Billion-Scale Similarity Search with GPUs

Jeff Johnson, Matthijs Douze, and Hervé Jégou, *Senior Member, IEEE*

Abstract—Similarity search finds application in database systems handling complex data such as images or videos, which are typically represented by high-dimensional features and require specific indexing structures. This paper tackles the problem of better utilizing GPUs for this task. While GPUs excel at data parallel tasks such as distance computation, prior approaches in this domain are bottlenecked by algorithms that expose less parallelism, such as k-min selection, or make poor use of the memory hierarchy. We propose a novel design for k-selection. We apply it in different similarity search scenarios, by optimizing brute-force, approximate and compressed-domain search based on product quantization. In all these setups, we outperform the state of the art by large margins. Our implementation operates at up to 55 percent of theoretical peak performance, enabling a nearest neighbor implementation that is 8.5 × faster than prior GPU state of the art. It enables the construction of a high accuracy k-NN graph on 95 million images from the YFCC100M dataset in 35 minutes, and of a graph connecting 1 billion vectors in less than 12 hours on 4 Maxwell Titan X GPUs. We have open-sourced our approach for the sake of comparison and reproducibility.

Index Terms—Similarity search, multimedia databases, indexing methods, graphical processing units

1 Introduction

Images and videos constitute a new massive source of data for indexing and search. Traditional media management systems are based on relational databases built on structured data. For example, an image is indexed by metadata like capture time and location, with possible manual additions like the names of people represented within. Images can thus be queried by name, date or location. This metadata can make it possible to automatically organize photo albums.

For large media collections, such metadata is harder to come by; producing content is so easy that data annotation is a significant bottleneck. A variety of machine learning and deep learning algorithms are being used to automatically interpret and annotate these complex, real-world entities. They produce representations or *embeddings*, typically real-valued, high-dimensional vectors of 50 to 1000+ dimensions. Popular examples include the text representations word2-vec [43] and FastText [34], image representations extracted from convolutional neural networks [25], [49], and image descriptors for instance search [7], [26], [53]. A traditional relational database cannot effectively deal with these descriptors, as they require machine learning tools like fuzzy matching, classifiers and similarity search.

Many of these vector representations can only effectively be produced on GPU systems, as the underlying processes either have high arithmetic complexity and/or high data bandwidth demands [36], or cannot be effectively partitioned due to communication overhead or representation quality [48]. Once produced, their manipulation is itself arithmetically intensive. How to utilize GPU assets is not straightforward. More generally, how to exploit new heterogeneous disk/CPU/GPU/FPGA architectures is a key subject for the database community [11].

In this context, searching by numerical *similarity* rather than via structured relations is more suitable. This could be to find the most similar content to a picture, or to find vectors that have the highest response to a linear classifier. One of the most expensive operations to be performed on large collections is to compute a k-NN graph, a directed graph where each vector of the database is a node and each edge connects a node to its k nearest neighbors. This is our flagship application. State of the art methods like NN-Descent [18] for this task have a large memory overhead on top of the dataset itself and cannot readily scale to the billion-sized databases we consider.

Such applications must deal with the *curse of dimensionality* [57], rendering both exhaustive search and exact indexing for non-exhaustive search impractical on billion-scale databases. This is why there is a large body of work on approximate search and/or graph construction. To handle huge datasets that do not fit in RAM, several approaches employ compressed representations of the vectors using an encoding. This is especially convenient for memory-limited devices like GPUs. It turns out that accepting a minimal accuracy loss can result in orders of magnitude of compression [28]. The most popular vector compression methods can be classified into either binary codes [24], [29], or quantization methods [32], [47]. Both have the desirable property that searching neighbors does not require reconstructing the vectors.

Traditional relational databases are stored on disk. However, we aim at response times of around 10 ms for operations that access hundreds of megabytes of data. Therefore, we will consider only databases that are stored in RAM.

- *J. Johnson is with Facebook AI Research, New York, NY 10003 USA. E-mail: jhj@fb.com.*
- *M. Douze and H. Jégou are with Facebook AI Research, Paris, France. E-mail: {matthijs, rvj}@fb.com.*

Manuscript received 30 Oct. 2017; revised 18 Apr. 2019; accepted 23 May 2019. Date of publication 7 June 2019; date of current version 29 June 2021.
(Corresponding author: Jeff Johnson.)
Recommended for acceptance by Y. Xia.
Digital Object Identifier no. 10.1109/TBDATA.2019.2921572

Modern servers routinely have tens to hundreds of gigabytes of memory, feasible for datasets of billions of entries, or trillions when distributed across multiple servers. Therefore, in the following we are concerned with search speed, for a given memory budget. Disk I/O performance is not relevant.

This paper focuses on methods based on product quantization (PQ) codes. They were shown to be more effective than Locality Sensitive Hashing (LSH) [15] or other variants producing binary codes [24], [37]. The binary coding variant of LSH is sub-optimal because it discards the norm of the vector components, in contrast to PQ, which satisfies Lloyd's conditions of optimality for a quantizer. LSH is also used as a partitioning technique. In particular E^{2}LSH provides a set of hash functions adapted to euclidean search [2]. However, this approach requires encoding the full dataset into several tables, with significant memory overhead. In contrast, PQ variants use a single code per vector. Side-by-side comparisons of LSH and PQ have validated PQ's superiority [27], [45]. PQ is particularly effective when query vectors are not encoded. There is a natural extension of the algorithm that does non-exhaustive search [32].

Several improvements were proposed over the original PQ-based technique, but most are difficult to implement efficiently on GPU. The inverted multi-index [5], useful for high-speed/low-quality operating points, depends on a complicated "multi-sequence" algorithm. The optimized product quantization or OPQ [23] is a linear transformation on the input vectors that improves the accuracy of the product quantization; it can be applied as a pre-processing. The SIMD-optimized implementation from André et al. [3] operates only with sub-optimal parameters (few coarse quantization centroids). Many other methods, like LOPQ and the Polysemous codes [20], [35] are too complex to be implemented efficiently on GPUs.

There are many implementations of similarity search on GPUs, but mostly with binary codes [46], small datasets [55], or exhaustive search [17], [50], [51]. To the best of our knowledge, only the work by Wieschollek et al. [58] appears suitable for billion-scale datasets with quantization codes. This is the prior state of the art on GPUs, which we compare against in Section 6.4 on the largest (billion-scale) public benchmarks for similarity search.

This paper makes the following contributions:

- a GPU k-selection algorithm, operating in fast register memory and flexible enough to be fusable with other kernels, for which we provide a complexity analysis. Hitherto this has been the limiting factor for similar GPU database applications;
- a near-optimal algorithmic layout for exact and approximate k-nearest neighbor search on GPU;
- a range of experiments that show that these improvements outperform previous art by a large margin on mid- to large-scale nearest-neighbor search tasks, in single or multi-GPU configurations.

A carefully engineered implementation of this paper's algorithms can be found in the open-source Faiss library. It implements many state-of-the-art indexing methods, and the most relevant algorithms are translated to the GPU. The Faiss repository (https://github.com/facebookresearch/faiss) contains the source scripts that reproduce most results of this paper.

The designs presented in this paper as implemented in Faiss have been applied to a wide variety of tasks. In a recent natural language processing work[38], k-NN search is used to match word embeddings, providing for translation between different languages without parallel texts. Caron et al. [14] employ the GPU clustering methods to unsupervised training of embeddings. In [19], [21], huge k-NN graphs are used for image classification. These applications are possible thanks to the order-of-magnitude performance improvement brought by the GPU implementation.

The paper is organized as follows. Section 2 introduces the context and notation. Section 3 reviews GPU architecture and discusses problems appearing when using it for similarity search. Section 4 introduces one of our main contributions, i.e., our GPU k-selection method, while Section 5 covers overall algorithm implementation. Finally, Section 6 provides extensive experiments for our approach, compares it to the state of the art, and shows concrete use cases for image collections.

2 Problem Statement

We are concerned with similarity search in vector collections. Given a query vector $\mathbf{x} \in \mathbb{R}^d$ and a database vector collection[1] $[\mathbf{y}_i]_{i=0:\ell}$ ($\mathbf{y}_i \in \mathbb{R}^d$), we search the k nearest neighbors of $\mathbf{x}$ in terms of L_2 (euclidean) distance:

$$L = k\text{-argmin}_{i=0:\ell} \|\mathbf{x} - \mathbf{y}_i\|_2. \qquad (1)$$

L_2 distance is used most often, as it is optimized by design when learning several embeddings [26], due to its attractive linear algebraic properties.

The minimum distances are collected by k-selection. For a scalar array $[a_i]_{i=0:\ell}$, k-selection finds the k lowest valued elements $[a_{s_i}]_{i=0:k}$, $a_{s_i} \leq a_{s_{i+1}}$, along with the indices $[s_i]_{i=0:k}$, $0 \leq s_i < \ell$, of those elements from the input array. The a_i will be 32-bit floating point values; the s_i are 32- or 64-bit integers. Other comparators are sometimes desired; for cosine similarity we search for *highest* values. The order between equivalent values $a_{s_i} = a_{s_j}$ is not specified.

2.1 Exact Search with Batching

Typically, searches are performed in batches of n_{q} query vectors $[\mathbf{x}_j]_{j=0:n_{\mathrm{q}}}$ ($\mathbf{x_j} \in \mathbb{R}^d$) in parallel, which allows for more flexibility when executing on multiple CPU threads or on GPU. Batching for k-selection entails selecting $n_{\mathrm{q}} \times k$ elements and indices from n_{q} separate arrays, where each array is of a potentially different length $\ell_i \geq k$.

The exact solution computes the full pairwise distance matrix $D = [\|\mathbf{x}_j - \mathbf{y}_i\|_2^2]_{j=0:n_{\mathrm{q}}, i=0:\ell} \in \mathbb{R}^{n_{\mathrm{q}} \times \ell}$. In practice, we use the decomposition

$$\|\mathbf{x}_j - \mathbf{y}_i\|_2^2 = \|\mathbf{x}_j\|^2 + \|\mathbf{y}_i\|^2 - 2\langle \mathbf{x}_j, \mathbf{y}_i \rangle. \qquad (2)$$

The two first terms are precomputed in one pass over the matrices X and Y whose rows are the $[\mathbf{x}_j]$ and $[\mathbf{y}_i]$. The

[1] To avoid clutter in 0-based indexing, we use the Python array notation $0:\ell$ to denote the range $\{0, \ldots, \ell-1\}$ inclusive.

bottleneck is to evaluate $\langle \mathbf{x}_j, \mathbf{y}_i \rangle$, equivalent to the matrix multiplication $XY^\top$. The k-nearest neighbors for each of the n_q queries are k-selected along each row of D.

