Int J Parallel Prog (2013) 41:595–620
DOI 10.1007/s10766-013-0239-8

Auto-tuning Similarity Search Algorithms on Multi-core Architectures

Buğra Gedik

Received: 30 June 2012 / Accepted: 17 January 2013 / Published online: 2 February 2013

Abstract In recent times, large high-dimensional datasets have become ubiquitous. Video and image repositories, financial, and sensor data are just a few examples of such datasets in practice. Many applications that use such datasets require the retrieval of data items similar to a given query item, or the nearest neighbors (NN or k-NN) of a given item. Another common query is the retrieval of multiple sets of nearest neighbors, i.e., multi k-NN, for different query items on the same data. With commodity multi-core CPUs becoming more and more widespread at lower costs, developing parallel algorithms for these search problems has become increasingly important. While the core nearest neighbor search problem is relatively easy to parallelize, it is challenging to tune it for optimality. This is due to the fact that the various performance-specific algorithmic parameters, or "tuning knobs", are inter-related and also depend on the data and query workloads. In this paper, we present (1) a detailed study of the various tuning knobs and their contributions on increasing the query throughput for parallelized versions of the two most common classes of high-dimensional multi-NN search algorithms: linear scan and tree traversal, and (2) an offline auto-tuner for setting these knobs by iteratively measuring actual query execution times for a given workload and dataset. We show experimentally that our auto-tuner reaches near-optimal performance and significantly outperforms un-tuned versions of parallel multi-NN algorithms for real video repository data on a variety of multi-core platforms.

Keywords Nearest neighbor search · Auto-tuning · Parallelization

B. Gedik (✉)
Department of Computer Engineering, Bilkent University, Ankara, Turkey
e-mail: bgedik@cs.bilkent.edu.tr

1 Introduction

For the last decade, we have witnessed the proliferation of multi-core processors, fueled by diminishing gains in processor performance from increasing operating frequencies. CPUs with eight or more cores have become common in the commodity and server-class general-purpose processor markets, and vendors promise to increase the number of cores per chip [11]. At the same time, the gap between the power of these modern computer systems and their programmability is widening. Even for parallel algorithms that are easy to express and understand, getting most performance out of a multi-core system is highly challenging. Due to the architectural variety of modern multi-core processors, algorithms would need to be individually tuned at the level of algorithmic parameters, for each processor. Performing this tuning in a manual manner is obviously not a scalable solution and more automated ways to tune algorithms for different systems are needed.

For instance, the number and sizes of caches and their configurations often differ significantly across different processors. For instance, in IBM's POWER6 processor [17], both cores share a common L2 cache, whereas in an AMD Athlon X2 processor [1], both cores have their own private L2 caches but a shared L3 cache. Yet another configuration is found in the Intel Itanium 2 processor [13] where each core has its own private L2 and L3 caches. Furthermore, differences in memory latencies, processor frequencies, pipeline lengths, branch prediction hardware, SIMD support, hyper-threading, etc., all add to the architectural variety. Each processor design has different performance characteristics and requires subtle modifications to a given parallel algorithm to fully use the processing power. For example, synchronizing two threads to work on the same data from main memory may show more benefits on systems with shared caches than on systems with core-local caches.

The complexity of modern CPUs can also be gauged from their programming manuals. The Intel 64 and IA-32 Architectures Optimization Reference Manual [14] is nearly 600 pages long, with 75 pages covering caching and multi-core optimizations. Similarly substantial manuals exist for other architectures, underlining the challenge of tuning algorithms for different multi-core systems.

In the context of multi k-NN search, we explore a different way to tackle this exponential tuning problem. We propose and validate experimentally a two-step approach consisting of:

(1) the extraction of promising tuning parameters from a parallel search algorithm, and
(2) the automatic tuning of the algorithm by systematically exploring the parameter space for a given architecture and workload.

In this paper, we investigate the parallel version of the multi k-NN search algorithm. This algorithm operates on a large set of multi-dimensional vectors. The input to the algorithm is a set of query vectors and the output is the set of the k closest vectors (in Euclidean distance) to each query vector. This algorithm has many applications. Here, we provide two examples.

Use case 1: Multimedia search. Multimedia search engines that provide capabilities to find the k most similar images to a given query image are based on this algorithm. They would typically extract so-called "feature-vectors" from each image that describe the image's properties, such as the color distribution, in the form of an (oftentimes high-dimensional) vector. Similarly, the query image is transformed into a feature-vector and the distance measure is chosen based on the desired image similarity criteria. Since such search engines typically serve hundreds or thousands of user requests per second, we can assume that not one but many queries are pending at any one time, leading to a multi k-NN search problem.

Use case 2: Copyright violation detection. Many video hosting sites are legally required to remove copyrighted material as quickly as possible in the context of the Digital Millennium Copyright Act [20]. Since the detection is only possible by inspecting the content of each video, this used to be a labor and cost-intensive task. By exploiting modern parallel architectures, one can employ the multi k-NN search algorithm to perform this task in a more cost-effective way as follows: Assume each copyrighted video is divided into shots and for each shot, a feature vector based on shot characteristics has been extracted. Once a new video gets uploaded to the hosting site, this video is also divided into shots and the same feature vectors are extracted per shot. To find similar videos in the copyright collection, one can now perform a multi k-NN search with all shot vectors of the new video as the query vectors. In a post-processing step, the returned k-NNs for each shot are compared to see if there is a majority that originates from the same copyrighted video. If that is the case, a potential copyright violation has occurred. In this example, the multiple queries occur due to the multiple shots, in the previous example it was due to the multiple users.

Besides its wide applicability, an interesting feature of the multi k-NN algorithm is that it can be optimized based on a large number of tuning decisions, whose interplay is far from obvious as we will illustrate. Such tuning decisions include SIMDization, multi-threading, load balancing, cache sharing, query partitioning, and data striping.

An important challenge in this context is to identify the right set of tuning knobs and to understand the interactions between them. Unlike the tuning of tight computational kernels like BLAS and FFT libraries, the multi k-NN tuning requires finding the right setting for various *algorithmic* knobs (such as short-cutting, partial result sharing, query batching) whose impact on performance is highly dependent on the characteristics of the dataset.

In summary, the contributions of this paper are:

1. A detailed study of the various tuning knobs, their interactions with each other, and their contributions on increasing the query throughput for parallelized versions of the two most common classes of high-dimensional multi k-NN search algorithms: linear scan and tree traversal.
2. An offline auto-tuner for setting these knobs by iteratively measuring actual query execution times for a given workload and dataset.

We show experimentally that our auto-tuner provides near optimal solutions and significantly outperforms un-tuned versions of parallel multi-NN algorithms for real video repository data on a variety of multi-core platforms.

The remainder of this paper is organized as follows. In Sect. 2, we discuss approaches to multi k-NN searching and other related work on tuning for multi-core architectures. Section 3 introduces the problem of multi k-NN search in more detail and gives an overview of the parallelization dimensions. In Sect. 4, we present the various tuning knobs for this algorithm in detail and explain how each of them affects the algorithm performance. Our heuristic to automatically tune the algorithm for a specific architecture using these tuning parameters is discussed in Sect. 5. Section 6 concludes the paper.

2 Related Work

Past work has investigated various ways to accelerate high-dimensional data access and more specifically k-NN searching. One solution is to improve the cache reuse when multiple k-NN queries are running concurrently in a batched fashion. Another solution is to use multi-dimensional index structures to reduce the amount of data that needs to be accessed. Finally, data compression has been proposed as a way to reduce accesses. We discuss past work from each area below.

Qiao et al. [21] present algorithms for improved cache reuse during concurrent table scan operations on multi-core architectures. Their goal is to improve the throughput of business intelligence queries that typically perform large amounts of table scans with aggregation operations. They are mainly concerned with ensuring that the aggregation data structures fit into the cache of each core in order to avoid thrashing. For that purpose, they run multiple queries in batches with the batch size based on cache size and expected data structure sizes. In contrast, our work focuses mainly on NN search queries and how the various optimization dimensions can be optimized in an architecture-agnostic manner. Some of these optimization dimensions are not present or are fixed in [21].

A large number of multi-dimensional index structures for k-NN searching have been proposed in the past. They can be categorized into roughly two groups: *scan-based* approaches and *index-based* approaches. An example for a scan-based approach is the VA-file [25] that keeps a quantized version of the data vectors in memory and first performs a fast scan on those before retrieving the remaining full candidate vectors from disk where the full dataset is kept. This technique tries to reduce disk accesses for a single query and does not deal with concurrent scans or CPU caching effects.

