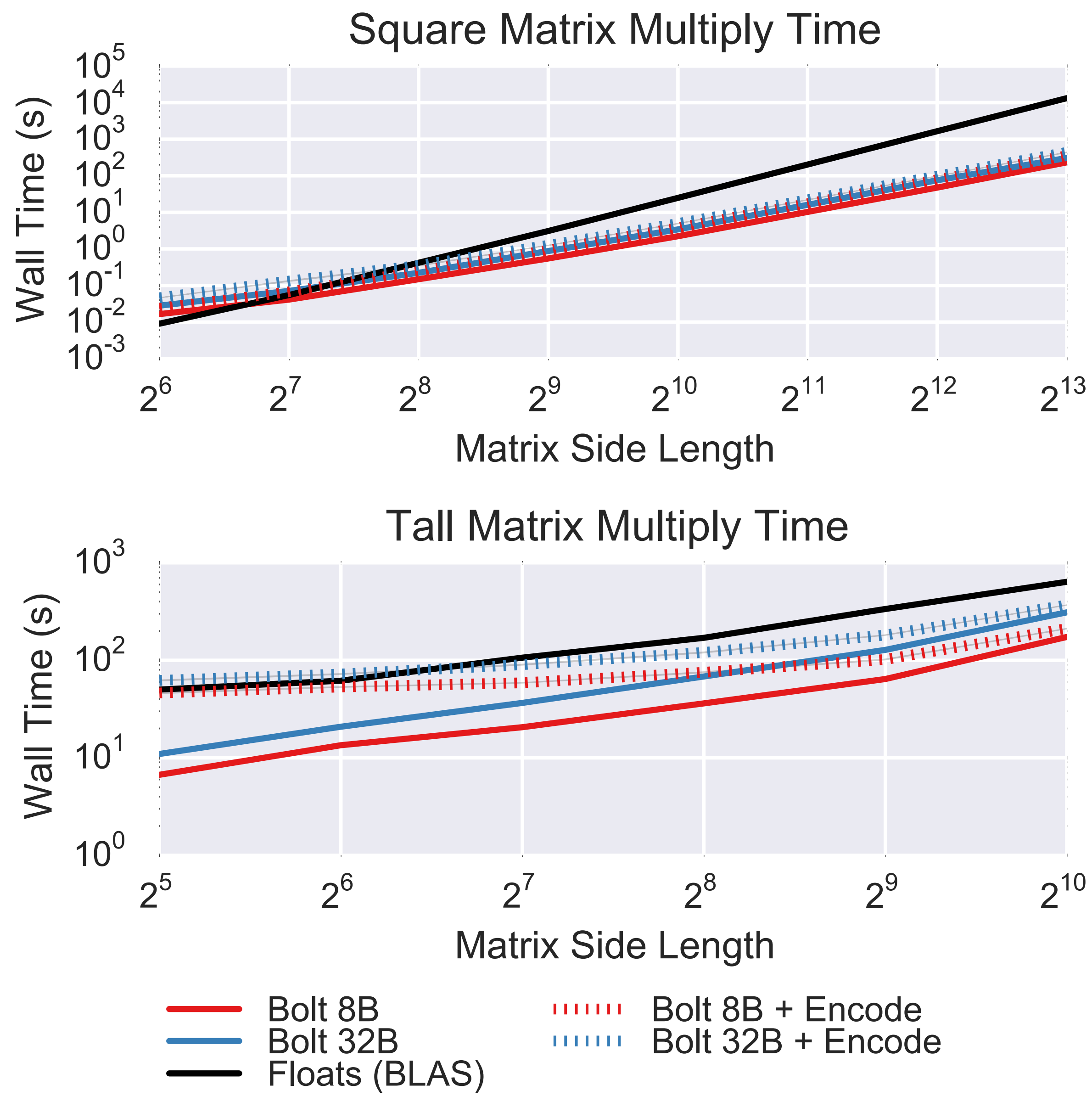
Dot Product Accuracy

- Bolt dot products are slightly less accurate than others
- But highly accurate in absolute terms



Matrix Multiplies

- Multiplying matrices via dot products in a for loop is *faster than BLAS*, even when we must encode the matrices first



Davis Blalock

Conclusion

- Bolt compresses vectors of data at multiple GB/s in a single thread
- Bolt's compressed representation can be used to compute approximate distances and dot products with:
  - High accuracy
  - Greater speed than any other algorithm, by a large margin
- The key is vectorized table lookups enabled by learned quantization functions

Questions?