Accelerating High-Dimensional Nearest Neighbors for Video Search

Carlo C. del Mundo and Alexa Yao
Dept. of Computer Science and Engineering
Dept. of Electrical Engineering
{cdel, jyao}@uw.edu

ABSTRACT

The k-nearest neighbor algorithm (kNN) is a critical algorithm used extensively in fields such as Computer Vision, Robotics, and Machine Learning. In this work, we address the performance of FLANN, a popular kNN library, at the node-level by co-designing indexing and search algorithms with software support. We characterize, profile, and optimize FLANN for high-dimensionality (e.g., ≥ 1096) for two systems: an enterprise-grade Intel E5520 CPU and a low-power ARM Cortex A15 CPU. Our profiling suggests that the priority queue k-means kNN algorithm is the best candidate for software acceleration as 98.8% of indexing and 97.5% of search time is spent on calculating the Euclidean distance between two vectors. We implemented vectorization with Intel SSE and ARM Neon extensions and multithreading with OpenMP. Overall, our software optimizations improve search and indexing by factors of 2.27-fold and 4.07-fold, respectively, for the ARM CPU. Finally, we projected the performance in offloading the computations to an on-board GPU in the ARM platform. Our projection suggests that even if GPU computation took zero time, data movement in a unified CPU/GPU SoC platform would still degrade performance by a factor of 9.90-fold compared to the optimized Intel CPU version.

1. INTRODUCTION

To index and search all of the videos in the Internet requires massive resources. Using commodity parts and existing libraries, a conservative estimate for a system requires 32 petabytes to store feature data, 480,000 CPU cores to build and update the index, and 1000 servers to service 1 million concurrent clients at a latency on the order of seconds. Such requirements in both power consumption and performance rival those of the top supercomputers [1]. While it may be feasible to add commodity CPUs to increase raw processing power, the exponential rates of multimedia content creation and consumption exacerbate the scalability of network infrastructure and memory subsystems. In 2013 alone, approximately 127 billion images and 620 million videos were uploaded to Facebook and YouTube respectively [2][3]. In the same year, traffic volumes from Netflix and YouTube accounted for more than 50% of global Internet traffic [4]. This growth in data has created a landscape where multimedia search engines will be essential to organize and reason about the available data.

This work seeks to enable video-based search (i.e., searching by video content instead of text-based keyword search) by characterizing and optimizing the core search algorithm: k-nearest neighbors. Although video-based search is the motivation of this work, we narrow our focus on a smaller problem: image-based search. The same principles apply to image-based search with vector size being the primary difference. A 5-minute videos is approximately 16-times larger than a single image in feature extraction. Using real image data gathered from the Yahoo! Flickr dataset [5], we developed an image-based search system and describe its operation in Section 2. A demo of our image search system in action can be viewed here: https://www.youtube.com/watch?v=iS84RKRZYt#t=395

The following project work was completed during CSE 548 and is summarized as follows.

1. We designed and implemented a prototype image search system for a single-node using 32,000 images drawn from Flickr (Section 3).
2. We characterized the FLANN library and we identified that program performance can be improved significantly by accelerating the Euclidean distance function (Section 4).
3. We evaluated two software-level optimizations — vectorization and multithreading — for two platforms: a low-power ARM SoC and a server grade x86 platform. Our software-level optimizations improve the performance of the baseline FLANN library by factors of 2.27-fold and 5.16-fold for searching and indexing, respectively, on the x86 platform, and 1.45-fold and 4.07-fold on the ARM platform (Sections 4.2 and 4.3).
4. We projected the performance of a fused CPU+GPU architecture SoC using the bandwidthTest SDK sample in the NVIDIA CUDA library. Despite a unified memory interface, our results indicate a slowdown in performance by a factor of 9.90-fold for search compared to the optimized CPU code (Section 4.4).

