Solving Multi-codebook Quantization in the GPU

Julieta Martinez(✉), Holger H. Hoos, and James J. Little

University of British Columbia, Vancouver, Canada
{julm,hoos,little}@cs.ubc.ca

Abstract. We focus on the problem of vector compression using multi-codebook quantization (MCQ). MCQ is a generalization of k-means where the centroids arise from the combinatorial sums of entries in multiple codebooks, and has become a critical component of large-scale, state-of-the-art approximate nearest neighbour search systems. MCQ is often addressed in an iterative manner, where learning the codebooks can be solved exactly via least-squares, but finding the optimal codes results in a large number of combinatorial NP-Hard problems. Recently, we have demonstrated that an algorithm based on stochastic local search for this problem outperforms all previous approaches. In this paper we introduce a GPU implementation of our method, which achieves a 30× speedup over a single-threaded CPU implementation. Our code is publicly available (https://github.com/jltmtz/local-search-quantization).

1 Introduction

Approximate nearest neighbour search is often a computational bottleneck of computer vision and machine learning applications. For example, in structure from motion [1] (the task of reconstructing the 3d structure of a scene given multiple images), the bottleneck arises when computing the relative location of several image pairs, which amounts to nearest neighbour searches in datasets of millions of SIFT [2] descriptors. Another task where nearest-neighbour search is the computational bottleneck is image retrieval, also known as image-based image search; in this task we are given an image of an object, and the goal is to find similar objects in a large collection of images. In practice, this amounts to finding the nearest neighbours of the descriptor of the image query among the stored descriptors of the images in the database.

In the past, the nearest-neighbour problem has been addressed using data structures that, with high confidence, prune the number of vectors to which the distance has to be computed. Two prominent examples of this approach include randomized kd-trees and hierarchical k-means (both implemented in the very successful FLANN library [3]). These approaches, however, do not scale well for large datasets, as they need to store all the database vectors in memory, and incur additional memory costs for the data structures that they use.

In the light of memory limitations, hashing approaches have become increasingly popular [4,5]. Hashing aims to learn a compact embedding of the data

G. Hua and H. Jégou (Eds.): ECCV 2016 Workshops, Part I, LNCS 9913, pp. 638–650, 2016.
DOI: 10.1007/978-3-319-46604-0_45

whose Hamming distance (which can be computed very fast using the `popcount` operation of modern processors) approximates the Euclidean distance of two original vectors.

Recently, however, it has been demonstrated that hashing approaches are significantly less accurate than multi-codebook quantization (MCQ) methods. Multi-codebook quantization is a generalization of k-means where, instead of assigning each point to a single centroid, a vector is assigned to multiple codebooks. The indices of such assignments can be used as a compact representation for the original vector, and the codes can be used to compute approximate distances to previously-unseen vectors using a few table lookups.

In this work we focus on a variant of MCQ known as Additive Quantization (AQ) [6], which has a simple, constraint-free formulation, but has proven hard to optimize. We have recently demonstrated [7] that the AQ formulation can be lifted to achieve state-of-the-art performance on a series of standard retrieval benchmarks by leveraging an encoding algorithm based on stochastic local search (SLS) [8]. In this work, we explore a GPU implementation of such algorithm that has allowed us to run our method on very large-scale datasets, and has lead us to obtain state-of-the-art performance on SIFT1B – a dataset of 1 billion visual descriptors.

2 Related Work

The work that introduced the idea of using vector quantization for nearest neighbour search was [9], and is called product quantization (PQ). In PQ, the optimization is actually trivial because the codebooks are assumed to be orthogonal to each other. A recent improvement to PQ was introduced by Babenko and Lempitsky [6], and is called additive quantization (AQ). In AQ, the codebooks are full-dimensional, which makes the encoding problem NP-hard. Babenko and Lempitsky suggested using beam search to solve the encoding problem, but this was identified as the main computational bottleneck of the system. Moreover, the performance of AQ has been largely surpassed by composite quantization [10], a method that also uses full-dimensional codebooks, but enforces the products of centroids in different codebooks to be constant.