2.2 Compressed-Domain Search

From now on, we focus on approximate nearest-neighbor search. We consider, in particular, the *IVFADC* indexing structure [32]. The IVFADC index relies on two levels of quantization, and the database vectors are encoded. The database vector $\mathbf{y}$ is approximated as:

$$\mathbf{y} \approx q(\mathbf{y}) = q_1(\mathbf{y}) + q_2(\mathbf{y} - q_1(\mathbf{y})), \tag{3}$$

where $q_1 : \mathbb{R}^d \rightarrow \mathcal{C}_1 \subset \mathbb{R}^d$ and $q_2 : \mathbb{R}^d \rightarrow \mathcal{C}_2 \subset \mathbb{R}^d$ are quantizers; i.e., functions that output an element from a finite set. Since the sets are finite, $q(\mathbf{y})$ is encoded as the index of $q_1(\mathbf{y})$ and that of $q_2(\mathbf{y} - q_1(\mathbf{y}))$. The first-level quantizer is a *coarse quantizer* and the second level a *fine quantizer* that encodes the residual vector after the first level.

The Asymmetric Distance Computation (ADC) search method returns an approximate result:

$$L_{\mathrm{ADC}} = k\text{-argmin}_{i=0:\ell} \|\mathbf{x} - q(\mathbf{y}_i)\|_2. \tag{4}$$

For IVFADC the search is not exhaustive. Vectors for which the distance is computed are preselected depending on the first-level quantizer q_1:

$$L_{\mathrm{IVF}} = \tau\text{-argmin}_{\mathbf{c} \in \mathcal{C}_1} \|\mathbf{x} - \mathbf{c}\|_2. \tag{5}$$

The *multi-probe parameter* τ is the number of coarse-level centroids we consider. The quantizer operates a nearest-neighbor search with exact distances, in the set of reproduction values. Then, the IVFADC search computes

$$L_{\mathrm{IVFADC}} = \underset{i=0:\ell \text{ s.t. } q_1(\mathbf{y}_i) \in L_{\mathrm{IVF}}}{k\text{-argmin}} \|\mathbf{x} - q(\mathbf{y}_i)\|_2. \tag{6}$$

Hence, IVFADC relies on the same distance estimations as the two-step quantization of ADC, but computes them only on a subset of vectors.

The corresponding data structure, the *inverted file*, groups the vectors $\mathbf{y_i}$ into $|\mathcal{C}_1|$ *inverted lists* $\mathcal{I}_1, \ldots, \mathcal{I}_{|\mathcal{C}_1|}$ with homogeneous $q_1(\mathbf{y}_i)$. Therefore, the most memory-intensive operation is computing L_{IVFADC}, and amounts to linearly scanning τ inverted lists.

The Quantizers. q_1 and q_2 have different properties. The quantizer q_1 needs to have a relatively low number of reproduction values so that the number of inverted lists does not explode. We typically use $|C_1| \approx \sqrt{\ell}$, trained via k-means. For q_2, we can afford more memory for a more extensive representation. The vector index (a 4- or 8-byte integer) is also stored in the inverted lists, so it makes no sense to have shorter codes than that; i.e., $\log_2 |\mathcal{C}_2| > 4 \times 8$.

Product Quantizer. We use a *product quantizer* (PQ) [32] for q_2, providing a large number of reproduction values for a limited memory and computational cost. It interprets $\mathbf{y}$ as b sub-vectors $\mathbf{y} = [\mathbf{y}^1 \ldots \mathbf{y}^b]$, where b is an even divisor of the dimension d. Each sub-vector is quantized with its own quantizer, yielding $(q^1(\mathbf{y}^1), \ldots, q^b(\mathbf{y}^b))$. The sub-quantizers typically have 256 reproduction values to fit in one byte. The quantization value of the product quantizer is then $q_2(\mathbf{y}) = q^1(\mathbf{y}^1) + 256 \times q^2(\mathbf{y}^2) + \ldots + 256^b \times q^b(\mathbf{y}^b)$, which from a storage point of view is just the concatenation of the bytes produced by each sub-quantizer. Thus, the product quantizer generates b-byte codes with $|\mathcal{C}_2| = 256^b$ reproduction values. The k-means dictionaries of the quantizers are small and quantization is computationally cheap.

3 GPU: Overview and k-Selection

This section reviews salient details of Nvidia's GPU architecture and programming model [40]. We then focus on less GPU-compliant parts involved in similarity search, namely k-selection, and discuss the literature and challenges.

3.1 Architecture

GPU Lanes and Warps. The Nvidia GPU is a general-purpose computer that executes instruction streams using a 32-wide vector of *CUDA threads* (the *warp*). The individual threads in the warp are referred to as *lanes*, with a *lane ID* in the range 0 – 31. Lanes within a single warp share a single warp-wide instruction counter. When warp lanes wish to take different execution paths despite the shared instruction counter, *warp divergence* occurs, reducing performance. Each lane has up to 255 32-bit registers in a shared register file. A CPU analogy is that each warp is a separate CPU hardware thread, with up to 255 SIMD vector registers of width 32, with warp lanes as SIMD vector lanes.

Collections of Warps. A configurable collection of 1 to 32 warps comprises a *block* or a *co-operative thread array* (CTA). Each block has a high speed *shared memory*, up to 48 KiB in size. Individual CUDA threads have a block-relative ID, called a *thread id*, which can be used to partition and assign work. Each block is run on a single core of the GPU called a *streaming multiprocessor* (SM), with functional units such as ALUs for execution. A GPU hides execution latencies by having many operations in flight on warps across all SMs. Each individual warp lane instruction throughput is low and latency is high, but the aggregate arithmetic throughput of all SMs together is 5 – 10× higher than typical CPUs.

Grids and Kernels. Blocks are organized in a *grid* of blocks in a *kernel*. Each block is assigned a grid relative ID. The kernel is the unit of work (instruction stream with arguments) scheduled by the host CPU for the GPU to execute. After a block runs through to completion, new blocks can be scheduled. Blocks from different kernels can run concurrently. Ordering between kernels is controllable via ordering primitives such as *streams* and *events*.

Resources and Occupancy. The number of blocks executing concurrently depends upon shared memory and register resources used by each block. CUDA thread register usage is determined at compilation time, while shared memory usage can be chosen at runtime. This affects *occupancy* on the GPU; greater register or shared memory resources in use will reduce execution concurrency.

Memory Types. Different blocks and kernels communicate through *global memory*, typically 4 – 32 GB in size, with 5 – 10 × higher bandwidth than CPU main memory. Shared memory is analogous to CPU L1 cache in terms of speed. GPU register file memory is the highest bandwidth memory. In order to maintain the high number of instructions in flight on a GPU, a vast register file is also required: 14 MB in the latest Pascal P100, in contrast with a few tens of KB on CPU. A ratio of $250 : 6.25 : 1$ for register to shared to global memory aggregate cross-sectional bandwidth is typical on GPU, yielding 10 – 100s of TB/s for the register file [12].

3.2 GPU Register File Usage

Structured Register Data. Shared and register memory usage involves efficiency tradeoffs. They lower occupancy but increase overall performance by retaining a larger working set in a faster memory. Making heavy use of register-resident data at the expense of occupancy or instead of shared memory is often profitable [54].

As the GPU register file is very large, storing structured data (not just temporary operands) is useful. A single lane can use its (scalar) registers to solve a local task, but with limited parallelism and storage. Instead, lanes in a warp can exchange register data using *warp shuffles*, enabling warp-wide parallelism and storage. A wide variety of access patterns (shift, any-to-any) are provided. In particular, we use the butterfly permutation [39] extensively.

Lane-stride Register Array. Warp shuffles are frequently used to manipulate *lane-stride register arrays*. That is, given elements $[a_i]_{i=0:\ell}$, each successive value is held in a register by neighboring lanes. The array is stored in $\ell/32$ registers per lane, with ℓ a multiple of 32. Lane j stores $\{a_j, a_{32+j}, \ldots, a_{\ell-32+j}\}$, while register r holds $\{a_{32r}, a_{32r+1}, \ldots, a_{32r+31}\}$.

For manipulating the $[a_i]$, the register in which a_i is stored (i.e., $\lfloor i/32 \rfloor$) and ℓ must be known at assembly time, while the lane (i.e., $i \bmod 32$) can be runtime knowledge. Thus, all configurations must be handled at compile time with extensive C++ templatization.

3.3 k-Selection on CPU versus GPU

k-selection has been a limiting performance factor for prior GPU similarity search applications (see Section 6), thus it deserves some discussion. Common CPU k-selection algorithms, often for arbitrarily large ℓ and k, can be translated to a GPU, including *radix selection* and *bucket selection* [1], *probabilistic selection* [44], *quickselect* [17], and *truncated sorts* [50]. Their performance is dominated by multiple passes over the input. For similarity search, input distances are typically computed on-the-fly or stored only in small blocks, not in their entirety. The full distance array may be too large to fit into any memory, and its size could be unknown at the start of the processing, making multiple pass algorithms impractical. Furthermore, algorithms that partition elements in global memory based on their value such as Quickselect result in excessive memory transactions as the warp-wide data access pattern is not uniform. Radix selection has no partitioning but multiple passes are still required.

Heap Parallelism. For similarity search, one is usually interested in a small number ($k < 1000$) of results. In this regime, selection via max-heap is a typical on CPU, but heaps do not expose much data parallelism due to serial tree update, and cannot saturate SIMD execution units. The *ad-heap* [41] takes better advantage of parallelism in heterogeneous systems, but still attempts to partition serial and parallel work between appropriate execution units. Despite the serial nature of heap update, for small k a CPU can maintain all of its state in the L1 cache with little effort, and L1 cache latency and bandwidth remains a limiting factor. Other similarity search components, like PQ code manipulation, tend to have greater impact on CPU performance [3].

GPU Heaps. Heaps can be implemented on a GPU [9], yet a straightforward GPU implementation has high warp divergence and irregular, data-dependent memory movement, since the path taken for each inserted element depends upon other values present in the heap.

GPU parallel priority queues [31] improve over serial heap update by allowing multiple concurrent updates, but they require a potential number of small sorts for each insert and data-dependent memory movement. They also require a significant coordination with the CPU host.

Other more novel GPU algorithms are available for small k, namely the selection algorithm in the *fgknn* library [51]. This is a complex algorithm that may suffer from too many synchronization points, greater kernel launch overhead, usage of slower memories, excessive use of hierarchy, partitioning and buffering. However, we take inspiration from this particular algorithm through the use of parallel merges as seen in their *merge queue* structure.