For this project, we leveraged the following tools. First, we used profiling tools including: (1) gprof, the GNU profiler; (2) pprof, the Google CPU profiler; and (3) Intel VTune Amplifier, a proprietary profiler for Intel x86 processors. Second, we used several software frameworks and libraries namely: (1) x86 SSE SIMD intrinsics, a lightweight C interface for using vector instructions on Intel processors [6], (2) the ARM Neon frameworks for vectorizing code on ARM platforms [7], (3) OpenMP, an API for shared memory multiprocessing [8], and (4) CUDA, an API for programming NVIDIA graphics processing units (GPUs) [9].
2. VISUAL SEARCH ENGINES

A visual search engine uses images or videos as search input and returns media of similar visual characteristics. Visual search consists of feature extraction (Figure 1a and b) and similarity search (Figure 1c). For feature extraction on images, we leverage the Berkeley Caffe framework [10], a convolutional neural networks based feature extractor generating 4K-dimensional feature vectors per image. In contrast to images, feature extraction for videos generates approximately 128K-dimensional feature vectors (a 16-fold difference from images) for 5-minute videos [11]. Similarity search compares each feature vector pairwise using k-nearest neighbors (kNN) embodied by Euclidean distance calculations. The region demarcated in Figure 1c is our region of interest and we seek to improve the performance of this stage in the visual search engine pipeline. We use the FLANN library as a baseline to perform the kNN operations [12].

The primary problem of nearest neighbor search is the curse of dimensionality. Several prior works have indicated that as the dimensionality of the dataset increases, hierarchical indexing algorithms such as kd-trees and k-means clustering degrades to a brute-force linear search. We evaluate this claim using two datasets: (1) a dataset that randomly samples from a uniform distribution, and (2) using images features generated from a convolutional neural network.

Parallel Randomized kd-trees. This approach approximates nearest neighbor search by constructing and searching multiple kd-trees in parallel. For each kd-tree, the index is created by randomly selecting the top-5 most varying dimensions, resulting in each tree having a different permutation of hyper plane cuts. In search, the priority queue is maintained and is shared across all trees keeping track of already visited branches. In our experiments, we used the default configuration of 4 parallel trees. We will refer to the parallel randomized kd-trees approach simply as kd-tree.

Priority Search k-means. In this approach, a priority queue and an indexing structure is used for approximate nearest neighbor search. The hierarchical indexing data structure is constructed by initially clustering the dataset into k clusters using the traditional k-means algorithm. Then, points in the same cluster are recursively clustered until the resulting clusters are individual points themselves. The resulting index is similar to the kd-tree index, but unlike the kd-tree, decompositions are performed across dimensions. In search, the tree is traversed by comparing the distance between the query point and each cluster center. The tree is traversed to the cluster centers with minimum distance to the query. This method does not guarantee exact searching as poor traversals could lead to falsely identifying neighbors.
in the wrong cluster. To compensate for this error, a priority queue is maintained and the algorithm traverses other branches of the index. Untraversed branches are added to the priority queue in the order of decreasing distance to the query point. The level of approximation is controlled by the maximum number of branches to be visited. In our experiments, we used the default configuration of $k = 32$ and iterations $= 11$ (per cluster step). We will refer to the priority search k-means approach simply as \textit{k-means}.

### 3.2 Microbenchmarking FLANN

To characterize each algorithm as a function of load, we devised a microbenchmark that varies: (1) number of images in the database or \textit{cardinality}, (2) number of features per images or \textit{dimensionality}, (3) number of requested results or the \textit{k most similar results}, (4) degree of precision or accuracy, and (5) number of cores. For each of the three kNN search algorithms, we measure: search and indexing time. Images were collected using the Yahoo! Flickr dataset [5]. We leveraged Berkeley Caffe [10], a popular convolutional neural network, to extract image features. We summarize our results below.

**Cardinality and Accuracy.** Figure 2 shows how search
time varies as the data set size increases. Accuracy is defined as the number of correctly returned neighbors with respect to the exact linear search. For exact 100% accuracy, the kd-tree and k-means approaches perform worst than the brute-force linear search substantiating the need for approximation. For the 40% precision k-nearest neighbors, the performance of both approaches outperform linear search by about 9-fold. We note that the 32,000 images drawn from Flickr do not represent a particular scene, object, or place. We expect that for a larger dataset size (> 1 million images), images with similar visual content will improve the performance of approximate nearest neighbor search at all approximation levels.

Dimensionality. Figure 3 depicts indexing and search time as dimensionality is varied. For the hierarchical kd-tree and k-means approaches, all feature vectors are indexed into an efficient search data structure which enables sublinear search time scaling with respect to data size. The one time cost of indexing is amortized by the number of searches performed over it. Indexing is three orders of magnitude slower than search time (not shown), and while search time grows linearly with the dimensionality of the data, the indexing time grows linearly with both dimensionality and cardinality. The index must also adapt and be rebuilt when new video content is uploaded to or deleted from the database. We note that FLANN does not have support for incremental indexing. Thus, if images are added or deleted, the entire index must be rebuilt from scratch.