Our main contribution is a method that solves the encoding problem in AQ using Stochastic Local Search. As we will show, this encoding problem is equivalent to multiple fully-connected pairwise Markov random fields (MRFs). MRFs are widely used in computer vision, where they are often used to model visual context. For example, Tung and Little [11] use an MRF to perform scene parsing (i.e., labelling every pixel in an image) on city landscapes, and Krähenbühl and Kultun [12] use filtering to perform MAP inference on MRFs with binary terms which are restricted to be Gaussian kernels, and apply their optimization method to scene parsing on natural images.

2.1 MRF Optimization on the GPU

There has been some previous work on porting different MRF optimization algorithms to the GPU. For example, Vineet and Narayanan [13] introduce

CUDACuts, a CUDA-based implementation of graph cuts, an algorithm for exact MRF optimization that is applicable to submodular MRFs. The method exploits shared memory in the push-relabel step of graph cuts, and is demonstrated on a Nvidia GTX 280 GPU. Zach et al. [14] introduce a data-parallel approach to MRF optimization based on plane sweeping, which is applicable when the pairwise terms follow a Potts prior. The work dates back to 2008, when CUDA was a more difficult language to program in than it is today, so the authors used the OpenGL API to program their solution. The authors demonstrate an implementation on an Nvidia Geforce 8800 Ultra GPU.

Most previous work focuses on MRFs as they are applied to image-related tasks such as scene segmentation. In these applications, the MRFs typically have a large number of nodes (1 per pixel), have a low number of labels (e.g., 3 in [14]), are sparsely connected, and have special constraints on form of their pairwise terms (e.g. Potts in [14] and submodular in [13]). Thus, it is unlikely that those implementations will fit our problem, which has a low number of nodes $m = \{7, 15\}$, a large number of labels ($h = 256$), is densely connected, and whose pairwise terms do not follow any particular formulation. Moreover, in our application all the MRFs share the pairwise terms, which is a rare occurrence in other MRF applications. We believe that the implementation that we are introducing in this paper is the first one to address this particular MRF case using graphics processors.

3 Problem Formulation

First, we introduce some notation, following mostly Norouzi and Fleet [15]. Formally, we denote the set to quantize as $X \in \mathbb{R}^{d \times n}$, having n data points with d dimensions each; MCQ is the problem of finding m codebooks $C_i \in \mathbb{R}^{d \times h}$ and the corresponding codes B_i that minimize quantization error:

$$\min_{C_i, B_i} \|X - [C_1, C_2, \ldots, C_m] \begin{bmatrix} B_1 \\ B_2 \\ \vdots \\ B_m \end{bmatrix} \|_2^2, \tag{1}$$

where $B_i = [\mathbf{b}_{i1}, \mathbf{b}_{i2}, \ldots, \mathbf{b}_{in}] \in \{0, 1\}^{h \times n}$, and each subcode $\mathbf{b}_i$ is limited to having only one non-zero entry: $\|\mathbf{b}_i\|_0 = 1$, $\|\mathbf{b}_i\|_1 = 1$. Letting $C = [C_1, C_2, \ldots, C_m]$ and $B = [B_1, B_2, \ldots, B_m]^\top$, we can rewrite expression 1 more succinctly as

$$\min_{C, B} \|X - CB\|_2^2. \tag{2}$$

It can be seen that, if we have only $m = 1$ codebooks, then the problem is reduced to k-means. This suggests an EM-like optimization for Eq. 2 inspired by Lloyd's algorithm – a popular method used to solve k-means.

3.1 Iterative Optimization

MCQ, as stated in Eq. 2 is non-convex and, in general NP-hard. The standard approach to solve this problem is an iterative 2-step process known as coordinate descent, akin to Lloyd's algorithm widely used in k-means. We iteratively solve two sub-problems

1. $C^{t+1} = \min_C \|X - CB^t\|_2^2$ (codebook update)
2. $B^{t+1} = \min_B \|X - C^{t+1}B\|_2^2$ (encoding).

Subproblem 1 is a quadratic program over C, and amounts to solving large-scale least-squares problem. In this paper we focus on subproblem 2 which, on the other hand, amounts to solving n independent NP-hard problems that can be expressed as fully-connected Markov Random Fields (MRFs). We now explain how this formulation is derived.

