The VLDB Journal (2014) 23:427–448
DOI 10.1007/s00778-013-0329-7

REGULAR PAPER

Approximate similarity search for online multimedia services on distributed CPU–GPU platforms

George Teodoro · Eduardo Valle · Nathan Mariano · Ricardo Torres · Wagner Meira Jr · Joel H. Saltz

Received: 30 August 2012 / Revised: 31 May 2013 / Accepted: 3 July 2013 / Published online: 27 July 2013

Abstract Similarity search in high-dimensional spaces is a pivotal operation for several database applications, including online content-based multimedia services. With the increasing popularity of multimedia applications, these services are facing new challenges regarding (1) the very large and growing volumes of data to be indexed/searched and (2) the necessity of reducing the response times as observed by end-users. In addition, the nature of the interactions between users and online services creates fluctuating query request rates throughout execution, which requires a similarity search engine to adapt to better use the computation platform and minimize response times. In this work, we address these challenges with Hypercurves, a flexible framework for answering approximate k-nearest neighbor (kNN) queries for very large multimedia databases. Hypercurves executes in hybrid CPU–GPU environments and is able to attain massive query-processing rates through the cooperative use of these devices. Hypercurves also changes its CPU–GPU task partitioning dynamically according to the observed load, aiming for optimal response times. In our empirical evaluation, dynamic task partitioning reduced query response times by approximately 50 % compared to the best static task partition. Due to a probabilistic proof of equivalence to the sequential kNN algorithm, the CPU–GPU execution of Hypercurves in distributed (multi-node) environments can be aggressively optimized, attaining *superlinear* scalability while still guaranteeing, with high probability, results at least as good as those from the sequential algorithm.

Keywords Descriptor indexing · Multimedia databases · Information retrieval · Hypercurves · Filter-stream · GPGPU

Electronic supplementary material The online version of this article (doi:10.1007/s00778-013-0329-7) contains supplementary material, which is available to authorized users.

G. Teodoro (✉)· J. H. Saltz
Center for Comprehensive Informatics, Emory University, Atlanta, GA, USA
e-mail: glmteodoro@gmail.com; gteodor@emory.edu

J. H. Saltz
e-mail: jhsaltz@emory.edu

E. Valle
Recod Lab/DCA/FEEC, State University of Campinas, Campinas, SP, Brazil
e-mail: dovalle@dca.fee.unicamp.br

N. Mariano · W. Meira Jr
Department of Computer Science, Universidade Federal de Minas Gerais, Belo Horizonte, MG, Brazil
e-mail: nathanr@dcc.ufmg.br
e-mail: meira@dcc.ufmg.br

R. Torres
Recod Lab/DSI/IC, State University of Campinas, Campinas, SP, Brazil
e-mail: rtorres@ic.unicamp.br

1 Introduction

A Similarity search is the process of finding the most similar objects, in a reference database, to a given query object. For multimedia retrieval, both the query and the database objects are represented by feature vectors in a high-dimensional space, and similarity search can be abstracted as the process of finding, in that space of features, the closest vectors to the query vector, according to some notion of distance (e.g., Euclidean distance). Similarity search is a fundamental operation for several applications in content-based multimedia retrieval (CBMR), including search engines for web images [52], image recognition on mobile devices [30], real-time song identification [17], photo tagging in social networks [61], recognition of copyrighted material [71] and many others.

The feature vectors give multimedia documents a meaningful geometry in terms of perceptual characteristics (color, texture, motion, etc.), helping to bridge the so-called "semantic gap": the disparity between the amorphous low-level multimedia coding (e.g., image pixels or audio samples) and the complex high-level tasks (e.g., classification or document retrieval) performed by CBMR engines. Searching for similar documents becomes equivalent to finding similar feature vectors. Query processing may consist of several phases and may be complex, but searching by similarity will often remain the first step and, because of the (in-)famous "curse of dimensionality", one of the most expensive.

The success of current CBMR services depends on their capacity to handle very large (and increasing) volumes of data and to keep low response times as observed by the end-user. The size of a database representing even a very small fraction of the images available on the Web exceeds the storage capacity of most commodity desktop machines. However, most indexing methods for similarity searching are designed to execute sequentially and are unable to take advantage of the aggregate power of distributed environments. The collection size and the consequent necessity of using distributed environments are not the only challenges in keeping response times low. The nature of user interaction with CBMR services creates large fluctuations in request rates throughout service execution, requiring these services to adapt dynamically at runtime to better use the computational resources available.

Motivated by these challenges, we propose Hypercurves, a distributed memory parallelization of the multi-dimensional sequential index Multicurves [72,73]. The parallelization is based on the filter-stream programming paradigm [10] implemented in Anthill [64,65]. Hypercurves' parallelization strategy for distributed memory machines splits the database, without need of replication, among computing nodes. The partitions are then accessed independently before a final reduction phase is employed to merge the partial results, in order to provide the final answer. The efficiency of the Hypercurves parallelization strategy is based upon the "sorted list-like" behavior of Multicurves' indexes, which allows a probabilistic equivalence (Sect. 4.2) between the distributed and the sequential version of the indexes. Thus, Hypercurves can reduce the number of elements retrieved from each partition as the number of nodes used increase, achieving superlinear scalability while still guaranteeing, with high probability, results at least as good as those obtained by the sequential algorithm.

Hypercurves executes on hybrid machines equipped with both CPUs and graphics processing units (GPUs). GPUs are massively parallel and power-efficient processors, which are widely used in high-performance computing. These devices are throughput-oriented processors equipped with a large number of lower-frequency computing cores, which are designed to execute a large number of tasks in parallel at the cost of longer computation times for individual tasks. Therefore, the use of GPUs by Hypercurves is challenging because the main goal of an online application is to minimize the response time for each of the user's individual queries (tasks). However, as the query rate increases and exceeds the CPUs' computing capabilities, the time a query spends waiting to be processed (queue time) quickly dominates its overall execution time, and under this circumstance, the use of GPUs becomes favorable as it improves system throughput and reduces or eliminates queueing times. To keep response times optimal in both situations of high and low request loads, we propose a dynamic CPU–GPU task scheduling strategy that takes into account the processors' characteristics and the instantaneous system load to continually retune the task partition throughout execution.

This paper addresses these challenges and significantly improves upon the CPU-only preliminary version of Hypercurves [69]. The main contributions of this work include the following:

- A GPU-enabled version of Hypercurves that uses multiple accelerators concurrently and is able to answer a massive number of requests in very large databases.
- A new dynamic scheduler for hybrid environments that adapts the CPU–GPU task partition under fluctuating request rates to optimize request response times. Compared to the best static task partition, the dynamic scheduler has obtained query response times that are up to 2.77× smaller.
- A set of optimizations for hybrid CPU–GPU environments that include cooperative execution in CPUs and GPUs (Sect. 5);
- Performance improvements (superlinear scale-up) that rest upon the ability of Hypercurves to partition the database without overlap, such that each data partition can be accessed independently, and we can safely reduce the number of elements (objects) to be accessed from each partition. We demonstrate the feasibility of this partitioning while maintaining results from the parallel Hypercurves at least as good as those from the sequential Multicurves with high probability (Sect. 4.2);

The remainder of the text is organized as follows. The next section discusses CBMR services and related GPU-enabled similarity search systems. Section 3 presents the sequential index Multicurves and the parallel framework Anthill that were used to build the parallel index Hypercurves. Hypercurves parallelization is detailed in Sect. 4 along with an analytical proof of the probabilistic equivalence between Multicurves and Hypercurves. In Sect. 5, we introduce Hypercurves on heterogeneous CPU–GPU environments. Section 6 discusses scheduling under fluctuating request rates. Section 7 presents an experimental evaluation, and we conclude in Sect. 8.

2 Related work

In textual data, low-level representations are strongly coupled to semantic meaning because the correlation between textual words and high-level concepts is strong. In multimedia, however, low-level coding (pixels, samples and frames) is distant from the high-level semantic concepts needed to answer user queries, which creates the much discussed "semantic gap". This problem is addressed by embedding multimedia documents in a space using *descriptors* and by using distances between descriptors to represent perceptual dissimilarities between documents.

Multimedia descriptors are diverse and include a wide choice of representations for perceptual properties. These properties include shape, color and texture for visual documents; tone, pitch and timbre for audio documents; flow and rhythm of movement for moving pictures; and many others. The descriptor gives these perceptual properties a precise representation by encoding them into a *feature vector*. The feature vector space induces a geometric organization where perceptually similar documents are given vectors close to each other, while perceptually dissimilar documents are given vectors that are further apart. Distances are usually established using simple metrics such as Euclidean and Manhattan distances, but more complex metrics may be chosen [52].

For images and videos, the last decade witnessed the ascent of descriptors inspired by Computer Vision, especially the *local descriptors* [44,70], with the remarkable success of the SIFT [39] descriptors. Local descriptors represent the properties of small areas of images or videos, as opposed to the traditional *global descriptors* that attempt to represent an entire document in a single feature vector. The success of local descriptors was followed by the use of compact representations based on their quantization using codebooks in the "bag-of-visual-words" model, which became one of the primary tools in the literature [14].

Regardless of the specific choice of descriptor, the retrieval of similar feature vectors is a core operation in almost all systems. Similar feature retrieval may be used directly, as in early content-based image-retrieval systems [60], or indirectly, as in cases where similarity search is part of a kNN classifier that retrieves a preliminary set of candidates to be refined by a more compute-intensive classifier. In one form or another, similar feature retrieval is a critical component of systems that handle real-world, large-scale databases [38].

In the framework of Böhm et al. [12], a multimedia description algorithm corresponds to an extractor of feature vectors, which is represented as a function F that maps a space of multimedia objects Obj into d-dimensional real vectors: $F : \mathrm{Obj} \rightarrow \mathbb{R}^d$. The dissimilarity between two objects obj_i and $\mathrm{obj}_j \in \mathrm{Obj}$ is established by their feature vectors distance (e.g., Euclidean distance): $\Delta(\mathrm{obj}_i, \mathrm{obj}_j) = \|F(\mathrm{obj}_i), F(\mathrm{obj}_j)\|$.

Objects' dissimilarity is used to establish various types of similarity queries [12]: range, nearest neighbor, k nearest neighbors (kNN), inverse k-nearest neighbors, etc. This work focuses on kNN queries. Given a database $B \subseteq \mathrm{Obj}$ and a query $q \in \mathrm{Obj}$, the k-nearest neighbors to q in B are the indexed set of the k objects in B closest to q:

$$\mathrm{kNN}(B, q, k) = \Big\{ b_1, \ldots, b_k \in B \;\Big|\; \forall i \le k \quad \forall b \in B \backslash \{b_1 \ldots, b_i\}, \Delta(q, b_i) \le \Delta(q, b) \Big\} \tag{1}$$

For large-scale multimedia services, however, the exact kNN is prohibitively time-consuming and its definition must be relaxed to account for approximate similarity search as discussed in next section.

2.1 Multimedia similarity search

Efficient query processing for multi-dimensional data has been pursued for at least four decades with myriad applications. These applications include satisfying multi-criteria searches, and searches with spatial and spatio-temporal constraints [21,23,51,77].

An exhaustive review would be overwhelming and beyond the scope of this article. One of the most comprehensive references to the subject is the textbook of Samet [55]. The book chapters of Castelli [16] and Faloutsos [25] provide a less daunting introduction, which is focused on CBMR for images. Another comprehensive reference is the survey of Böhm et al. [12], which provides an excellent introduction with a formalization of similarity queries, the principles involved in their indexing process and their cost models. A book edited by Shakhnarovich et al. [57] focuses on computer vision and machine-learning applications. In the topic of metric methods, which are able to process non-vector features as long as they are embedded in a metric space, the essential reference is the textbook of Zezula et al. [79]. The survey of Chávez et al. [18] is also an excellent introduction to similarity search in metric spaces.

Despite the diversity of methods available, those of practical interest in the context of large-scale content-based multimedia services are surprisingly few. Because of the "curse of dimensionality" (explained below), methods that insist on exact solutions are only adequate for low-dimensional spaces, but multimedia feature vectors often have hundreds of dimensions. Most methods assume that they may use shared main memory, which cannot be the case in very large databases. Other methods, such as those based on clustering, have prohibitively high index building times, with forced rebuilding if the index changes too drastically, and are, therefore, adequate only for static databases of moderate size.

Since performing exact kNN search on high-dimensional datasets of multimedia descriptors is not viable, several scalable methods to approximate the search have been proposed. The approximation may imply different compromises: sometimes the compromise is finding elements that are not too far from the exact answers, i.e., guaranteeing that the distance to the elements returned will be up to a factor from the distance to the correct elements; sometimes the compromise is a bounded probability of missing the correct elements. The guarantee may also be more complex, for example: it may be a bounded probability of finding the correct answer, provided it is sufficiently closer to the query than the closest incorrect answer [32].

Approximation on a bounded factor is formalized as follows: given a database $B \subseteq \text{Obj}$ and a query $q \in \text{Obj}$, the $(1+\epsilon)$ k-nearest neighbors to q in B are an indexed set of objects in B whose distance to the true kNN is at most a $(1+\epsilon)$ factor higher:

$$\begin{aligned}\epsilon\text{-kNN}\ (B, q, k) = \Big\{ b_1, \ldots, b_k \in B \ \Big| \\ \forall i \le k, \ \Big[\forall b \in B \backslash \{b_1, \ldots, b_i\}, \\ \Delta(q, b_i) \le (1+\epsilon)\, \Delta(q, b) \Big] \Big\} \end{aligned} \quad (2)$$

Even if perfect accuracy can be sacrificed, the efficiency requirements of kNN search remain very challenging: the method should perform well for high-dimensional data (up to hundreds of dimensions) in very large databases (at least millions of records); and it should be dynamic, i.e., it should allow data insertion and deletion without performance degradation.