4 Fast k-Selection on the GPU

For any CPU or GPU algorithm, either memory or arithmetic throughput should be the limiting factor as per the *roofline performance model* [59]. For input from global memory, k-selection cannot run faster than the time required to scan the input once at peak memory bandwidth. We aim to get as close to this limit as possible. Thus, we wish to perform a single pass over the input data.

We want to keep intermediate state in the fastest memory, namely the register file. The main drawback of doing so is the lane-stride register array indexing constraint mentioned in Section 3.2, providing limitations on algorithm feasibility.

4.1 In-Register Sorting

We use an in-register sorting primitive as a building block. Sorting networks are commonly used on SIMD architectures [16], as they exploit vector parallelism. They are easily implemented on the GPU, and we build sorting networks with lane-stride register arrays.

We use a variant of *Batcher's bitonic sorting network* [10], which is a set of parallel merges on an array of size 2^k. Each merge takes s arrays of length t (s and t a power of 2) to $s/2$ arrays of length $2t$, using $\log_2(t)$ parallel steps. A bitonic sort applies this merge recursively: to sort an array of length ℓ, merge ℓ arrays of length 1 to $\ell/2$ arrays of length 2, to $\ell/4$ arrays of length 4, successively to 1 sorted array of length ℓ, leading to $\frac{1}{2}(\log_2(\ell)^2 + \log_2(\ell))$ parallel merge steps.

Odd-size Merging and Sorting Networks. If some input data is already sorted, we can modify the network to avoid merging steps. If we do not have a full power-of-2 set of data, we efficiently shortcut to deal with the smaller size.

Algorithm 1 is an odd-sized merging network that merges already sorted *left* and *right* arrays, each of arbitrary length. While the bitonic network merges *bitonic* sequences, we start with *monotonic* sequences: sequences sorted monotonically. A bitonic merge is made monotonic by reversing the first comparator stage.

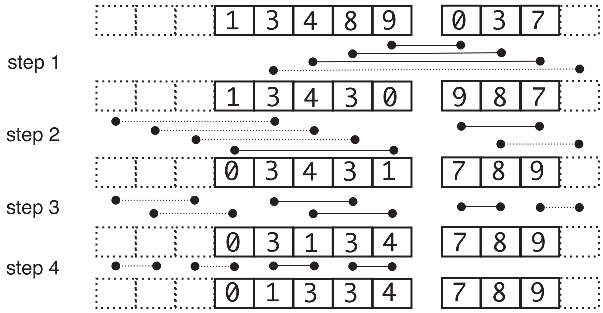

Fig. 1. Odd-size network merging arrays of sizes 5 and 3. Bullets indicate parallel compare/swap. Dashed lines are elided elements or comparisons.

Algorithm 1. Odd-Size Merging Network

```
function MERGE-ODD($[L_i]_{i=0:\ell_L}, [R_i]_{i=0:\ell_R}$)
  parallel for $i \leftarrow 0 : \min(\ell_L, \ell_R)$ do
    ▷ inverted 1st stage; inputs are already sorted
    COMPARE-SWAP$L_{\ell_L-i-1}, R_i$
  end for
  parallel do
    ▷ If $\ell_L = \ell_R$ and a power-of-2, these are equivalent
    MERGE-ODD-CONTINUE($[L_i]_{i=0:\ell_L}$, left)
    (MERGE-ODD-CONTINUE($[R_i]_{i=0:\ell_R}$, right)
  end do
end function
function MERGE-ODD-CONTINUE($[x_i]_{i=0:\ell}, p$)
  if $\ell > 1$ then
    $h \leftarrow 2^{\lceil \log_2 \ell \rceil - 1}$          ▷ largest power-of-2 $< \ell$
    parallel for $i \leftarrow 0 : \ell - h$ do
      ▷ Implemented with warp shuffle butterfly
      COMPARE-SWAP($x_i, x_{i+h}$)
    end for
    parallel do
      if $p$ = left then                 ▷ left side recursion
        MERGE-ODD-CONTINUE($[x_i]_{i=0:\ell-h}$, left)
        MERGE-ODD-CONTINUE($[x_i]_{i=\ell-h:\ell}$, right)
      else                               ▷ right side recursion
        MERGE-ODD-CONTINUE($[x_i]_{i=0:h}$, left)
        MERGE-ODD-CONTINUE($[x_i]_{i=h:\ell}$, right)
      end if
    end do
  end if
end function
```

The odd size algorithm is derived by considering arrays to be padded to the next highest power-of-2 size with dummy elements that are never swapped (the merge is monotonic) and are already properly positioned; any comparisons with dummy elements are elided. A left array is considered to be padded with dummy elements at the start; a right array has them at the end. A merge of two sorted arrays of length ℓ_L and ℓ_R to a sorted array of $\ell_L + \ell_R$ requires $\lceil \log_2(\max(\ell_L, \ell_R)) \rceil + 1$ parallel steps. Fig. 1 shows Algorithm 1's merging network for arrays of size 5 and 3, with 4 parallel steps.

The COMPARE-SWAP is implemented using warp shuffles on a lane-stride register array. Swaps with a stride a multiple of 32 occur directly within a lane as the lane holds both elements locally. Swaps of stride ≤ 16 or a non-multiple of 32 occur with warp shuffles. In practice, used array lengths are multiples of 32 as they are held in lane-stride arrays.

Algorithm 2 extends the merge to a full sort. Assuming no structure present in the input data, it requires $\frac{1}{2}(\lceil \log_2(\ell) \rceil^2 + \lceil \log_2(\ell) \rceil)$ parallel steps for sorting a data array of length ℓ.

Algorithm 2. Odd-Size Sorting Network

```
function SORT-ODD($[x_i]_{i=0:\ell}$)
  if $\ell > 1$ then
    parallel do
      SORT-ODD($[x_i]_{i=0:\lfloor \ell/2 \rfloor}$)
      SORT-ODD($[x_i]_{i=\lfloor \ell/2 \rfloor:\ell}$)
    end do
    MERGE-ODD($[x_i]_{i=0:\lfloor \ell/2 \rfloor}, [x_i]_{i=\lfloor \ell/2 \rfloor:\ell}$)
  end if
end function
```

4.2 WarpSelect

Our k-selection implementation, WARPSELECT, maintains state entirely in registers and requires only a single pass over input. It uses MERGE-ODD and SORT-ODD as primitives. Since the register file provides much more storage than shared memory, it supports $k \leq 1024$. Each warp is dedicated to k-selection to a single one of the n arrays $[a_i]$. If n is large enough, a single warp per each $[a_i]$ will result in full GPU occupancy. Large ℓ per warp is handled by recursive decomposition, if ℓ is known in advance.

Overview. Our approach (Algorithm 3 and Fig. 2) operates on values, with associated indices carried along (omitted from the description for simplicity). It selects the k least values that come from global memory, or from intermediate value registers if fused into another kernel providing the values. Let $[a_i]_{i=0:\ell}$ be the sequence provided for selection.

The elements (on the left of Fig. 2) are processed in groups of 32, i.e., the warp size. Lane j is responsible for processing the elements $\{a_j, a_{32+j}, ...\}$. Thus, if the elements come from global memory, the reads are contiguous and coalesced into a minimal number of memory transactions. Each time one of the lanes becomes full (WARP-BALLOT is triggered) the merging routine is called on all lanes, and the thread queue is flushed.

Algorithm 3. WARPSELECT Pseudocode for Lane j

```
function WARPSELECT($a$)
  if $a < W_k$ then ▷ each lane $j$ does this independently
    $T^j_{C_j} \leftarrow a$
    $C_j \leftarrow C_j + 1$
  end if
  if WARP-BALLOT($C_j = t$) then
      ▷ Reinterpret thread queues as lane-stride array
    $[\alpha_i]_{i=0:32t} \leftarrow$ CAST($[T^j_i]_{i=0:t,j=0:32}$)
    ▷ Warp sorts thread queues together, $W_i$ updated
    SORT-ODD($[\alpha_i]_{i=0:32t}$)
    MERGE-ODD($[W_i]_{i=0:k}, [\alpha_i]_{i=0:32t}$)
    $[T^j_i]_{i=0:t,j=0:32} \leftarrow +\infty$
    $[C_j]_{j=0:32} \leftarrow 0$
  end if
end function
```

Data Structures. The warp shares a lane-stride register array of k smallest seen elements, $[W_i]_{i=0:k}$, called the *warp*

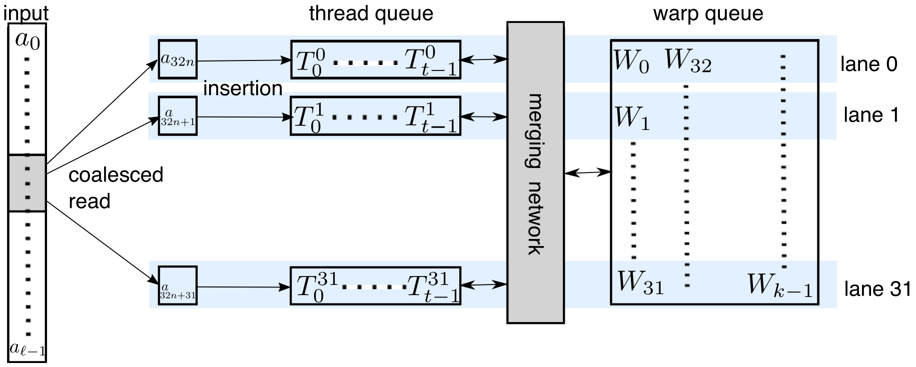

Fig. 2. Overview of WarpSelect. The input values stream in on the left, and the warp queue on the right holds the output result.

queue. It is ordered from smallest to largest ($W_i \leq W_{i+1}$); if the requested k is not a multiple of 32, we round it up.

Each lane j maintains a small set of t elements in registers, called the *thread queues* $[T_i^j]_{i=0:t}$, along with a count of the number of elements C_j ($0 \leq C_j \leq t$) currently valid. All elements in the thread queues with index $< C_j$ are guaranteed to be $\leq W_{k-1}$. Other elements are initialized to maximum sentinel values, e.g., $+\infty$.

The thread queue is a first-level filter for new values coming in; only new potential min-k elements are retained during a scan, until we have collected a set of t elements in any lane that need to be considered as possible true min-k elements. The choice of t is made relative to k, see Section 4.3. The warp queue is a second level that maintains all of the min-k warp-wide observed values. The warp queue is initialized to maximum sentinel values as well.