3.2 MCQ Encoding as MAP Estimation in Multiple MRFs

To make the formulation of MCQ encoding as MRFs more evident, we focus on the problem of finding the codes $[\mathbf{b}_1, \mathbf{b}_2, \dots, \mathbf{b}_m]$ that best approximate a single vector in $\mathbf{x}$. The goal in this case is to minimize the distance between $\mathbf{x}$ and its approximation $\hat{\mathbf{x}}$:

$$\min_{\hat{\mathbf{x}}} \|\mathbf{x} - \hat{\mathbf{x}}\|_2^2 = \min_{\hat{\mathbf{x}}} \|\mathbf{x}\|_2^2 - 2\langle \mathbf{x}, \hat{\mathbf{x}} \rangle + \|\hat{\mathbf{x}}\|_2^2 \tag{3}$$

In MCQ, the approximation $\hat{\mathbf{x}}$ is constrained to be the sum of single entries in m different subcodebooks C_i; that is, $\hat{\mathbf{x}} = \sum_{i=1}^m C_i \mathbf{b}_i$, therefore, we can rewrite the previous equation – during the encoding phase – as a minimization over the subcodebooks $\mathbf{b}_i$ that represent $\hat{\mathbf{x}}$

$$\min_{\{\mathbf{b}_1, \mathbf{b}_2, \dots, \mathbf{b}_m\}} \|\mathbf{x} - \sum_{i=1}^m C_i \mathbf{b}_i\|_2^2 = \|\mathbf{x}\|_2^2 - 2\langle \mathbf{x}, \sum_{i=1}^m C_i \mathbf{b}_i \rangle + \|\sum_{i=1}^m C_i \mathbf{b}_i\|_2^2 \tag{4}$$

$$= \|\mathbf{x}\|_2^2 - 2\sum_{i=1}^m \langle \mathbf{x}, C_i \mathbf{b}_i \rangle + \|\sum_{i=1}^m C_i \mathbf{b}_i\|_2^2. \tag{5}$$

Finally, we expand the norm of the approximation as

$$\|\sum_{i=1}^m C_i \mathbf{b}_i\|_2^2 = \sum_{i=1}^m \|C_i \mathbf{b}_i\|_2^2 + \sum_i^m \sum_{j \neq i}^m \langle C_i \mathbf{b}_i, C_j \mathbf{b}_j \rangle \tag{6}$$

It now becomes apparent that we can express the objective function as a sum of unary (univariate) and pairwise (bi-variate) terms. First of all, we can drop the $\|\mathbf{x}\|_2^2$ term, because it is not a function of $\mathbf{b}_i$. Next, we can sum both $-2\sum_{i=1}^m \langle \mathbf{x}, C_i \mathbf{b}_i \rangle$ and $\sum_{i=1}^m \|C_i \mathbf{b}_i\|_2^2$, which are both functions of a single code $\mathbf{b}_i$, and make them the unary term of our MRF. Finally, the term $\sum_i^m \sum_{j \neq i}^m \langle C_i \mathbf{b}_i, C_j \mathbf{b}_j \rangle$, which is a function of two codes, becomes our pairwise term. Since there is a term for every pair of codes, the resulting MRF is fully connected.

For each vector that we are encoding, the codes may take any value from 1 to h; thus our search space consists of h^m possible solutions (typically, $m = \{7, 15\}$ and $h = 256$), which renders the problem inherently combinatorial and, in general, NP-Hard.

Not only is each problem hard, but the number of problems, n, usually scales with big data. For example, imagine that we want to build a visual search engine that searches for visual content on every image on the Internet – in that case, n can easily be in the order of several trillions. The scale of this problem calls for methods that are effective (i.e., achieve low-cost solutions), and efficient (do so without taking much time). We also need to make use of state-of-the-art computational resources to tackle the problem within reasonable timelines. In this paper, we present a GPU implementation of a promising algorithm that addresses this problem.