An example for an index-based approach to k-NN searching is the R-tree index family introduced by Guttman [10] and the SS-tree introduced by White and Jain [27]. Due to the spherical nature of NN regions, the latter uses nested bounding spheres to describe the data, rather than bounding boxes used by R-trees. Both index structures are intended for large disk-based datasets and would perform sub-optimally if used as pure in-memory structures. In our experiments, we therefore focus on the k-d-tree [8] index which is an in-memory data structure that provides good performance for k-NN searching.

To reduce the amount of data to be kept in either scan-based or index-based data structures, many compression schemes have been proposed. One especially tailored to high-dimensional data is the FastMap algorithm presented by Faloutsos and Lin [7].

Compression is very useful for typical high-dimensional data due to the fact that few dimensions contain most of the useful information. In this paper, we do not consider compression since it introduces additional post-processing. However, we do discuss how striping and principle component analysis [15] can improve the query performance due to these data characteristics.

Automatic tuning for multi-core systems is a fairly new research area. Williams [28], Datta et al. [6], Vuduc et al. [24], and Whaley et al. [26] present various approaches for autotuning scientific applications. Their focus is on automatic generation of optimal code for a specific architecture for regular scientific applications such as FFT, PDE solvers, and sparse matrix kernels. Whaley et al.'s ATLAS project [26] proposes using timing measurements to generate optimal code for any linear algebra kernel on any target architecture. Unlike for the computation kernels discussed in these works, the performance strongly depends on the data characteristics for our multi k-NN search problem. Furthermore, parallel multi k-NN search involves several algorithmic optimizations.

The work on the general area of auto-tuning can be divided into three categories. The first is the compiler-based auto-tuners. These automatically or semi-automatically generate a set of alternative implementations and search this space of implementations using a model, in order to locate a good solution [4,9,30]. The second is the application-level auto-tuners. These systems perform an empirical search across the set of parameter values to locate the best setting in terms of performance [12,18]. Usually these parameters are generated by the application programmer and their ideal values are located by the auto-tuner. Our work is an example of this. The parameterizations we provide for the multi kNN search algorithm do not require recompiling code and are taken as submission time parameters to the algorithm. The third is the run-time auto-tuners. These provide on-the-fly adaptation of application-level and/or code-transformation level parameters [3,23]. The goal is to react to the changing conditions of the system at run-time.

3 Problem Definition

This section discusses the problem addressed in this work in detail. We also describe the various optimization dimensions that will be used to tune our algorithm in the next section.

3.1 Multi Nearest Neighbor Query

Since this paper focuses on accelerating nearest neighbor queries, we first give the traditional definition of the problem:

Definition 1 **(k-nearest neighbor query)** Given a dataset V of N d-dimensional vectors and a query vector q, the *k-nearest neighbor query (NN-query)* returns the set $NN_q(k) \subseteq V$ that contains exactly k vectors and for which the following condition holds:

$$\forall v \in NN_q(k), \forall v' \in V - NN_q(k) : d(v,q) \leq d(v',q).$$

Here, d is a distance function, typically measuring the Euclidian distance between two vectors. We note that $NN_q(k)$ is non-deterministic if there are multiple vectors with equal distance from q among the k-th nearest neighbors. For real datasets from image repositories for example, it would be extremely rare to find identical distances. On the other hand, this property allows us to upper-bound the memory footprint required to keep the search state within each thread.

In many applications, it is possible to batch multiple NN-queries together. For example, when searching for movie clips similar to a given clip, one may obtain feature vectors for each frame or shot of the query clip, and run NN-queries for each in order to find similar frames or shots and ultimately similar clips. Another example are image search engines with thousands of users issuing queries concurrently. From an algorithmic point of view, batching of queries can be advantageous because the concurrently running queries may be able to share cache content, thereby increasing the query throughput, as we will show later.

A formal definition of the multi k-NN search problem is given next:

Definition 2 (Multi k-nearest neighbor query) Given a dataset V of N d-dimensional vectors and Q query vectors $q_1, \ldots, q_Q$, the *Multi k-nearest neighbor query (MNN-query)* returns the set $\{NN_{q_1}(k), \ldots, NN_{q_Q}(k)\}$ of k-nearest neighbors for each q_i.

3.2 Scan and Tree Algorithms

The most straightforward way to compute the k-NNs for a given query vector q is by scanning all vectors, computing the distances, and storing the closest k. Besides simplicity, this has the additional benefit of improving cache locality since CPUs read a whole cache line of vectors into L2 cache with the access of the first vector of that line. Subsequent vector reads can therefore be served directly from cache. We will refer to this algorithm as *scan* in the remainder of the paper.

Another common approach is the use of tree-based index structures to reduce the amount of vectors that have to be inspected. The *k-d-tree* [8] is oftentimes the structure of choice for in-memory high-dimensional NN-search since it provides $O(logN)$ search time for N randomly distributed vectors. A k-d-tree partitions the data space recursively into two half-spaces at each tree node such that both halves contain the same number of vectors. The tree node itself represents the vector that defines the axis-parallel partitioning plane. There are different strategies for picking the dimension along which the partitioning happens. In this paper, we assume the dimension with the largest value range is picked for partitioning. This resulted in the best performance in our experiments.

The algorithm to search for the k-NNs of a query vector q works as shown in Fig. 1. It repeatedly prunes half-spaces that are further away than the current kth NN, since these cannot contain any closer vectors. More details can be found in [8]. We will refer to this algorithm as *tree* in the remainder of the paper.

It should be noted that this paper does not intend to compare a scan and a tree algorithm. We rather pick the two algorithm as examples for "scan-like" and

```
let $c :=$ root node, $R := \emptyset$ before the first call.

function getKNN( $c$, $q$, $k$, $R$ )
(1)  if $c$ is an inner node
(2)        if distance $d(q, H1)$ between $q$ and half-space
           of first child is smaller than $k$th NN in $R$,
(3)              call getKNN( first child, $q$, $k$, $R$ )
(4)        if distance $d(q, H2)$ between $q$ and half-space
           of second child is smaller than $k$th NN in $R$,
(5)              call getKNN( second child, $q$, $k$, $R$ )
(6)  else
(7)        let $v$ be the vector at $c$
(8)        if distance $d(q, v)$ is smaller than $k$th NN in $R$,
(9)              $R := R \cup v$
(10) return
endfunction
```

Fig. 1 The k-NN search algorithm on k-d-tree

"tree-like" algorithms and how our autotuned optimizations help in their respective query throughput. The proposed optimizations would apply to many other search algorithms that share a similar nature to these scan-like and tree-like algorithms.

3.3 Parallel Multiquery Algorithm

In this paper, we explore various ways to parallelize both the scan and tree algorithm to benefit the MNN problem.

Figure 2 illustrates a scenario that mimics the usage of multi-NN algorithms in practice. The scenario envisions a shared address-space parallel machine processing multiple simultaneous NN queries using multiple threads on a shared-memory resident dataset. We use Fig. 2 to explain various optimization parameters that we consider during parallelization of the scan and tree NN algorithms. These parameters affect the way the input queries and the base data are partitioned.

In Fig. 2, the input shown on top consists of a set of k-NN queries. The ideal size, Q, of this set depends on the cache sizes per core, since we need to keep status information per query and if Q is too large, thrashing may occur. If Q is too small, there will be less benefit from sharing read vector data.

The query vectors are further partitioned into P sets. Each of the P sets is individually processed by a group of T threads. Again, the optimal values for P and T will depend on various architectural details such as the number of cores, cache sizes, and so on. All threads in the group working on the same query subset, P, (e.g., **T1a** and **T1b**) will then read distinct blocks of the dataset in a block-cyclic fashion. This approach leads to balancing the workload across the threads in a group. The base data first gets partitioned into blocks of size B each. Whenever a thread finishes, it will pick the next block of data not yet processed by the other threads in the group. No two threads process the same data block. For the tree algorithm, we build one tree index per data block.

