Multicore Construction of k-d Trees for High Dimensional Point Data

Victor Lu[1] and John C. Hart[2]

[1]HERE, a Nokia business unit
[2]University of Illinois at Urbana-Champaign

Abstract

This paper presents the first parallelization of FLANN's k-d tree for approximate nearest neighbor finding of high dimensional data. We propose a simple node-parallel strategy that acheives surprisingly scalable speedups on a range of inputs and hardware platforms. When combined with speedups from SIMD vectorization, our approach can achieve up to 91x total speedup over the existing FLANN implementation on a 32-core machine.

1. Introduction

The prevalence of digital cameras and Internet image hosting services, such as flickr, has created an explosion of online digital imagery, and with it many exciting new ways to utilize these large image databases. For example, 3-D reconstructions of the city of Rome have been built by finding and registering matching elements in the hundreds of thousands of photos on flickr tagged with the keyword "rome" [1]. Other examples use millions of photographs to perform scene completion [16], recognise panoramas in image collections [6], and infer labels on unknown images given a collection of labeled images[5].

These techniques are all built around the ability to find similar images to a given image, based on some kind of large vector representation of the image. Entire images can be represented by a low-resolution version of the image [26] or by a GIST descriptor [24]. Localized regions within an image can be represented by the concatenation of its underlying RGB values or by vectors computed using SIFT [19] or HOG feature transforms [10]. Similarity between images or their regions can then be measured by the Euclidean distance of their vectors.

Hence, finding similar images or parts of images amounts to solving the *nearest neighbor problem*: Given P a set of n k-dimensional data points in R^k, construct a data structure that helps us quickly find the *nearest neighbor* $p^* = \min_{p \in P} d(p, q)$ to any query point $q \in R^k$.

When dealing with high dimensional point data, such as image and region descriptors, existing nearest neighbor methods invariably suffer from the *curse of dimensionality,* which degrades search time to that of a brute force search. To regain algorithmic efficiency, *approximate nearest neighbor* methods find query results within a user specified error bound of the exact nearest neighbor. This is often acceptable for image searches since the descriptor vector distance is not necessarily the "perceptual distance" between two images or image regions.

The k-d tree[3] is a popular method for finding exact and approximate nearest neighbors. Its hierarchical data structure can be constructed in $O(n \log n)$ time and supports queries in $O(\log n)$ time. Once constructed, the nearest neighbor to a query point can be quickly found by examining only the data points residing in nearby leaf nodes and culling entire subtrees that are too far away. By examining a restricted number of leaf nodes, search is further accelerated, but at the risk of missing the exact nearest neighbor and being left instead with an approximate one.

While k-d trees speedup an otherwise brute force search, its construction can still represent a significant bottleneck in a variety of applications. Several recent methods in example-based inpainting [9], super resolution upsampling [15], non-local mean denoising [7], and object detection [20] must first construct a k-d tree for each received image in order to facilitate subsequent nearest neighbor queries into the image. The interactivity of these methods thus depends very much on how quickly the k-d tree can be constructed.

The parallelism found in modern multicore CPUs offers the hope of accelerating k-d tree construction, but is not yet realized in any existing high dimensional k-d tree implementations, such as the widely used FLANN (Fast Library for Approximate Nearest Neighbors) [22] and ANN [21]. The latest version 1.8.0 of FLANN parallelizes across separate queries and contains a GPU k-d tree builder specifically for 3-d points, but construction of high dimensional k-d trees remains single threaded.

We present here the first parallelization of FLANN's high dimensional k-d tree builder. This paper is structured as follows. Section 2 reviews previous methods for parallelizing k-d tree construction. Section 3 describes our node-parallel strategy and its implementation. Section 4 demonstrates the scalability of our approach. Section 5 illustrates the importance of our work in a concrete real world application: logo detection.

2. Related Work

Many methods exist for finding nearest neighbors in high dimensional space, some more useful than others in specific situations. For example, hashing approaches, such as locality sensitive hashing (LSH) [11], have been investigated for their theoretical and qualitative benefits though they can underperform compared to alternatives in practical situations [23]. Vantage point (VP) tree [29] methods have been shown to achieve favorable search efficiency on image patches [18], but may take longer to build than k-d trees: when partitioning, VP trees must compute full vector distances to a chosen *vantage point*, whereas k-d trees split on an axis aligned plane which requires examining only a single vector component. A brute force search using the GPU can find exact nearest neighbors more quickly than a k-d tree [14], but cannot benefit from the further speedups enabled by approximate methods. Special purpose methods such as PatchMatch [2] outperform alternatives on their special cases (for PatchMatch that of finding similar image regions). Our work does not claim to be the "size that fits all," but instead we accelerate the situations where k-d trees are most useful.