Update. The three invariants maintained are:

- all $T_i^j \leq W_{k-1}$ for $i < C_j$, $T_i^j = +\infty$ otherwise;
- no thread will accumulate more than t elements in their thread queue ($C_j \leq t$);
- all a_i seen so far in the true min-k are contained in either some lane's thread queue ($[T_i^j]_{i=0:C_j,j=0:32}$), or in the warp queue.

Lane j receives a new a_{32i+j}, and we compare against the current W_k. If $a_{32i+j} > W_k$, then the new element is by definition not in the min-k, and is rejected. Otherwise, we add it to the thread queue, overwriting the sentinel value previously at $W_{C^j}^j$. If any lane has accumulated t values, then we cannot process any new a_i without being able to determine if it is in the min-k for all elements seen so far, as we have no place to keep the element if it is $< W_k$. Using the *warp ballot* instruction, we determine if any lane has accumulated t values, in which case the ballot is "won". If not, we are free to continue processing new elements.

Maintaining the Invariants. Some or all elements in the thread queues may now be in the true min-k. In order to make W_k the true kth lowest element seen so far, the warp uses odd-merge to merge and sort the thread and warp queues together. The new warp queue will be the min-k elements across the merged, sorted queues, and the thread queues are reinitialized to the maximum sentinel value. We are then free to continue processing subsequent elements without violating the invariants.

The warp queue is already sorted, but the thread queues are not. The thread queues are sorted together, and the set of $32t$ sorted elements are merged with the sorted warp queue of length k. Supporting odd-sized merges is important because Batcher's formulation would require that $32t = k$ and is a power-of-2. Thus if $k = 1024$, t must be 32. We found that the optimal t is way smaller (see below), which means that we are merging two different sizes.

Handling the Remainder. If there are remainder elements because ℓ is not a multiple of 32, those are considered for the thread queues for the lanes covering the remainder, after which we proceed to the output stage.

Output. After handling all elements, a final sort and merge is made of the thread and warp queues, after which the warp queue holds the min-k of all a_i.

4.3 Complexity and Parameter Selection

For each incoming group of 32 elements, WarpSelect performs 1, 2 or 3 constant-time operations, all happening in warp-wide parallel time:

1) read 32 elements, compare to W_k, cost $\mathcal{C}_1$, happens N_1 times;
2) insert in thread queue, cost $\mathcal{C}_2$, happens N_2 times;
3) if $\exists j$ such that $\mathcal{C}_j = t$, sort and merge queues, cost $\mathcal{C}_3 = \mathcal{O}(t\log(32t)^2 + k\log(\max(k, 32t)))$, happens N_3 times.

Thus, the total cost is $N_1\mathcal{C}_1 + N_2\mathcal{C}_2 + N_3\mathcal{C}_3$. $N_1 = \ell/32$. For N_2, because we retain the $[T_i^j]$ in registers which requires compile-time indexing, we use an unrolled loop over t to find the proper register to overwrite the current C_j value, so $N_2 = \mathcal{O}(t)$. For N_3, we derive an estimate for random data drawn independently.

Let the input to k-selection be a sequence $\{a_1, a_2, \ldots, a_\ell\}$ (1-based indexing), a randomly chosen permutation of a set of distinct elements. Elements are read sequentially in c groups of size w (the warp, so $w = 32$). Assume ℓ is a multiple of w, so $c = \ell/w$. Recall that t is the maximum thread queue length. We call elements prior to or at position n in the min-k seen so far the *successive min-k (at n)*. The likelihood that a_n is in the successive min-k at n is:

$$\alpha(n,k) := \begin{cases} 1 & \text{if } n \leq k \\ k/n & \text{if } n > k \end{cases}, \tag{7}$$

as each a_n, $n > k$ has a k/n chance as all permutations are equally likely, and all elements in the first k qualify.

Counting the Thread Queue Insertions. In a given lane, an insertion is triggered if the incoming value is in the successive min-$k+t$ values, but the lane has "seen" only $wc_0 + (c - c_0)$ values, where c_0 is the previous won warp ballot. The probability of this happening is:

$$\alpha(wc_0 + (c - c_0), k+t) \approx \frac{k+t}{wc} \text{ for } c > k. \tag{8}$$

The approximation considers that the thread queue has seen *all* the wc values, not just those assigned to its lane. The probability of *any* lane triggering a queue insertion is then:

$$1 - \left(1 - \frac{k+t}{wc}\right)^w \approx \frac{k+t}{c}. \tag{9}$$

Here the approximation is a first-order Taylor expansion. Summing up the probabilities over c gives an expected number of insertions of $N_2 \approx (k+t)\log(c) = \mathcal{O}(k\log(\ell/w))$.

Counting Full Sorts. We seek $N_3 = \pi(\ell, k, t, w)$, the expected number of full sorts required for WarpSelect.

Single Lane. For now, we assume $w=1$, so $c=\ell$. Let $\gamma(\ell, m, k)$ be the probability that in an sequence $\{a_1, \ldots, a_\ell\}$, exactly m of the elements as encountered by a sequential scanner ($w=1$) are in the successive min-k. Given m, there are $\binom{\ell}{m}$ places where these successive min-k elements can occur. It is given by a recurrence relation:

$$\gamma(\ell, m, k) := \begin{cases} 1 & \ell = 0 \text{ and } m = 0 \\ 0 & \ell = 0 \text{ and } m > 0 \\ 0 & \ell > 0 \text{ and } m = 0 \\ (\gamma(\ell-1, m-1, k)\cdot\alpha(\ell, k) + \\ \gamma(\ell-1, m, k)\cdot(1-\alpha(\ell, k))) & \text{otherwise.} \end{cases} \tag{10}$$

The last case is the probability of encountering the following situation: there is a $\ell - 1$ sequence with $m-1$ successive min-k elements preceding us, and the current element is in the successive min-k, or the current element is not in the successive min-k, m ones are before us.

We then develop a recurrence relationship for $\pi(\ell, k, t, 1)$. We first note that

$$\delta(\ell, b, k, t) := \sum_{m=bt}^{\min((bt+\max(0,t-1)),\ell)} \gamma(\ell, m, k), \tag{11}$$

for b where $0 \le bt \le \ell$ is the fraction of all sequences of length ℓ that will force b sorts of data by winning the thread queue ballot, as there have to be bt to $(bt + \max(0, t-1))$ elements in the successive min-k for these sorts to happen (as the min-k elements will overflow the thread queues). There are at most $\lfloor \ell/t \rfloor$ won ballots that can occur, as it takes t separate sequential current min-k seen elements to win the ballot. $\pi(\ell, k, t, 1)$ is thus the expectation of this over all possible b:

$$\pi(\ell, k, t, 1) = \sum_{b=1}^{\lfloor \ell/t \rfloor} b \cdot \delta(\ell, b, k, t). \tag{12}$$

This quantity can be computed by dynamic programming. Analytically, note that for $t=1$, $k=1$, $\pi(\ell, 1, 1, 1)$ is the harmonic number $H_\ell = 1 + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{\ell}$, which converges to $\ln(\ell) + \gamma$ (the Euler-Mascheroni constant γ) as $\ell \to \infty$.

For $t = 1, k > 1, \ell > k$, $\pi(\ell, k, 1, 1) = k + k(H_\ell - H_k)$ or $\mathcal{O}(k\log(\ell))$, as the first k elements are in the successive min-k, and the expectation for the rest is $\frac{k}{k+1} + \frac{k}{k+2} + \ldots + \frac{k}{\ell}$.

For $t > 1, k > 1, \ell > k$, note that there are some number D, $k \le D \le \ell$ of successive min-k determinations D made for each possible $\{a_1, \ldots, a_\ell\}$. The number of won ballots for each case is by definition $\lfloor D/t \rfloor$, as the thread queue must fill up t times. Thus, $\pi(\ell, k, t, 1) = \mathcal{O}(k\log(\ell)/t)$.

Multiple Lanes. The $w > 1$ case is complicated by the fact that there are joint probabilities to consider (if more than one of the w workers triggers a sort for a given group, only one sort takes place). However, the likelihood can be bounded. Let $\pi'(\ell, k, t, w)$ be the expected won ballots assuming no mutual interference between the w workers for winning ballots (i.e., we win b ballots if there are $b \le w$ workers that independently win a ballot at a single step), but with the shared min-k set after each sort from the joint sequence. Assume that $k \ge w$. Then we have

$$\begin{aligned} \pi'(\ell, k, 1, w) &\le w\left(\left\lceil \frac{k}{w} \right\rceil + \sum_{i=1}^{\lceil \ell/w \rceil - \lceil k/w \rceil} \frac{k}{w(\lceil k/w \rceil + i)}\right) \\ &\le w\pi(\lceil \ell/w \rceil, k, 1, 1) = \mathcal{O}(wk\log(\ell/w)), \end{aligned} \tag{13}$$

where the likelihood of the w workers seeing a successive min-k element has an upper bound of that of the first worker at each step. As before, the number of won ballots is scaled by t, so $\pi'(\ell, k, t, w) = \mathcal{O}(wk\log(\ell/w)/t)$. Mutual interference can only reduce the number of ballots, so we obtain the same upper bound for $\pi(\ell, k, t, w)$. Assuming w fixed for the warp size, we have $N_3 = \pi(\ell, k, t, 32) = \mathcal{O}(k\log(\ell)/t)$.

Selection of t. The trade-off is to balance a cost in $N_2\mathcal{C}_2$ and one in $N_3\mathcal{C}_3$. The practical choice for t given k and ℓ was made by experimenting on a variety of k-NN data. For $k \le 32$, we use $t=2$, $k \le 128$ uses $t=3$, $k \le 256$ uses $t=4$, and $k \le 1024$ uses $t=8$, all irrespective of ℓ.

5 Implementing the Index on a GPU

This section explains efficient GPU implementation of Section 2's similarity search methods, devoting particular attention to IVFADC which targets the largest databases. It is one of the indexing methods originally built upon product quantization [32]. Details on distance computations and articulation with k-selection are the key to understanding why this method can outperform recent GPU approximate nearest neighbor strategies [58].

5.1 Exact Search

We briefly come back to the exhaustive search method, often referred to as exact brute-force. It is interesting on its own for exact nearest neighbor search in small datasets. It is also a component of many indexes in the literature; we use it for the IVFADC coarse quantizer q_1.