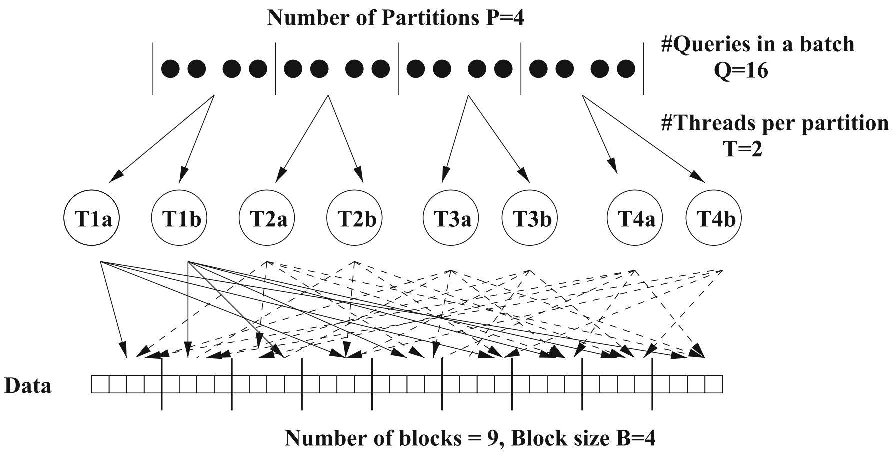


Fig. 2 Optimization parameters for scan and tree nearest neighbor algorithms

Finally, we allow the data to be stored in stripes of a certain size S. With data striping, we first store the first S dimensions of each vector, followed by the next S dimensions, and so on. This is useful in further improving the cache locality. By setting $Q = 1,\ T = 1,\ P = 1,\ B = 1,\ S = d$, the parallel multi k-NN algorithm reduces to the sequential multi k-NN query algorithm.

3.3.1 Discussion of Parameter Trade-offs

More details on each parameter and how they impact the query throughput is given in the next section. However, here we briefly describe the fundamental aspects of the parameters we have chosen for optimization.

(1) *The number of partitions*, i.e., the parameter P, enables us to perform query partitioning. Making a loose analogy with parallel processing, queries can be seen as instructions, whereas vectors could be seen as data. Via query partitioning, we are executing different sets of queries on the same data in parallel, in a multiple-instructions, single data approach.
(2) *The number of threads per partition*, i.e. parameter T, enables us to perform data partitioning. This means we execute the same queries in a partition over different segments of the data in parallel, akin to single instruction, multiple-data, i.e. SIMD,[1] processing. When we have $T > 1$ and $P > 1$, we get the most general case of multiple instructions, multiple data, i.e. MIMD, processing.
(3) *The block size*, i.e. parameter B, enables us to implement a fundamental algorithmic optimization: sharing of partial results between the parallel computations. This parameter creates a tradeoff between data filtering efficiency and synchronization overhead. Too many blocks will result in frequent synchronization across

[1] This is just an analogy, not to be confused with the SIMD instructions, like SSE, which we discuss later.

Table 1 Parameters used in the paper

Input	Parameter description	
N	Number of vectors in the dataset	
d	Number of dimensions per vector	
k	Number of nearest neighbors	
Tuning	Parameter description	Goal
C	Shortcutting (if set to 1)	Reduce the amount of data touched
B	Block size (# of vectors per block)	Improve shortcutting via result sharing
P	Number of query partitions	Enable distribution of queries over cores
T	Number of threads per partition	Enable distribution of data over cores
Q	Query batch size	Improve temporal locality of data access
S	Data striping size in dimensions	Improve spatial locality of data access

threads. Too few blocks will prevent different threads from learning each other's progress, reducing the effectiveness of filtering.

(4) *The stripe size*, i.e. the parameter S, enables us to exploit spatial data locality, a crucial performance factor in today's multi-core processors with complex memory hierarchies.

(5) *The query batch size*, i.e. the parameter Q, provides temporal data locality during query processing, somewhat dually to stripe size. As we discuss later in the paper, the parameters Q, P, and S have interesting interactions, as the diversity in the batched queries impacts their data access patterns and thus the best stripe size S to use.

3.3.2 Summary of Parameters

We close this section with a summary of all parameters used in this paper in Table 1 and a description of the data sets used for experiments presented in the rest of the paper.

It is important to note that the benefit that can be achieved from tuning these parameters is highly dependent on the workload characteristics, such as the query and data distributions. As such, our auto-tuning approach provides performance benefits that go beyond code optimization techniques that rely on modeling the hardware [29].

3.4 Basic Performance with Shortcutting

We now discuss an algorithmic optimization called *shortcutting* that is applied to both scan and tree algorithms in order to reduce the cost of distance computations even for sequential execution. The resulting optimized algorithms serve as baseline for the following optimization steps.

The shortcutting optimization discussed here is important since the evaluation time of quadratic form distance functions grows quadratically with the number of dimensions [22]. Reducing this component of the overall cost can improve high-dimensional k-NN-search performance. Once k-NN candidates are determined, their largest distance from the query vector can be used to shortcut all following distance calculations. If, for example, the Euclidean distance is used, the summing of the squares can be terminated once the partial sum exceeds the largest possible NN-distance found so far. When introducing parallelism further down, this shortcutting will be even more useful since multiple largest k-NN distances from different parts of the dataset are determined concurrently thus leading to a higher likelihood of early termination of the distance calculation.

For validation of all optimizations, we use two real-world datasets in this paper. They consist of high-dimensional feature vectors extracted from various video sources of the TREC 2007 Text Retrieval Conference [19]. Specifically, we use the following feature vector sets:

- **Color Histogram (CH)**: global color of $700,353$ video shots represented as 166-dimensional histograms in HSV color space
- **Histogram of Oriented Gradient** [5] **(HOG)**: features related to edge orientation histograms of $21,532$ video shots with $3,780$ dimensions each.

Both datasets have very different data distributions, with the HOG dataset being much more high-dimensional than the CH dataset. In the remainder of the paper up to Sect. 5, we will use the CH dataset for our experiments exclusively.

We implemented the shortcutting optimization for both the scan and tree algorithms. The resulting performance will be considered as our baseline performance (see Fig. 3). The figure plots the search time for 1,000 queries as a function of the data size. Like the rest of the experiments in Sect. 4, we use the CH dataset for this experiment and a dual quad-core Intel machine with a total of 8 cores.

As expected, the tree algorithm is significantly faster than the scan algorithm. While the scan time grows linearly with the data size, the tree search time is slightly sub-linear since it can prune larger amounts of data that do not require any distance calculations subsequently.

Even though we leave k (the number of NNs requested) constant in this experiment ($k = 10$), we want to point out a subtle dependency between k and the search time. Both the scan and the tree algorithm's search times increase with increasing k. In the case of trees, this is due to the reduced pruning power with more NNs requested. In the case of scans, this is due to the fact that we store the current best k NNs in a heap data structure. The height of this heap grows logarithmically with k. Since the overall search time consists of a constant part (i.e., the reading of the vectors and distance calculations) and a logarithmic part (i.e., the heap operations), we see a logarithmic behavior even for scans. This shows that cost components that were negligible for disk-based search algorithms, can have a significant impact on memory-based algorithms.

4 Discussion of Optimizations

We now discuss the six optimization dimensions that showed most impact on the overall algorithm performance: SIMDization, multi-threading, load balancing, cache sharing, query partitioning, and data striping. We discuss each parameter in detail to

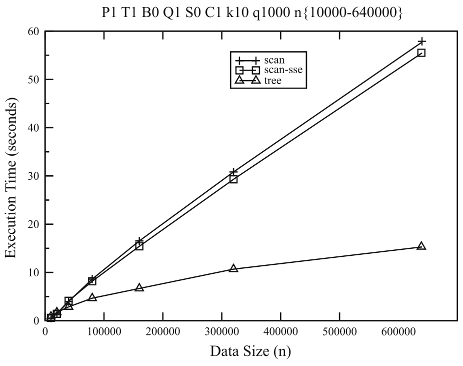


Fig. 3 Basic un-optimized performance with shortcutting. Unlike the scan-version, the tree-version performs sub-linearly. SSE leads to marginal performance gains

demonstrate its impact on performance and to explain the subtle differences they have on scan and tree algorithms.

4.1 Use of SIMD Instructions

At the inner-most core of the k-NN search algorithm lies the vector distance computation, which is amenable to acceleration via SIMD instructions, such as the Intel SSE extensions. By performing the equivalent of 4 scalar floating point operations using one instruction, we could accelerate the vector distance computation by a factor close to 4.[2] However, there are practical obstacles to achieving this ideal speedup. First, the algorithmic shortcutting optimization mentioned in the previous section introduces branches into the inner most loop, as we need to compare the partial distance with the current kth shortest distance. This reduces the pipeline utilization and results in increased number of cycles per instruction (CPI). However, the cost savings obtained from applying algorithmic shortcutting far out-weights the potential loss of performance during the SIMDization stage. Second, again due to shortcutting, we start chasing memory as we are not performing a true linear scan anymore (we do not touch all the data). In the rest of the paper, we will discuss striping in detail, which tries to solve the resulting spatial access locality bottleneck. However, striping introduces additional branching into the code. Striping and shortcutting together make it difficult to use unrolling to get the true benefit out of SIMDization. Finally, the number of hardware threads available on a processor may not necessarily match the number of SIMD units available, making it a potential bottleneck on some architectures.