Deciding on the proper nearest neighbor method for a given task may require much trial and error. The FLANN library implements a variety of these methods, while providing a mechanism for their automatic selection [23]. Our parallel k-d tree builder can be used as a drop in replacement for FLANN's k-d tree builder, once again benefitting the situations where k-d trees are most useful.

Much of the previous work on parallel construction of k-d trees have focused on low dimensional (3-d) versions, and focus their parallel performance on the computation of a surface area heuristic (SAH) over all elements to find the appropriate position of each splitting plane. For example, GPU methods for computing SAH k-d trees for accelerating ray tracing [30, 8, 28] construct the top levels of the tree in a breadth-first manner that streams through all elements at each level to compute the best splitting plane positions. Such an approach would not work well for FLANN-style computation of approximate nearest neighbors, which uses a small (e.g. 100-element) subset of points to discover the dimensions of greatest variance.

3. Method

Briefly, FLANN's recursive k-d tree construction algorithm proceeds as follows. On each recursive step, the algorithm picks one of the five dimensions with highest variance, estimated using a random subset (e.g. 100) of the node's data points, and splits its data along this dimension at its mean value, estimated on the same 100 element subset. Random subset selection is achieved by randomizing the list of vectors just once at the start of build and picking the first 100 at each recursive step. A node is made a leaf if it contains exactly one point.

We parallelize computations *across* nodes by mapping nodes to parallel *tasks* and *within* nodes by vectorizing its mean and variance estimation steps. Parallel tasks are spawned dynamically as new child nodes are created, while a task scheduler (here TBB [17]) takes care of mapping their executions onto physical cores. Section 3.1 describes the details of implementing this strategy.

Standard k-d tree builders such as in FLANN expect an explicit listing of its input vectors. When feature vectors are defined on overlapping windows in an image (*e.g.* 32×32 patches), explicit listings become especially memory *inefficient*, as each pixel value is relisted each time it is overlapped by a window. For example, a 1024×768 RGB image takes just 2.25MB, whereas an explicitly listing of its 32×32 patches requires up to 2.09GB! Section 3.2 describes modifications for avoiding this explicit listing, thus achieving orders of magnitude savings in memory.

3.1. Parallelization and Vectorization

Our implementation leverages two recent advances in programming tools for utilizing multicore parallelism:

Support for nested task parallelism in the form of libraries and language extensions such as TBB, Cilk Plus, OpenMP and WOOL allow programs to dynamically spawn tasks and tasks to spawn additional tasks, while a runtime scheduler, such as[13] [4], takes care of mapping tasks to physical processors. This style of parallel programming maps naturally to node parallel k-d tree construction, where tasks encapsulate the processing of a node and tasks are spawned when recursing on children nodes.

Auto vectorization capabilities of modern compilers coupled with preprocessor directives and the `restrict` keyword provide an almost effortless way in many cases for utilizing the wide vector units (now 8-wide) in recent processors. In our implementation, we vectorized the mean and variance computation during tree build and the distance computations during traversal. Specifically, with the Intel C++ Compiler (`icc`), we use the `restrict` keyword to assure the compiler that source and target arrays do not overlap, we added `#pragma simd`'s before for-loops and used the `-vec-report2` compiler option to check whether vectorization took place. Vectorization speedups is slightly sublinear due to overheads such as moving single byte `chars` into 4 byte vector register slots.

We avoid racing on a global random number generator (RNG) state and suffering the penalties of false sharing, by using a reentrant RNG. We explicitly pass a RNG state into each node task, and pass the updated RNG state to the left child task and an arbitrarily offseted RNG state to the right child task. In practice, search performance does not degrade from this pseudo-random hack.

During parallel tree build, all threads will be simultaneously making requests to allocate new nodes. To handle this in a scalable fashion, we use TBB's scalable allocator.