As stated in Section 2, the distance computation boils down to a matrix multiplication. We use optimized GEMM routines in the cuBLAS library to calculate the $-2\langle \mathbf{x}_j, \mathbf{y}_i \rangle$ term with respect to L_2 distance, resulting in a partial distance matrix D'. To complete the distance calculation, we use a fused k-selection kernel that adds the $\|\mathbf{y}_i\|^2$ term to each partial distance result and immediately submits the value to k-selection in registers. The $\|\mathbf{x}_j\|^2$ term needs not be taken into account before k-selection. k-selection that can be fused with other GPU computations allows for only 2 passes

(GEMM write, k-select read) over the matrix D', compared to other implementations that typically require 3 or more.

As the matrix D' does not fit in GPU memory for realistic problem sizes, the problem is tiled over the batch of queries, with $t_q \leq n_q$ queries being run in a single tile.

5.2 IVFADC Indexing

PQ Lookup Tables. IVFADC requires computing the distance from a vector to a set of PQ reproduction values. By developing Equation (6) for a database vector y, we obtain:

$$\|\mathbf{x} - q(\mathbf{y})\|_2^2 = \|\mathbf{x} - q_1(\mathbf{y}) - q_2(\mathbf{y} - q_1(\mathbf{y}))\|_2^2. \tag{14}$$

If we decompose the residual vectors left after q_1 as:

$$\mathbf{y} - q_1(\mathbf{y}) = [\widetilde{\mathbf{y}^1} \cdots \widetilde{\mathbf{y}^b}] \quad \text{and} \tag{15}$$

$$\mathbf{x} - q_1(\mathbf{y}) = [\widetilde{\mathbf{x}^1} \cdots \widetilde{\mathbf{x}^b}], \tag{16}$$

then the distance is rewritten as:

$$\|\mathbf{x} - q(\mathbf{y})\|_2^2 = \|\widetilde{\mathbf{x}^1} - q^1(\widetilde{\mathbf{y}^1})\|_2^2 + ... + \|\widetilde{\mathbf{x}^b} - q^b(\widetilde{\mathbf{y}^b})\|_2^2. \tag{17}$$

Each quantizer $q^1, \ldots, q^b$ has 256 reproduction values, so when $\mathbf{x}$ and $q_1(\mathbf{y})$ are known, all distances can be precomputed and stored in tables $T_1, \ldots, T_b$ each of size 256 [32]. Computing the sum (17) consists of b look-ups and additions. Comparing the cost to compute n distances:

- Explicit computation: $n \times d$ mutiply-adds;
- With lookup tables: $256 \times d$ multiply-adds and $n \times b$ lookup-adds.

This is the key to PQ efficiency. In our GPU implementation, b is any multiple of 4 up to 64. The codes are stored as sequential groups of b bytes per vector within lists.

IVFADC Lookup Tables. When scanning over inverted list elements $\mathcal{I}_L$, the lookup table method can be applied, as the query $\mathbf{x}$ is known and by definition $q_1(\mathbf{y})$ is constant. Moreover, the computation of the tables $T_1 \ldots T_b$ is further optimized [6]. The expression of $\|\mathbf{x} - q(\mathbf{y})\|_2^2$ in Equation (14) is decomposed as:

$$\underbrace{\|q_2(...)\|_2^2 + 2\langle q_1(\mathbf{y}), q_2(...)\rangle}_{\text{term 1}} + \underbrace{\|\mathbf{x} - q_1(\mathbf{y})\|_2^2}_{\text{term 2}} - 2\underbrace{\langle \mathbf{x}, q_2(...)\rangle}_{\text{term 3}}. \tag{18}$$

The objective is to minimize inner loop computations. The computations we can do in advance and store in lookup tables are as follows:

- Term 1 is independent of the query. It can be precomputed from the quantizers, and stored in a table $\mathcal{T}$ of size $|\mathcal{C}_1| \times 256 \times b$;
- Term 2 is the distance to q_1's reproduction value. It is thus a by-product of the first-level quantizer q_1;
- Term 3 is computed independently of the inverted list. Its computation costs $d \times 256$ multiply-adds.

This decomposition is used to produce the lookup tables $T_1 \ldots T_b$ used during the scan of the inverted list. For a single query, computing the $\tau \times b$ tables from scratch costs $\tau \times d \times 256$ multiply-adds, while this decomposition costs $256 \times d$ multiply-adds and $\tau \times b \times 256$ additions. On the GPU, the memory usage of $\mathcal{T}$ can be prohibitive, so we enable the decomposition only when memory is a not a concern.

5.3 GPU Implementation

Algorithm 4 summarizes the process as one would implement it on a CPU. The inverted lists are stored as two separate arrays, for PQ codes and associated IDs. IDs are resolved only if k-selection determines k-nearest membership. This lookup is a few sparse memory reads in a large array, thus for the GPU the IDs can optionally be stored on CPU for tiny performance cost.

Algorithm 4. IVFPQ Batch Search Routine

```
function IVFPQ-SEARCH($[\mathbf{x_1}, ..., \mathbf{x_{n_q}}], \mathcal{I}_1, \ldots, \mathcal{I}_{|\mathcal{C}_1|}$)
  for $i \leftarrow 0 : n_q$ do ▷ batch quantization of Section 5.1
    $L^i_{\text{IVF}} \leftarrow \tau\text{-argmin}_{c \in \mathcal{C}_1} \|\mathbf{x} - \mathbf{c}\|_2$
  end for
  for $i \leftarrow 0 : n_q$ do
    $L \leftarrow []$                                   ▷ distance table
    Compute term 3 (see Section 5.2)
    for $L$ in $L^i_{\text{IVF}}$ do                      ▷ $\tau$ loops
      Compute distance tables $T_1, \ldots, T_b$
      for $j$ in $\mathcal{I}_L$ do
                ▷ distance estimation, Equation (17)
        $d \leftarrow \|\mathbf{x_i} - q(\mathbf{y_j})\|_2^2$
        Append $(d, L, j)$ to $L$
      end for
    end for
    $R_i \leftarrow$ k-select smallest distances $d$ from $L$
  end for
return R
end function
```

List Scanning. A kernel scans the τ closest inverted lists for each query, and calculates per-vector pair distances using the lookup tables T_i. The T_i are stored in shared memory: up to $n_q \times \tau \times \max_i |\mathcal{I}_i| \times b$ lookups are required for a query set (trillions of accesses in practice), and are random access. This limits b to at most 48 (32-bit floating point) or 96 (16-bit floating point) with current GPU architectures. In case we do not use the decomposition of Equation (18), the T_i are calculated by a separate kernel before scanning.

Multi-Pass Kernels. Each $n_q \times \tau$ pairs of query against inverted list can be processed independently. At one extreme, a block is dedicated to each of these, resulting in up to $n_q \times \tau \times \max_i |\mathcal{I}_i|$ partial results being written back to global memory, which is then k-selected to $n_q \times k$ final results. This yields high parallelism but can exceed available GPU global memory. As with exact search, we choose a tile size $t_q \leq n_q$ to reduce memory consumption, bounding its complexity by $\mathcal{O}(2 t_q \tau \max_i |\mathcal{I}_i|)$ with multi-streaming.

End-to-end Performance. For our IVFADC + PQ implementation, Table 1 shows global memory bandwidth and arithmetic utilitization relative to the roofline model peak of a Maxwell Titan X GPU. It is evaluated on the YFCC100M index (Section 6.4) for a query of 4096 vectors, with $\tau = 32$ and $k = 100$, without the decomposition of the T_i.The end-to-end workload is quite heterogeneous, with both high arithmetic (IVFADC q_1) and memory bandwidth demands (list scanning). k-selection is only 20.6 percent of the total

TABLE 1
GPU IVFADC + PQ End-to-End Performance

% of time (1.3s total)	kernel	arithmetic (% peak)	gmem (% peak)	limiting factor
19.4%	IVFADC q_1	95%	44%	arithmetic
5.9%	k-select q_1	40%	64%	gmem b/w
23.4%	T_i distance	60%	66%	gmem b/w
34.5%	list scanning	26%	87%	gmem b/w
14.7%	k-select lists	55%	52%	arithmetic
2.1%	ID lookup	23%	13%	inst latency
weighted average		**52.4%**	**65.7%**	**gmem b/w**

time of 1.3 s, but both the k-select q_1 and k-select list kernels take advantage of kernel fusion. Otherwise, multiple passes in global memory would be required, leaving the workload to be dominated by k-selection.

5.4 Multi-GPU Parallelism

Modern servers can support several GPUs. We employ this capability for both speed and memory.

Replication. If an index fits in the memory of a single GPU, it can be replicated across $\mathcal{R}$ different GPUs. To query $n_{\rm q}$ vectors, each replica handles a fraction $n_{\rm q}/\mathcal{R}$ of the queries. Replication has near linear speedup, except for a potential loss in efficiency for small $n_{\rm q}$.

Sharding. If an index does not fit in the memory of a single GPU, an index can be sharded across $\mathcal{S}$ different GPUs. For adding ℓ vectors, each shard receives $\ell/\mathcal{S}$ of the vectors, and for query, each shard handles the full query set $n_{\rm q}$, joining the partial results (an additional round of k-selection is still required) on a single GPU or in CPU memory. For a given index size ℓ, sharding will yield a speedup (sharding has a query of $n_{\rm q}$ against $\ell/\mathcal{S}$ versus replication with a query of $n_{\rm q}/\mathcal{R}$ against ℓ), but is usually less than pure replication due to fixed overhead and cost of subsequent k-selection.

Replication and sharding can be used together ($\mathcal{S}$ shards, each with $\mathcal{R}$ replicas for $\mathcal{S} \times \mathcal{R}$ GPUs in total). Sharding or replication are both fairly trivial, and the same principle can be used to distribute an index across multiple machines.

6 Experiments and Applications

This section compares our GPU k-selection and nearest-neighbor approach to existing libraries. Unless stated otherwise, experiments are carried out on a 2×2.8GHz Intel Xeon E5-2680v2 with 4 Maxwell Titan X GPUs on CUDA 8.0.

6.1 k-Selection Performance

We compare against two other GPU small k-selection implementations: the row-based Merge Queue with Buffered Search and Hierarchical Partition extracted from the *fgknn* library of Tang et al. [51] and Truncated Bitonic Sort (*TBiS*) from Sismanis et al. [50]. Both were extracted from their respective exact search libraries. These implementations were chosen because they do not require multiple passes over the input data. Any implementation that requires more than one pass over the input data will be largely bound by global memory bandwidth, and in the cases where our utilization exceeds 50 percent, cannot win. Both fgknn and TBiS require additional temporary global memory for intermediate calculations unlike our implementation.