Figure 3 shows a slight improvement (around 6–7 %) in performance with explicit SIMD instructions embedded in the code using g++ provided intrinsics. Note that the compiler is free to emit SSE code even in the absence of intrinsics in the user code, using auto-vectorization. As a result, the improvement in the performance should be inter-

[2] Horizontal addition of a vector of 4 floats, which is an operation needed for distance computation via SIMD instructions, requires 2 SIMD instructions with SSE3, which is only 2 ($<$ 4) times better than the scalar version of the same computation.

preted as the benefit of manual SIMDization over the auto-vectorization performed by the compiler.

In the experiments reported in this paper, we have used the SSE instructions to accelerate the vector distance computations on the Intel processors. We relied on SSE3 instructions that are 128-bit. Experiments on more modern Intel hardware that supports 256-bit AVX instructions showed mixed results. The performance for the HOG dataset showed 3 % reduction in performance, whereas for the CH dataset it showed a marginal 2 % improvement. We do not use SIMD as a tuning knob.

4.2 Multi-threading

The most obvious tuning step is the introduction of multiple concurrent threads. There are two ways to extend the algorithms to multi-threading: by concurrently performing the data exploration for a single query and by allowing multiple pending queries to run concurrently. In this section we focus on the former, the latter will be discussed in a later section on multi-query optimization.

In order to execute a single query with T concurrent threads, we split the data into T equally sized blocks, i.e. $B = N/T$. For the scan algorithm, we then run T concurrent scans, one on each data block. For the tree algorithm, we first build T index structures and then run the tree traversal algorithm from Fig. 1 concurrently on each index. Since the tree construction is done offline, we will not count this step into the query time. The resulting query response time for different dataset sizes is shown in Fig. 4a (note that the x-axis is in logarithmic scale).

The execution time for scan algorithm with both one thread (cross symbol) and eight threads (box symbol) is linear in the number of data vectors, with the multi-threaded version with $T = 8$ being nearly twice as fast. The reason why we do not see perfect scale-up with T is two-fold: First, the memory subsystem is becoming saturated since each thread has to fetch data from main memory. Second, the scans in different data blocks finish at different times due to different amounts of shortcutting

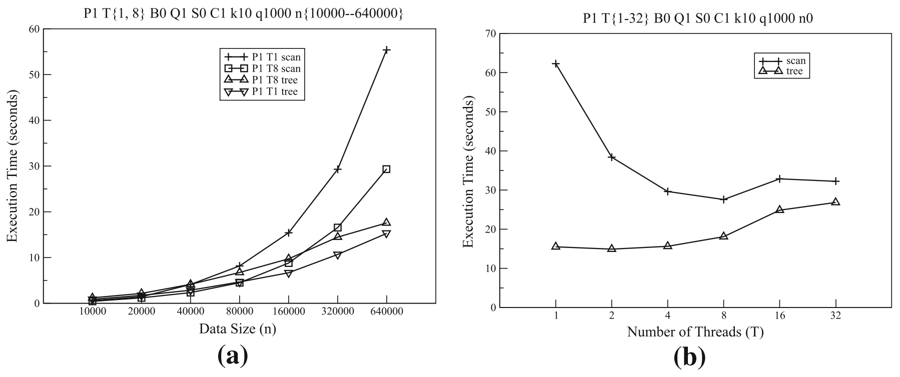


Fig. 4 Multi-threading with Data Partitioning. **a** Data size (n) versus execution time. Multi- threading benefits the Scan algorithm but hurts the tree algorithm. **b** Number of threads (T) versus execution time. Only the scan algorithms scales with the number of threads

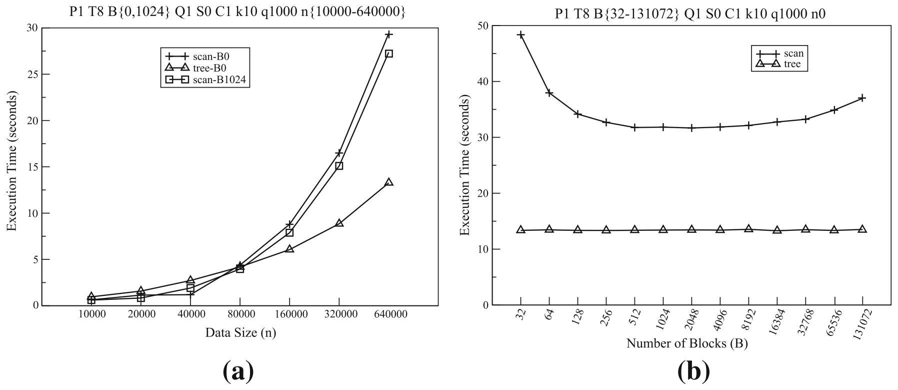


Fig. 5 Effect of sharing across data partitions. **a** Data size (n) versus execution time. Larger number of blocks improve the scan performance. **b** Number of blocks (B) versus execution time. Too many or too few blocks degrade the scan performance

(see previous section). The thread that encounters many shortcuts will finish sooner than a thread with few shortcuts. As a result, the former thread will sit idle until all threads have finished. In summary, the scale-up is sub-optimal because of memory bandwidth limitations and uneven load across the T threads. The next sections will address these problems by introducing multi-query cache sharing and finer-grained data partitioning.

As expected, the tree algorithm is sub-linear in the data size. However, surprisingly, the multi-threaded version performs slightly worse than the sequential version. This is due to the decreased cache locality. Each of the multiple threads running concurrently performs random memory accesses during the tree traversal. There are $O(T \log(\frac{N}{T}))$ such random accesses compared to $O(\log(N))$ random accesses for the sequential tree algorithm. With more read requests, the memory subsystem bottleneck increases, eventually slowing down all threads. In contrast, the scan algorithm has slightly better cache locality since each thread reads vectors sequentially, leading to a memory access only for the first vector of a cache line. The memory bottleneck is, therefore, less pronounced.

Figure 4b shows the query performance for different degrees of parallelism. The scan algorithm performance improves until about 8 threads, after that it degrades. This indicates that with more than 8 threads, the conflicting memory read requests from the threads start to saturate the memory subsystem. For this architecture, using 8 threads is therefore the best choice. The tree graph shows that the tree algorithm saturates the memory already after 1 or 2 threads. The performance degrades continuously with more parallelism. For this architecture, the sequential tree algorithm is apparently best. We point out that for different cache sizes and memory bandwidths, these numbers could be very different. The right tuning of T is therefore very architecture-dependent and not obvious.

4.3 Block-Cyclic Load Balancing and Shortcut Sharing

In order to better balance the load for the different threads, we now introduce another tuning knob, the block size B. Previously, the number of blocks were always equal to T. By turning it into a separate parameter, we can perform block-cyclic load balancing as follows: whenever a thread finishes its assigned data block, it picks the next unassigned block and processes it. This way, threads that finish their block early due to shortcutting, will not sit idle as long as there are more blocks to process.

A second improvement is the sharing of shortcutting information between threads. Once a thread finishes its current data block, it inspects a global variable storing the smallest kth NN-distance found so far. If its local kth NN-distance is smaller, it updates the global variable. If the global variable is smaller, it updates its local kth NN-distance and uses that for shortcutting during processing of the next data block. This way, shortcutting information is shared among all threads while requiring global memory access only during data block switching. Since this improvement can never impact runtime negatively, we use shortcut sharing as the default when block-cyclic load balancing is used.

Figure 5a shows the improved scale-up for 8 threads. For all dataset sizes, the scan algorithm with $B = 1,024$ (denoted by "scan-B1024") is about 10 % faster than the version with $B = N/T$ (denoted by "scan-B0"). This indicates that there is some amount of uneven load between the threads for $B = N/T$, due to the data distribution. The tree graph in Fig. 5a is shown mainly for reference since multi-threading of single queries is not helpful on this architecture.