A common strategy found in methods useful for large-scale multimedia is to project the data onto different subspaces and to create *subindexes* for each of the subspaces. These subindexes can typically be independently queried, and their results are aggregated to find the final answer.

Locality-sensitive hashing (LSH) uses locality-aware hashing functions, organized in several "hash tables", to index data [32]. LSH is supported by a theoretical background, which allows the prediction of the approximation bounds for the index for a given set of parameters. The well-succeeded family of pStable locality-sensitive hash functions [19] allowed LSH to directly index Euclidean spaces, and its geometric foundation is also strongly based on the idea of projection onto random straight lines. LSH works very well when the goal is to minimize the number of distances to be evaluated, and can rely upon uniform random data access cost. However, in situations where the cost of accessing the data dominates the cost of computation, the efficiency of LSH is compromised. The parameterization of LSH tends to favor the use of a large number of hash functions (and thus subindexes), which also poses a challenge for scalability.

MEDRANK is also based on the use of multiple sub indexes. It projects data onto several random straight lines. The data are indexed by their one-dimensional positions in the line [22]. This method has an interesting theoretical analysis that establishes, under certain hypotheses, the bounds on approximation error. The techniques used by the algorithm succeeded in moderately dimensional multi-criteria databases, in which it is still feasible to search for exact solutions. In those cases, many of the choices are provably optimal [23]. For high-dimensional multimedia information, however, MEDRANK fails, primarily due to the lack of correlation between distances in straight lines and distances in high-dimensional space [71].

Multicurves [72,73] uses fractal space-filing curves to map a multi-dimensional vector onto an one-dimensional key representing a position in the curve (which is referred to here as *extended-key*). The position is then used to perform a search by similarity. One important characteristic of Multicurves is its use of multiple curves where each curve maps a projection of vectors onto a moderate-dimensional subspace. That dimensionality reduction results in an efficient implementation, reducing the effects of the "curse of dimensionality". Because of the exponential nature of the "curse", it is more efficient to process several low- or moderate-dimensional indexes than a single high-dimensional one. This result is explained by the fact that we not only gain the intrinsic advantages of using multiple curves (i.e., elements that are incorrectly separated in one curve may be close in another), but we also mitigate the boundary effects in each curve. Because Multicurves is the foundation for the distributed algorithm proposed in this work, it is described in detail in Sect. 3.1.

2.2 Scheduling and similarity search in systems with accelerators

The use of hybrid accelerated computing systems is quickly growing in the field of high-performance computing [74]. However, maximizing the use of these systems is a complex task, which requires the use of elaborated software instruments to handle the peculiar aspects of each type of processor available in a machine. The benefits and challenges brought by accelerators motivated several projects in two fields that are particularly related to our work (1) Multi-/Many-core *scheduling techniques* and (2) GPU *accelerated similarity search*.

2.2.1 Scheduling in multi-/many-core systems

Mars [29] and Merge [37] have evaluated the cooperative use of CPUs and GPUs to increase the speed of MapReduce computations. Mars has performed an initial evaluation on the benefits of partitioning Map and Reduce tasks statically between CPUs and GPUs. Merge has extended that approach with the dynamic distribution of work at runtime. The Qilin [40] system has further included an automated methodology to map computation tasks to CPUs and GPUs. The Qilin strategy was based on an early profiling phase, which is used for building a performance model that estimates the best division of work. These solutions (Mars, Merge, and Qilin), however, are unable to take advantage of distributed systems.

Other projects have focused on execution in distributed CPU–GPU equipped platforms [13,31,53,65,67]. Ravi et al. [31,53] have proposed techniques for the automatic translation of generalized reductions to CPU–GPU environments via compiling techniques, which are coupled with runtime support that coordinates execution. DAGuE [13] and StarPU [5] are frameworks that focus on the execution of regular linear algebra applications on CPU–GPU machines. These systems represent the application as a directed acyclic graph (DAG) of operations and ensure that dependencies are respected. They offer different scheduling policies, including those that prioritize the computation of critical paths in the application dependency graph in order to maximize parallelism. The work of Disher et al. [20] also accelerates LU factorization, from linear algebra, in hybrid machines equipped with multi-/many-core processors. Differing from DaGuE and StarPU, Disher et al. have investigated the use of the new *intel many integrated core* (MIC) architecture, and have implemented dynamic task partitioning to improve load balancing among MIC and multi-core CPUs.

These previous studies have focused on minimizing the execution time of an entire application run by partitioning the application's internal subtasks among available devices. In Hypercurves, however, we are interested in minimizing the execution time of each application's internal subtask (query), as the execution time of each query directly impacts the waiting times observed by the end-users. In addition, Hypercurves is an interactive online service and, consequently, it is affected by fluctuating workloads. The different levels of load observed during an execution require the CPU–GPU task partition to be retuned dynamically to provide better use of the hardware and minimize query response times. To the best of our knowledge, Hypercurves is the first system to propose scheduling techniques for minimizing query execution times for online services in hybrid CPU–GPU equipped machines. The proposed techniques are applicable to other online applications.

2.2.2 GPU accelerated similarity search

In the last few years, a number of studies have used many-core processors to accelerate the typically computationally-intensive process of similarity searching in high-dimensional spaces. Most of these studies have focused on accelerating the exact (brute force) kNN search [4,27,33,35,59], while a few have used GPUs to speed up efficient approximate similarity search algorithms [34,49,50].

The existing GPU-based exact (brute force) kNN parallelizations follow a common two-stage strategy. The first phase consists of computing similarity/distance between the query and the reference dataset. The second stage, which is the main difference between the methods, selects the k-nearest neighbors using the similarity metric previously calculated. The work of Garcia et al. [27] was seminal in GPU-based exact kNN; Garcia's work evaluated the use of CUDA and CUDA basic linear algebra subroutines (CUBLAS) [47] to perform distance computations in the first phase, showing favorable results in the CUBLAS version. In the second stage, a parallel version of insertion sort that was modified to select only the k nearest elements was used. Kuang et al. [35] have further identified that the insertion sort performance degrades as k increases, and thus, they improved that phase with the use of a GPU accelerated radix-sort [56]. Additionally, Sismanis et al. [59] have presented a study on the performance of various sorting algorithms for selecting kNN elements.

Other projects have introduced approaches to the multi-GPU parallelization of kNN in shared [33] and distributed [62] memory machines. For example, in the work of Sun et al. [62], a dataset is partitioned into disjoint data chunks that are assigned to multiple GPU-enable machines. These machines compute kNN in the chunks to which they are assigned, and a reduction is employed among the nodes to merge the results and generate the global kNN answer.

A GPU-enabled variant of the locality-sensitive hashing (LSH) nearest neighbors search was proposed by Pan et al. [34,49,50]. This implementation includes parallelization for both the index building and the query-processing phases. The authors have reported performance improvements of approximately 40× on top of the single-core CPU version. Unfortunately, the datasets their parallelization is able to handle are limited by the size of the GPU on-chip memory, which is a significant barrier for use in large multimedia datasets. In Hypercurves, we achieve comparable increases in speed, but we are additionally capable of (1) scaling to multiple GPUs in a node and (2) using multiple machines to increase the speed of a search, attaining superlinear scalability in multinode executions.

Finally, the recent work of Kruliš et al. [34] has employed CPUs and GPUs cooperatively to accelerate the sophisticated and computationally-intensive signature quadratic form distance (SQFD) [6–8] similarity search method. The performance attained by their parallelization, which carefully considers the balance of the workload between CPUs and GPUs, outperforms the increases in speed of a 48-core server. Their work, however, is limited to execution on a single node and does not investigate scheduling under variable workloads.

3 Background

This section presents the sequential index Multicurves, parallelized in this work, and the Anthill framework used in the parallelization.

3.1 The sequential index Multicurves

Multicurves [72,73] is an index for accelerating kNN queries based on space-filling curves. One of the main challenges in the use of space-filling curves in similarity search regards the boundary effects that are a result of the existence of regions that violate the curves' neighborhood-relation preserving property (i.e., the property that points that are close in the space should be mapped to points that are close in the curve). To overcome this problem, Multicurves uses multiple curves, expecting that in at least one of the curves the neighborhood-relations will be preserved. Each of the curves, however, is responsible for a subset of the dimensions, rather than all of the dimensions as is seen in other methods [3,24,26,36,41,58]. Because of the exponential nature of the "curse", it is more effective to process several low-dimensional queries than a single high-dimensional query.

Algorithm 1 Multicurves index construction

input:

B: the database elements to be indexed
curves: number of curves or subindexes
dims$[i]$: dimensionality of the i^{th} curve
$a[i, j]$: the coordinate in the feature vector to be assigned to the j^{th} dimension of subindex i
$C^{-1}()$: the space-filling curve projection
$F()$: Computes/returns a feature vector

output: an array of *curves* sorted lists, which composes the index (each element is a subindex)

```
1: subindexes[] ← new array with curves empty sorted lists;
2: for all $b \in B$ do
3:    $v \leftarrow F(b)$;
4:    for $c \leftarrow 1$ to curves do
5:       proj[] ← new array with dims[c] empty elements;
6:       for $d \leftarrow 1$ to dims[c] do
7:          $proj[d] \leftarrow v[a[c, d]]$;
8:       $key \leftarrow C^{-1}(proj)$;
9:       Insert $< key, b >$ into subindexes[c];
10: return subindexes[];
```

Multicurves index construction is presented in Algorithm 1. The feature vector for each database element is obtained in Line 3 (it is usually precomputed). Each curve projects the data onto a corresponding subspace and then computes the key (short name for extended-key), which is the one-dimensional position in the curve. The tuples $\langle$key, $b\rangle$ are stored in lists sorted by key, with one list per curve. Geometrically, the algorithm is projecting the feature vector in a subspace and mapping it using a curve that fills the subspace. For simplicity, the algorithm is presented as a "batch" operation, but the index may be built incrementally.

Algorithm 2 Multicurves search phase

input : (in addition to *curves*, *dims*[], a[], $C^{-1}()$ and $F()$ explained in Algorithm 1)

k: the number of desired nearest neighbors
depth: the probe-depth, i.e., the number of elements to examine per subindex
q: the data element to be queried
subindexes[]: array of sorted lists composing the index, generated in Algorithm 1

output : a list with the k approximate nearest neighbors

```
1: $v \leftarrow F(q)$;
2: candidates ← ∅;
3: for $c \leftarrow 1$ to curves do
4:    proj[] ← new array with dims[c] empty elements;
5:    for $d \leftarrow 1$ to dims[c] do
6:       $proj[d] \leftarrow v[a[c, d]]$;
7:    $key \leftarrow C^{-1}(proj)$;
8:    candidates ← candidates ∪ {depth closest vectors to key in subindex[c]};
9: knn ← {k closest vectors to q in candidates};
10: return knn ;
```

The search phase (Algorithm 2) is conceptually similar: the query is decomposed into projections, using the same subspaces as in the index construction, and each projection has its key computed. The algorithm then obtains a number of candidate elements (*probe-depth*) from each subindex. The elements returned are those nearest to the key in each curve. Finally, the distances between the candidate elements and the query are calculated, and the k nearest elements are selected as the results. The index creation and search processes are illustrated in Fig. 1.

For simplicity, in the scheme above, both the query and the database elements are associated with a single feature vector by the description function $F()$. The extension for using multiple descriptors per multimedia object, as used by local descriptors, is trivial. In the latter case, each descriptor vector is indexed and queried independently. For example, if a query object generates 10 feature vectors, the kNN search will produce 10 sets of k nearest neighbors, one for each query vector. The task of taking a final decision (classification result, retrieval ranking) from those multiple answers is very application-dependent and is beyond the scope of this article, which is concerned with the basic infra-structure of the search engine.