Search Load. Figure 4 varies the search load as a function of the number of queries (left) and the number of results returned per query (right). We note that the number of results returned per query significantly affects the performance of approximate nearest neighbor search. At k = 64 neighbors, kd-trees and k-means degrade to linear search. Beyond k = 64 neighbors, both hierarchical algorithms perform worse than linear search.

Parallelism. Varying the number of cores only affects the search time and not indexing (not shown). The speedup is proportional to the number of cores used as the search queries can be subdivided evenly between the cores.

3.3 Profiling FLANN

Figure 5 and Figure 6 depicts the profile of the two hierarchical search algorithms: kd-trees and k-means. Euclidean distance calculations represent a majority of the execution time for search in kd-trees and k-means. We note especially that for k-means, the Euclidean distance function comprises 98.8% of execution time for searching and 97.5% for indexing. Thus, we focus our efforts on the k-means algorithm by optimizing the Euclidean Distance function.

4. APPROACH & RESULTS

4.1 Experimental Testbed

Our experimental testbed consists of two CPU platforms: (1) an Intel x86 E5520 CPU (Quad Core), and an (2) ARM Cortex A15 CPU. We used g++ 4.4 as our core compiler, gperftools v2.2.1, and gprof 2.2 as our CPU profiler. Results are shown with optimization flags, -O2, enabled.

For the Intel platform, we used 32,000 dataset images each at 4096 dimensions, 256 query images, with k = 16 neighbors. Due to the small amount of DRAM present in the ARM Cortex A15 SoC, we used 16,384 dataset images.

4.2 Vectorization

In block-based looping structure such as the Euclidean distance, a good candidate for optimization is vectorization. Inside the processor’s compute subsystem lies computational units that can perform arithmetic operations in a vector-wise fashion. Instead of computing one arithmetic value per cycle, vectorization allows you to compute the same operation with multiple data values per cycle. This single-instruction multiple-data (SIMD) style of computation allows theoretical speedups up to the length of the vector. With processors supporting Intel SSE/SSE2, up to four 32-bit data operands can be executed in the same cycle.

Listing 1 lists the Euclidean Distance function in FLANN.

```c
RType EuclideanDistance (DType * A, DType * B)
{
    RType diff , result ;
    for (int i = 0 ; i < DIM ; i++)
    {
        diff = RType ( A[i] - B[i]) ;
        result += diff * diff ;
    }
    return result ;
}
```

Figure 5: kd-tree: Search and Index breakdown by function.

Figure 6: k-means: Search and Index breakdown by function.
RType EuclideanDistance(DType* A, DType* B) {
    RType diff, result;
    for (int i = 0; i < dim; i += 4) {
        __mm128 a_i = __mm_load_ps(&A[i]);
        __mm128 b_i = __mm_load_ps(&B[i]);
        c_i = __mm_sub_ps(a_i, b_i);
        result += sum;
    }
}

Listing 2: x86: Naive

RType EuclideanDistance(DType* A, DType* B) {
    RType result;
    for (int i = 0; i < dim; i += 4) {
        __mm128 a_i = __mm_load_ps(&A[i]);
        __mm128 b_i = __mm_load_ps(&B[i]);
        c_i = __mm_sub_ps(a_i, b_i);
        __mm_store_ps(&sum, c_i);
    }
    return result;
}

Listing 3: x86: Dot Product

dType and RType can be a single-precision or double-precision value, and DIM is the dimensionality of the feature vector \((\geq 4096)\). The calculation can be summarized as the dot product between the differences of two vectors. For the k-means algorithm, the indexing algorithm requires double-precision computation while the search algorithm requires only single-precision. We focus on single-precision floating point Euclidean distance calculations.

Intel x86 E5520. In Intel x86 SSE2 instructions, we implemented vectorization in three different ways: (1) Naive (Listing 2), (2) Unrolled (not shown), and (3) Dot Product (Listing 3). Each version yielded a performance improvement of 1.75x, 2.27x, and 1.78, respectively. In Naive, two sets of 4x32-bit floating point operand groups are loaded into SIMD registers. The subtraction and multiplication operations are done in SIMD registers with the sub and mul instructions. The squares are summed and reduced into a single scalar value via the vadd instruction. In Unrolled (not shown), we explicitly unroll the loop by loading two sets of sixteen 32-bit floating point operands. Savings occurs primarily by reducing the number of horizontal add SIMD instructions. Finally, in Dot Product, we use a special instruction for dot products, dp. Overall, the Unrolled version outperformed the other schemes as loop counter overhead is reduced by a factor of four.