Computing mean and variance requires scratch space with size proportional to k. To avoid dynamically allocating this space for each task, we maintain preallocated per thread scratch space using TBB's `enumerable thread specific` template.

3.2. Memory Efficient Indexing of Image Patches

The problem at hand is stated in the following generalized setting: Given R a raster grid of length d subvectors $v_{i,j} \in R^d$ (Eq. 1), we define the vector $\mathbf{v} \in R^{M\times N\times d}$, at each $M \times N$ window on the raster grid, as the concatenation of the $v_{i,j}$ subvectors covered by the window (Eq. 2). There may be multiple R's of different rectangular shapes, but all must have the same subvector length d. The goal is then to constuct a k-d tree on the set of all such $\mathbf{v}$'s without having to explicit list them but by instead operating directly on the raster grids.

$$R = \begin{bmatrix} & \vdots & & \vdots & \\ \cdots & v_{i,j} & \cdots & v_{i,j+N-1} & \cdots \\ & \vdots & \ddots & \vdots & \\ \cdots & v_{i+M-1,j} & \cdots & v_{i+M-1,j+N-1} & \cdots \\ & \vdots & & \vdots & \end{bmatrix} \quad (1)$$

$$\mathbf{v} = [v_{i,j}, \cdots, v_{i+M-1,j}, \cdots, v_{i,j+N-1}, \cdots, v_{i+M-1,j+N-1}] \quad (2)$$

To make concrete, for 32 × 32 RGB patches, we have $M = N = 32$ and $d = 3$. When considering the Felzenszwalb variant [12] of the HOG feature vector, we have $M = N = 8$ and $d = 31$. In both cases, the plurality of raster grids may correspond to different images or separate levels in a pyramid.

We assume in memory the raster grids are laid out in a single array `pyr` as the concatenation of the raster grids, each of which is itself a concatenation of its subvectors in column major order.

In standard builders, each vector is represented by an `offset` into the array of vectors and its `i`-th component is indexed by `offset + i`. When reordering a list of vectors, such as during partitioning or during the initial randomizing of list ordering, the array of `offset`'s is rearranged to avoiding the massive data movement of directly rearranging the array of vectors.

In our modified builder, in addition to an `offset` into `pyr` specifying the start of the top left subvector of a window, we also record for each vector a `stride`, which is the number of array elements in a column of subvector in the level that the vector is in. The `index` of the `i`-th component of a vector represented by `offset` and `stride` is then computed in C/C++ as (see Figure 1):

$$\begin{aligned} \mathtt{index} &= \mathtt{offset} \\ &+ \mathtt{i/d/M} * \mathtt{stride} \\ &+ \mathtt{i/d\%M} * \mathtt{d} \\ &+ \mathtt{i\%d} \end{aligned} \quad (3)$$

In practice, one never has to evaluate the full expression each time a vector component is accessed. During partitioning, when a set of vectors is split along the `i`-th dimension, a large portion of the computation in Eq. 3 is constant across iterations and can thus be moved outside the loop (Code 1). When computing the mean and variance of a set of vectors, indexing becomes even simpler, since most components in the same vector are in fact contiguous in `pyr` (Code 2). We also observed that vectorizing the inner loop of Code 2 is profitable since the loop usually iterates over a sufficiently large number of contiguous elements in `pyr` (248 for HOGs, 96 for 32 × 32 RGB patches).

It is sometimes useful to consider the set of all unit normalized vectors (*i.e.* $\frac{\mathbf{v}}{||\mathbf{v}||}$), as in [20]. Since each subvector is shared by multiple vectors, the unit length normalization cannot be pre-applied to the subvectors beforehand. Instead, we can precompute and store per vector "normalization constants" in a separate array and index it with `offset / d` each time a component is accessed and normalize it using the retrieved constant.

4. Results

We evaluated our parallel k-d tree builder by characterizing its performance on a range of inputs and hardware platforms.

Test inputs. We considered 128-d *SIFT* keypoint descriptors [19], 384-d *GIST* image descriptors [24], 1024-d 32 × 32 tiny images, and 4096-d 64 × 64 image patches. For SIFT, we used the first 0.5M, 1M and 5M SIFT vectors from `cd` in Stewenius *et al.*'s dataset [25]. For GIST and tiny images, we used the first 0.5M, 1M and 5M GIST vectors and tiny images from the Tiny Images Dataset[26]. For image patches, we randomly selected two subsets of size 0.1M and 1M from Winder *et al.*'s dataset [27].