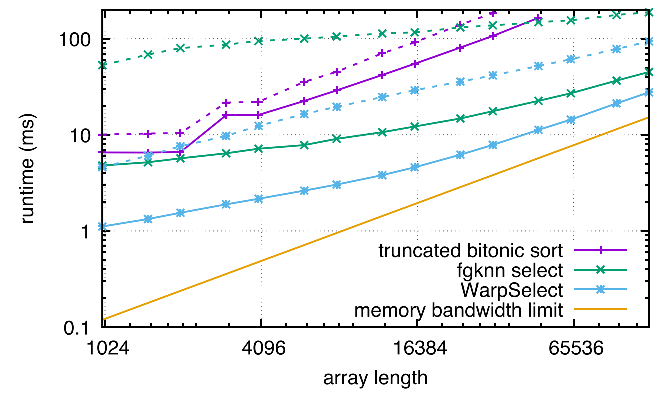

Fig. 3. Runtimes for different k-selection methods, as a function of array length ℓ. Simultaneous arrays processed are $n_q = 10000$. $k = 100$ for full lines, $k = 1000$ for dashed lines.

We evaluate k-selection for k = 100 and 1000 of each row from a row-major matrix $n_q \times \ell$ of random 32-bit floating point values on a single Titan X. The batch size n_q is fixed at 10000, and the array lengths ℓ vary from 1000 to 128000. Inputs and outputs to the problem remain resident in GPU memory, with the output being of size $n_q \times k$, with corresponding indices. Thus, the input problem sizes range from 40 MB (ℓ = 1000) to 5.12 GB (ℓ = 128k). TBiS requires large auxiliary storage and is limited to $\ell \leq$ 48k in our tests.

Fig. 3 shows our relative performance against TBiS and fgknn. It also includes the peak possible performance given by the memory bandwidth limit of the Titan X. The relative performance of WarpSelect over fgknn increases for larger k; even TBiS starts to outperform fgknn for larger ℓ at $k = 1000$. We look especially at the largest $\ell = 128000$. WarpSelect is $1.62\times$ faster at $k = 100$, $2.01\times$ at $k = 1000$. Performance against peak possible drops off for all implementations at larger k. WarpSelect operates at *55 percent of peak* at $k = 100$ but only 16 percent of peak at $k = 1000$. This is due to additional overhead assocated with bigger thread queues and merge/sort networks for large k.

Differences from fgknn. WarpSelect is influenced by fgknn, but has several improvements: all state is held in registers (no shared memory), no inter-warp synchronization, multiple kernel launches or buffering is used, no "hierarchical partition", and odd-size networks provide more efficient merging and sorting. These improvements allow k-selection to be fused directly into other GPU kernels, a significant performance advantage for exact similarity search (Section 5.1) and IVFADC list traversal (Section 5.2).

6.2 k-Means Clustering

Exact search with $k = 1$ can be used by a k-means clustering method in the assignment stage, to assign $n_{\rm q}$ training vectors to $|\mathcal{C}_1|$ centroids. Despite the fact that it does not use IVFADC and $k = 1$ selection is trivial (a parallel min-reduction, not WarpSelect), k-means is a good benchmark for the clustering used to train the quantizer q_1.

We apply the algorithm on MNIST8m images. The 8.1M images are graylevel digits in 28x28 pixels, linearized to vectors of 784-d. In Table 2 we compare this

TABLE 2
MNIST8m k-Means Performance

method	# GPUs	# centroids 256	 4096
BIDMach [13]	1	320 s	735 s
Ours	1	140 s	316 s
Ours	4	84 s	100 s

k-means implementation to the GPU k-means of BIDMach [13], which was shown to be more efficient than several distributed k-means implementations that require dozens of machines.[2] Both algorithms were run for 20 iterations. Our implementation is more than 2× *faster*, although both are built upon cuBLAS. Our implementation receives some benefit from the k-selection fusion into L_2 distance computation. For multi-GPU execution via replicas, the speedup is close to linear for large enough problems (3.16× for 4GPUs with 4096 centroids). Note that this benchmark is somewhat unrealistic, as one would typically sub-sample the dataset randomly when so few centroids are requested.

Large Scale. We also compare to the approximate method of Avrithis et al. [4]. It clusters 10^8 128-d vectors to 85K centroids. Their clustering method runs in 46 minutes, but requires 56 minutes at least of pre-processing to encode the vectors. Our method performs *exact* k-means on 4 GPUs in 52 minutes without any pre-processing.

6.3 Exact Nearest Neighbor Search

We consider a classical dataset used to evaluate nearest neighbor search: SIFT1M [32]. Its characteristic sizes are $\ell = 10^6$, $d = 128$, $n_{\rm q} = 10^4$. Computing the partial distance matrix D' costs $n_{\rm q} \times \ell \times d = 1.28$ Tflop, which runs in less than one second on current GPUs. Fig. 4 shows the cost of the distance computations against the cost of our tiling of the GEMM for the $-2 < \mathbf{x_j}, \mathbf{y_i} >$ term of Equation (2) and the peak possible k-selection performance on the distance matrix of size $n_{\rm q} \times \ell$, which additionally accounts for reading the tiled result matrix D' at peak memory bandwidth.

In addition to our method from Section 5, we include times from the two GPU libraries evaluated for k-selection performance in Section 6.1. We make several observations:

- for k-selection, the naive algorithm that sorts the full result array using `thrust::sort_by_key` is more than 10× slower than comparison methods;
- L_2 distance and k-selection cost is dominant for all but our method, which has 85 *percent of the peak* possible performance, assuming GEMM usage and our tiling of the partial distance matrix D' on top of GEMM is close to optimal. The cuBLAS GEMM itself has low efficiency for small reduction sizes ($d = 128$);
- Our fused L_2 and k-selection kernel is important. The same exact algorithm without fusion (requiring an additional pass through D') is at least 25 percent slower.

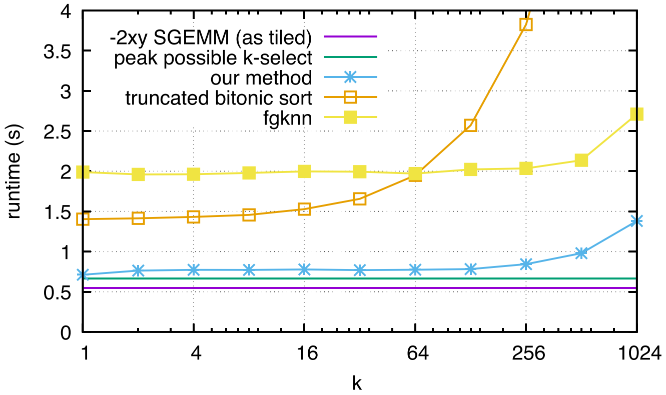

Fig. 4. Exact search k-NN time for the SIFT1M dataset with varying k on 1 Titan X GPU.

Efficient k-selection is even more important in situations where approximate methods are used to compute distances, because the relative cost of k-selection with respect to distance computation increases.

6.4 Billion-Scale Approximate Search

There are few studies on approximate nearest-neighbor search on large datasets ($\ell \gg 10^6$). We report a few comparison points here on index search, using standard datasets and evaluation protocol in this field. The statistics of these datasets are provided in Table 3. We are most interested in SIFT1B and DEEP1B, which are to the best of our knowledge the largest datasets publicly available for the task of similarity search.

SIFT1M. For the sake of completeness, we first compare our GPU search speed on SIFT1M with the implementation of Wieschollek et al. [58]. They obtain a nearest neighbor recall at 1 (fraction of queries where the true nearest neighbor is in the top 1 result) of R@1 = 0.51, and R@100 = 0.86 in 0.02 ms per query on a Titan X. For the same time budget, our implementation obtains R@1 = 0.80 and R@100 = 0.95.

SIFT1B. We compare again with Wieschollek et al., on the SIFT1B dataset [33] of 1 billion SIFT image features at $n_{\rm q} = 10^4$. We compare the search performance in terms of same memory usage for similar accuracy (more accurate methods may involve greater search time or memory usage). On a single GPU, with m = 8 bytes per vector, R@10 = 0.376 in 17.7 μs per query vector, versus their reported R@10 = 0.35 in 150 μs per query vector. Thus, our implementation is more accurate at a speed *8.5× faster*.

DEEP1B. We also experimented on the DEEP1B dataset [8] of ℓ = 1 billion CNN representations for images at $n_{\rm q} = 10^4$. The paper that introduces the dataset reports CPU results (1 thread): R@1 = 0.45 in 20 ms search time per vector. We

TABLE 3
Properties of the Datasets in Our Evaluation

dataset	# dataset vectors	# query vectors	# training vectors	dims	data size
SIFT1M	1000000	10000	100000	128	128 MiB
SIFT1B	1000000000	10000	100000000	128	128 GiB
DEEP1B	1000000000	10000	350000000	96	384 GiB
YFCC100M	95074575	n/a	n/a	128	48.6 GiB

[2] BIDMach numbers from https://github.com/BIDData/BIDMach/wiki/Benchmarks#KMeans

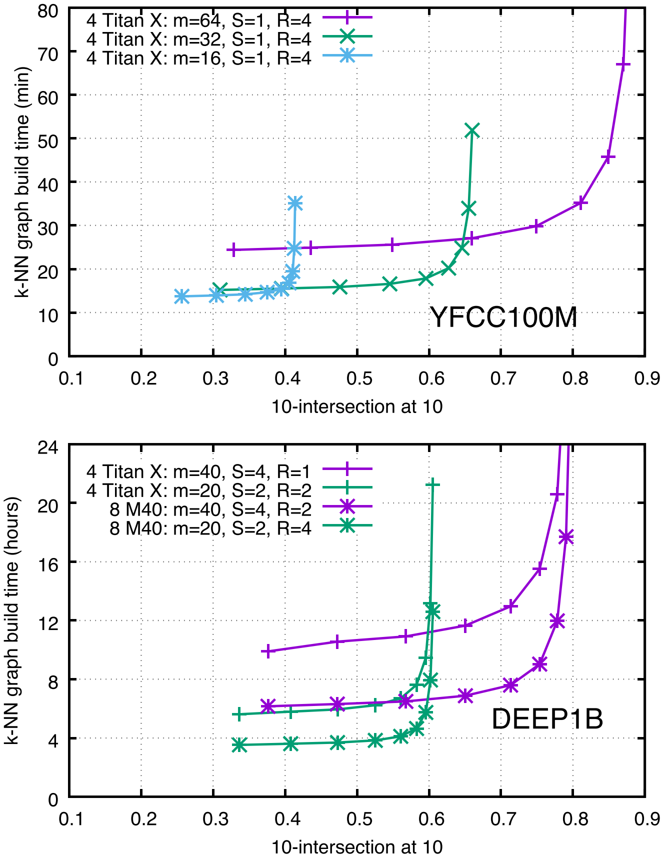

Fig. 5. Speed/accuracy trade-off of brute-force 10-NN graph construction for the YFCC100M and DEEP1B datasets.

use a PQ encoding of m = 20, with d = 80 via OPQ [23], and $|\mathcal{C}_1|$ = 2^{18}, which uses a comparable dataset storage as the original paper (20 GB). This requires multiple GPUs as it is too large for a single GPU's global memory, so we consider 4 GPUs with $\mathcal{S}$ = 2, $\mathcal{R}$ = 2. We obtain a R@1 = 0.4517 in 0.0133 ms per vector. While the hardware platforms are different, it shows that making searches on GPUs is a game-changer in terms of speed achievable on a single machine.