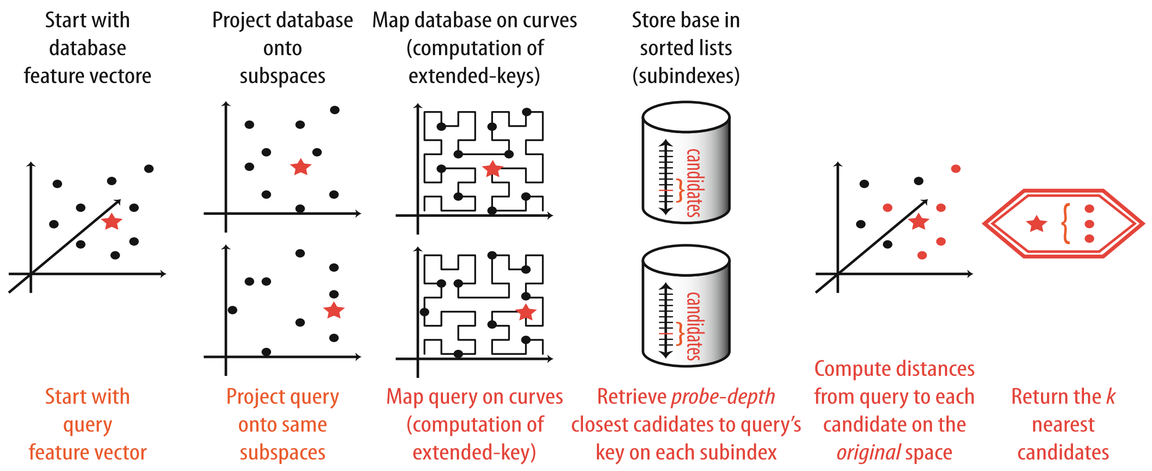

Start with database feature vectore
Project database onto subspaces
Map database on curves (computation of extended-keys)
Store base in sorted lists (subindexes)
candidates
Start with query feature vector
Project query onto same subspaces
Map query on curves (computation of extended-key)
Retrieve probe-depth closest cadidates to query's key on each subindex
Compute distances from query to each candidate on the original space
Return the k nearest candidates

Fig. 1 Multicurves execution workflow. *In black* The index is created by projecting the database feature vectors (*small dots*) onto different subspaces and mapping each projection onto a space-filling curve to obtain the extended-keys. Each subspace induces an independent subindex where the vectors are stored, sorted by extended-keys. *In red* Searching is performed by projecting the query feature vector (*red star*) onto the same subspaces and computing the extended-keys of the projections. A number (probe-depth) of candidates closest to the query's extended-key are retrieved from each subindex. Finally, the true distance of the candidates to the query is evaluated and the k closest candidates are returned

3.2 The parallel environment anthill

Anthill [64,65,68] is a runtime system based on the filter–stream programming model [10]. As such, applications developed in this paradigm are decomposed into processing stages, called *filters*, which communicate with each other using unidirectional *streams*. Additionally, at runtime, Anthill is able to create multiple copies (instances) of each of the application filters on the nodes of a distributed memory machine. The streams are then responsible for providing a set of high-level communication policies (e.g., round-robin, broadcast, labeled stream, etc.), which abstract message routing complexities among filter instances (Fig. 2).

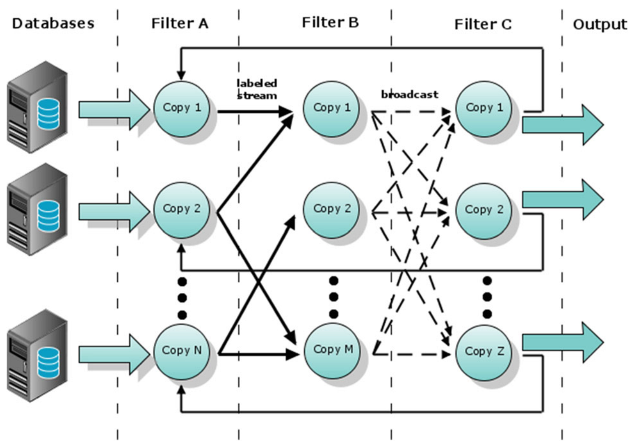


Fig. 2 The deployment of an Anthill application. Filters (*columns*) cooperate to process the data. The filters copies (*circles*) are created by Anthill's runtime environment, and communication is mediated by unidirectional streams (*arrows*)

The development of applications in the filter-stream programming model naturally leads to pipeline and data parallelism. Pipeline parallelism is attained with the decomposition of the application into a set of filters that are executed concurrently in a pipeline fashion. Data parallelism is achieved with the creation of multiple copies of each filter, which may process different partitions of the input data.

The filter programming abstraction in Anthill is multi-threaded and event-oriented, deriving from the message-oriented programming model [11,48,75]. Anthill runtime is responsible for controlling the non-blocking I/O flow through streams, and messages arriving at a filter instance create associated computing events. The developer then writes the application as a set of event-processing handlers that perform the application-specific transformations in events and may, consequently, output messages to the next stage in the pipeline. Each filter copy is implemented as a multithreaded program that concurrently computes events. This allows a single copy of each filter to fully utilize a multi-core machine, reducing the number of filters created in a distributed environment. This feature is especially important for the construction of Hypercurves, because it allows a dataset to be divided into smaller number of partitions (one per machine instead of one per CPU core or GPU), which reduces the total number of elements to be evaluated in a distributed execution.

The multithreaded event-oriented filter interface also enables events to be computed by heterogeneous devices (e.g., CPUs and GPUs). This is accomplished by allowing the programmer to provide, for the same event type, handler functions targeting different processors, which are then invoked to use the appropriate processor. Figure 3 illustrates the architecture of a typical filter. The Filter receives data

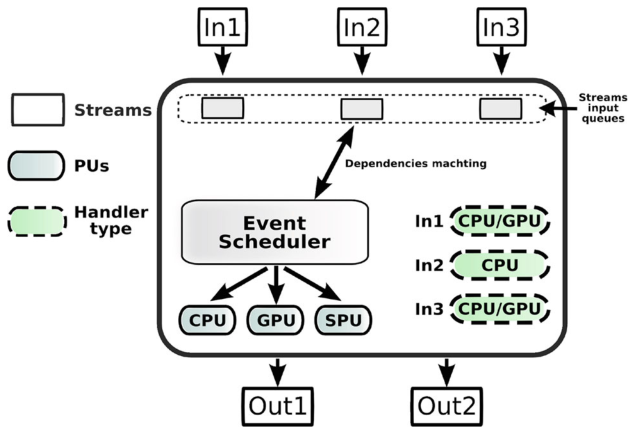


Fig. 3 The architecture of a single filter. Input streams (*top blocks*) generate events that must be handled by the filter. Different handler functions (*dashed round boxes*) can be provided by the programmer for each type of event and processing unit. The event scheduler coordinates the filter operation, selecting events to be processed by worker threads (WTs) that invoke the handling functions according to the available processing units (*round boxes*). As processing progresses, data are sent to the output streams (*bottom blocks*), generating events on the next filter (not shown)

from multiple input streams (In1, In2 and In3), each of which has an associated event queue and handler functions. As depicted in the figure, handlers may be implemented for different types of processors.

The filter *Event Scheduler* runs independently for each filter instance and is responsible for selecting events to be executed. In our implementation, one worker thread (WT) is created for each CPU core, and when GPUs are available, one WT is assigned to manage each GPU. When an event is created as the result of a received message, it is not immediately assigned to a WT. Instead, the event is queued in a list of events ready for execution, and the assignment of events to WTs occurs on-demand as WTs become idle and request work from the scheduler. In the current implementation, a demand-driven, first-come, first-served (DDFCFS) task assignment policy is used as the default strategy, but other policies are available [65].

All filters run concurrently, typically on different machines, and communication between them is managed by the runtime system. Anthill has two implementations for the communication layer, responsible for transferring data between machines, which are built on top of message passing interface (MPI) [1] and parallel virtual machine (PVM) [63]. The choice of the implementation does not affect the application code, as the same filter interface is provided in both cases.

4 The distributed index Hypercurves

Hypercurves is a parallel version of the sequential Multicurves (Sect. 3.1) that is built on top of the filter-stream programming paradigm implemented by Anthill. This section provides details on the Hypercurves parallelization strategy, which was supported by a probabilistic proof of equivalence between Multicurves and Hypercurves (Sect. 4.2).

In Hypercurves, the database is partitioned without replication among the nodes in the execution environment. Searching is performed locally in the subindexes managed by each node, and a reduction stage merges the results. The cost of the algorithm is dominated by the local subindexes searches, which are further dependent on the probe-depth used (the number of candidates to retrieve from each subindex). When using the same probe-depth as the sequential algorithm for each local index of the distributed environment, the answer provided by Hypercurves is guaranteed to be at least as good as that provided by the sequential algorithm. However, this is an extremely pessimistic and costly choice for the distributed probe-depth. We have shown that the quality of Hypercurves is equivalent to that of Multicurves with very high probability when dividing the original probe-depth (that used in the sequential execution) by the number of nodes used and adding a small “slack”. The user can also modify the probe-depth of the parallel algorithm (Hypercurves) according to Eq. 6 (Sect. 4.2) to guarantee that the quality of Hypercurves is equivalent to that of Multicurves with any required probability.

This section focuses on a description of the CPU-only Hypercurves, and the GPU accelerated approach is presented in Sect. 5.

4.1 Hypercurves parallelization strategy

The parallelization strategy used in Hypercurves consists of partitioning the database without any replication among the nodes (*filter instances or copies*). The queries are broadcast to all filter copies, each of which finds a local answer in its database subsets. The local answers are then merged into a global answer in a later reduction step. To better exploit the Anthill execution environment, Hypercurves is created by decomposing Multicurves into four types of filters, organized in two parallel computation pipelines (Fig. 4).

The first pipeline is conceptually an index builder/updater with the filters input reader (IRR) and index holder/local searcher (IHLS). IRR reads the feature vectors from the input database and partitions them in a round-robin fashion among the copies of IHLS, which store the vectors received in their local index according to Algorithm 1. The filters may execute concurrently and after the input is exhausted they may interact to update the index, for instance, in the case that the database is mutable.

The second pipeline, which is conceptually the query processor, contains three filters: (1) Query Receiver (QR); (2) IHLS (shared with the first pipeline); and (3) Aggregator. QR is the entry point to the search server, receiving

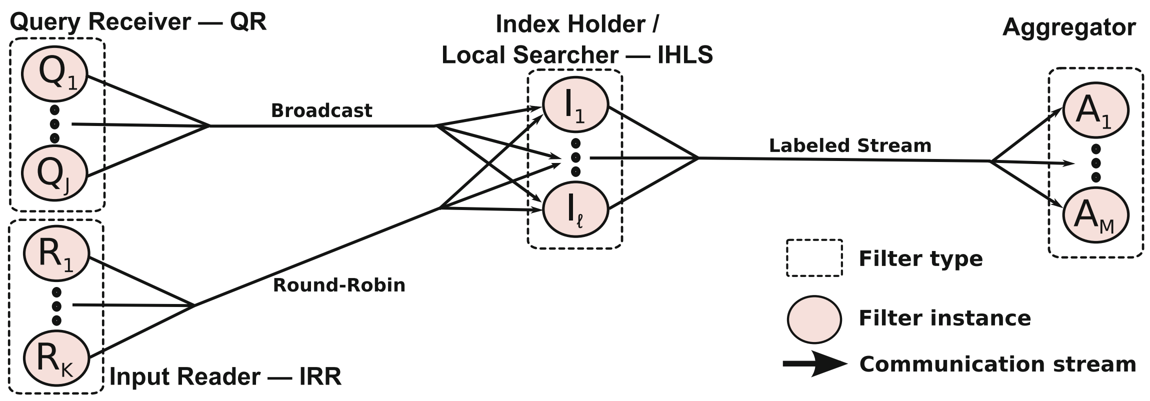


Fig. 4 Hypercurves parallelization design. Four filter types are involved: IRR, which reads data elements from the database and divides them in a round-robin fashion among the IHLS filters to be indexed; QR, which reads queries from the user and dispatches them to the IHLS to be processed; IHLS, which provides a "local" index and query processing for a subset of the data; and Aggregator, which collects local kNN answers to the queries and aggregates them into a global kNN answer. Transparent copies of those filters are instantiated as needed by Anthill's runtime. Several types of streams are used in communications between those copies: for example, during a search, a query is broadcast from QR to all copies of IHLS; then, all local answers relative to that query are sent to the same Aggregator filter, using the "labeled-stream" facility. Each filter is multithreaded and a single instance of each filter can concurrently use all the cores that are available in a node

and broadcasting the queries to all IHLS copies. For each query, IHLS instances independently perform a search on their local index partitions, retrieving *local k* nearest feature vectors, similar to the sequential Multicurves (Algorithm 2). The final answer is obtained by the Aggregator filter, which reduces the IHLS local answers into *global k* nearest vectors. As several Aggregator filter copies may exist, it is crucial that the messages related to a particular query (with the same query-id) are sent to the same Aggregator instance. This correlation is guaranteed using the Anthill Labeled-Stream communication policy, which computes the particular copy of the Aggregator filter that will receive a given message sent from IHLS based on a hash calculated on the query-id. In this context, the query-id corresponds to the label of the message. The computation pattern performed between IHLS and Aggregator is very similar to a generalized parallel data reduction [78].

Hypercurves exploits four dimensions of parallelism: task, data, pipeline, and intra-filter. Task parallelism occurs as the two pipelines are executed in parallel (e.g., index updates and searches). Data parallelism is achieved as the database is partitioned among the IHLS filter copies. Pipeline parallelism results from Anthill"s ability to execute filters of a single computational pipeline (e.g., IRR and IHLS for updating the index) in parallel. Intra-filter parallelism refers to the ability of a single filter copy to process events in parallel as a multithreaded process, thereby efficiently exploiting modern multi- and many-core computers. Intra-filter parallelism is important in Hypercurves, as it allows the creation of a single copy of IHLS per node, instead of one per CPU core as in typical MPI-only implementations. The creation of a single copy of IHLS per node reduces the total number of database partitions and, as a consequence, improves the system's scalability and efficiency.

The broadcast from QR to IHLS has little impact on performance because the cost of the algorithm is dominated by the local searches. Therefore, communication latency is offset by the increases in computation speed. The cost of local searches depends heavily on the probe-depth used (the number of candidates to retrieve from each subindex). As discussed, Hypercurves can be made equivalent to Multicurves by employing on each parallel node a probe-depth at least as large as the one used in the sequential algorithm. However, this over-pessimistic choice is unnecessarily costly, as is presented in the next section.

4.2 Probabilistic equivalence Multicurves–Hypercurves

Multicurves is based upon the ability of space-filling curves to give a total order to data. This ability means that each subindex is a sorted list from which a number of candidates can be retrieved and then verified against the query to obtain the k nearest candidates (Algorithm 2).

