Building a Scalable Multimedia Search Engine Using Infiniband

Qi Chen

Yisheng Liao, Christopher Mitchell, Jinyang Li, Zhen Xiao

Peking University  NYU
Online search must be scalable

Example Search Engines: Google, Bing
How multimedia search is done

<table>
<thead>
<tr>
<th></th>
<th>Feature 1</th>
<th>Feature 2</th>
<th>...</th>
<th>Feature x</th>
</tr>
</thead>
<tbody>
<tr>
<td>Img 1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Img 2</td>
<td></td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Img n</td>
<td></td>
<td>1</td>
<td></td>
<td>1</td>
</tr>
</tbody>
</table>

billions of images

billions of Features

Indexing (usually done offline)
How multimedia search is done

- **Query image**
- **Search features** $f_1, f_2, \ldots, f_n$

**Indexing** (usually done offline)

<table>
<thead>
<tr>
<th></th>
<th>Feature 1</th>
<th>Feature 2</th>
<th>\ldots</th>
<th>Feature x</th>
</tr>
</thead>
<tbody>
<tr>
<td>Img 1</td>
<td>1</td>
<td>1</td>
<td>\ldots</td>
<td></td>
</tr>
<tr>
<td>Img 2</td>
<td></td>
<td></td>
<td>\ldots</td>
<td>1</td>
</tr>
<tr>
<td>\ldots</td>
<td></td>
<td></td>
<td>\ldots</td>
<td></td>
</tr>
<tr>
<td>\ldots</td>
<td></td>
<td></td>
<td>\ldots</td>
<td></td>
</tr>
<tr>
<td>Img n</td>
<td></td>
<td></td>
<td>\ldots</td>
<td>1</td>
</tr>
</tbody>
</table>

- Billions of images
- Billions of features

Billions of images are indexed to allow for efficient search. Each image is represented by a set of features, and each feature is associated with a score indicating its relevance to the image or other images.
Two ways to distribute: horizontal partition

Not scalable because a query must contact all servers
Two ways to distribute: vertical partition

<table>
<thead>
<tr>
<th>Feature 1</th>
<th>Feature 2</th>
<th>...</th>
<th>...</th>
<th>Feature f</th>
</tr>
</thead>
<tbody>
<tr>
<td>Img 1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Img 2</td>
<td></td>
<td>1</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Img n</td>
<td>1</td>
<td></td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

Expensive because a query may look up tens of thousands of features
Horizontal vs. vertical: State-of-art and new opportunity

- Horizontal beats vertical partitioning on the Ethernet
- But..
- Ultra-low latency network is coming to data centers
  - InfiniBand, RoCE
  - RTT ≈ 10us. (compared to RTT >100us on Ethernet)
- Our insight: Vertical beats horizontal on low-latency networks
  - Why latency matters: Use more roundtrips to reduce feature lookups
Outlines

1. Motivation
2. VertiCut Design
3. Evaluation
4. Related Work
Overview of VertiCut Image Search

- Indexing
  - Offline indexer transforms images to 128-bit binary codes

- Searching
  - Online k-nearest-nbr (KNN) algorithm finds k codes with smallest hamming distance to a query code
How to do KNN in binary space?

- VertiCut uses Multi-index Hashing [CVPR ’12]
- To index
  - Break a 128-bit code into 4 pieces
  - Insert i-th piece in hash table Ti

\[
\text{Code}(x) = 011\ldots111 \ 000\ldots101 \ 000\ldots101 \ 001\ldots110
\]
How to do KNN in binary space?

- VertiCut uses Multi-index Hashing [CVPR’ 12]
- To index
  - Break a 128-bit code into 4 pieces
  - Insert i-th piece in hash table Ti

\[
\text{Code}(x) = 011\ldots111 | 000\ldots101 | 000\ldots101 | 001\ldots110
\]
How to do KNN in binary space?

- VertiCut uses Multi-index Hashing [CVPR ’12]
- To index
  - Break a 128-bit code into 4 pieces
  - Insert i-th piece in hash table Ti

Code(x) = 011…111 | 000…101 | 000…101 | 001…110

<table>
<thead>
<tr>
<th>index</th>
<th>img list</th>
</tr>
</thead>
<tbody>
<tr>
<td>000…000</td>
<td>…</td>
</tr>
<tr>
<td>…</td>
<td>…</td>
</tr>
<tr>
<td>011…111</td>
<td>…</td>
</tr>
<tr>
<td>…</td>
<td>…</td>
</tr>
</tbody>
</table>

T1
How to do KNN in binary space?

- VertiCut uses Multi-index Hashing [CVPR ’12]
- To index
  - Break a 128-bit code into 4 pieces
  - Insert i-th piece in hash table Ti

$$\text{Code}(x) = \begin{array}{c}
011\ldots111 \\
000\ldots101 \\
000\ldots101 \\
001\ldots110
\end{array}$$

<table>
<thead>
<tr>
<th>index</th>
<th>img list</th>
</tr>
</thead>
<tbody>
<tr>
<td>000...000</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>011...111</td>
<td>... , Code(x)</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

T1
How to do KNN in binary space?