YFCC100M. This dataset [52] contains 99.2 million images and 800,000 videos. We could download 95 million of the images. We compute CNN descriptors as the one-before-last layer of a ResNet [30], reduced to $d = 128$ with PCA. Since we use it only for the k-NN graph experiments, we do not distinguish a training and a query set.

6.5 The k-NN Graph

An example usage of our similarity search method is to construct a *k-nearest neighbor graph* of a dataset via brute force (all vectors queried against the entire index).

Experimental Setup. We evaluate the trade-off between speed, precision and memory on the Yfcc100M and Deep1B datasets:

- Speed: How much time it takes to build the IVFADC index from scratch and construct the whole k-NN graph ($k = 10$) by searching nearest neighbors for all vectors in the dataset. Thus, this is an end-to-end test that includes indexing as well as search time;
- Quality: We sample 10,000 images for which we compute the exact nearest neighbors. We measure the fraction of 10 returned nearest neighbors that are within the ground-truth 10 nearest neighbors.

For Yfcc100M, we use a coarse quantizer (2^{16} centroids), and consider $m = 16$, 32 and 64 byte PQ encodings for each vector. For Deep1B, we pre-process the vectors to $d = 120$ via OPQ, use $|\mathcal{C}_1| = 2^{18}$ and consider $m = 20, 40$. For a given encoding, we vary τ from 1 to 256, to obtain trade-offs between efficiency and quality, as seen in Fig. 5. For the k-NN graph experiments on Deep1B, we did not use the training set and the query vectors, we sampled them from the main dataset. We experimented with two fairly common multi-GPU workstation configurations for data-intensive applications: 4 Maxwell-class Titan X GPUs or 8 M40 GPUs.

Discussion. For Yfcc100M we used $\mathcal{S}$ = 1, $\mathcal{R}$ = 4. An accuracy of more than 0.8 is obtained in 35 minutes. For Deep1B, a lower-quality graph can be built in 6 hours, with higher quality in about half a day. We also experimented with more GPUs by doubling the replica set, using 8 Maxwell M40s (the M40 is roughly equivalent in performance to the Titan X). Performance is improved sub-linearly ($\sim 1.6\times$ for $m = 20$, $\sim 1.7\times$ for $m = 40$).

For comparison, the largest k-NN graph construction we are aware of used a dataset comprising 36.5 million 384-d vectors, which took a cluster of 128 CPU servers 108.7 hours of compute [56], using NN-Descent [18]. Note that NN-Descent could also build or refine the k-NN graph for the datasets we consider, but it has a large memory overhead over the graph storage, which is already 80 GB for Deep1B. Moreover it requires random access across all vectors (384 GB for Deep1B).

The largest GPU k-NN graph construction we found is a brute-force construction using exact search with GEMM, of a dataset of 20 million 15,000-d vectors, which took a cluster of 32 Tesla C2050 GPUs 10 days [17]. Assuming computation scales with GEMM cost for the distance matrix, this approach for Deep1B would take an impractical 200 days of computation time on their cluster.

6.6 Using the k-NN Graph

When a k-NN graph has been constructed for an image dataset, we can find paths in the graph between any two images, provided there is a single connected component (this is the case). For example, we can search the shortest path between two images of flowers, by propagating neighbors from a starting image to a destination image. Denoting by S and D the source and destination images, and d_{ij} the distance between nodes, we search the path $P = \{p_1, \ldots, p_n\}$ with $p_1 = S$ and $p_n = D$ such that:

$$\min_{P} \max_{i=1..n} d_{p_i p_{i+1}}, \qquad (19)$$

i.e., we want to favor smooth transitions. An example result is shown in Fig. 6 from Yfcc100M[3]. It was obtained after 20 seconds of propagation in a k-NN graph with $k = 15$ neighbors. Since there are many flower images in the dataset, the transitions are smooth.

[3] The mapping from vectors to images is not available for Deep1B

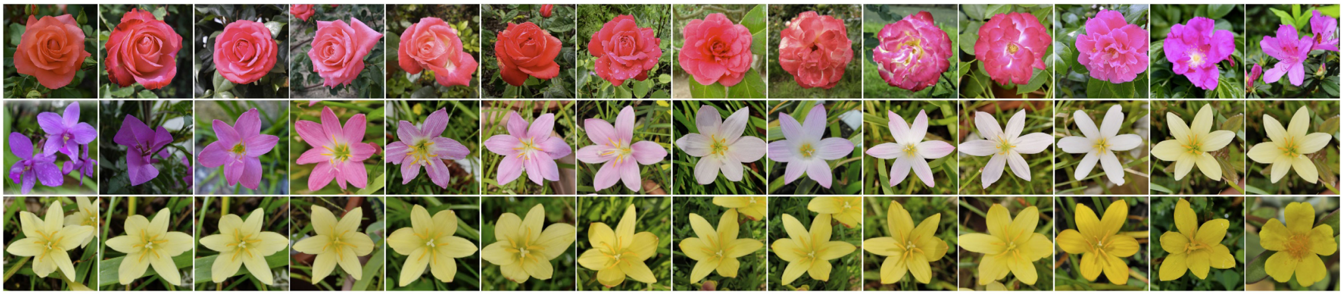

Fig. 6. Path in the k-NN graph of 95 million images from YFCC100M. The first and the last image are given; the algorithm computes the smoothest path between them.

7 Conclusion

The arithmetic throughput and memory bandwidth of GPUs are well into the teraflops and hundreds of gigabytes per second. However, implementing algorithms that approach these performance levels is complex and counter-intuitive. In this paper, we presented the algorithmic structure of similarity search methods that achieves near-optimal performance on GPUs.

This work enables applications that needed complex approximate algorithms before. For example, the approaches presented here make it possible to do exact k-means clustering or to compute the k-NN graph with simple brute-force approaches in less time than a CPU (or a cluster of them) would take to do this approximately.

The limitations of this work are inherent to GPU architectures. The throughput oriented, non-latency optimized execution model of GPUs are efficient for brute-force computations or linear scans of memory arrays (as with IVFADC). Other approximate k-NN approaches like the recent graph-based methods HNSW [42] and NSG [22] depend upon pointer chasing and sparse memory accesses, and will likely not map as efficiently to GPU hardware. As with our work here for efficient k-selection on the GPU SIMD architecture, clever ways to extract parallelism from apparently serial algorithms should be explored to see if HNSW and NSG are worthwhile on GPUs. Also, GPUs still have an order of magnitude less memory than the RAM of a typical CPU server. Additional quantization and compression techniques to expand the toolbox of memory/speed tradeoffs available would also prove useful in this memory-constrained environment.

GPU hardware is now very common on scientific workstations, due to their popularity for machine learning algorithms. We believe that our work further demonstrates their interest for Big Data applications.

References

[1] T. Alabi, J. D. Blanchard, B. Gordon, and R. Steinbach, “Fast k-selection algorithms for graphics processing units,” *ACM J. Exp. Algorithmics*, vol. 17, pp. 4.2:4.1–4.2:4.29, Oct. 2012.

[2] A. Andoni and P. Indyk, “Near-optimal hashing algorithms for near neighbor problem in high dimensions,” in *Proc. Symp. Foundations Comput. Sci.*, 2006, pp. 459–468.

[3] F. André, A.-M. Kermarrec, and N. L. Scouarnec, “Cache locality is not enough: High-performance nearest neighbor search with product quantization fast scan,” in *Proc. Int. Conf. Very Large DataBases*, 2015, pp. 288–299.

[4] Y. Avrithis, Y. Kalantidis, E. Anagnostopoulos, and I. Z. Emiris, “Web-scale image clustering revisited,” in *Proc. IEEE Int. Conf. Comput. Vis.*, 2015, pp. 1502–1510.

[5] A. Babenko and V. Lempitsky, “The inverted multi-index,” in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, Jun. 2012, pp. 3069–3076.

[6] A. Babenko and V. S. Lempitsky, “Improving bilayer product quantization for billion-scale approximate nearest neighbors in high dimensions,” *CoRR*, abs/1404.1831, Apr. 2014.

[7] A. Babenko and V. Lempitsky, “Aggregating local deep features for image retrieval,” in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2015, pp. 1269–1277.

[8] A. Babenko and V. Lempitsky, “Efficient indexing of billion-scale datasets of deep descriptors,” in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, Jun. 2016, pp. 2055–2063.

[9] R. Barrientos, J. Gómez, C. Tenllado, M. Prieto, and M. Marin, “knn query processing in metric spaces using GPUs,” in *Proc. Int. Eur. Conf. Parallel Distrib. Comput.*, Sep. 2011, pp. 380–392.

[10] K. E. Batcher, “Sorting networks and their applications,” in *Proc. Spring Joint Comput. Conf.*, 1968, pp. 307–314.

[11] P. Boncz, W. Lehner, and T. Neumann, “Special issue: Modern hardware,” *VLDB J.*, vol. 25, no. 5, pp. 623–624, 2016.

[12] J. Canny, D. L. W. Hall, and D. Klein, “A multi-teraflop constituency parser using GPUs,” in *Proc. Empirical Methods Natural Language Process.*, 2013, pp. 1898–1907.

[13] J. Canny and H. Zhao, “Bidmach: Large-scale learning with zero memory allocation,” in *Proc. BigLearn Workshop*, 2013. [Online]. Available: https://nips.cc/Conferences/2013/Schedule?showEvent=3698

[14] M. Caron, P. Bojanowski, A. Joulin, and M. Douze, “Deep clustering for unsupervised learning of visual features,” *CoRR*, abs/1807.05520, Jul. 2018.