Figure 5b shows that the choice of B is not obvious: If B is too large (i.e., few large blocks), load balancing will be limited as seen in the 10 % overhead for $B = N/T$. If B is too small (i.e., many small blocks), read sequentiality within a thread is diminished, leading to increased cache misses due to data thrashing. This leads to the u-shaped graph in the figure with a minimum at around $B = 1,024$. Again, this value depends on various architectural parameters (e.g., cache line size and memory bandwidth). The tree graph in this figure is again shown for reference, as no multi-threading is used and therefore $B = 1$ for the tree algorithm.

4.4 Multi-query Cache Sharing

Up to this point, we have considered only single-query optimizations. In reality, it is not uncommon to have a list of multiple pending k-NN queries. Examples are multi-user systems with thousands of query requests per second and problems that generate a list of NN queries (such as video or music searching). In such scenarios we can further improve the query throughput by performing one data scan for multiple pending queries rather than one scan for each query. The advantage of this "query batching" is that each data vector has to be retrieved only once from main memory for each batch, thereby reducing main memory contention.

For the tree algorithm, sharing of read memory vectors among multiple queries is difficult since each query may explore a different subspace of the data vectors. Therefore, during the tree traversal as shown in Fig. 1, each query may descend down

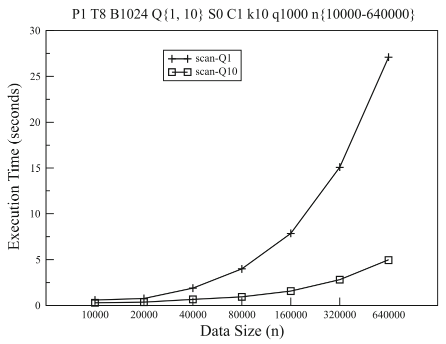


Fig. 6 Effect of query batching. Reusing data across queries improves performance significantly

Table 2 Effect of query batching on low-level metrics. Cache misses (both L2 and L1D) as well as cycles per instruction (CPI) are reduced

Metrics	Q1	Q10
CPI	2.43	1.04
L2 misses	142M	32.6M
L1D misses	152.8M	41.8M
Mispred. rate	1 %	2 %

to a different child at any given node. It is possible to collect pending query requests at the different tree nodes and only traverse further down once enough requests have been collected, similar in spirit to the buffer tree approach [2]. However, if the buffers at each node are small, not much data sharing is possible. If the buffers are large, the query response time will suffer since it takes longer to collect enough pending requests in a node. Another possibility would be to pick a B that is small enough such that up to T trees fit into L2 cache, thereby reducing cache misses. This however leads to a degradation of the logarithmic tree search complexity because now every thread will have to traverse a large number of trees, turning the search essentially into a scan again. For these reasons, we do not consider multi-query cache sharing for the tree algorithm. We will see in the next section that running concurrent queries on the trees of different data partitions is more beneficial for the tree algorithm.

Figure 6 shows the effect of query batching on the scan algorithm for varying dataset sizes. In this experiment, we compare two query batch sizes: $Q = 1$ and $Q = 10$. Across all dataset sizes, the throughput is increased by a factor of approximately 5. As can be seen from the figure, with $Q = 10$, the scan algorithm is 3–5 times faster than the tree algorithm. We will see in the next section that a larger Q is not always better, however.

Table 2 shows low-level metrics for the same experiment presented in Fig. 6. We observe that query batching with $Q = 10$ reduces L2 cache misses by a factor of 4.6× and L1 data cache misses by a factor of 3.66×, compared to no query batching. It also reduces the number of cycles spent per instruction (CPI) by a factor of 2.3×. On the downside, query batching increases branch misprediction rate from 1 % to 2 % due to the inner query loop introduced in query batching. However, the misprediction rate is quite low in general.

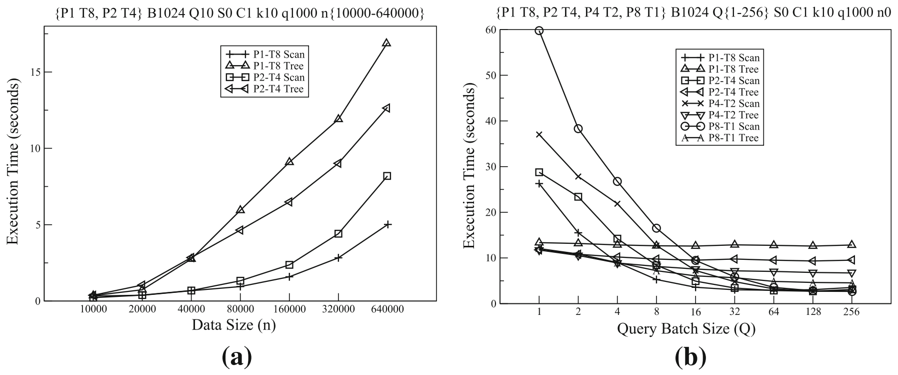


Fig. 7 Effect of query partitioning. **a** Data size (n) versus execution time. More partitions benefit the tree but hurt the scan performance. **b** Query batch size (Q) versus execution time. Effect of partitioning is more pronounced for small query batch sizes for scans and larger batch size for trees

4.5 Query Partitioning

As discussed in the previous section, some search algorithms, such as the tree algorithm, benefit less or not at all from multi-query cache sharing due to variations in the data access pattern between queries. In this section, we introduce a way to parallelize multiple queries by partitioning the set of pending queries into P partitions and processing each subset independently of the other subsets. While query batching is somewhat data-driven, as each thread traverses its portion of the data and calculates distances for all queries in the batch, query partitioning is somewhat query-driven, as each query partition triggers a traversal of the data.

Another way to look at the two approaches is by consulting the example in Fig. 2. For $P = 1$, all threads are working on the same 16 queries. Each data block read by a thread is used to compute 16 distances per data vector in it. On the other hand, for $P = Q$, each query has its own threads. Each data block read by a thread is used to compute only the distances to that thread's query vector. If we use the tree algorithm, the trees of the data blocks are now traversed by each query's thread independently to obtain the k NNs per block.

In order to keep the model general, we also allow for a mix of these two extremes. In the example in Fig. 2, the query batch is partitioned into $P = 4$ partitions with each partition being assigned $T = 2$ threads. In this case, all four partitions are processed in parallel but within each partition, the two threads follow the multi-query cache sharing approach (and the block-cyclic load balancing) for their assigned 4 queries. This way, our query framework can adapt to various modes of data/result sharing and data/query-driven execution.

Some effects of the P and T parameters are shown in Fig 7a. For the scan algorithm, using one partition with 8 threads is noticeably better than using two partitions with 4 threads each, for all dataset sizes. This is due to the fact that with more partitions, each thread processes smaller number of queries and thus more data scans are performed overall. On the other hand, the tree algorithm performs better with two partitions than one because it cannot share data reads for multiple queries in a partition and has to therefore process all queries in a partition sequentially for each data block.[3]

Figure 7b plots the execution time for various query batch sizes. For the scan algorithm, the difference in execution time due to partitioning diminishes with increasing Q, because the number of queries per partition gets too large, making it impossible to fit a data vector and the entire set of queries into cache. The tree algorithm performs best with 8 partitions for all batch sizes.

4.6 Data Striping

Multi-media related feature vectors oftentimes have a high number of dimensions. Due to our shortcutting optimization, typically only a subset of these dimensions needs to be read in order to rule out a vector. As an example, assume we store N vectors $v^1, v^2, \ldots v^N$ with 100 dimensions as

$$v_1^1, \ldots, v_{100}^1, v_1^2, \ldots, v_{100}^N, \quad \ldots$$

Further assume the cache line can hold 50 dimensions and reading of all vectors after the first one can be shortcut after 8 dimensions. Then after accessing v_1^1, all components v_1^1 through v_{50}^1 are brought into the cache. Once v_1^2 is accessed, a new memory access is required and $v_1^2, \ldots, v_{50}^2$ is brought into the cache. When shortcutting at v_8^2 and switching to v_1^3, yet another memory access is needed, leading to N accesses overall.

On the other hand, assume we were to store the vectors in a striped layout with a stripe size of 10 dimensions:

$$v_1^1, \ldots, v_{10}^1, v_1^2, \ldots, v_{10}^2, \quad \ldots, \quad v_{11}^1, \ldots, v_{20}^1, v_{11}^2, \ldots, v_{20}^2, \quad \ldots$$

For this layout, reading the first vector requires 10 memory accesses since it is striped across 10 locations. However, reading the next (shortcut) vectors is much cheaper since after reading v_1^2, the first 10 dimensions of the next 4 vectors have been brought into the cache as well. Therefore, reading the remaining $N-1$ vectors requires only $\lceil \frac{N-1}{5} \rceil$ accesses.