ARM Cortex A15. In ARM NEON instructions, we implemented vectorization in three approaches: (1) Naive (Listing 4), (2) Intermediate (Listing 5), and (3) Advanced (Listing 6). Version 2 and 3 yielded a performance improvement of 1.27 and 1.45 respectively. In Naive, two sets of 4x32-bit floating point operand groups are transferred back to CPU level individually using the vgetq_lane instruction, then summed and accumulated into a single floating value. The last step to read out four values from the SIMD registers one by one and then calculates the sum in the CPU level degrades the performance. In our simulation, this method shows a performance hit of nearly double the run-time compared to the regular non-SIMD implementation (Listing 4). As we narrowed down the overhead part in Version 1, in Intermediate (Listing 5), and (3) Advanced (Listing 6). Version 2 and 3 yielded a performance improvement of 1.27-fold and 1.45-fold respectively. In Naive (Listing 4), we focused on ensuring the sum operation is completed in SIMD via instructions vadd and vpd to yield a 32-bit scalar value, and then transfer it back to the CPU level. This implementation, the savings of reducing the number of SIMD register read instructions and horizontal sum time showed promising improvement from twice the non-vectorized run-time down to three quarter of the non-vectorized run-time. Unfortunately, the NEON Extension does not offer dot product related instructions natively, so in Advanced, our final approach of vectorization in ARM architecture, we extended from the intermediate implementation, and focused on finding an alternative solution in place of the Dot Product approach as in the X86. To perform a dot product similar operation, we first changed the number of unrolled sets to \(2^c\) (\(2 < c \leq 5\), \(4 < 2^c \leq 64\)), which loads a total \(2^c\) operand groups (each group consists of two sets of 4x32-bit floating point operands) in one loop, and every 4x32-bit operand group yields a vector of four squared floating values, and these vectors are accumulated during the multiplication step via the vmlaq instruction. Overall, after comparing the run-time by scaling c, when c is 4 and the number of unrolled sets is four times the previous implementation, it yielded the best run-time and also outperformed the non-vectorized implementation by a factor of 1.45.

Our overall results are listed in Table 1. We evaluated our optimizations with K-means using 32,768 dataset images, 256 query images, 4096 dimensions, \(k = 16\) neighbors at 4 cores and 60% accuracy. For the Intel platform, we improve performance by a factor of 2.27-fold and 5.16-fold for searching and indexing, respectively. This translates into a rate of 2\(\mu s\) per single-precision Euclidean distance call (or 13.91\(GB/s\)), and 3\(\mu s\) per double-precision Euclidean distance call (or 9.67\(GB/s\)). For the ARM platform, we improve performance by a factor of 1.45-fold and 2.4-fold for searching and indexing, respectively.

4.3 Multithreading

Today’s CPUs employ multiple-cores typically on the range of 4-16 cores. These cores are autonomous work units and program execution is not limited to one processing core. To make use of all resources inside the CPU, we apply another optimization, multithreading, using the OpenMP framework.

In k-means, there are several opportunities to use multiple threads to improve program performance. Our profiling suggests that the code snippet listed in Listing 7 comprised a
majority of execution time during k-mean’s indexing phase. We added OpenMP annotations and improved the performance of indexing by a factor of 5.16-fold using 4-way multithreading on the server platform.

We also attempted to apply multithreading to the Euclidean distance stage. Unfortunately, adding multiple threads to execute a relatively small function (i.e., 4096 loop iterations) introduced too much overhead and degraded performance.

4.4 NVIDIA Jetson TK1

The Jetson TK1 board is a heterogeneous CPU/GPU system-on-chip (SoC) consisting of an ARM Cortex A15 CPU and a 192-core GPU based on the NVIDIA Maxwell architecture [17]. The “4+1” CPU consists of four cores supported by a fifth companion core implemented in a low-leakage technology. The idea is to run the first four cores in a high-operating speed with the fifth “shadow” core acting as a supporting unit. The main quad core contains a 2 MB shared L2 cache, and the shadow core contains its separate 512 KB L2 cache. Our introductory board consists of 2 GB DRAM with 16 GB of storage capacity running Ubuntu 14.04. It draws a power of 5 watts.