4 Solution Methodology

Our approach to MCQ encoding is based on stochastic local search (SLS), a prominent class of algorithms that have, at several points in time, achieved state-of-the-art performance on several NP-hard problems. The basic idea behind SLS is to iteratively alternate between a greedy (local) search procedure, and randomized perturbations to the current solution. SLS methods are typically anytime, trading off computation for solution quality, and are also often easy to implement.

Our algorithm is defined by

1. An initialization procedure
2. A perturbation procedure
3. A local search algorithm
4. An acceptance criterion

Our initialization procedure gives a starting solution by initializing each subcode uniformly at random between 1 and h – this gives rise to an initial solution s. Similarly, our perturbation procedure takes the current solution, and changes $k \leq m$ subcodes to uniformly random values between 1 and h. Our local search algorithm is iterated conditional modes (ICM); an iterative algorithm itself that updates each subcode to its optimal value, while keeping the rest fixed. After perturbation and local search, we have a candidate solution s'. Our acceptance criterion simply keeps the solution with lower cost out of s and s'.

The pseucodode for our solution is given in the appendix as Algorithm 1.

4.1 Implementating SLS for Encoding

The initial implementation of our algorithm was done in Julia [16], a recent high-level language for scientific computing that runs on top of the LLVM compiler infrastructure[1], and as such is more suitable for low-level optimization. This implementation makes use of SIMD instructions where possible, and was heavily profiled and optimized.

A crucial observation that we obtained during the development of this single-threaded implementation was the importance of cached memory access when implementing ICM. In order to explain this subtlety, it is necessary that we talk about lookup tables in our implementation.

Shared Pairwise Tables. As we have mentioned, the objective function in encoding can be broken down in unary (single-parameter) and pairwise (two-parameter) terms – in fact, this is the easiest way to show that the problem is equivalent to an MRF. A first observation is that, although we have n different MRFs to solve, *they all actually share the same binary terms.* That is, the expression $\sum_i^m \sum_{j \neq i}^m \langle C_i \mathbf{b}_i, C_j \mathbf{b}_j \rangle$ is independent of $\mathbf{x}$, the vector that we are encoding. This means that we can precompute the values of the expression $\sum_i^m \sum_{j \neq i}^m \langle C_i \mathbf{b}_i, C_j \mathbf{b}_j \rangle$ for all the possible values of $C_i \mathbf{b}_i$ and $C_j \mathbf{b}_j$. For m codebooks, we have in total $m \cdot (m-1)/2$ tables of size $h \times h$. Handily, these tables are computed once and reused for all the MRFs that we are solving. For large n, the cost of computing these tables is negligible.

Unary Tables. The unary terms in our formulation correspond to the terms $-2\sum_{i=1}^m \langle \mathbf{x}, C_i \mathbf{b}_i \rangle$ and $\sum_{i=1}^m \|C_i \mathbf{b}_i\|_2^2$. While the second unary term is independent of the encoded vector, $\mathbf{x}$, the first term actually has to be computed for each vector that we are encoding. To speed up SLS, we compute m tables of size $h \times 1$ that store the sum of both unary terms for each value of $\mathbf{b}_i \in \{1, 2, \ldots, h\}$ for each $i \in \{1, 2, \ldots, m\}$.

Batched Implementation. A straightforward implementation of Algorithm 1 encodes each point sequentially. The innermost statement of the algorithm, that is, the search for the best value of a single code given the rest (line 19), becomes the computational bottleneck of the system. Moreover, one can see that every time the expression

$$\mathbf{b}_k := \arg\min_{\mathbf{b}_k} -2\langle \mathbf{x}, C_k \mathbf{b}_k \rangle + \|C_k \mathbf{b}_k\|_2^2 + \sum_{i \neq k}^{m} \langle C_k \mathbf{b}_k, C_i \mathbf{b}_i \rangle \tag{7}$$

is evaluated for different k, a different pairwise $h \times h$ lookup table has to be loaded into cache. A straightforward optimization for cache performance of this bottleneck results from optimizing multiple codes at the same time. This is a common pattern in large-scale machine learning systems, which are typically throughput – not latency – oriented, and is commonly known as "batching". This results in about 4× speedup in our CPU implementation, and is also used in our GPU implementation.