In Hypercurves, the index is partitioned and each IHLS filter stores a subset of the database. Therefore, a single filter cannot guarantee the equivalent approximate k-nearest neighbors, and the Aggregator filter collects the local best answers to return a final solution. In terms of an equivalence between Multi- and Hypercurves, it matters little how the candidates are distributed among the IHLS instances because the reduction steps performed after the candidates are selected are conservative: they will never discard one of the "good" answers after it is retrieved. Both Multi- and Hypercurves will only miss a correct answer if they fail

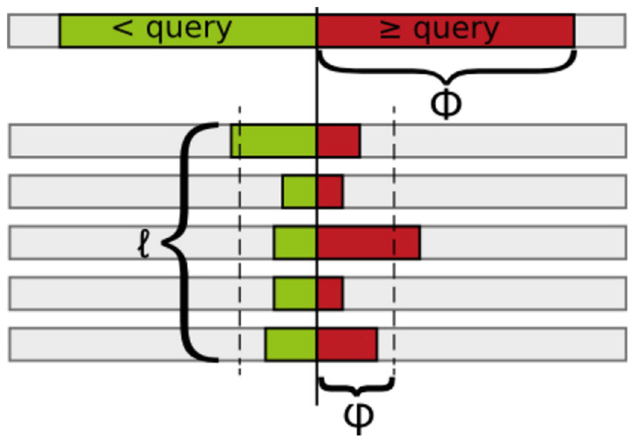


Fig. 5 The probabilistic equivalence between Multi- and Hypercurves corresponds to the following model. In a sorted list, for a query q, we retrieve Φ elements $< q$ and Φ elements $\geq q$. If we distribute the elements of that list randomly into ℓ sorted lists, how many 2φ elements must we retrieve in each of those new lists, in order to ensure that we miss none of the original elements? Because the elements $< q$ and $\geq q$ cannot exchange positions, each "half-list" can be analyzed independently. In the example shown, the equivalence is not guaranteed, because some elements "spill over" the φ limit in two of the half-lists

to retrieve it from the subindexes. Therefore, Hypercurves can be guaranteed to be at least as good as Multicurves by employing on each IHLS filter copy the same probe-depth as used in the sequential Multicurves. However, this choice is costly and over-pessimistic.

Consider the same database, either in one of Multicurves' subindexes (sequential) with probe-depth $= 2\Phi$, or partitioned among ℓ of Hypercurves' IHLS filter instances, each with probe-depth $= 2\varphi$ (even probe-depths make the analyses more symmetric, although the argument is essentially the same for odd values). For any query, the candidates that would be in a single sorted list in Multicurves are now distributed among ℓ sorted lists in Hypercurves. In more general terms, we start with a single sorted list and retrieve the 2Φ elements closest to a query vector. If we randomly distribute that single sorted list into ℓ sorted lists, how many elements must we retrieve from each of the new lists (i.e., which value for 2φ must we employ) to ensure that none of the originally retrieved elements is missed? Note that: (1) due to the sorted nature of the list, the elements before the query cannot exchange positions with the elements after the query; (2) no element of the original list can be lost if all those 2ℓ "half-lists" are shorter than φ. These observations, which are essential to understand the equivalence proof, are illustrated in Fig. 5.

Due to (1), we can analyze each half of the list independently. The distribution of the elements among the ℓ lists is given by a Multinomial distribution with Φ trials and with all probabilities equal to ℓ^{-1}. The exact probability of any list being longer than φ involves computing a truncated part of the distribution, but the exact formulas are exceedingly

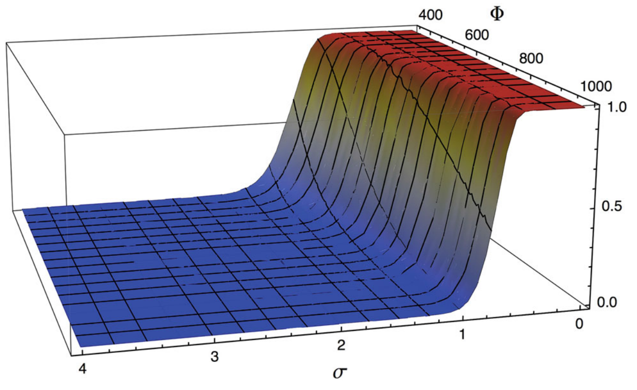


Fig. 6 Equivalence between sequential Multicurves and parallel Hypercurves with distributed probe-depth of 2φ, with $\varphi = (1+\sigma)\lceil\Phi/\ell\rceil$ and number of IHLS filter copies $\ell = 50$. The probability of missing any of the candidate vectors drops sharply close to zero for values of σ that are still small in every configuration

complex and not much elucidative. We can, however, bound that probability. First, an upper bound for the probability of a single list having more than φ elements can be derived using Chernoff bounds. Note that $\text{List}_i \sim \text{Binomial}(\Phi; \ell^{-1})$, i.e., that $\text{List}_i = \sum_{j=1}^{\Phi} X_j$ is a sum of Φ independent Bernoulli trials X_j with probability ℓ^{-1}. The Chernoff upper bounds for List_i are then given by:

$$\Pr[\text{List}_i > (1+\delta)\mu] < \left(\frac{e^{\delta}}{(1+\delta)^{(1+\delta)}}\right)^{\mu} \tag{3}$$

where $\mu = \text{E}[\text{List}_i] = \Phi/\ell$. Making $(1+\delta)\mu = \varphi$ we have the desired formulation:

$$p = \Pr[\text{List}_i > \varphi] < \left(e^{\frac{\ell\varphi}{\Phi}-1}\left(\frac{\ell\varphi}{\Phi}\right)^{-\frac{\ell\varphi}{\Phi}}\right)^{\Phi/\ell} \tag{4}$$

Because the covariance between any two different multinomial components is negative, we can assume independence and still bound the error from above. Therefore, the probability of any missed element in any of the 2ℓ half-lists is bounded by:

$$P = \Pr[\text{List}_i > \varphi, \forall\, 1 \leq i \leq \ell] < 1 - (1-p)^{2\ell} \tag{5}$$

This probability tends to zero for very reasonable values of φ, which are still much lower than Φ. The idea of "11overflowing" or "spilling over" can be made more explicit by taking $\varphi = (1+\sigma)\lceil\Phi/\ell\rceil$, i.e., if we "distribute" the probe-depth among the filters, adding a "safety" or "slack" factor of $(1+\sigma)$:

$$P < 1 - \left(1 - \left(e^{\sigma}(\sigma+1)^{-\sigma-1}\right)^{\Phi/\ell}\right)^{2\ell} \tag{6}$$

For all reasonable scenarios, this probability tends to zero very quickly, even for small positive σ (Fig. 6).

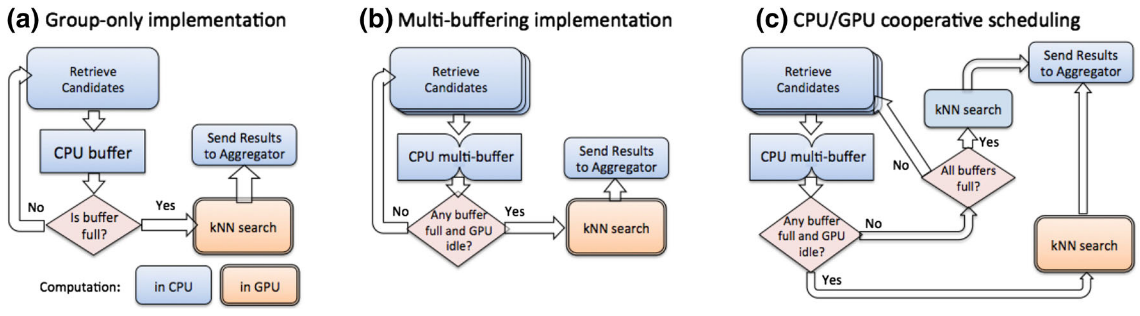

(a) Group-only implementation
Retrieve Candidates
CPU buffer
Is buffer full?
No
Yes
kNN search
Send Results to Aggregator
Computation:
in CPU
in GPU
(b) Multi-buffering implementation
Retrieve Candidates
CPU multi-buffer
Any buffer full and GPU idle?
No
Yes
kNN search
Send Results to Aggregator
(c) CPU/GPU cooperative scheduling
Retrieve Candidates
CPU multi-buffer
Any buffer full and GPU idle?
No
Yes
All buffers full?
kNN search
Send Results to Aggregator
kNN search

Fig. 7 The three progressively sophisticated parallelization strategies of the IHLS filter on CPU–GPU machines. **a** Grouping queries in batches (group-size) to better utilize the GPU. **b** Employing multi-buffer to avoid idle phases in the GPU. **c** Using the CPU to perform kNN when it is idle

5 Hypercurves on CPU–GPU machines

This section presents the design and implementation of Hypercurves on hybrid machines, equipped with CPUs and GPUs, starting with a brief introduction to the GPU computation model. The advanced scheduling, targeting the minimization of response times under fluctuating request loads, will be addressed in Sect. 6.

5.1 Graphics processing units (GPUs)

The use of GPUs as general-purpose processors represents a major paradigm shift toward massively parallel and power-efficient systems, and is a growing trend in high-performance computing. Contemporary NVIDIA GPUs, such as *Fermi* or *Kepler*, have higher peak performances than most multi-core CPUs. At a high level, the GPU hardware is a collection of multiprocessors, each of which consists of a group of scalar processors. For example, the NVIDIA Tesla M2090 used in this study has 16 multiprocessors, each of which has 32 scalar processors, and a total of 512 processing units. The execution model of GPUs differs in some ways from the execution model of multi-core CPUs: GPUs have (1) a hierarchical memory managed by the programmer; and (2) an execution model in which all processors in the same multiprocessor execute the same instructions.

Developers of GPU accelerated applications may employ programming abstractions and frameworks, such as NVIDIA CUDA[1] and OpenCL.[2] CUDA organizes computation into multiple *thread blocks*, which may be mapped for parallel execution in the available multiprocessors. Each of the thread blocks consists of several threads organized in *thread warps*. A warp of threads executes in lock-step, and divergent branching must be avoided in order to maximize performance. Threads in a thread block execute on the same multiprocessor and may communicate using shared or global memory. The code that is launched by an application for execution in a GPU is named *kernel*. We have used CUDA in our GPU-enabled implementations.

5.2 GPU-enabled Hypercurves

This section first presents the baseline GPU parallelization strategy for Hypercurves (Sect. 5.2.1). The implementation of kNN, which is used in our parallelization, on GPUs is discussed in Sect. 5.2.2. Finally, a set of optimizations is proposed to improve the performance of the basic GPU-enabled Hypercurves in Sect. 5.2.3.

5.2.1 Parallelization strategy

The IHLS filter is the most compute-intensive stage of the Hypercurves pipeline, and therefore, it is our target for GPU acceleration. The most expensive computation performed by IHLS is the kNN search in the candidates returned from the subindexes (*probe-depth* × *curves* candidates per query). However, the execution of a single query is usually not sufficient to fully utilize the parallelism of a GPU. Thus, to efficiently use GPUs with IHLS, the IHLS filter must aggregate a batch of queries (*group-size* queries) and dispatch them as a group for the parallel computation of the kNN search on a GPU. The CPU is further used to execute the less compute-intensive operations of IHLS such as searching the subindexes and aggregating the batches of queries in a buffer.

The baseline implementation of the GPU-enabled IHLS filter is divided into three main stages, as presented in Fig. 7a. The first is *Retrieve Candidates*, which will retrieve probe-depth candidates for each query from each of the subindexes (Lines 3 to 8 in Algorithm 2) and copy the candidates to a buffer in the CPU memory. While the buffer of queries is still not full, the first stage is repeated. When the buffer is full, or when candidates for group-size queries are copied to the buffer, the *kNN search* stage computes the k-nearest neighbors for that batch of queries using a GPU. Finally, after

[1] http://nvidia.com/cuda.

[2] http://www.khronos.org/opencl/.

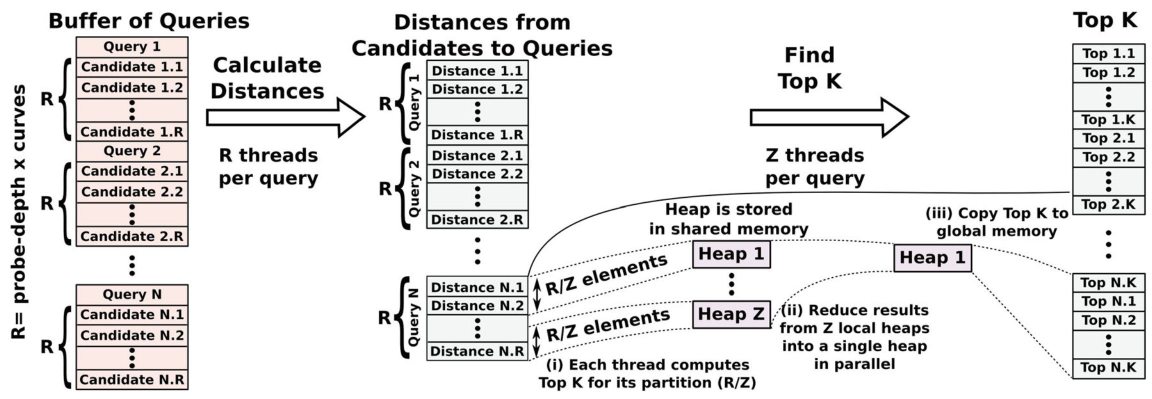


Fig. 8 kNN execution workflow in a GPU. In the first phase, *Calculate Distances*, the distance for the query objects (N) to their respective reference dataset of objects (R) is calculated using a total of N × R threads. Further, in the *Find Top-k* phase, the *k* nearest objects to each of the queries are selected. This phase uses multiple threads (Z) for selecting nearest elements of each query. Each of the Z threads uses an independent heap to select the nearest objects in a partition of the R distances to that query, calculated in the previous phase. Further, the Z heaps with local answers are merged in parallel to find the global *k* nearest neighbors for that query. A total of Z × N threads are used by this *kernel*

the kNN search is executed, the results are sent downstream to the Aggregator filter.