The exact stripe size depends on the cache line size, the cache size, and the data characteristics. If the data was transformed via PCA [15] for example, most of the information needed for NN searching may be concentrated in the first few dimensions and a shorter stripe size would be better. If the dimensions are not sorted by "information content", a larger stripe size would be better.

Figure 8a shows how striping affects the scan algorithm's performance (we do not perform striping for the tree algorithm since the traversed tree nodes are rarely clustered on the same cache lines anyway). For all dataset sizes and query batch sizes Q, choosing a stripe size $S = 16$ cuts the query response time by nearly half compared to no striping (indicated by "S0"). Figure 8b shows how different stripe

[3] If the blocks are sufficiently small, this may still be advantageous if the index trees fit inside L2 or L3 cache.

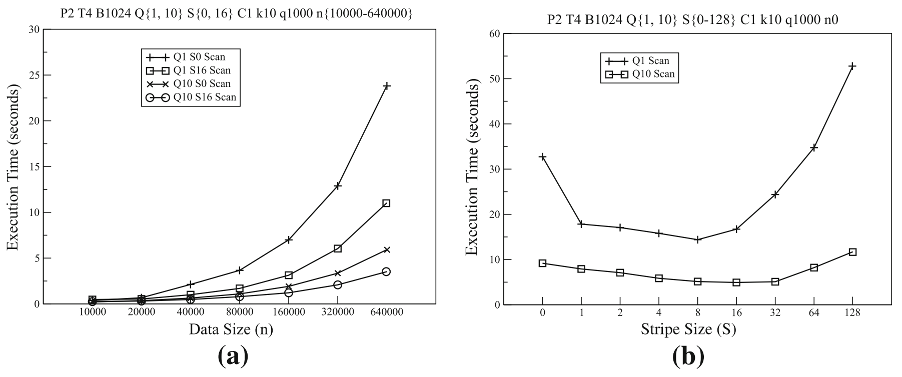


Fig. 8 Effect of data striping with query batching. **a** Data size (n) versus execution time. Data striping benefits the scan algorithm for all data and query batch sizes. **b** Stripe size (S) versus execution time. Data striping is sensitive to query batch size

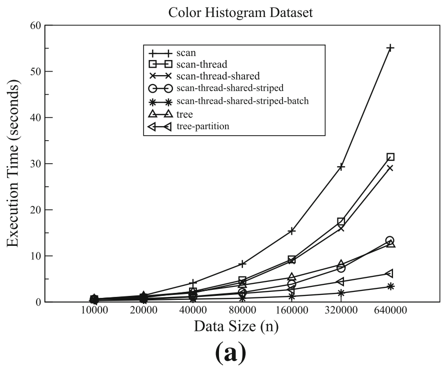
sizes affect the query performance for two different query batch sizes. Both graphs are u-shaped because too small stripe sizes lead to extra memory accesses because the vector components are too scattered across memory, while too large stripe sizes lead to extra memory accesses because cache line space is wasted with unnecessary vector components. For $Q = 1$, the optimal stripe size is at around 8, while for $Q = 10$, the best stripe size is 32. The stripe size is bigger for larger query batch sizes because when many query vectors are considered together, the maximum shortcutting for all query vectors determines the optimal stripe size. This is a good example for the complicated interplay between the various optimization parameters.

4.7 Discussion

In the previous sections, we have discussed the various tuning parameters that we have externalized for both scan and tree algorithm. While there may be other parameters that could be introduced, we believe that the discussed ones are the most beneficial. In this section, we compare the contribution of each tuning parameter on the overall query performance.

Figure 9a presents the query response times for both scan and tree algorithm for varying samples of the color-histogram dataset. For the scan algorithm, we first add multi-threading with $T = 4$ and $P = 2$ which reduces the response time by about 44 %. When adding multi-query cache sharing with $B = 1,024$, the response time drops by another 8 %. Adding striped data layout with $S = 16$ reduces the response time by approximately 60 %. Finally, executing multiple queries in a batch of size $Q = 10$ lowers the query time by additional 75 %. We can see that for the scan algorithm and this dataset, query batching and data striping are the two most beneficial optimization parameters.

The performance of the tree algorithm on the same dataset is shown as graphs with triangle symbols. In order to reduce clutter in the figure, we only show the base line performance (for a single tree index over the data) and the improvement obtained by



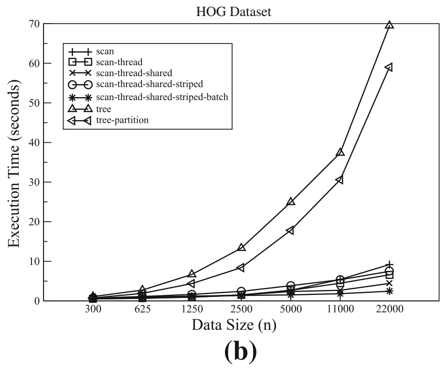


Fig. 9 Contributions of individual optimizations (Data size (n) versus execution time). Effect of individual optimizations varies as per the data set properties. **a** Color Histogram Dataset, **b** HOG Dataset

using $Q = 128$ queries per batch, $B = 1,024$ data block size, $P = 8$ partitions and $T = 1$ thread per partition. For these settings, we see a more than 50 % drop in query execution time across all dataset sizes.

While the basic tree algorithm initially largely outperforms the scan algorithm, with all optimizations applied, the scan algorithm is nearly twice as fast (with 3.33 s) as the tree algorithm (with 6.2 s) for executing 1, 000 NN-queries. Overall, the scan algorithm was improved more than 16 times due to the successive optimizations.

Figure 9b illustrates that applying the same optimization parameter settings for a different dataset does not necessarily lead to the same gains. The figure shows the same plots as the previous figure but for the extremely high-dimensional HOG-dataset. The most obvious difference is that the tree algorithm performs much worse than the scan algorithm, even with all optimizations applied. This is expected because tree-based index structures that recursively partition the data space tend to traverse most of the tree nodes because less pruning is possible (due to the similarity of distances in high-dimensional spaces).

But also within each algorithm class, we observe different gains for the different optimizations. For the scan algorithm, multi-threading with $T = 4$ and $P = 2$ improves query response time only about 25 %. Multi-query cache sharing with $B = 1024$ reduces the time by 33 %. Adding striped data layout with $S = 16$ degrades (!) the performance about 2×. Finally, query batching with $Q = 10$ reduces the time by more than 65 %. For the HOG-dataset, multi-query cache sharing seems to be more beneficial than for the color-histogram dataset. On the other hand, data striping with $S = 16$ and multi-threading with $T = 4$ and $P = 2$ are less beneficial than for the color-histogram data. Similarly, using $Q = 128$, $B = 1024$, $P = 8$, $T = 1$ for the tree algorithm improves the performance by only 17 %.

These two examples show both the potential of the proposed tuning parameters as well as the high dependency of these parameters on the algorithm used, the data distribution, and, as we will see later, the machine configuration. All parameter combinations shown here are merely a small slice in the overall parameter space. Other parameter settings may yield much better (or much worse) performance results. By now, we hope to have convinced the reader that (1) having a small set of tuning parameters is useful for making the tuning task tractable, and (2) it is still very hard for humans to pick the right parameter settings because of the many intricate dependencies of parameters on the problem configuration and on other parameter values.

5 Auto-tuner

In this section we discuss automatic configuration of the optimization parameters used in our scan and tree-based search algorithms. This is achieved via developing a simulated annealing [16] based offline auto-tuner.

5.1 Simulated Annealing

The complexity and diversity of modern processor and memory hardware, involving large number of cores, deep cache hierarchies, instruction pipelines, branch prediction units, vector instructions, etc., coupled with the algorithmic and systems related complexity inherent in multi-threaded software running on these hardware, make optimization approaches based on precise modeling not only extremely difficult but also unportable from one hardware to another.

Most importantly, such a model would need to incorporate the impact of the data set on the performance of the algorithm. As an example, the variance in the number of dimensions used before shortcutting kicks in, across different queries in the same batch, impacts the effectiveness of data stripping. This depends highly on the query distribution. This example also illustrates another difficulty: modeling the subtle interactions between different parameters.