[1] http://llvm.org/.

CUDA Kernel Breakdown. Our GPU implementation consists of three main kernels: (a) a setup kernel, which initializes the solutions, computes unary lookup tables and creates a series of random number generators; (b) a perturbation kernel, which alters random codes in the solution, and (c) a local search kernel, which implements ICM.

Initialization Kernel. Our initialization kernel creates a CUDA random (`curand`) state for each data point in the database and, during the first iteration, uses the samples to create a random initial solution. The implementation is fairly straightforward and takes less than 1/1000 of the total optimization time.

Perturbation Kernel. In this kernel, we have one thread per data point, with the goal of maximizing the work/thread ratio. The task of each kernel is to choose $k = 4$ (choosen via validation) entries in the current solution, and perturb them to random values between 1 and h.

The main challenge in this kernel is that the choice of $k = 4$ entries to perturb amounts to sampling *without replacement* from the uniform distribution between 1 and m. Since there are no off-the-shelf functions for sampling without replacement in the `curand` library, we implemented our own version of *reservoir sampling*[2]. The pseudocode for this kernel is shown in the appendix as Algorithm 2.

It is easy to show by induction that each code has an equal probability k/m of being perturbed, and the algorithm finishes when k samples have been perturbed – which is most likely achieved before visiting every code. Although elegant, reservoir sampling leads to large thread divergence, which is likely suboptimal for the GPU architecture. In any case, the time spent in this perturbation step is negligible compared to the ICM kernel that we discuss next.

ICM Kernel. The job of the ICM kernel is to perform local search after the solution has been perturbed; this corresponds to lines 16–21 in Algorithm 1. By far, this is the most expensive part of our pipeline, so we discuss the implementation in more detail.

In short, we have three main tasks: (a) copying the pre-computed unary terms to shared memory, (b) adding the corresponding pairwise terms to the unary terms for each possible value of each code, and (c) finding the minimum of all h possibilities in each code.

Task (a) is a only done once; its implementation is straightforward and makes bootstrapping the process faster when moving from one code to another. Task (b) is memory-limited – the pairwise tables are too big to fit in shared memory, so they have to be read from global memory. Task (c) amounts to a reduce operation, and is thus compute-limited.

The implementation of (b) is slightly complicated, as it requires us to use more global memory in order to achieve coalesced access in main memory. Now it is important that we remember the pairwise tables of size $h \times h$ that store

[2] https://en.wikipedia.org/wiki/Reservoir_sampling.

the pairwise terms $\sum_i^m \sum_{j \neq i}^m \langle C_i \mathbf{b}_i, C_j \mathbf{b}_j \rangle$. While in theory we only require $m \cdot (m-1)/2$ tables (and they need not be filled completely, as the dot products are symmetrical), in practice we create $m^2 - m$ tables, and fill all their h^2 entries. The purpose of this extra-memory use is to allow coalesced memory access to the pairwise terms. We keep a matrix of pointers to the tables, such that the table indexed by $[i, j]$ contains the dot-products between the ith and the jth code, such that the jth row contains all h possible values of the jth code for a constant value of the ith code. This means that, when adding the pairwise terms for all values of the jth code, we can access the precomputed values in a coalesced manner. However, since ICM optimizes each code sequentially, it is likely that we will at some point need the transpose of this table, so we store a copy of the $[i, j]$ table such that $[j, i] = [i, j]^\top$. Thanks to this transpose, we can preserve coalesced access when searching all the h possible values of the ith code w.r.t. a fixed value for the jth code.

Once all the pairwise terms have been added, we run task (c) as a reduce `findmin` operation on the unary + pairwise terms. This is a reduction over the h values that we computed in task (b). Note, however, that the output is not the minimum value itself, but its index (a number from 1 to h), so this requires an extra array of indices to keep track of during reduction. We implemented all the optimizations available in the CUDA reduce guide[3] for this step.