- VertiCut uses Multi-index Hashing [CVPR ’12]
- To index
  - Break a 128-bit code into 4 pieces
  - Insert i-th piece in hash table Ti

\[ \text{Code}(x) = 011\ldots111 \quad 000\ldots101 \quad 000\ldots101 \quad 001\ldots110 \]
How to do KNN in binary space?

- VertiCut uses Multi-index Hashing [CVPR’ 12]
- To index
  - Break a 128-bit code into 4 pieces
  - Insert i-th piece in hash table Ti

\[
\text{Code}(x) = 011\ldots111 \| 000\ldots101 \| 000\ldots101 \| 001\ldots110
\]
How to do KNN in binary space?

- VertiCut uses Multi-index Hashing [CVPR ’12]
- To index
  - Break a 128-bit code into 4 pieces
  - Insert i-th piece in hash table Ti

\[
\text{Code}(x) = 011\ldots111 000\ldots101 000\ldots101 001\ldots110
\]
VertiCut search architecture

\[ Q = 011\ldots110 \]

Search nodes

Get (~10us)

Pilaf DHT [USENIX ATC'13]
How to do KNN in binary space?

- To search 100 KNNs given a query code q

query code q

00…11 01…01 10…01 11…10
How to do KNN in binary space?

- To search 100 KNNs given a query code $q$

  Find KNNs with hamming distance $< 4$

<table>
<thead>
<tr>
<th>query code $q$</th>
</tr>
</thead>
<tbody>
<tr>
<td>00⋯11</td>
</tr>
<tr>
<td>01⋯01</td>
</tr>
<tr>
<td>10⋯01</td>
</tr>
<tr>
<td>11⋯10</td>
</tr>
</tbody>
</table>
How to do KNN in binary space?

To search 100 KNNs given a query code \( q \)

Find KNNs with hamming distance < 4

For each hash table \( T_i \)
  \( S \leftarrow \text{Enum indices with distance} = 0 \)
  For each \( idx \) in \( S \):
    \( C \leftarrow C \cup T_i.\text{lookup}(idx) \)
How to do KNN in binary space?

To search 100 KNNs given a query code \( q \)

Find KNNs with hamming distance < 4

For each hash table \( T_i \)

\[ S \leftarrow \text{Enum indices with distance} = 0 \]

For each \( idx \) in \( S \):

\[ C \leftarrow C \cup T_i.\text{lookup}(idx) \]
How to do KNN in binary space?

To search 100 KNNs given a query code q

Find KNNs with hamming distance < 4

For each hash table Ti
  S ← Enum indices with distance = 0
  For each idx in S:
    C ← C ∪ Ti.lookup(idx)
How to do KNN in binary space?

To search 100 KNNs given a query code $q$

Find KNNs with hamming distance $< 4$

For each hash table $T_i$

$S \leftarrow$ Enum indices with distance $= 0$

For each idx in $S$:

$C \leftarrow C \cup T_i.\text{lookup}(\text{idx})$
How to do KNN in binary space?

To search 100 KNNs given a query code q

Find KNNs with hamming distance < 4

For each hash table Ti
  S ← Enum indices with distance = 0
  For each idx in S:
    C ← C ∪ Ti.lookup(idx)

For each image code x in C:
  if distance(x, q) < 4:
    add x to result
  if |result| >= 100:
    return KNN in result
How to do KNN in binary space?

- To search 100 KNNs given a query code $q$

Find KNNs with hamming distance $< 8$

query code $q$

$00\ldots11$  $01\ldots01$  $10\ldots01$  $11\ldots10$
How to do KNN in binary space?

To search 100 KNNs given a query code q

Find KNNs with hamming distance \(< 8\)

For each hash table Ti:
- \(S \leftarrow \text{Enum indices with distance} = 1\)
- For each idx in S:
  - \(C \leftarrow C \cup Ti.\text{lookup}(idx)\)

query code q

00...11 01...01 10...01 11...10
How to do KNN in binary space?

To search 100 KNNs given a query code $q$

Find KNNs with hamming distance $< 8$

For each hash table $T_i$

- $S \leftarrow$ Enum indices with distance $= 1$
- For each $idx$ in $S$:
  - $C \leftarrow C \cup T_i.lookup(idx)$
How to do KNN in binary space?

To search 100 KNNs given a query code $q$

Find KNNs with hamming distance < 8

For each hash table $T_i$

$S \leftarrow$ Enum indices with distance = 1

For each $idx$ in $S$:

$C \leftarrow C \cup T_i.$lookup($idx$)
How to do KNN in binary space?

To search 100 KNNs given a query code $q$

Find KNNs with hamming distance $< 8$

For each hash table $T_i$

$S \leftarrow \text{Enum indices with distance } = 1$

For each $idx$ in $S$:

$C \leftarrow C \cup T_i\text{.lookup}(idx)$
How to do KNN in binary space?