5.2.2 Efficient kNN on a GPU

This section presents the GPU-enabled kNN that was implemented using CUDA. The kNN used in Hypercurves is a variant of the traditional kNN, which has been ported to accelerators in several works [27,33,35,59]. The traditional kNN compares the same reference dataset to a set of queries. However, in the kNN used by IHLS each one of N queries in the set of queries is compared to a different reference dataset retrieved from the indexing curves. The size of the reference dataset for each query is defined as R = probe-depth × curves, where probe-depth is the number of candidates retrieved from each curve and curves refers to the number of indexing curves used. The computation workflow of our kNN, presented in Fig. 8, is organized into two phases, each of which is implemented as a different *kernel*. The first phase, *Calculate Distances*, computes the distances between each query and its R candidates. In this stage, one GPU thread is assigned to each candidate, accounting for a total of $R \times N$ threads in that GPU *kernel*. In addition, the GPU shared memory is employed to store the queries, as each query is reused in R distance calculations.

The second phase of the kNN, *Find Top-k*, selects the k nearest points' to each query using the distances to the R candidates. In this stage, a set of Z threads cooperate in the top-k calculation of each query; thus, a total of $Z \times N$ threads are used by this *kernel*. The bottom-right area of Fig. 8 presents the top-k computation for a single query. Initially, the candidates associated with a query are partitioned among the Z threads, which use a thread local heap to calculate the k nearest points to the query in their partitions. Roughly, each thread will evaluate R/Z points in this phase. A parallel reduction is employed to reduce the Z heaps into a single heap, which will contain the global k nearest points to the query. This reduction is organized in the form of a loop, similar to a parallel prefix sum [28], where each active thread merges two heaps per iteration. As a consequence, the number of active threads and heaps will be cut in half after each iteration. This process continues until a single heap exists.

In our parallelization, the heaps are stored in shared memory for fast access, unless their size (k) is so large that it no longer fits in that memory. Insertions in a heap are data-dependent and may lead to divergent branching among threads in a warp. However, in practice, we have observed that the probability of inserting elements in a heap decreases quickly as candidates are evaluated, minimizing the number of divergent branches. The *Find Top-k* phase distinguishes our work from most of the previous GPU-enabled kNN searches, which employ a coarse-grained parallelization of this phase by assigning a single thread per query. However, the use of one thread per query is a strong limiting factor to the full utilization of GPUs when the number of queries to be computed is not very large.

In addition to optimizing the computation, our method also reduces the impact of data transfer costs by overlapping computation with data movements. In our implementation, the group-size queries dispatched for execution by the IHLS filter are divided into multiple partitions and the communication and kNN computation processes for those partitions are dispatched for parallel execution using multiple CUDA streams.

5.2.3 IHLS optimizations

This section presents optimizations to the basic IHLS implementation (Sect. 5.2.1), which include techniques to maximize the use of a GPU, the utilization of CPUs in computationally-intensive kNN operations in cases where the CPUs would otherwise remain idle, and the use of multiple GPUs in a node.

1) *Minimizing GPU idle phases*: In the basic IHLS parallelization, the Retrieve Candidates and the kNN search phases are computed sequentially. This strategy creates idle periods in the GPU after a batch of queries is processed, as the GPU idly waits for the CPU to fill up the buffer with the next batch of queries. To minimize these GPU idle periods, the IHLS filter was modified to (1) use multiple CPU threads in the execution of CPU phases, while retrieving candidates, and while copying to the buffer; and (2) employ a multi-buffer scheme that allows CPU threads to fill up a buffer of queries while a second buffer is processed by the GPU-enabled kNN. With this strategy, a batch of queries will be ready for GPU computation after the accelerator has finished the current kNN search (Fig. 7b).

2) *Cooperative CPU–GPU kNN execution*: Due to the large number of computing cores available in current CPUs and the low computational costs involved in retrieving and buffering candidates for kNN computation, the use of CPUs only in these phases may not be sufficient to fully utilize all CPU cores in a machine. Therefore, whenever the CPUs complete the process of filling up all available buffers, they are used to perform kNN computations instead of remaining idle (Fig. 7c). We have noticed that using two buffers per GPU is sufficient.

3) *Execution on multi-GPU nodes*: The ability to efficiently use multiple GPUs in a single node is another important feature of Hypercurves. To employ multiple GPUs, we allow the IHLS filter to dispatch batches of queries (buffers) concurrently for execution with the available accelerators, assigning one batch of queries to each GPU. The pipeline for the multi-GPU IHLS filter remains similar to that shown in Fig.7c, except that threads will check whether at least one of the GPUs is available for computation when deciding whether the GPU should be used. Machines with multiple multi-core CPUs and multiple GPUs may also have heterogeneous configurations of data paths between CPUs and GPUs to reduce bottlenecks in data transfer between these devices. The multi-GPU node used in our evaluation is built with three GPUs and two multi-core CPUs that are connected to each other through a nonuniform memory architecture (NUMA) configuration. Multiple I/O hubs exist in this configuration, and the number of links traversed to access a GPU varies based on the CPU used by the calling process (see Fig. 9 in Sect. 7) [66].

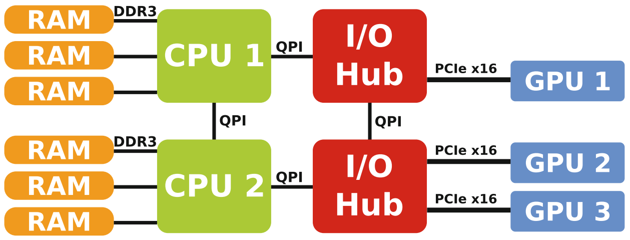


Fig. 9 Multi-GPU node architecture

Therefore, the efficient use of these multiple I/O hubs requires that CPU threads using the GPUs are mapped to appropriate CPU cores. In Hypercurves' implementation, the placement of CPU threads is performed so as to minimize the number of links traversed to access the GPU. In other words, in our example multi-GPU machine, the CPU thread using GPU 1 is bind to CPU 1, and threads using GPU 2 and GPU 3 execute in CPU 2.

6 Response-time aware task partition

This section addresses the problem of using CPU–GPU equipped machines to accelerate online services. GPUs are throughput-oriented devices that typically deliver their best performances in scenarios of high parallelism. In Hypercurves, however, the computation of a single user request is not sufficient to completely utilize a GPU and, as previously discussed, multiple queries must be grouped and concurrently processed to maximize GPU utilization. Although the GPU throughput increases when a batch of queries is processed in parallel, this performance gain is attained at the cost of longer execution times for that group of queries as compared to the execution time of a single query. Thus, the grouping of queries required to maximize the utilization of GPUs may contradict the primary goal of online applications: minimizing the response time of each individual query.

The response time observed by a query ($T_{\text{resp}}(t)$), submitted to the system at timestamp t, consists of two main components. The first component is the *query execution time* ($T_{\text{exec.}}(\text{proc})$) that refers to the time a processor spends executing a query. This time varies according to the device used, e.g., CPU or GPU. The second component is the *query queue time*, which is the interval in which a query remains queued in the system waiting to start its execution. The queue time is derived from the system load (load(t): the number of events waiting for execution) at the query submission time t and the system throughput (the number of events processed per second).

$$T_{\text{resp.}}(t) = T_{\text{exec.}}(\text{proc}) + \frac{\text{load}(t)}{\text{throughput}} \tag{7}$$

An analysis of the response-time components (presented in Eq. 7) allows us to systematically evaluate configurations in which the use of GPUs is pertinent. First, if the system load is low, the execution time will be the most costly component of the request response time. Thus, the selection of a processor with a shorter execution time for a single query or a small number of queries will minimize the response times. For fine-grained tasks (queries) as computed in Hypercurves, the execution of a single query in either of the processors would lead to very low response times. The CPU would achieve execution times comparable to those of a GPU because of the costly overheads to start a computation in the latter (e.g., the cost of calling a GPU kernel and of data transfers from devices). Therefore, regardless of the processor used, the response times observed by the users are low in these circumstances and there is little room for improvement with scheduling.

During periods in which the load of the system (load(t)) is higher than the throughput, the system is saturated and queue time tends to quickly dominate the query response times, as observed in queuing theory [2,43]. This is the system condition that would presumably lead to the highest response times and negatively affect the application quality of the service. Therefore, in setups in which the CPUs are not sufficient for the prompt execution of the incoming requests, and congestion exists in the queue of events waiting for execution, the use of GPUs would be beneficial as a mechanism to increase the system throughput and, as a consequence, reduce or eliminate the expensive queue time component.

In summary, the use of GPUs is advantageous under certain load conditions. Because the load of the system varies throughout execution, the appropriate use of GPUs by online applications requires a dynamic CPU–GPU task scheduling that should take into account the processors' characteristics and the instantaneous system load to retune task partitioning dynamically.

The CPU–GPU task scheduling algorithm for online applications that we propose, presented in Algorithm 3, is named the Dynamic Task Assigner for Heterogeneous Environments (DTAHE). This algorithm runs independently in each instance of the IHLS filter and is executed concurrently by the multiple WTs created in an IHLS instance. The main role of the DTAHE is to decide (1) the processor that should be used to compute each event (query) retrieved from the waiting queue by the WTs and (2) when a batch of queries should be dispatched for GPU computation.

Each WT in a filter executes until there remain no additional work or tasks to be processed (*EndOfWork* is received). When an event is returned for computation (Line 1), the WT will execute that event using the CPU if the load of the system is low and the number of events waiting for execution is smaller than or equal to the number of idle CPU threads, or if all buffers used to aggregate queries' candidates for GPU

Algorithm 3 DTAHE

```
IdleCPUs(): number of CPU cores idle
GetEvent(): pop event from the waiting queues
Buffers: set of buffers used to store query candidates
BufferEvent(e, b): store candidates to event(query) e in one of the
    buffers in b
AnyGPUIdle(): true if any GPU is idle
AllFull(b): true if all buffers in b are full
AnyFull(b): true if any buffer in b is full
NotEmpty(b): true if a buffer in b is not empty
ProcessInCPU(e): execute event e in the CPU
ProcessInGPU(b): execute buffer(batch) b of queries in the GPU
1: while (($e \leftarrow GetEvent()) \neq EndOfWork$) do
2:   if $load(now) \leq IdleCPUs()$ or
     $AllFull(Buffers)$ then
3:     $ProcessInCPU(e)$
4:   else
5:     $BufferEvent(e, Buffers)$
6:   if AnyGPUIdle() then
7:     if ($load(now) < IdleCPUs()$ and
       $NotEmpty(Buffer)$) or $AnyFull(Buffers)$ then
8:       $ProcessInGPU(Buffers)$
```

execution are already full (Line 2–3). If these conditions are not met, the query will be routed for GPU execution. In that case, query candidates are retrieved from the indexes and copied to the buffer of queries (Line 5).

Finally, before trying to retrieve another event for computation from the waiting queues, the WTs verify whether a buffer of queries should be dispatched for GPU execution. GPU execution occurs when at least one GPU is idle and either (1) the load in the system is low and there are queries buffered for GPU execution or (2) there exists a full buffer in the set *Buffers*.

Notice that the optimization of the group sizes is implicit. By dispatching the GPU buffers for execution before they are full, the system is matching the instantaneous incoming request rates to the processing rates and, as a consequence, is greedily selecting the optimal number of queries to be grouped for GPU execution under the instantaneous system load.

7 Experimental results

In this section, we evaluate the performance of Hypercurves in CPU–GPU equipped machines. Experiments were executed in three configurations of machines with different generations of GPUs. The *first setup* is a node with two quad-core *AMD Opteron* 2.0 GHz *2350* processors, 16 GB of main memory, and a *NVIDIA GeForce GTX260* GPU. The *second setup* is an eight-node cluster connected with Gigabit Ethernet in which each node is equipped with two quad-core *Intel Xeon E5520* processors, 24 GB of main memory, and one *NVIDIA GeForce GTX470*. The *third setup* is a machine equipped with a dual socket *Intel X5660* 2.8 GHz *Westmere* processor, three *NVIDIA Tesla M2090* (*Fermi*) GPUs, and 24 GB of DDR3 RAM (See Fig. 9). All machines used *Linux*.

The main database used in the evaluation contains 130,463,526 SIFT local feature vectors [39] with 128 dimensions. The SIFT vectors were extracted from 233,852 background Web images and 225 foreground images from our personal collections. The foreground images were used to compute sets of feature vectors that must be matched, while the background images have generated the feature vectors used to confound the method. The foreground images, after strong transformations (rotations, changes in scale, nonlinear photometric transformations, noise addition, etc.), were also used to create 187,839 query feature vectors.

The experiments concentrate on issues of efficiency because, as demonstrated in Sect. 4.2, Hypercurves returns the same results as Multicurves with very high probability. Thus, by construction, Hypercurves inherits the good trade-off between precision and speed of Multicurves [73]. We also provide a brief comparison between Multicurves and other popular method as supplementary material in the Appendix A.3.