5 Experimental Setup

We are interested in MCQ as a means to fast and accurate approximate nearest neighbour search. The standard protocol in the literature uses three partitions of the data: (1) train, (2) query, and (3) base. Typically, the train partition is used to learn the codebooks C; then, one must encode (that is, derive B for) the base vectors using the previously learned codebooks C. Finally, one uses a smaller query set to search for approximate nearest neigbours in the base partition.

Performance is measured with recall@N curves. These curves are monotonically increasing, and reflect the empirical probability distribution, computed over the query set, that the first N computed approximate neighbours contain the actual nearest neighbour. In information retrieval, recall@1 is considered the most important point in this curve.

5.1 Datasets

We focus on two large-scale datasets: SIFT1B and SIFT10M. SIFT1B consists of 128-dimensional SIFT descriptors; the set has 100 million vectors for train, 10 thousand vectors for query, and 1 billion vectors for database. For expedience, we only used the first million vectors from the training set – this is a shorthand commonly used in previous work [10,15]. SIFT10M has the same data as SIFT1B, except that the first 10 million vectors of SIFT1B are used as base. The dataset is publicly available at http://corpus-texmex.irisa.fr/.

[3] https://docs.nvidia.com/cuda/samples/6_Advanced/reduction/doc/reduction.pdf.

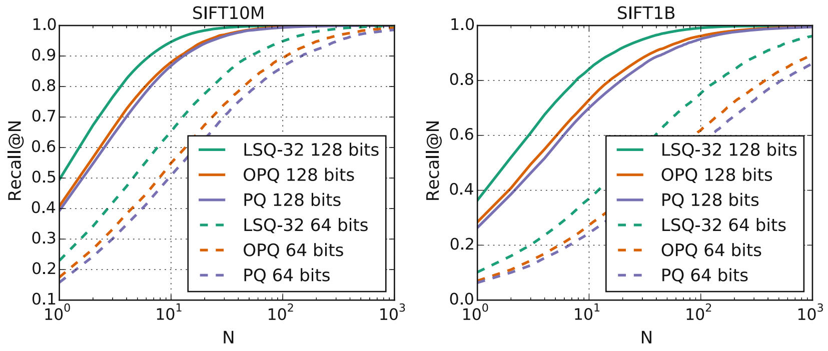

Fig. 1. Recall@N curves for very large-scale datasets: (left) the SIFT10M, and (right) SIFT1B.

5.2 Baselines

To compare nearest neighbour search performance, we use 3 baselines: product quantization (PQ) [9], which assumes that the codebooks are orthogonal, but initiated research in this area; optimized product quantization (OPQ) [15], which still assumes orthogonality, but also optimizes the subspace allocation via a rotation matrix $R \in SO(d)$, which can be solved for directly solving a Procrustes problem. Finally, we compare against composite quantization [10], a method that enforces the products between the codebooks (i.e., our pairwise terms), to be constant, and uses constrained optimization to find the codebooks. In CQ, codes are found using ICM, similar to our method. To reproduce PQ and OPQ we use the publicly available implementation of Norouzi and Fleet.[4] For CQ, we reproduce the values reported in [10]. We refer to our method as SLSQ-x, where x represents the number of ILS iterations used during database encoding. The number of ICM iterations (J in Algorithm 1, was set to 4).

6 Results

We report our results in recall@N, reproduced from our previous work [7] in Fig. 1, with detailed result on Table 1. Our method, when using 16 ILS iterations outperforms all the baselines, and when using 32 iterations further widens the performance advantage with respect to previous work. These results are shown here just for completeness, but are not the main focus of our work – when we set out to port our algoritm to the GPU, we already knew that it performed very well. Next, we report the speedups of our GPU implementation.

6.1 Speedups

For this particular report, the most important part for us to report is the speedup observed by our GPU implementation. We report those results on Table 2.

[4] https://github.com/norouzi/ckmeans.

Table 1. Detailed recall@N values for our method on large-scale datasets: SIFT10M and SIFT1B.