In this work, we use an offline auto-tuner that explores the parameter space by running the search algorithm on a sample data set and measuring the time it takes to execute sample queries on this data. On the up side, this approach does not attempt to model the hardware, the software, the workload, or the interactions between them, thus it is quite portable under changes to any combination of these components. On the down side, it requires that the auto-tuner is run offline, as it takes more time than modeling based approaches. This is however not a major concern for the real word application (video copyright violation detection) that motivated this work.

Specifically, we employ a simulated annealing-based algorithm that is run offline to come up with a set of parameters that result in “good” performance. Even though there is no optimality guarantee, in practice we found that the configurations derived from the simulated annealing runs are markedly superior to what we were able to achieve via manual optimization based on the insight we have gained form the experiments reported in the earlier sections.

One of the most important aspects of the simulated annealing algorithm is the technique used for moving from one solution to the next. This requires us to formulate a set of parameter constraints that define the space of possible moves given the current settings of the parameters.

5.2 Parameter Constraints

At each simulated annealing step, we randomly pick one of the parameters to change and assign a random value to it from the valid range of values it could take, respecting the constraints imposed on it based on the current values of the remaining parameters. Denoting the maximum query batch size by Q_m and the maximum number of total threads available by T_m, we summarize the parameter constraints as follows:

1) The query batch size should be at least equal to the number of partitions, so that each partition has at least one query. On the other hand, there should be an application specific value for the maximum number of queries that can be batched (Q_m). In the worst case, a delay sensitive application with low query rate could define $Q_m = 1$, disabling the query batching, whereas a throughput sensitive application with high query rate could provide a relatively large value for Q_m, benefiting fully from the query batching optimization. In summary, we have:

$$P \le Q \le Q_m.$$

2) The number of partitions times the number of threads per partition should be less than the total number of threads available in the system (T_m). Since the in-memory nearest neighbor search is CPU-bound, one may set T_m equal to the maximum number of hardware threads available. In summary, considering the P and Q relationship from item 1, we have:

$$1 \le P \le min(Q, \lfloor T_m/T \rfloor).$$

3) The number of threads per partition should be less than or equal to the number of blocks used to partition the data (N/B), since each thread needs to process at least one block. In summary, considering the P and T relationship from item 2, we have:

$$1 \le T \le min(\lfloor T_m/P \rfloor, \lfloor N/B \rfloor).$$

4) The block size constraint directly follows from item 3:

$$1 \le B \le \lfloor N/T \rfloor.$$

5) Finally, the stripe size is subject to the constraint $1 \le S \le d_t$, where d_t is the number of dimensions that cover $(1 - \epsilon) * 100\,\%$ of the energy, in case the vector space is transformed via PCA (otherwise $d_t = d$ could be used).

5.3 Experimental Results

In this section, we present our results from running the auto-tuner algorithm on scan-based nearest neighbor search using a Power SMP machine (p595) with 64-processors and 128-cores (2 cores per processor), as well as the dual quad-core Intel machined described earlier. We use two data sets for these experiments, CH and HOG. However, before we move on with these results, we first look at some of the interesting interactions between key parameters using the same setup from the earlier sectionsx. The understanding gained from this analysis would help us interpret the results from the simulated annealing more effectively.

5.3.1 Interplay Between Key Parameters

As Figure 10 shows, there is a subtle interplay between the number of partitions P, the number of threads per partition T, the query batch size Q, and to some degree the stripe size S. This figure shows four graphs for four different P–T combinations. For each combination, we measure the query execution time for varying Q.

All four graphs exhibit a u-shape: smaller query batches provide less data sharing among queries and larger query batches not only hurt query access locality, but also have a negative impact on striping. The latter effect is more pronounced when the number of partitions is small. Too many queries in a given batch (which happens when P is small and Q is large) increases the variance in the number of dimensions explored before the shortcutting takes effect. This reduces the effectiveness of striping. For this very reason, when the query batch size is small, $P1$-$T8$ seems to be the best setting, but as the query batch size increases, more partitions start to provide better results.

5.3.2 Results from Auto-tuner

Figure 11 shows a sampling of the simulated annealing steps, sorted by the amount of time it takes to execute 1, 000 10NN queries (randomly sampled following the data distribution) over the CH data set for the Power machine. The figure also shows individual settings for the 5 key parameters we have. Table 3 shows summary information about the annealing runs, including the maximum and median execution times, as well as the best setting found, for both HOG and CH data sets. While it would be interesting to compare the best result found via annealing with that of an exhaustive search over the configuration space, the latter is prohibitive due to immense size of the configuration space.

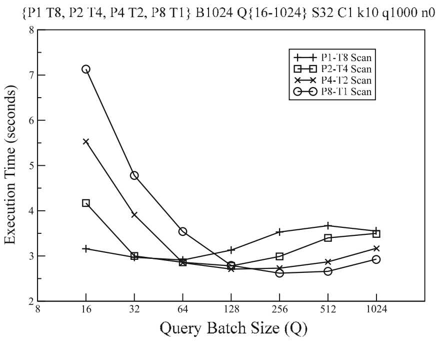


Fig. 10 Interplay between Q, P, S, and T. No individual parameter determines the optimal performance

Table 3 Summary of annealing results for the Power machine (CH dataset)

Metrics	CH	HOG
exec. time, max	48.724 s	132.285 s
exec. time, med	0.855 s	3.184 s
exec. time, anneal	0.133 s	0.416 s
Anneal setting	P32 T4 B128 Q64 S32	P16 T8 B32 Q32 S256

There are a number of interesting observations from the results. First and foremost, the best setting provides around 300× improvement over the worst setting ($P1\ T1\ B1\ Q1\ S1$) and 7.7× improvement over the median setting for the HOG data set. Similarly, the best setting provides around 360× improvement over the worst setting and 6.4× improvement over the median setting for the CH data set. Second, the

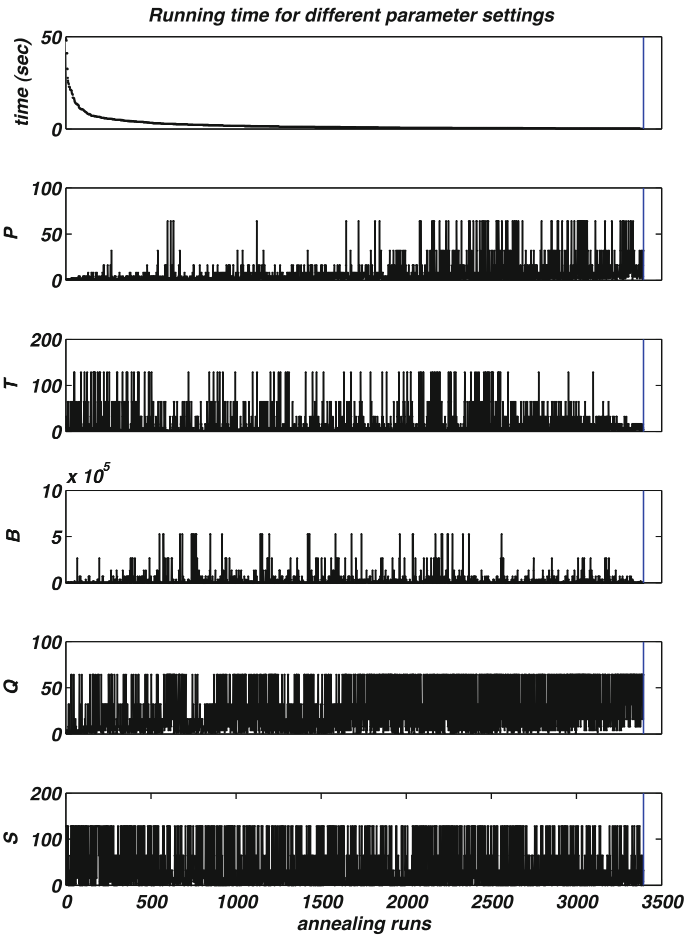


Fig. 11 Explored Configurations for the Power Machine

Table 4 Summary of annealing results for the Intel machine

Metrics	CH	HOG
exec. time, max	53.205 s	278.123 s
exec. time, med	2.541 s	6.783 s
exec. time, anneal	0.421 s	1.112 s
exec. time, best	0.421 s	1.097 s
Anneal setting	P2 T4 B512 Q64 S32	P8 T1 B256 Q32 S128
Best setting	P2 T4 B512 Q64 S32	P4 T2 B256 Q32 S256

best setting for both the HOG and the CH data sets use exactly 128 threads (T_m was set to 4×128). This is justified as there is no reason to use more than the available number of hardware threads when the CPU is fully utilized. Third, the number of queries per partition happen to be the same (2) for both data sets, whereas the stripe size is 8 times larger for the HOG data set. The latter could be explained by the fact that the HOG data set has ≈ 20 times more dimensions compared to CH.