Hardware platforms. Experiments were performed on a `desktop` machine representative of a consumer level

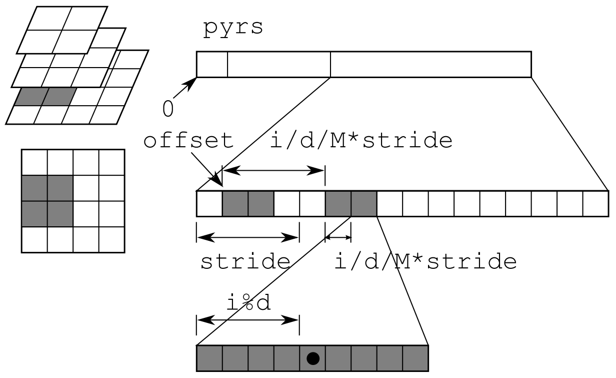

Figure 1. Layout of HOG pyramid in memory. A vector defined on a 2×2 cell window and its array elements in memory are shaded in gray.

```
int c1 = i / d / M;
int c2 = i / d % M * d + i % d;
for (int j = 0; j < n; j++) {
   int idx = offsets[j] +
      c1 * strides[j] + c2;
   if (pyr[idx] < split_val) {
      ... // sort left
   } else {
      ... // sort right
   }
}
```

Code 1. Iterating over i-th components of a set of `n` vectors specified by arrays `offsets` and `strides`

```
int idx = offset;
int width = M * d;
for (int j = 0; j < N; j++) {
   // following loop can be easily vectorized
   for (int l = 0; l < width; l++) {
     ... // work on pyr[idx + l]
   }
   idx += stride;
}
```

Code 2. Iterating over components of single vector specified by `offset` and `stride`

computer and a high end `server` machine (Table 1). All programs were compiled using `icc` version 12.1.0, with options `-O3` and `-xSSE4.2` on `server` and `-xAVX` on `desktop`.

Figure 2 compares the single threaded running time ($P = 1$) of our parallel builder against FLANN version 1.7.1's k-d tree builder. FLANN compiled "fresh out of the box" was not auto-vectorized but was easily modified (Section 3.1) to allow for the compiler to do so. Figure 2 shows the huge speedups achievable by ensuring the compiler indeed vectorizes. Unsurpisingly, once vectorized, FLANN's k-d tree builder runs at virtually the same speed as our parallel builder at $P = 1$. This comparison verifies that our single threaded running time, relative to which subsequent parallel speedups shall be computed, is indeed competitive.

Name	Machine description
`desktop`	Intel Core i5-3550 @ 3.30GHz (4 cores, 8 vector lanes) 16 GB RAM 64-bit Fedora Linux 16, kernel 3.2.9-2
`server`	Intel Xeon L7555 @ 1.87 GHz (4×8 cores, 4 vector lanes, 24 MB L3) 64 GB RAM 64-bit Sci. Linux 6.2, kernel 2.6.32-220

Table 1. Machines used in this work

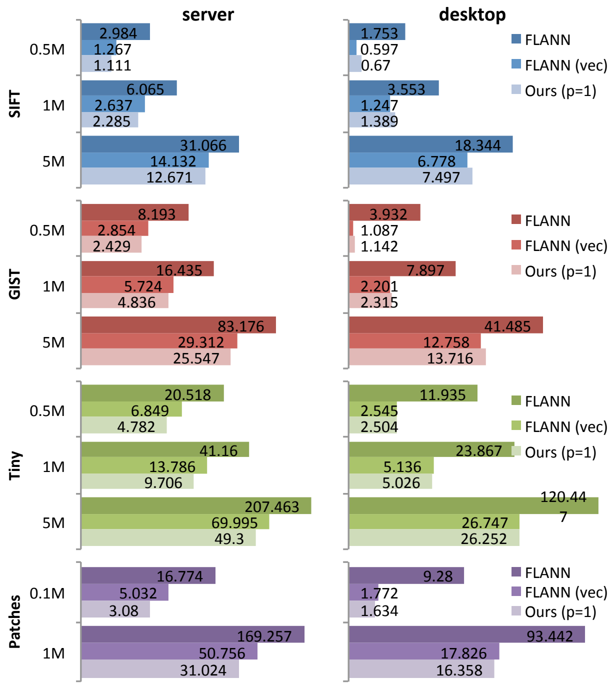

Figure 2. Comparison of serial k-d tree build times in seconds achieved by FLANN version 1.7.1 compiled "fresh out of the box," a FLANN modified to ensure auto vectorization by the compiler, and our parallel builder with number of threads set at $P = 1$.