	SIFT10M – 64 bits			SIFT10M – 128 bits		
	R@1	R@10	R@100	R@1	R@2	R@5
PQ	15.79	50.86	86.57	39.31	54.74	74.89
OPQ	17.49	54.92	89.41	40.80	56.73	77.15
CQ [10]	21	63	93	47	64	84
LSQ-16	22.51	64.62	94.67	49.26	66.75	85.36
LSQ-32	**22.94**	**65.20**	**94.85**	**49.50**	**67.31**	**86.33**
	SIFT1B – 64 bits			SIFT1B – 128 bits		
	R@1	R@10	R@100	R@1	R@2	R@5
PQ	06.34	24.41	56.92	26.38	38.57	56.45
OPQ	07.02	27.34	61.89	28.43	40.80	59.53
CQ [10]	09	33	70	34	48	68
LSQ-16	09.73	35.82	73.84	35.32	50.84	70.66
LSQ-32	**10.18**	**36.96**	**75.31**	**36.35**	**51.99**	**72.13**

Table 2. Time spent per vector during encoding in our approach. "Sequential" refers to an LSQ implementation where ICM encodes each point sequentially (*i.e.*, does not take advantage of the shared pairwise terms). "Batched" is our LSQ implementation, which performs conditioning of shared pairwise terms among several data points.

Method	Sequential		Batched	
codebooks (m)	7	15	7	15
LSQ-16 (ms.)	1.52	7.02	0.53	2.01
LSQ-32 (ms.)	3.01	13.93	1.05	4.03

Method	GPU (batched)	
codebooks (m)	7	15
LSQ-16 (μs)	17.9	67.2
LSQ-32 (μs)	35.7	134.3

Method	Exhaustive NN	
codebooks (m)	8	16
PQ (μs)	42.6	77.9
OPQ (μs)	49.2	90.3

Focusing on the LSQ-32 results when using 15 codebooks, we see that our sequential implementation in Julia took almost 14 ms. per vector, our batched implementation reduced that time to roughly 4 ms. These times were measured on an Intel i7-3930K CPU, which runs at 3.2 GHz and has 12 MB of cache. The batched implementation was ported to a Titan X GTX GPU, and we observed a reduction in time to 134.3 µs, which results in a speedup of 30×. This means that encoding the SIFT1B base dataset can be done in slightly above 37 h, instead of the over 45 days that it would take with our single-threaded CPU implementation. Moreover, this GPU implementation has a running time comparable to that of PQ and OPQ in the CPU.

7 Conclusions and Future Work

We have demonstrated a GPU implementation of a recent SLS-based algorithm for encoding in multi-codebook quantization. The implementation takes advantage of the large-scale parallelism of modern GPUs by being throughput-oriented; it increases the load per thread during perturbation by making use of reservoir sampling, and ensures coalesced access to global memory by using multiple copies of the pairwise tables. We have shown that, in a typical scenario, our GPU implementation achieves a 30× speedup over a single-threaded CPU implementation. This has allowed us to tackle a dataset with 1 billion points in slightly over a day and a half of computation, instead of the month and half that it would take to do the same task with a single CPU.

A shortcoming of our comparison is that our CPU implementation is probably far from optimal. Ideally, we would like our CPU implementation to be based in C or C++, and even more ideally to be parallelized using `pthreads`. We dread the idea of doing the latter, and in any case we believe that the effort/speedup ratio will still highly favour the GPU implementation. On the other hand, we do not see any obvious ways of speeding up our current GPU implementation.

Future work may include further exploring the design space of stochastic local search algorithms for this problem, or improving the design of our CUDA program to obtain better performance from the GPU.