7.1 Elementary kNN performance in a GPU

This section evaluates the performance of our kNN parallelization for GPUs (detailed in Sect. 5.2.2). We first present a performance comparison of the *Find Top-k* phase used in our implementation to other GPU-enabled kNNs, as this is the stage that distinguishes most of the available kNN GPU implementations (Sect. 2.2.2). Our evaluation is compared to the *Vector-* [4] and *Heap-* [50] based implementations, which employ a coarse-grained parallelization in which a single thread calculates the top-k answers to a query using, respectively, an unordered vector and a heap to maintain the k nearest points during the search. The *Heap-Reduce* version, which was proposed and implemented in this work, employs a fine-grained parallelization in which multiple threads cooperate in the top-k calculation of a single query (See Fig. 8 in Sect. 5.2.2). The fourth approach, named *Sort* [50], uses a GPU-based sorting algorithm (our implementation uses Thrust [9]) to sort the distances from all queries and their reference datasets together. To avoid mixing distances to reference points relative to different queries during the sorting phase, the query-ids are attached to the most significative bits of the distances. After sorting, the nearest elements are the first k elements in the batch of distances belonging to each query.

The execution times for the top-k strategies are presented in Fig. 10. This set of experiments used a single M2090 GPU, 400 queries, a probe-depth of 250, and eight curves. As presented, the execution times of the Vector and Heap coarse-grained parallelizations increase as the value of k grows and,

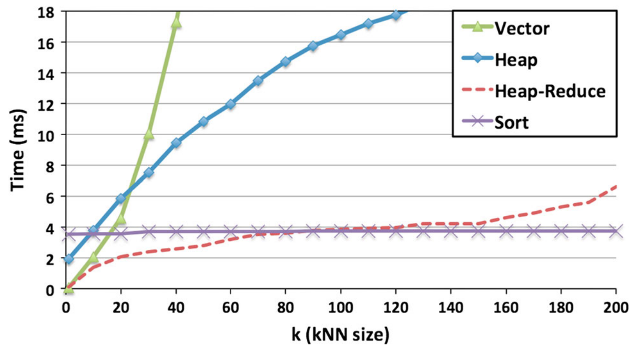


Fig. 10 A study of the find top-k strategies

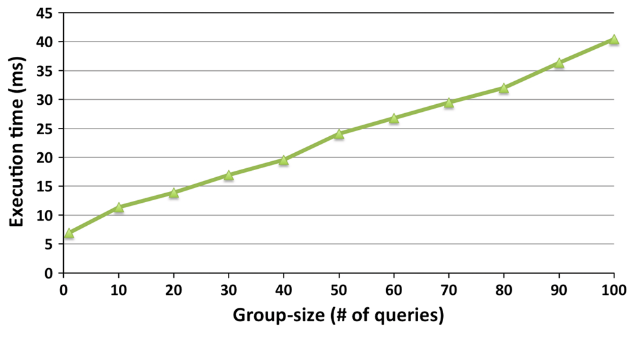
as expected, the behavior of the Heap is better than that of the Vector. Further, the performance of our Heap-Reduce implementation attained better results than did Vector and Heap for all values of k. In our experimental evaluation, the value Z used by the Heap-Reduce has been set to be the size of a CUDA warp of threads, since it empirically attained the best performance. The Heap-Reduce implementation additionally achieved the best performance among all methods for values of k up to 80. For larger k values, the performance of the Sort strategy became competitive, but such large values are uncommon in multimedia applications. Values of interest in such applications are approximately 10–20.

It is worth mentioning that the top-k phase is a challenging operation for GPU parallelization, and one could consider copying the distances to the system memory and using the CPU for this computation. We have observed, however, that this strategy is not worthwhile in our case because the efficient GPU-based top-k calculation is still faster than the cost of copying the distances to the CPU (which requires 3.4 ms) for most k values of interest.

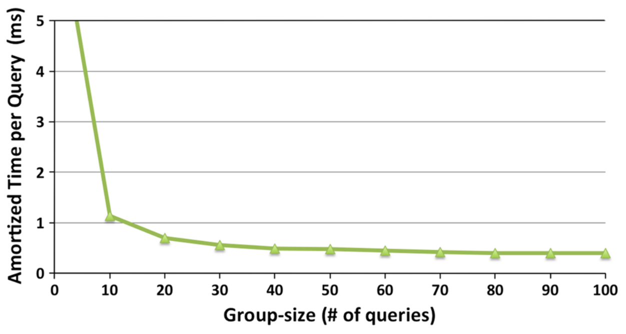
In Fig. 11, we present the kNN execution times (including data transfers) for different numbers of queries. These results show, as expected, that the overall execution time increases as group-size increases (Fig. 11a). However, the amortized cost of computing a query decreases as group-size increases, as a consequence of the higher parallelism that leads to a better GPU use (Fig. 11b).

7.2 Hypercurves in hybrid machines

In this section, we evaluate the performance of Hypercurves implementation on GPUs. We first present the baseline version of Hypercurves using query grouping that is detailed in Sect. 7.2.1. Optimizations for minimizing the GPU idle time, for using CPUs in kNN computations, and for executing on multi-GPU nodes are presented, respectively, in Sects. 7.2.2, 7.2.3, and 7.2.4. This set of experiments employs a subset of the main database containing 1,000,000 feature vectors



(a) Execution time of a group of queries.



(b) Amortized execution time per query.

Fig. 11 kNN execution as group-size varies (k = 10)

and 30,000 queries. A smaller database was selected with the intention of reducing the overall execution time of our experiments. Moreover, a single machine is used in each experiment and all queries are dispatched for computation at the beginning of the execution. In the CPU-only version of the application, the queries are processed in the same order that they are received by the filters, without aggregation of the queries in batches. Essentially, the CPU-only Hypercurves scheduler consists of Lines 1 and 3 of Algorithm 3 (Sect. 6). We emphasize that a large dataset with approximately 130 million feature vectors is used in our multi-node scalability evaluation (Sect. 7.4).

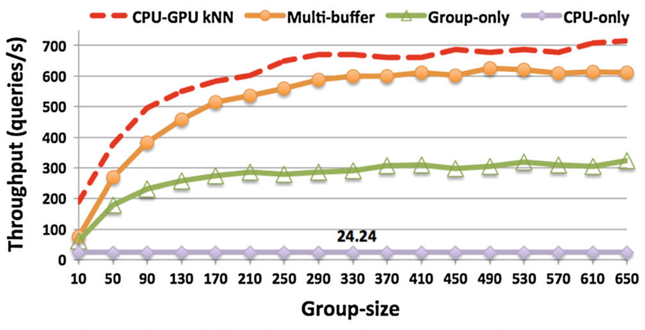
7.2.1 Effect of group-size on performance

The throughput (queries executed per second) of the GPU-enabled Hypercurves for a typical value of probe-depth is presented in Fig. 12. The execution scheme evaluated in this section refers to the strategy shown in Fig. 7a, labeled “Group-only” in the results, which is the baseline version of the application. The results show that a small number of queries is not sufficient to fully use a GPU and that performance improves as group-size increases. Additionally, the value of group-size differentially affects the GPUs used. A larger number of queries is necessary for achieving peak performance with the GTX260 GPU, compared to the other GPUs used. This is a result of the lower data transfer rates and higher communication latency of the GTX260, which requires larger chunks of data to maximize performance.

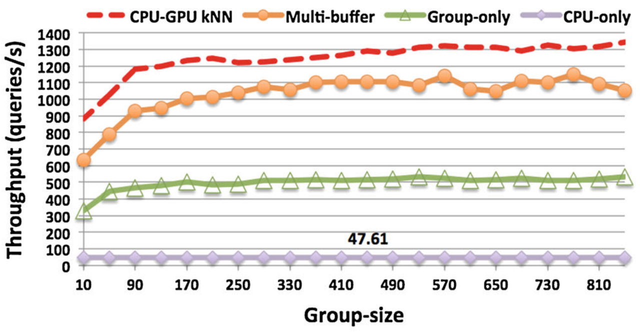
A comparison of the throughputs achieved by the GPUs show different levels of performance, which are derived from the improvements in data transfer rates and computing capabilities among these generations of GPUs. The following sections evaluate optimizations on top of this baseline parallelization.

7.2.2 Impact of minimizing GPU idle times

This section evaluates our strategy to reduce the idle periods of GPUs (depicted in Fig. 7b in Sect. 5.2.3). The throughput of Hypercurves for this optimization is presented in Fig. 12, using the curve labeled “Multi-buffer”. As shown, the performance of this version of Hypercurves nearly doubles the throughput of the baseline version (“Group-only”) presented in the previous section. These improvements are the result of a better use of GPUs as batches of queries for GPU computation are filled up by the CPUs in parallel with the GPU execution and, as a consequence, the GPUs do not experience idle periods waiting for the CPU to prepare a group of queries for execution. Moreover, speedups of about 26× were attained over the single CPU core version of the application,



(a) AMD/GTX260 — Probe-depth=250



(b) Intel/GTX470 — Probe-depth=250

Fig. 12 Hypercurves performance as group-size varies on both machine configurations (AMD/GTX260 and Intel/GTX470). Dynamic scheduling is *not* employed in these experiments: the parameters used are fixed for each execution (each data point)

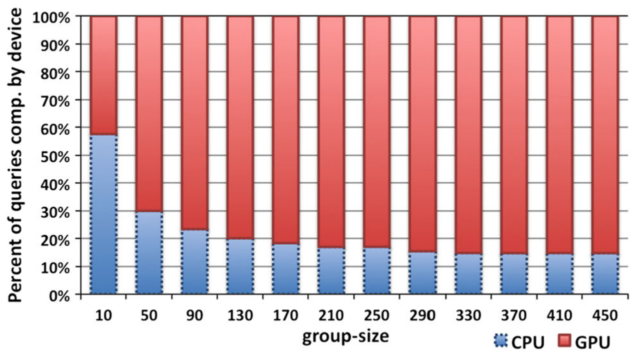


Fig. 13 kNN partitioning between CPU and GPU as group-size varies

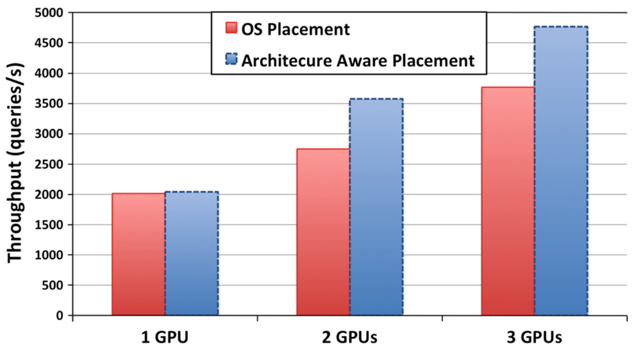


Fig. 14 Single node Multi-GPU scalability

for probe-depth values of 250, 350, and 450 (probe-depths of 350 and 450 are not presented in Fig. 12). Additionally, the GPU accelerated version of Hypercurves attained speedups of approximately 3.4× that of the parallel CPU version of the application using the eight cores available in either the AMD or Intel equipped machines.

In these experiments, the CPU cores were able to fill up the candidate buffers faster than the GPUs were able to consume them, and as a consequence the CPU cores experienced idle periods after the buffers were full. The ability to use these idle CPU cycles to improve the performance is discussed in the next section.

7.2.3 Gains due CPU–GPU kNN execution

The improvements in the kNN phase execution on CPUs during periods in which otherwise the CPUs would remain idle are evaluated in this section. This execution strategy is presented in Fig. 7c and discussed in Sect. 5.2.3. The throughput of this version of Hypercurves is shown in Fig. 12, using the "CPU-GPU kNN" label. As presented, the use of the CPU to compute the kNN phase significantly improved the throughput of Hypercurves, achieving an increase of performance of approximately 1.23× that of the Multi-buffer version. In addition, this version is 4.2× faster than the CPU-only multicore parallel version of Hypercurves.

The gains from the use of CPUs to compute the kNN phase are higher for smaller values of group-size. This result is explained by the lower efficiency of the GPU for small group sizes, which consequently require fewer candidates to be buffered per unit of time and increase the CPU idle periods that are used in kNN computations. Fig. 13 illustrates that the percentage of kNN tasks processed by the GPU increases as group-size grows, which is a result of better GPU efficiency for higher group-size values.

7.2.4 Multi-GPU execution

This section assesses the scalability of Hypercurves in a machine equipped with multiple GPUs. We use two strategies for the placement of CPU threads that are managing the GPUs: (1) OS: threads are not bound to any specific CPU core: the Operating System chooses the placement of threads; and (2) Architecture Aware: threads that manage a GPU are bound to the CPU that is "closest" to that GPU (regarding the number of links traversed to access the GPU) (Sect. 5.2.3).

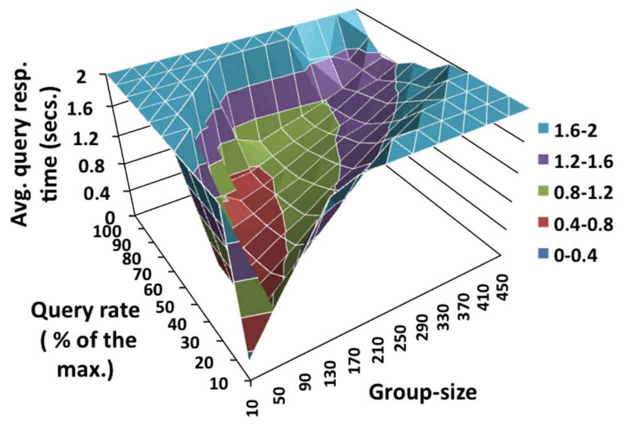
The results of single node multi-GPU execution are presented in Fig. 14. The performance of both placement strategies increases with the number of GPUs used. In addition, for the configuration with 3 GPUs, the architecture-aware placement attains a throughput that is 1.24× higher than the one delivered by the OS placement. The improved scalability of the architecture-aware placement is a result of its ability to efficiently use CPU-GPU communication channels. For example, the cost of transferring data increases 56 and 74 % in comparison with the single GPU execution, respectively, when two and three GPUs are used with the OS placement. Architecture-aware placement is able to reduce the data transfer cost increments to only 16 and 34 %, respectively, for two and three GPUs.