Figure 3 reports parallel speedups relative to "Ours($P = 1$)" in Figure 2. As shown, our parallel k-d tree builder achieves scalable speedup and tremendous time savings across all chosen test inputs and hardware platforms. Compared to the non-vectorized "fresh out of the box" FLANN, our parallel k-d tree builder is up to 91.5x faster (Figure 4).

Not shown in figure 3, is that for smaller input sizes, we actually experience a slight parallel *slow down*. This is probably due to excessive stealing and limited parallelism in small inputs. But in most use cases this is okay since small inputs already build in less than a second.

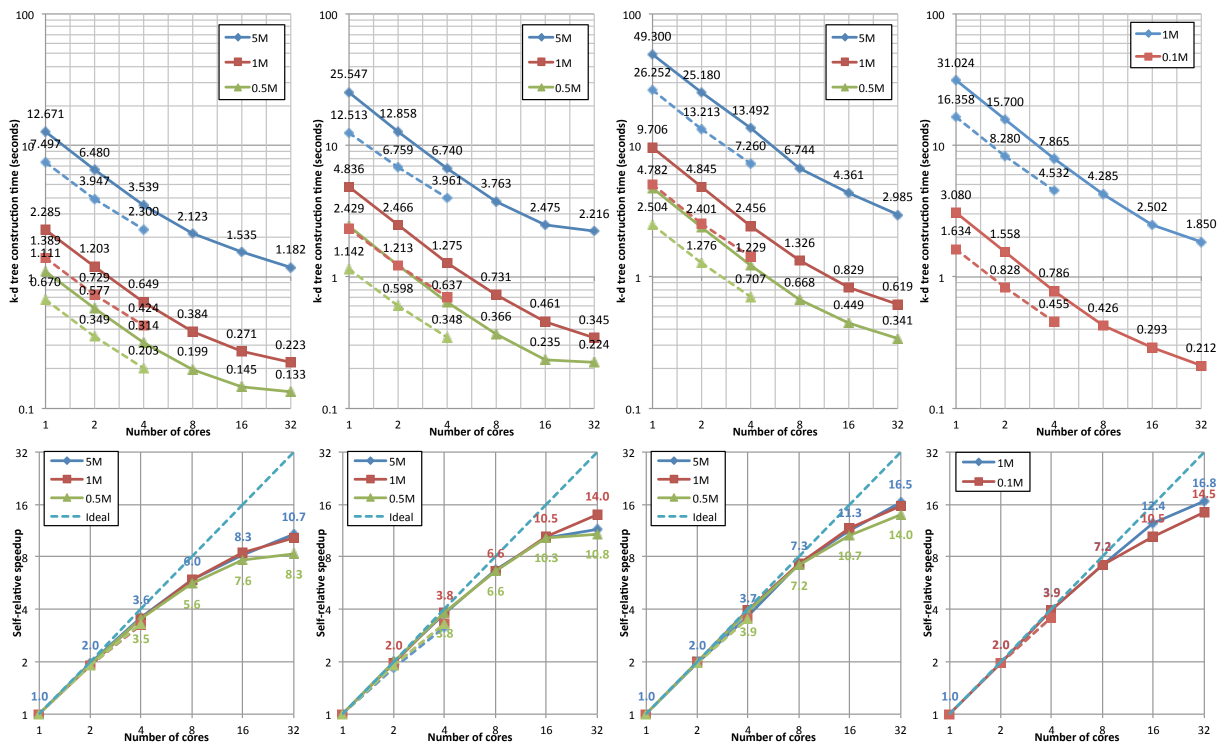

Figure 3. Absolute k-d tree build time in (top row) and self relative speedup (bottom row) for various input sizes, point dimensions and machine configurations. From left to right, point data type are SIFT feature descriptors (128-d, `uchar`), GIST image descriptor (384-d, `float`), 32×32 tiny images (1024-d, `uchar`), and 64×64 image patches (4096-d, `uchar`). Data points corresponding to solid lines were collected on `desktop` while dashed lines on `linux-server` (see Table 1). Speedup is relative to "Ours" in Figure 2