In contrast to discrete CPU+GPU systems, the ARM CPUs and on-board NVIDIA GPU uses the same DRAM memory space, but have separate partitions of the data. Thus, explicit data transfers between CPU and GPU do not rely on a separate PCI-e bus. To test the throughput between memory transfers between the CPU and GPU, we leveraged the NVIDIA CUDA SDK sample program bandwithTest. This program evaluates the data transfer throughput to and from the GPU by measuring the time it takes to perform a memcpy operation. Table 2 depicts the benchmarks results.

Performance Projection. Assuming a streaming model)

Listing 5: NEON: Intermediate

```c
RType EuclideanDistance(DType A, DType B) {
    RType result;
    for (int i = 0; i < DIM; i+=4) {
        float32x4_t a_vec = vld1q_f32(A);
        float32x4_t b_vec = vld1q_f32(B);
        float32x4_t diff_vec = vsubq_f32(a_vec, b_vec);
        float32x4_t sq_vec = vmulq_f32(diff_vec, diff_vec);
        result += vgetq_lane_f32(sq_vec, 0) + vgetq_lane_f32(sq_vec, 1) + vgetq_lane_f32(sq_vec, 2) + vgetq_lane_f32(sq_vec, 3);
        A += 4;
        B += 4;
    }
    return result;
}
```

Listing 6: NEON: Advanced

```c
#pragma omp parallel for num_threads(4), schedule(static)
for (int i=0; i<indices_length; ++i) {
    DType sq_dist = euclidean_distance(points[i], dcenters[0]);
    for (int j=1; j<branching; ++j) {
        DType new_sq_dist = euclidean_distance(points[i], dcenters[j]);
        ...
}
```
Table 1: Overall Results for K-Means with Optimizations Applied. For the Intel E5520, 32,000 images were used. Due to resource constraints, only 16,384 images were used for the ARM Cortex A15.

<table>
<thead>
<tr>
<th></th>
<th>Search</th>
<th>Indexing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel E5520 CPU</td>
<td>FLANN (Baseline) 15.88 s</td>
<td>193.00 s</td>
</tr>
<tr>
<td></td>
<td>Vectorization Only 6.99 s</td>
<td>96.33 s</td>
</tr>
<tr>
<td></td>
<td>Multithreading Only</td>
<td>71.49 s</td>
</tr>
<tr>
<td>Speedup (w/ All)</td>
<td>2.27x</td>
<td>5.16x</td>
</tr>
<tr>
<td>ARM Cortex A15CPU</td>
<td>FLANN (Baseline) 13.70 s</td>
<td>139.99 s</td>
</tr>
<tr>
<td></td>
<td>Vectorization Only 9.45 s</td>
<td>96.3 s</td>
</tr>
<tr>
<td></td>
<td>Multithreading Only</td>
<td>58.51 s</td>
</tr>
<tr>
<td>Speedup (w/ All)</td>
<td>1.45x</td>
<td>4.07x</td>
</tr>
</tbody>
</table>

Table 2: bandwidthTest provided by the NVIDIA CUDA SDK. H = Host (CPU), D = Device (GPU), A -> B = memcpy from A to B.

<table>
<thead>
<tr>
<th></th>
<th>Pinned</th>
<th>Pageable</th>
</tr>
</thead>
<tbody>
<tr>
<td>H -&gt; D</td>
<td>995 MB/s</td>
<td>1415 MB/s</td>
</tr>
<tr>
<td>D -&gt; H</td>
<td>5405 MB/s</td>
<td>1405 MB/s</td>
</tr>
<tr>
<td>D -&gt; D</td>
<td>9001 MB/s</td>
<td>11738 MB/s</td>
</tr>
</tbody>
</table>

5. CONCLUSION

After analyzing the FLANN library, we narrowed down our optimization focus on Euclidean distance because it takes the majority of run-time for both indexing and searching in k-means. We boosted program performance by using two software-level optimizations: vectorization and multithreading. When dealing with a large set of data in vectorization, unrolling the data is a good practice. The FLANN library, by default, unrolls in groups of four which allows the compiler to optimize code during compilation. Our initial naive implementations could not outperform the unrolled versions until we introduced more advanced optimizations. Both Intel SSE and ARM NEON extensions utilizes a 128-bit SIMD architecture. However, we learned that NEON instructions lack double-precision SIMD support as well as specialized dot product instructions complicating our efforts. In terms of productivity, multithreading was easier to implement because the k-means indexing structure is block-based. Only two lines of code were necessary to gain a speedup for indexing. Lastly, working within the FLANN library was complicated. To be productive and effective, we created standalone test modules to accelerate our workflow, and they were later integrated into the FLANN library for overall evaluation.

6. REFERENCES