7.3 Evaluating query response times

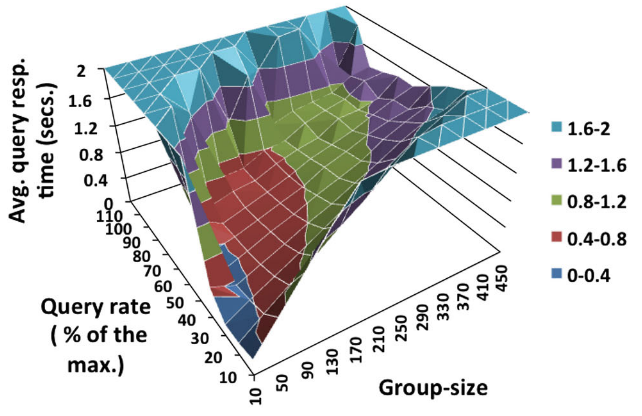
In this section, we analyze Hypercurves' query response times in CPU–GPU equipped machines. First, the impact of the group-size values on query response times is evaluated in Sect. 7.3.1. In Sect. 7.3.2, we assess the performance of the CPU-GPU dynamic scheduling algorithm (DTAHE) under scenarios of runtime variable workloads.

7.3.1 Effects of group-size on response times

This section evaluates the effects of the group-size on the query response times under different query rates. We vary both the number of queries submitted per second and the group-size value used by Hypercurves across experiments, but we *keep them fixed within each run*. Workloads varying the query rates throughout a single execution are analyzed in the next section.



(a) AMD node — GPU-only



(b) AMD node — CPU–GPU

Fig. 15 Average query response times as query rates (% of the maximum) and group sizes vary, for a probe-depth of 350. The parameters are kept fixed for each execution (each data point)

The average response times of Hypercurves using only the GPU to compute kNN operations are presented in Fig. 15a as the group-size values and the query rates (% of maximum delivered by the application) are varied. It is important to note that (1) a single value for group-size would not minimize the response times under different query rates; (2) the response-time function to be optimized is complex; and (3) the response-time function also depends on other aspects such as the application parameters and the configuration of the hardware used.

The Hypercurves response times in the CPU–GPU cooperative execution of kNN operations are presented in Fig. 15b. The average response times for the CPU–GPU executions are smaller for all configurations of query rates (load) and group sizes. The average reduction in response times across all experiments with the use of CPU–GPU was approximately 58 % compared to the version of the application that uses only GPUs in kNN computations. This improvement is a consequence of the fact that queries computed by the CPU have a shorter execution times compared to batches of queries computed in a GPU.

7.3.2 Response times with variable workloads

The ability of the DTAHE dynamic scheduler (Sect. 6) to adapt the Hypercurves CPU–GPU work partition under scenarios with stochastic, variable workloads is assessed in this section. In this evaluation, the load/request rate submitted to Hypercurves varied during the execution following a Poisson distribution with an expected average rate (λ) varying from 20 to 100 % of the maximum throughput of the application. That maximum throughput was computed in a preliminary run in which all the queries were sent for computation at the beginning of the execution.

These experiments also included a static scheduler, named *best static*, as a baseline for the evaluation of DTAHE. The static scheduler uses a fixed value of group-size throughout the entire execution and is therefore not able to change the task partitioning dynamically according to the instantaneous system load, as is performed by DTAHE. The fixed group-size value used by the *best static* scheduler for each query rate configuration is the value that minimizes the average response times. Therefore, for each average load level (λ), we performed an exhaustive search for the value of group-size that minimized the average response times in preliminary executions of the application.

Finally, the average response times for multiple request rate levels using the best static and the dynamic DTAHE schedulers are presented in Table 1. The queries' average response times were significantly reduced with the use of DTAHE compared to the best static scheduler. The response times attained with the use of DTAHE are up to 2.77×, 1.66×, and 1.55× smaller than the response times of the best static scheduler, respectively, for nodes equipped with the GTX260, GTX470, and M2090 GPUs.

The evaluation with a Poisson request rate (λ) equal to 100 % of the application maximum throughput is the only case in which DTAHE does not deliver better results than a static scheduler. However, in this configuration a dynamic scheduler was not expected to beat the static scheduler because the load of the systems is very high during the entire execution and the static best value of group-size coincides with the best instantaneous choice. Even in this extreme case, however, the performance penalty of the dynamic scheduling is small and the dynamic scheduling leads to better performance in all other configurations.

7.4 Evaluating the scalability of Hypercurves

In this section, we analyze the performance of Hypercurves in distributed memory machines. This set of experiments was executed using the eight-node Intel cluster with the

Table 1 Avg. query response times (in s) for static and DTAHE schedulers under stochastic loads

Scheduling	Poisson λ (% of max. throughput)				
	20	40	60	80	100
(a) First setup (GTX260 node)					
Best static	0.11	0.4	0.42	0.61	0.98
DTAHE	0.06	0.13	0.14	0.22	1.02
(b) Second setup (GTX470 8-node cluster)					
Best static	0.054	0.098	0.12	0.25	0.65
DTAHE	0.034	0.089	0.10	0.16	0.68
(c) Third setup (3 GPUs M2090 node)					
Best static	0.046	0.075	0.10	0.20	0.52
DTAHE	0.030	0.061	0.08	0.12	0.53

Table 2 Throughput as database size and number of nodes increase proportionally (probe-depth = 350)

Number of nodes	1	2	3	4	5	6	7	8
# of cores / GPUs	16 / 1	32 / 2	48 / 3	64 / 4	70 / 5	86 / 6	102 / 7	118 / 8
Optimist (queries/s)	964	1904	2649	3598	4490	5397	6297	7483
Pessimist (queries/s)	964	1683	2197	2849	3416	3968	4498	5135
Network traffic (MB/s)	0.51	2.03	4.24	7.68	11.98	17.29	23.54	31.97

CPU–GPU cooperative version of the application that uses all CPU cores and GPUs available on the nodes. The main database with 130, 463, 526 feature vectors was used in this evaluation. This evaluation focuses on scale-up experiments in which the database is increased proportionally to the number of nodes in each run. Therefore, $n/8$ of the database is used for the experiment, where n is the number of nodes. The database is partitioned among IHLS filters in a round-robin fashion without any replication. The communication between filters is managed by the Anthill framework and can use either MPI or PVM. We have evaluated the Anthill implementations using MPI and PVM, but no significant performance difference between the two was observed.

We consider the compromise between the performance of parallelism *vs.* precision equivalence to the sequential algorithm. A scale-up evaluation is appropriate in our application scenario because we expect to obtain an abundant volume of data for indexing. Therefore, a speed-up evaluation starting with a single node holding the entire database may not be realistic.

The query rate delivered by the algorithm considers two parameterization scenarios named Optimist and Pessimist (Table 2), which differ in their guarantees of equivalence (in terms of the precision of the kNN search) to the sequential Multicurves algorithm. The Optimist parameterization divides the probe-depth equally among the nodes, without any slack: it will only be equivalent to Multicurves in the unlikely case that all candidates of that query are equally distributed on the nodes. The Pessimist parameterization uses a slack that guarantees a probability smaller than 2 % that a candidate vector selected by the sequential algorithm will be missing from the distributed version (See Sect. 4.2 for details). Note that this choice is extremely conservative; to effectively affect the answer, the missed feature vectors from the candidate set must be among the actual top-k set, and k is much smaller than the probe-depth.

The query-processing rates attained by Hypercurves on the scale-up evaluations are presented in Table 2. As shown, the scalability of the algorithm is impressive for both Optimist and Pessimist configurations, achieving *superlinear scale-ups* in all setups. This strong performance of Hypercurves is a result of (1) the fact that the retrieve candidates from indexes phase grows logarithmically with the size of the dataset and (2) the costly phase involving the computation of the distances from the query to the retrieved candidates can, as a result of the probabilistic equivalence (Sect. 4.2), be efficiently distributed among the nodes with a relatively small overhead.

Table 2 also presents the network traffic (MB/s) generated among all filters of the application as the number of nodes used increases. As shown, the amount of data exchanged is relatively small and, as a consequence, the application would be able to scale to a large number of machines. The intra-filter parallelism, the ability to use all CPU cores and GPUs in a node with a single filter instance, is an important feature of Hypercurves (a single IHLS instance is used per machine) that reduces its communication demands compared to using a single instance per CPU core or GPU.

Alongside the very good scalability achieved by the algorithm, the raw query-processing rates (queries/s) are also very high. For example, the numbers of queries that Hypercurves would be able to answer per day are: 646 and 443 million, respectively, for the Optimist and the Pessimist configurations. These query-processing capabilities indicate that, by employing the technology proposed, a large-scale image search system could be built at reasonably low hardware and power costs per request, as GPU accelerators are very computationally efficient and power-efficient platforms.

8 Conclusions and future work

In this work, we have proposed, implemented, and evaluated Hypercurves, an online similarity search engine for very large high-dimensional multimedia databases. Hypercurves executes on CPU–GPU equipped machines and is able to fully utilize these systems. Hypercurves has attained speed increases of approximately 80× the speeds of the single-core CPU version, by using a multi-GPU node. In addition, Hypercurves has achieved *superlinear* scale-ups in all multi-node distributed memory experiments while maintaining a high probability guarantee of equivalence with the sequential Multicurves algorithm, as asserted by the proof of probabilistic equivalence.

We have also investigated the problem of request response times under scenarios with stochastically variable workloads and proposed a dynamic scheduling (DTAHE) that is able to adapt the CPU–GPU task partitioning during execution according to the instantaneous system load to minimize request response times. DTAHE reduced request response times by up to 2.77×, in comparison with the best static scheduling results.

We are currently interested in the complex interactions between algorithmic design and parallel implementation for services such as Hypercurves. We are also investigating how a complete system for content-based image retrieval can be built upon our indexing services and optimized using our techniques and scheduling algorithms. We consider this approach a promising future direction, since Hypercurves implementations in heterogeneous environments offer very good reply rates.

Acknowledgments We would like to express our gratitude to the reviewers for their valuable comments, which helped us to improve our work both in terms of content and presentation. E. Valle thanks CNPq and FAPESP for the financial support of this work. R. Torres thanks CAPES, CNPq (grants 306580/2012-8, 484254/2012-0), and FAPESP for the financial support. W. Meira Jr. thanks CNPq, FAPEMIG and InWeb for financial support of this work. E. Valle and G. Teodoro thank the CENAPAD/UNICAMP for making available the computational resources required by the expensive experiments of this work. This research also used resources of the Keeneland Computing Facility at the Georgia Institute of Technology, which is supported by the National Science Foundation under Contract OCI-0910735.

References

1. The message passing interface (MPI). http://www-unix.mcs.anl.gov/mpi/
2. Adan, I., Resing, J.: Queueing theory. Eindhoven University of Technology, Department of Mathematics and Computing Science, Eindhoven, The Netherlands, Lecture notes (2001)
3. Akune, F., Valle, E., Torres, R.: MONORAIL: a disk-friendly index for huge descriptor databases. In: 20th international conference on pattern recognition (ICPR) (2010)
4. Arefin, A.S., Riveros, C., Berretta, R., Moscato, P.: GPU-FS-kNN: a software tool for fast and scalable kNN computation using GPUs. PLoS ONE **7**(8), e44000 (2012)
5. Augonnet, C., Thibault, S., Namyst, R., Wacrenier, P.A.: StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. In: International Euro-Par conference on parallel processing, pp. 863–874 (2009)
6. Beecks, C., Seidl, T.: On stability of adaptive similarity measures for content-based image retrieval. In: Schoeffmann, K., Mérialdo, B., Hauptmann, A.G., Ngo, C.W., Andreopoulos, Y., Breiteneder, C. (eds) MMM, Lecture Notes in Computer Science, vol. 7131. Springer (2012)
7. Beecks, C., Uysal, M.S., Seidl, T.: Signature quadratic form distances for content-based similarity. In: Proceedings of the 17th ACM international conference on multimedia, MM '09, pp. 697–700. ACM (2009)
8. Beecks, C., Uysal, M.S., Seidl, T.: Signature quadratic form distance. In: Proceedings of the ACM international conference on image and video retrieval, CIVR '10, pp. 438–445. ACM (2010)
9. Bell, N., Hoberock, J.: Thrust: a productivity-oriented library for CUDA. In: Mei, W., Hwu, W. (ed.) GPU Gems. Jade Edition (2011)
10. Beynon, M., Ferreira, R., Kurc, T.M., Sussman, A., Saltz, J.H.: DataCutter: Middleware for filtering very large scientific datasets on archival storage systems. In: IEEE symposium on mass storage systems, pp. 119–134 (2000)
11. Bhatti, N.T., Hiltunen, M.A., Schlichting, R.D., Chiu, W.: Coyote: a system for constructing fine-grain configurable communication service. ACM Trans. Comput. Syst. **16**(4), 321–366 (1998)
12. Böhm, C., Berchtold, S., Keim, D.A.: Searching in high-dimensional spaces: index structures for improving the performance of multimedia databases. ACM Comput. Surv. **33**(3), 322–373 (2001)
13. Bosilca, G., Bouteiller, A., Herault, T., Lemarinier, P., Saengpatsa, N., Tomov, S., Dongarra, J.: Performance portability of a GPU enabled factorization with the DAGuE framework. In: IEEE international conference on cluster computing (CLUSTER) (2011)
14. Boureau, Y.L., Bach, F., LeCun, Y., Ponce, J.: Learning mid-level features for recognition, pp. 2559–2566. IEEE conference on computer vision and pattern recognition (2010)
15. Butz, A.R.: Alternative algorithm for Hilbert's space-filling curve. IEEE Trans. Comput. **100**(4), 424–426 (1971)
16. Castelli, V.: Multidimensional indexing structures for content-based retrieval, pp. 373–433. Wiley, New York (2002)
17. Chandrasekhar, V., Sharifi, M., Ross, D.A.: Survey and evaluation of audio fingerprinting schemes for mobile query-by-example applications. In: Klapuri, A., Leider, C. (eds.) ISMIR, pp. 801–806. University of Miami, Miami (2011)
18. Chávez, E., Navarro, G., Baeza-Yates, R., Marroquín, J.L.: Searching in metric spaces. ACM Comput. Surv. **33**(3), 273–321 (2001)
19. Datar, M., Immorlica, N., Indyk, P., Mirrokni, V.S.: Locality-sensitive hashing scheme based on p-stable distributions. In: Proceedings of the twentieth annual symposium on computational geometry, SCG '04. ACM (2004)
20. Deisher, M., Smelyanskiy, M., Nickerson, B., Lee, V.W., Chuvelev, M., Dubey, P.: Designing and dynamically load balancing hybrid