5. Application: Logo Detection

We examine the benefits of using our parallel k-d tree builder in a larger application by applying it to the logo detector described in [20], which works as follows. First, a set of *part vectors* are trained, each corresponding to a specific part of a specific logo class. Given a novel image, the image is then searched for patches whose HOG vector is sufficiently close in Euclidean Distance to any of the part vectors. This search can then be performed using either a k-d tree or any other nearest neighbor method.

We used a 4×10 core Intel Xeon E7-4860 machine running at 2.27 GHz to measure detection time over a range of core counts. We reimplemented the logo detector in [20] entirely in C++ and compiled using gcc 4.4.7 with option `-O2`. Both brute force and k-d tree based logo detection are parallelized across part vectors and distance computations vectorized using SSE intrinsics. The unit length normalization required in [20] was implemented as described at the end of Section 3.2. Following [20], we train a set of 512 part vectors. Detection was performed on a 1024×768 image from the FlickrLogos32 dataset.

Figure 5 shows detection time with and without a parallelized k-d tree builder. As core count increases, the time to build a k-d tree serially quickly dominates the overall detection running time, thus limiting further parallel speedups. And as the easily parallelized brute force detection continues to scale linearly, the k-d tree detectors advantage over a brute force detection quickly diminishes and is in fact overtaken at 32 cores. Thus, as the number of cores increase, a parallelized k-d tree construction is crucial for helping k-d tree methods stay competitive with a massively parallelizable brute force method.

References

[1] S. Agarwal, N. Snavely, I. Simon, S. Seitz, and R. Szeliski. Building Rome in a day. In *ICCV*, 2009.

[2] C. Barnes, E. Shechtman, D. B. Goldman, and A. Finkelstein. The generalized PatchMatch correspondence algorithm. In *ECCV*, 2010.

[3] J. L. Bentley. Multidimensional binary search trees used for associative searching. *C. ACM*, 18(9):509–517, Sept. 1975.

[4] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46(5):720–748, Sept. 1999.

[5] O. Boiman, E. Shechtman, and M. Irani. In defense of Nearest-Neighbor based image classification. In *CVPR*, 2008.

[6] M. Brown and D. Lowe. Recognising panoramas. In *ICCV*, 2003.

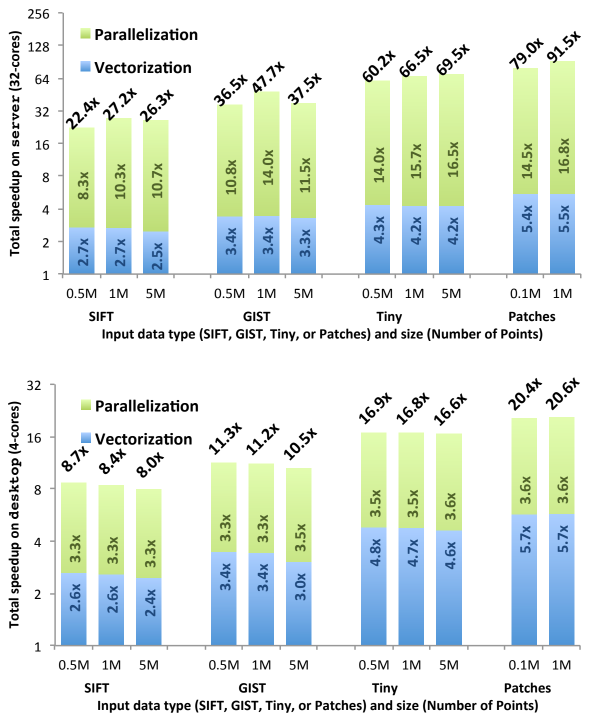

Figure 4. Total speedup of our node parallel k-d tree builder after parallelization and vectorization. Speedups here computed relative to "FLANN" in Figure 2