Appendix

```
// Initialize each code to a random value between 1 and $h$.
for $i := 1, 2, \ldots, m$ do
    $\mathbf{b}_i \sim \mathcal{U}\{1, h\}$ ;
end
// Save the current solution
$s := [\mathbf{b}_i, \mathbf{b}_2, \ldots, \mathbf{b}_m]$
// Loop over ILS iterations.
for $i := 1, 2, \ldots, I$ do
    // Perturb $k$ codes.
    for $j := 1, 2, \ldots, k$ do
        $idx \sim \mathcal{U}\{1, m\}$ // Sample a code to perturb (without replacement)
        $\mathbf{b}_{idx} \sim \mathcal{U}\{1, h\}$ // Perturb the code to a random value
    end
    // Run iterated conditional modes
    // Loop over the number of ICM iterations
    for $j := 1, 2, \ldots, J$ do
        // Update each code, keeping the rest fixed
        for $k := 1, 2, \ldots, m$ do
            $\mathbf{b}_k := \arg\min_{\mathbf{b}_k} -2\langle \mathbf{x}, C_k \mathbf{b}_k \rangle + \|C_k \mathbf{b}_k\|_2^2 + \sum_{i \neq k}^{m} \langle C_k \mathbf{b}_k, C_i \mathbf{b}_i \rangle$
        end
    end
    // Compute the cost of the newly-found solution
    $s' := [\mathbf{b}_i, \mathbf{b}_2, \ldots, \mathbf{b}_m]$
    // If the new solution is better, keep it
    if $cost(s') < cost(s)$ then
        $s := s'$
    end
end
return $s$
```

Algorithm 1. Our SLS algorithm for encoding in MCQ

```
needed := k // The number of codes that we still have to perturb
left := m // The number of codes that we can still visit
for $i := 1, 2, \ldots, m$ do
    // With probability needed/left, alter the ith code
    $r \sim \mathcal{U}(0,1)$;
    if $r < needed/left$ then
        $\mathbf{b}_i \sim \mathcal{U}\{1, h\}$
        $needed := needed - 1$
        if $needed \leq 0$ then
            return
        end
    end
    $left := left - 1$
end
```

Algorithm 2. Our perturbation method using reservoir sampling

References

1. Crandall, D.J., Owens, A., Snavely, N., Huttenlocher, D.P.: Sfm with mrfs: discrete-continuous optimization for large-scale structure from motion. IEEE Trans. Pattern Anal. Mach. Intell. **35**(12), 2841–2853 (2013)
2. Lowe, D.G.: Distinctive image features from scale-invariant keypoints. IJCV **60**(2), 91–110 (2004)
3. Muja, M., Lowe, D.G.: Fast approximate nearest neighbors with automatic algorithm configuration. In: VISApp, vol. 1 (2009)
4. Gong, Y., Lazebnik, S.: Iterative quantization: a procrustean approach to learning binary codes. In: CVPR (2011)
5. Norouzi, M., Fleet, D.J.: Minimal loss hashing for compact binary codes. In: ICML (2011)
6. Babenko, A., Lempitsky, V.: Additive quantization for extreme vector compression. In: CVPR (2014)
7. Martinez, J., Clement, J., Hoos, H., Little, J.: Revisiting additive quantization. In: ECCV (2016)
8. Hoos, H.H., Stützle, T.: Stochastic local search: Foundations & applications. Elsevier, Amsterdam (2004)
9. Jégou, H., Douze, M., Schmid, C.: Product quantization for nearest neighbor search. TPAMI **33**(1), 117–128 (2011)
10. Zhang, T., Du, C., Wang, J.: Composite quantization for approximate nearest neighbor search. In: ICML (2014)
11. Tung, F., Little, J.J.: CollageParsing: nonparametric scene parsing by adaptive overlapping windows. In: Fleet, D., Pajdla, T., Schiele, B., Tuytelaars, T. (eds.) ECCV 2014. LNCS, vol. 8694, pp. 511–525. Springer, Heidelberg (2014). doi:10.1007/978-3-319-10599-4_33
12. Krähenbühl, P., Koltun, V.: Efficient inference in fully connected crfs with gaussian edge potentials. arXiv preprint arXiv:1210.5644 (2012)
13. Vineet, V., Narayanan, P.: Cuda cuts: fast graph cuts on the gpu. In: IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops, CVPRW 2008, pp. 1–8. IEEE (2008)

14. Zach, C., Gallup, D., Frahm, J.M., Niethammer, M.: Fast global labeling for real-time stereo using multiple plane sweeps. In: VMV, pp. 243–252 (2008)
15. Norouzi, M., Fleet, D.J.: Cartesian k-means. In: CVPR (2013)
16. Bezanson, J., Edelman, A., Karpinski, S., Shah, V.B.: Julia: a fresh approach to numerical computing. arXiv preprint arXiv:1411.1607 (2014)