To search 100 KNNs given a query code $q$

Find KNNs with hamming distance $< 8$

For each hash table $T_i$
  $S \leftarrow$ Enum indices with distance $= 1$
  For each idx in $S$:
    $C \leftarrow C \cup T_i.lookup(idx)$

For each image code $x$ in $C$:
  if $\text{distance}(x, q) < 8$:
    add $x$ to result
  if $|\text{result}| \geq 100$:
    return KNN in result
How to do KNN in binary space?

To search 100 KNNs given a query code q

For each d = 4, 8, 12, 16, ….

For each hash table Ti
   S ← Enum indices with distance = \( \frac{d}{4} - 1 \)
   For each idx in S:
      C ← C ∪ Ti.lookup(idx)

For each image code x in C:
   if distance(x, q) < d :
      add x to result
   if |result| ≥ 100:
      return KNN in result
To search 100 KNNs given a query code $q$

For each $d = 4, 8, 12, 16, \ldots$.

For each hash table $T_i$
- $S \leftarrow$ Enum indices with distance $= \frac{d}{4} - 1$
- For each idx in $S$:
  - $C \leftarrow C \cup T_i$.lookup(idx)

For each image code $x$ in $C$:
- if $\text{distance}(x, q) < d$:
  - add $x$ to result
- if $|\text{result}| \geq 100$:
  - return KNN in result

Problem:
Large $d \rightarrow$ numerous (combinatorial) lookups

Typically, $d=20 \rightarrow$ #lookups = 165K
Optimization #1: approx. KNN

To search 100 KNNs given a query code q

For each $d = 4, 8, 12, 16, \ldots$.

For each hash table $T_i$:
\[ S \leftarrow \text{Enum indices with distance } = \frac{d}{4} - 1 \]
For each idx in $S$:
\[ C \leftarrow C \cup T_i.\text{lookup}(\text{idx}) \]

For each image code $x$ in $C$:
\[ \text{if } \text{distance}(x, q) < d: \]
\[ \text{add } x \text{ to result} \]
\[ \text{if } |\text{result}| \geq 100: \]
\[ \text{return KNN in result} \]

Problem:
Large $d \rightarrow$ numerous (combinatorial) lookups

Typically, $d=20 \rightarrow$
#lookups = 165K

Our insight:
• Stop search as soon as the candidate set $C$ is big enough
• KNNs in $C$ approximates the true KNNs
Optimization #1: approx. KNN

To search 100 KNNs given a query code q

For each \( d = 4, 8, 12, 16, \ldots \).

For each hash table \( T_i \)

\[ S \leftarrow \text{Enum indices with distance } = \frac{d}{4} - 1 \]

For each idx in \( S \):

\[ C \leftarrow C \cup T_i.\text{lookup}(\text{idx}) \]

if \( |C| \geq f \times 100 \):

return KNN in result

Our insight:

- Stop search as soon as the candidate set \( C \) is big enough
- KNNs in \( C \) approximates the true KNNs
Optimization #1: approx. KNN

Experiments show:

- To obtain k results, we can stop search when $|C| > 20 \times k$
- Results contain 80% of true KNNs
- Avg. distance of results is close to that of true KNNs (<1)
- Reduces # of lookups by a factor of 40
Optimization #2: avoid null lookups

- Observation: >90% lookups return empty result
- Each search node keeps a bitmap for each hash table
  - Do a lookup in DHT only after the bitmap returns a hit
  - Bitmap size (4*500MB) does not increase with # of images indexed

\[ Q = 011\ldots110 \]
Experiment Environment

- Experimental Setup

- 12 servers connected with 20Gbps Infiniband

- 1 billion image descriptors from BIGANN dataset

- Each query retrieves 1000 KNNs
Vertical scales better than Horizontal

- ~22000 DHT gets ~5500 RTTs
- ~10800 DHT gets ~2700 RTTs

10 million images

120 million images
VertiCut is only feasible on low-latency network

![Graph showing query latency with number of servers and latency comparison between Ethernet and Infiniband](image)

- ~2700 RTTs
- 8 times slower on Ethernet
Effects of Optimizations

# of DHT lookups

<table>
<thead>
<tr>
<th>No opt</th>
<th>Approx</th>
<th>Bitmap</th>
<th>VertiCut</th>
</tr>
</thead>
<tbody>
<tr>
<td>60.5 s</td>
<td>0.75 s</td>
<td>2.3 s</td>
<td>0.11 s</td>
</tr>
</tbody>
</table>

550X latency reduction
Related Work

- Bag-of-features based search
  - Ji et al. [TM’13], MARÉE et al. [MIR’10], YAN et al. [SenSys’08], MIH [CVPR’12], Rankreduce [LSDS-IR’10]
  - Traditionally use horizontal partition for distribution

- High-dimensional search trees (e.g. KD-tree)
  - ALY et al. [BMVC’11]
  - Build a distributed tree offline → Cannot be incrementally updated
Conclusion

- Ultra low-latency networks allow vertical partition to perform better than traditional horizontal partition
- VertiCut: a scalable image search engine
  - Built on top of Pilaf DHT on Infiniband
  - Use two optimizations to reduce DHT lookups
    - Approximate nearest neighbor search
    - Eliminate empty lookups
Thank You!