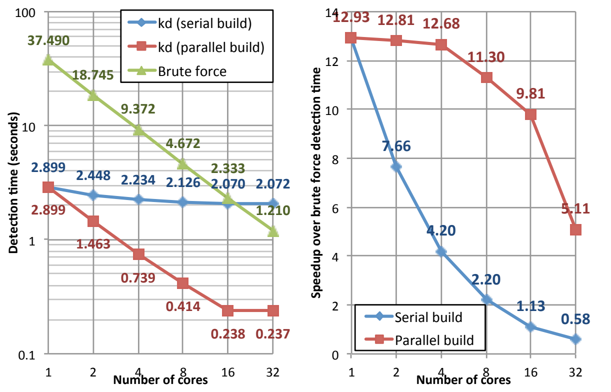

Figure 5. Scalability of overall detection time (left) and speedup over brute force detection (right), with and without parallel k-d tree build. In both plots, horizontal axis is the the number of cores.

[7] A. Buades, B. Coll, and J.-M. Morel. A non-local algorithm for image denoising. In *CVPR*, 2005.

[8] B. Choi, R. Komuravelli, V. Lu, H. Sung, R. L. Bocchino, S. V. Adve, and J. C. Hart. Parallel SAH k-D tree construction. In *HPG*, 2010.

[9] A. Criminisi, P. Perez, and K. Toyama. Region filling and object removal by exemplar-based image inpainting. *Transactions on Image Processing*, 2004.

[10] N. Dalal and B. Triggs. Histograms of oriented gradients for human detection. In *CVPR*, 2005.

[11] M. Datar, N. Immorlica, P. Indyk, and V. S. Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. In *SCG*, 2004.

[12] P. F. Felzenszwalb, R. B. Girshick, D. McAllester, and D. Ramanan. Object detection with discriminatively trained part based models. *PAMI*, 2010.

[13] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the Cilk-5 multithreaded language. *SIGPLAN Not.*, 33(5):212–223, May 1998.

[14] V. Garcia, E. Debreuve, F. Nielsen, and M. Barlaud. K-nearest neighbor search: Fast GPU-based implementations and application to high-dimensional feature matching. In *ICIP*, 2010.

[15] D. Glasner, S. Bagon, and M. Irani. Super-resolution from a single image. In *ICCV*, 2009.

[16] J. Hays and A. A. Efros. Scene completion using millions of photographs. In *SIGGRAPH*, 2007.

[17] Intel. Threading Building Blocks. http://www.threadingbuildingblocks.org/.

[18] N. Kumar, L. Zhang, and S. Nayar. What is a good nearest neighbors algorithm for finding similar patches in images? In *ECCV*, 2008.

[19] D. G. Lowe. Distinctive image features from scale-invariant keypoints. *IJCV*, 60(2):91–110, Nov. 2004.

[20] V. Lu, I. Endres, M. Stroila, and J. C. Hart. Accelerating Linear Classifiers with Approximate Range Queries. In *WACV*, 2014.

[21] D. M. Mount and S. Arya. ANN. A Library for Approximate Nearest Neighbor Searching. http://www.cs.umd.edu/~mount/ANN/.

[22] M. Muja and D. G. Lowe. FLANN - Fast Library for Approximate Nearest Neighbors. http://www.cs.ubc.ca/research/flann/.

[23] M. Muja and D. G. Lowe. Fast approximate nearest neighbors with automatic algorithm configuration. In *VISSAPP*, 2009.

[24] A. Oliva and A. Torralba. Modeling the shape of the scene: A holistic representation of the spatial envelope. *IJCV*, 42(3):145–175, May 2001.

[25] H. Stewenius. UK-Bench Recognition Homepage. http://vis.uky.edu/~stewe/ukbench/data/.

[26] A. Torralba, R. Fergus, and W. Freeman. 80 million tiny images: A large data set for nonparametric object and scene recognition. *PAMI*, 30(11):1958 –1970, nov. 2008.

[27] S. Winder, G. Hua, and M. Brown. Learning Local Image Descriptors Data. http://www.cs.ubc.ca/~mbrown/patchdata/patchdata.html.

[28] Z. Wu, F. Zhao, and X. Liu. SAH KD-tree construction on GPU. In *HPG*, 2011.

[29] P. N. Yianilos. Data structures and algorithms for nearest neighbor search in general metric spaces. In *SODA*, 1993.

[30] K. Zhou, Q. Hou, R. Wang, and B. Guo. Real-time KD-tree construction on graphics hardware. In *SIGGRAPH Asia*, 2008.