LU for multi/many-core. Comput Sci Res Dev **26**(3–4), 211–220 (2011)
21. Du Mouza, C., Litwin, W., Rigaux, P.: Large-scale indexing of spatial data in distributed repositories: the SD-Rtree. VLDB J. **18**, 933–958 (2009)
22. Fagin, R., Kumar, R., Sivakumar, D.: Efficient similarity search and classification via rank aggregation. In: Proceedings of the 2003 ACM SIGMOD international conference on management of data, SIGMOD '03. ACM (2003)
23. Fagin, R., Lotem, A., Naor, M.: Optimal aggregation algorithms for middleware. In: Proceedings of the 20th ACM SIGMOD-SIGACT-SIGART symposium on principles of database systems, PODS '01, pp. 102–113. ACM (2001)
24. Faloutsos, C.: Gray codes for partial match and range queries. IEEE Trans. Softw. Eng. **14**, 1381–1393 (1988)
25. Faloutsos, C.: Multimedia Indexing, pp. 435–464. Wiley, New York (2002)
26. Faloutsos, C., Roseman, S.: Fractals for secondary key retrieval. In: Proceedings of the Eighth ACM SIGACT-SIGMOD-SIGART, PODS '89, pp. 247–252. ACM (1989)
27. Garcia, V., Debreuve, E., Barlaud, M.: Fast k nearest neighbor search using GPU. In: CVPR workshop on computer vision on GPU (CVGPU). Anchorage, Alaska, USA (2008)
28. Harris, M., Sengupta, S., Owens, J.D.: Parallel Prefix Sum (Scan) with CUDA. In: Nguyen, H. (ed.) GPU Gems 3, chap. 39, pp. 851–876. Addison Wesley, Reading (2007)
29. He, B., Fang, W., Luo, Q., Govindaraju, N.K., Wang, T.: Mars: A mapreduce framework on graphics processors. In: Parallel architectures and compilation techniques (2008)
30. Hua, G., Fu, Y., Turk, M., Pollefeys, M., Zhang, Z.: Introduction to the special issue on mobile vision. Int. J. Comput. Vis. **96**, 277–279 (2012)
31. Huo, X., Ravi, V., Agrawal, G.: Porting irregular reductions on heterogeneous CPU–GPU configurations. In: 18th international conference on high performance computing (HiPC) (2011)
32. Indyk, P., Motwani, R.: Approximate nearest neighbors: towards removing the curse of dimensionality. In: STOC, pp. 604–613 (1998)
33. Kato, K., Hosino, T.: Solving k-Nearest neighbor problem on multiple graphics processors. In: Proceedings of the 2010 10th IEEE/ACM international conference on cluster, cloud and grid computing, CCGRID '10 (2010)
34. Kruliš, M., Skopal, T., Lokoč, J., Beecks, C.: Combining CPU and GPU architectures for fast similarity search. Distrib. Parallel Databases **30**, 179–207 (2012)
35. Kuang, Q., Zhao, L.: A practical GPU based kNN algorithm. In: International symposium on computer science and computational technology (ISCSCT), pp. 151–155 (2009)
36. Liao, S., Lopez, M.A., Leutenegger, S.T.: High dimensional similarity search with space filling curves. In: Proceedings of the 17th international conference on data, engineering, pp. 615–622 (2001)
37. Linderman, M.D., Collins, J.D., Wang, H., Meng, T.H.: Merge: a programming model for heterogeneous multi-core systems. SIGPLAN Not. **43**(3), 287–296 (2008)
38. Liu, Y., Zhang, D., Lu, G., Ma, W.Y.: A survey of content-based image retrieval with high-level semantics. Pattern Recogn. **40**(1), 262–282 (2007)
39. Lowe, D.G.: Distinctive image features from scale-invariant keypoints. Int. J. Comput. Vis. **60**, 91–110 (2004)
40. Luk, C.K., Hong, S., Kim, H.: Qilin: Exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In: 42nd international symposium on microarchitecture (MICRO) (2009)
41. Mainar-Ruiz, G., Perez-Cortes, J.C.: Approximate nearest neighbor search using a single space-filling curve and multiple representations of the data points. In: Proceedings of the 18th international conference on pattern recognition, pp. 502–505 (2006)
42. Megiddo, N., Shaft, U.: Efficient nearest neighbor indexing based on a collection of space filling curves. Technical Report IBM Research Report RJ 10093 (91909), IBM Almaden Research Center, San Jose California (1997)
43. Menascé, D., Almeida, V.: Capacity planning for web services: metrics, models and methods. Prentice Hall, Englewood (2002)
44. Mikolajczyk, K., Schmid, C.: A performance evaluation of local descriptors. IEEE Trans. Pattern Anal Mach Intel **27**, 1615–1630 (2005)
45. Morton, G.M.: A computer oriented geodetic data base and a new technique in file sequencing. Technical Report, IBM Ltd., Ottawa, Ontario, Canada (1966)
46. Muja, M., Lowe, D.G.: Fast approximate nearest neighbors with automatic algorithm configuration. In: In VISAPP international conference on computer vision theory and applications, pp. 331–340 (2009)
47. nVidia corporation: CUDA CUBLAS library (2010). http://developer.nvidia.com/
48. O'Malley, S.W., Peterson, L.L.: A dynamic network architecture. ACM Trans. Comput. Syst. **10**(2), 110–113 (1992)
49. Pan, J., Lauterbach, C., Manocha, D.: Efficient nearest-neighbor computation for GPU-based motion planning. In: 2010 IEEE/RSJ international conference on intelligent robots and systems (IROS), p. 2243–2248. IEEE (2010)
50. Pan, J., Manocha, D.: Fast GPU-based locality sensitive hashing for k-nearest neighbor computation. In: 19th ACM SIGSPATIAL international conference on advances in geographic information systems, GIS '11. ACM (2011)
51. Pang, H., Ding, X., Zheng, B.: Efficient processing of exact top-k queries over disk-resident sorted lists. VLDB J. **19**, 437–456 (2010)
52. Penatti, O.A.B., Valle, E., Torres, RdS: Comparative study of global color and texture descriptors for web image retrieval. J. Vis. Comun. Image Rep. **23**(2), 359–380 (2012)
53. Ravi, V., Ma, W., Chiu, D., Agrawal, G.: Compiler and runtime support for enabling generalized reduction computations on heterogeneous parallel configurations. In: Proceedings of the 24th ACM international conference on supercomputing, pp. 137–146. ACM (2010)
54. Sagan, H.: Space-filling curves. Springer, New York (1994)
55. Samet, H.: Foundations of Multidimensional and Metric Data Structures (The Morgan Kaufmann Series in Computer Graphics and Geometric Modeling). Morgan Kaufmann Publishers Inc, San Francisco (2005)
56. Satish, N., Harris, M., Garland, M.: Designing efficient sorting algorithms for manycore GPUs. In: IEEE international parallel and distributed processing symposium (IPDPS) (2009)
57. Shakhnarovich, G., Darrell, T., Indyk, P.: Nearest-Neighbor Methods in Learning and Vision: Theory and Practice (Neural Information Processing). The MIT Press, Cambridge (2006)
58. Shepherd, J., Zhu, X., Megiddo, N.: A fast indexing method for multidimensional nearest neighbor search. In: SPIE conference on storage and retrieval for image and video databases VII, pp. 350–355 (1999)
59. Sismanis, N., Pitsianis, N., Sun, X.: Parallel search of k-nearest neighbors with synchronous operations (2012)
60. Smeulders, A., Worring, M., Santini, S., Gupta, A., Jain, R.: Content-based image retrieval at the end of the early years. IEEE Trans. Pattern Anal. Mach. Intell. **22**(12), 1349–1380 (2000)
61. Stone, Z., Zickler, T., Darrell, T.: Autotagging facebook: social network context improves photo annotation. In: IEEE computer vision and pattern recognition workshops (2008)
62. Sun, L., Stoller, C., Newhall, T.: Hybrid MPI and GPU approach to efficiently solving large kNN problems. Tera Grid Poster. URL http://www.isgtw.org/pdfs/kNNposter.pdf (2010)
63. Sunderam, V.S.: PVM: a framework for parallel distributed computing. Concurr. Pract. Exp. **2**(4), 315–340 (1990)

64. Teodoro, G., Fireman, D., Guedes, D., Jr., W.M., Ferreira, R.: Achieving multi-level parallelism in filter-labeled stream programming model. In: The 37th international conference on parallel processing (ICPP) (2008)
65. Teodoro, G., Hartley, T.D.R., Catalyurek, U., Ferreira, R.: Run-time optimizations for replicated dataflows on heterogeneous environments. In: Proceedings of the 19th ACM international symposium on high performance distributed computing (HPDC) (2010)
66. Teodoro, G., Kurç, T.M., Pan, T., Cooper, L.A.D., Kong, J., Widener, P.M., Saltz, J.H.: Accelerating large scale image analyses on parallel, CPU-GPU equipped systems. In: IPDPS, pp. 1093–1104 (2012)
67. Teodoro, G., Pan, T., Kurc, T.M., Kong, J., Cooper, L.A., Podhorszki, N., Klasky, S., Saltz, J.H.: High-throughput analysis of large microscopy image datasets on CPU-GPU cluster platforms. In: IPDPS '13 (2013)
68. Teodoro, G., Sachetto, R., Sertel, O., Gurcan, M., Jr., W.M., Catalyurek, U., Ferreira, R.: Coordinating the use of GPU and CPU for improving performance of compute intensive applications. In: IEEE cluster (2009)
69. Teodoro, G., Valle, E., Mariano, N., Torres, R., Meira Jr., W.: Adaptive parallel approximate similarity search for responsive multimedia retrieval. In: Proceedings of the 20th ACM international conference on information and knowledge management, CIKM '11. ACM (2011)
70. Tuytelaars, T., Mikolajczyk, K.: Local invariant feature detectors: a survey. Found. Trends. Comput. Graph. Vis. **3**, 177–280 (2008)
71. Valle, E., Cord, M., Philipp-Foliguet, S.: Fast identification of visual documents using local descriptors. In: Proceeding of the eighth ACM symposium on document engineering, DocEng '08. ACM (2008)
72. Valle, E., Cord, M., Philipp-Foliguet, S.: High-dimensional descriptor indexing for large multimedia databases. In: Proceeding of the 17th ACM conference on information and knowledge management, CIKM '08. ACM (2008)
73. Valle, E., Cord, M., Phillip-Folliguet, S., Gorisse, D.: Indexing personal image collections: a flexible, scalable solution. IEEE Trans. Consum. Elect. **56**, 1167–1175 (2010)
74. Vetter, J.S., Glassbrook, R., Dongarra, J., Schwan, K., Loftis, B., McNally, S., Meredith, J., Roth, P., Spafford, K., Yalamanchili, S.: Keeneland: bringing heterogeneous GPU computing to the computational science community. Comput. Sci. Eng. **13**(5), 90–95 (2011)
75. Welsh, M., Culler, D., Brewer, E.: SEDA: an architecture for well-conditioned, scalable internet services. SIGOPS Oper. Syst. Rev. 35(5) (2001)
76. Winder, S.A.J., Brown, M.: Learning local image descriptors. In: CVPR (2007)
77. Yiu, M.L., Mamoulis, N.: Multi-dimensional top-k dominating queries. VLDB J. **18**, 695–718 (2009)
78. Yu, H., Rauchwerger, L.: Adaptive reduction parallelization techniques. In: Proceedings of the 14th international conference on supercomputing, ICS '00 (2000)
79. Zezula, P., Amato, G., Dohnal, V., Batko, M.: Similarity search: the metric space approach, 1st edn. Springer Publishing Company, Springer (2010)