Results for the dual quad-core Intel machine are similar, where improvements compared to the worst and median settings are reaching up to 250 times and 6 times, respectively for the HOG data set. The most striking difference in the dual quad-core results, compared to the Power results, is the best parameter settings. Again this stresses the importance of architecture specific tuning, which we are able to perform in a portable way using our auto-tuner.

Since the Intel machine parameter search space is smaller (less number of cores), we have also run an exhaustive search for this case. The exhaustive search took 3.5 days for the CH dataset and around 6 days for the HOG dataset. Table 4 shows the results. We see that for the CH dataset the simulated annealing approach has found the exact optimal solution. However, for the HOG dataset the solution found by the annealing is different than the optimal one (based on exhaustive search). The good news is that, the performance of the best configuration is only 1.3 % lower than that of the configuration found by the annealing solution.

In summary, the tuning knobs we have chosen impact fundamental aspects of the nearest neighbor search, either through interacting with the algorithmic aspects, architecture specific aspects, or the data dependent aspects of the processing involved. These parameters show complex relationships that are both hardware and workload specific, and as a result quite difficult to model in a parametric way. Our experience with the auto-tuner, as presented in this paper, shows that these optimization parameters could be configured automatically, providing two orders of magnitude improvement over a non-optimized scenario and close to an order of magnitude improvement over an average case configuration.

6 Conclusion

In this paper, we presented a novel way to deal with the complexity of tuning algorithms, specifically search algorithms, for the large variety of today's multi-core architectures. We focused on the parallelized multi k-NN search problem for high-dimensional datasets as found in image and video repositories. For this application, we identified a set of tuning parameters that capture different dimensions of performance optimization and validated them on a real-world dataset. Examples of these parameters are query batch size, data stripe size, number of threads, number of query partitions, and the data block size. This validation conclusively demonstrated the complex interplay among these parameters and the need for an automated tuning mechanism. We proposed a simulated annealing based autotuner that explores the tuning parameter space to identify the optimal set of parameters. Our experimental evaluation of the autotuner on different architectures and datasets showed that the resulting parameter settings provide up to two orders of magnitude improvement over the worst settings.

References

1. Advanced Micro Devices: AMD Athlon 64 X2 Dual-Core Processor Product Data Sheet. http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/33425.pdf (2007)
2. Arge, L.: The buffer tree: a new technique for optimal i/o-algorithms (extended abstract). In: Proceedings of the 4th International Workshop on Algorithms and Data Structures (WADS), pp. 334–345. Springer, London (1995)
3. Cascaval, C., Duesterwald, E., Sweeney, P., Wisniewski, R.W.: Multiple page size modeling and optimization. In: Proceedings of the Parallel Architectures and Compilation, Techniques (PACT). pp. 339–349 (2005)
4. Chen, C., Chame, J., Hall, M.W.: Combining models and guided empirical search to optimize for multiple levels of the memory hierarchy. In: Proceedings of the International Symposium on Code Generation and Optimization (CGO) (2005)
5. Dalal, N., Triggs, B.: Histograms of oriented gradients for human detection. In: Proceedings of the Computer Vision and Pattern Recognition, Workshop (CVPR), pp. 886–893 (2005)
6. Datta, K., Murphy, M., Volkov, V., Williams, S., Carter, J., Oliker, L., Patterson, D., Shalf, J., Yelick, K.: Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing (SC). pp. 1–12 (2008)
7. Faloutsos, C., Lin, K.-I.: Fastmap: a fast algorithm for indexing, data-mining and visualization of traditional and multimedia datasets. SIGMOD Rec. **24**(2), 163–174 (1995)
8. Friedman, J.H., Bentley, J.L., Finkel, R.A.: An algorithm for finding best matches in logarithmic expected time. ACM Trans. Math. Softw. **3**(3), 209–226 (1977)
9. Girbal, S., Vasilache, N., Bastoul, C., Cohen, A., Parello, D., Sigler, M., Temam, O.: Semi-automatic composition of loop transformations for deep parallelism and memory hierarchies. Int. J. Parallel Program. **34**(3), 261–317 (2006)
10. Guttman, A.: R-trees: a dynamic index structure for spatial searching. In: Yormark, B. (ed) Proceedings of the ACM International Conference on Management of Data (SIGMOD), pp. 47–57 (1984)
11. Hill, M.D., Marty, M.R.: Amdahl's law in the multicore era. IEEE Comput. **41**, 33–38 (2008)
12. Chungand, I.-H., Hollingsworth, J.: Using information from prior runs to improve automated tuning systems. In: Proceedings of the ACM/IEEE Conference on Supercomputing (SC) (2004)
13. Intel Corporation. Intel Itanium 2 Processor Reference Manual. http://download.intel.com/design/Itanium2/manuals/25111003.pdf (2004)
14. Intel Corporation: The Intel 64 and IA-32 Architectures Optimization Reference Manual. http://download.intel.com/design/processor/manuals/248966.pdf (2008)
15. Jolliffe, I.T.: Principal Component Analysis. Springer Series in, Statistics (1986)
16. Kirkpatrick, S., Gelatt, C.D., Vecchi, M.P.: Optimization by simulated annealing. Science **220**(4598), 671–680 (1983)
17. Le, H.Q., Starke, W.J., Fields, J.S., O'Connell, F.P., Nguyen, D.Q., Ronchetti, B.J., Sauer, W.M., Schwarz, E.M., Vaden, M.T.: IBM POWER6 microarchitecture. IBM J. Res. Dev. **51**(6), 639–662 (2007)

18. Nelson, Y., Bansal, B., Hall, M., Nakano, A., Lerman, K.: Model-guided performance tuning of parameter values: a case study with molecular dynamics visualization. In: Proceedingsof the International Symposium on Parallel and Distributed Processing, pp. 1–8 (2008)
19. NIST: NIST Special Publication: SP 500–274 (Proceedings of The Sixteenth Text REtrieval Conference (TREC) 2007). http://trec.nist.gov/pubs/trec16/t16_proceedings.html. 2007
20. NIST: The Digital Millennium Copyright Act of 1998. http://www.copyright.gov/legislation/dmca.pdf (2011)
21. Qiao, L., Raman, V., Reiss, F., Haas, P.J., Lohman, G.M.: Main-memory scan sharing for multi-core cpus. Very Larg Data Bases J (VLDBJ) **1**(1), 610–621 (2008)
22. Seidl, T., Kriegel, H.-P.: Optimal multi-step k-nearest neighbor search. In: Proceedings of the ACM International Conference on Management of Data (SIGMOD), pp. 154–165 (1998)
23. Voss, M., Eigenmann, R.: ADAPT: automated de-coupled adaptive program transformation. In: Proceedings of the International Conference on Parallel Processing, pp. 163–170 (2000)
24. Vuduc, R.W.: Automatic performance tuning of sparse matrix kernels. PhD thesis, University of California, Berkeley, Dec 2003
25. Weber, R., Schek, H.-J., Blott, S.: A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In: Proceedings of the International Conference on Very Large Data Bases (VLDB) (1998)
26. Whaley, R.C., Petitet, A., Dongarra, J.J.: Automated empirical optimizations of software and the ATLAS project. Parallel Comput **27**(1–2), 3–35 (2001)
27. White, D.A., Jain, R.: Similarity indexing with the ss-tree. In: Proceedings of the IEEE International Conference on Data, Engineering (ICDE). pp. 516–523 (1996)
28. Williams, S.W.: Auto-tuning performance on multicore computers. Technical report, Electrical Engineering and Computer Sciences, University of California at Berkeley (2008)
29. Yotov, K., Li, X., Ren, G., Cibulskis, M., DeJong, G., Garzarn, M.J., Padua, D.A., Pingali, K., Stodghill, P., Wu, P.: A comparison of empirical and model-driven optimization. In: Proceedings of the ACM Programming Language Design and Implementation Conference (PLDI) (2003)
30. Yotov, K., Li, X., Ren, G., Garzaran, M., Padua, D., Pingali, K., Stodghill, P.: Is search really necessary to generate high-performance BLAS? In: Proceedings of the IEEE: Special Issue on Program Generation, Optimization, and Platform Adaptation, vol. 93(2). pp. 358–386 (2005)