[15] M. S. Charikar, “Similarity estimation techniques from rounding algorithms,” in *Proc. ACM Symp. Theory Comput.*, May 2002, pp. 380–388.

[16] J. Chhugani, A. D. Nguyen, V. W. Lee, W. Macy, M. Hagog, Y.-K. Chen, A. Baransi, S. Kumar, and P. Dubey, “Efficient implementation of sorting on multi-core simd CPU architecture,” *Proc. VLDB Endow.*, vol. 1, no. 2, pp. 1313–1324, Aug. 2008.

[17] A. Dashti, “Efficient computation of k-nearest neighbor graphs for large high-dimensional data sets on GPU clusters,” Master’s thesis, Engineering, Univ. Wisconsin Milwaukee, Milwaukee, WI, Aug. 2013.

[18] W. Dong, M. Charikar, and K. Li, “Efficient k-nearest neighbor graph construction for generic similarity measures,” in *Proc. Int. Conf. World Wide Web*, Mar. 2011, pp. 577–586.

[19] M. Douze, H. Jégou, and J. Johnson, “An evaluation of large-scale methods for image instance and class discovery,” in *Proc. Thematic Workshops ACM Multimedia*, 2017, pp. 1–9.

[20] M. Douze, H. Jégou, and F. Perronnin, “Polysemous codes,” in *Proc. Eur. Conf. Comput. Vis.*, Oct. 2016, pp. 785–801.

[21] M. Douze, A. Szlam, B. Hariharan, and H. Jégou, “Low-shot learning with large-scale diffusion,” *CoRR*, abs/1706.02332, Jun. 2017.

[22] C. Fu, C. Xiang, C. Wang, and D. Cai, “Fast approximate nearest neighbor search with the navigating spreading-out graph,” *Proc. VLDB Endowment*, vol. 12, no. 5, pp. 461–474, 2019.

[23] T. Ge, K. He, Q. Ke, and J. Sun, “Optimized product quantization,” *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 36, no. 4, pp. 744–755, Apr. 2014.

[24] Y. Gong and S. Lazebnik, “Iterative quantization: A procrustean approach to learning binary codes,” in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, Jun. 2011, pp. 817–824.

[25] Y. Gong, L. Wang, R. Guo, and S. Lazebnik, “Multi-scale orderless pooling of deep convolutional activation features,” in *Proc. Eur. Conf. Comput. Vis.*, 2014, pp. 392–407.

[26] A. Gordo, J. Almazan, J. Revaud, and D. Larlus, "Deep image retrieval: Learning global representations for image search," in *Proc. Eur. Conf. Comput. Vis.*, 2016, pp. 241–257.

[27] A. Gordo, F. Perronnin, Y. Gong, and S. Lazebnik, "Asymmetric distances for binary embeddings," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 36, no. 1, pp. 33–47, Jan. 2014.

[28] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding," *CoRR*, abs/1510.00149, Oct. 2015.

[29] K. He, F. Wen, and J. Sun, "K-means hashing: An affinity-preserving quantization method for learning binary compact codes," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, Jun. 2013, pp. 2938–2945.

[30] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, Jun. 2016, pp. 770–778.

[31] X. He, D. Agarwal, and S. K. Prasad, "Design and implementation of a parallel priority queue on many-core architectures," in *Proc. IEEE Int. Conf. High Perform. Comput.*, 2012, pp. 1–10.

[32] H. Jégou, M. Douze, and C. Schmid, "Product quantization for nearest neighbor search," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 33, no. 1, pp. 117–128, Jan. 2011.

[33] H. Jégou, R. Tavenard, M. Douze, and L. Amsaleg, "Searching in one billion vectors: Re-rank with source coding," in *Proc. Int. Conf. Acoust., Speech Signal Process.*, May 2011, pp. 861–864.

[34] A. Joulin, E. Grave, P. Bojanowski, and T. Mikolov, "Bag of tricks for efficient text classification," arXiv:1607.01759, 2016.

[35] Y. Kalantidis and Y. Avrithis, "Locally optimized product quantization for approximate nearest neighbor search," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, Jun. 2014, pp. 2329–2336.

[36] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Proc. Advances Neural Inf. Process. Syst.*, 2012, pp. 1097–1105.

[37] B. Kulis and T. Darrell, "Learning to hash with binary reconstructive embeddings," in *Proc. Advances Neural Inf. Process. Syst.*, Dec. 2009, pp. 1042–1050.

[38] G. Lample, A. Conneau, M. Ranzato, L. Denoyer, and H. Jégou, "Word translation without parallel data," *CoRR*, abs/1710.04087, Oct. 2017. [Online]. Available: https://iclr.cc/Conferences/2018/Schedule?showEvent=336

[39] F. T. Leighton, *Introduction to Parallel Algorithms and Architectures: Array, Trees, Hypercubes*, San Francisco, CA, USA: Morgan Kaufmann, 1992.

[40] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "NVIDIA Tesla: A unified graphics and computing architecture," *IEEE Micro*, vol. 28, no. 2, pp. 39–55, Mar. 2008.

[41] W. Liu and B. Vinter, "Ad-heap: An efficient heap data structure for asymmetric multicore processors," in *Proc. Workshop Gen. Purpose Process. Using GPUs*, 2014, pp. 54:54–54:63.

[42] Y. A. Malkov and D. Yashunin, "Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs," *CoRR*, abs/1603.09320, Mar. 2016.

[43] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," in *Proc. Advances Neural Inf. Process. Syst.*, 2013, pp. 3111–3119.

[44] L. Monroe, J. Wendelberger, and S. Michalak, "Randomized selection on the GPU," in *Proc. ACM Symp. High Perform. Graph.*, 2011, pp. 89–98.

[45] M. Norouzi and D. Fleet, "Cartesian k-means," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, Jun. 2013, pp. 3017–3024.

[46] J. Pan and D. Manocha, "Fast GPU-based locality sensitive hashing for k-nearest neighbor computation," in *Proc. ACM Int. Conf. Advances Geographic Inf. Syst.*, 2011, pp. 211–220.

[47] L. Paulevé, H. Jégou, and L. Amsaleg, "Locality sensitive hashing: A comparison of hash function types and querying mechanisms," *Pattern Recognit. Lett.*, vol. 31, no. 11, pp. 1348–1358, Aug. 2010.

[48] O. Shamir, "Fundamental limits of online and distributed algorithms for statistical learning and estimation," in *Proc. Advances Neural Inf. Process. Syst.*, 2014, pp. 163–171.

[49] A. Sharif Razavian , H. Azizpour, J. Sullivan, and S. Carlsson, "CNN features off-the-shelf: An astounding baseline for recognition," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. Workshops*, 2014, pp. 512–519.

[50] N. Sismanis, N. Pitsianis, and X. Sun, "Parallel search of k-nearest neighbors with synchronous operations," in *Proc. IEEE High Perform. Extreme Comput. Conf.*, 2012, pp. 1–6.

[51] X. Tang, Z. Huang, D. M. Eyers, S. Mills, and M. Guo, "Efficient selection algorithm for fast K-NN search on GPUs," in *Proc. IEEE Int. Parallel Distrib. Process. Symp.*, 2015, pp. 397–406.

[52] B. Thomee, D. A. Shamma, G. Friedland, B. Elizalde, K. Ni, D. Poland, D. Borth, and L.-J. Li, "YFCC100M: The new data in multimedia research," *Commun. ACM*, vol. 59, no. 2, pp. 64–73, Jan. 2016.

[53] G. Tolias, R. Sicre, and H. Jégou, "Particular object retrieval with integral max-pooling of CNN activations," *CoRR*, abs/1511.05879, Nov. 2015. [Online]. Available: https://iclr.cc/archive/www/doku.php%3Fid=iclr2016:accepted-main.html

[54] V. Volkov and J. W. Demmel, "Benchmarking GPUs to tune dense linear algebra," in *Proc. ACM/IEEE Conf. Supercomputing*, 2008, pp. 31:1–31:11.

[55] A. Wakatani and A. Murakami, "GPGPU implementation of nearest neighbor search with product quantization," in *Proc. IEEE Int. Symp. Parallel Distrib. Process. Appl.*, 2014, pp. 248–253.

[56] T. Warashina, K. Aoyama, H. Sawada, and T. Hattori, "Efficient k-nearest neighbor graph construction using mapreduce for large-scale data sets," *Efficient k-Nearest Neighbor Graph Construction Using MapReduce Large-Scale Data Sets Trans.*, vol. 97-D, no. 12, pp. 3142–3154, 2014.

[57] R. Weber, H.-J. Schek, and S. Blott, "A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces," in *Proc. Int. Conf. Very Large DataBases*, 1998, pp. 194–205.

[58] P. Wieschollek, O. Wang, A. Sorkine-Hornung , and H. P. A. Lensch, "Efficient large-scale approximate nearest neighbor search on the GPU," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, Jun. 2016, pp. 2027–2035.

[59] S. Williams, A. Waterman, and D. Patterson, "Roofline: An insightful visual performance model for multicore architectures," *Commun. ACM*, vol. 52, no. 4, pp. 65–76, Apr. 2009.

Jeff Johnson received the BSE degree in computer science from Princeton University, in 1998. He is a research engineer at Facebook AI Research (FAIR) since 2014, and at Facebook since 2013. From 2001-2012 he worked on performance-sensitive, real-time physics simulation algorithms and low-latency distributed computing techniques for video games and interactive media. At Facebook prior to joining FAIR, he developed novel distributed database systems for web-scale applications. He is the author of many internal GPU systems in use at Facebook, and external GPU systems as found in the PyTorch deep learning framework. He currently performs research on hardware, software and algorithmic co-design for machine learning.

Matthijs Douze received the MS degree in computer science from the ENSEEIHT, and the PhD degree in computer vision from the University of Toulouse, in 2004. He is a research scientist at Facebook AI Research (FAIR) since 2015. From 2005-2015 he was an engineer on the LEAR project-team at INRIA Grenoble. His main research topics are large-scale algorithms for computer vision and similarity search.

Hervé Jégou received the PhD degree from the University of Rennes, in 2005, for his thesis on error-resilient compression and joint source channel coding after attending the École Normale Supérieure de Cachan. He is a research scientist at Facebook AI Research (FAIR) since 2015. Since then he has worked primarily in computer vision and pattern recognition. He joined INRIA as a permanent researcher in 2006, where he was involved in or led several projects for large image and video collections. In 2018, he received the ECCV Koenderink Test of Time Prize. He is a senior member of the IEEE.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/csdl.