PNNU: Parallel Nearest-Neighbor Units for Learned Dictionaries

H.T. Kung, Bradley McDanel and Surat Teerapittayanon

Harvard University, Cambridge, MA 02138, USA
kung@harvard.edu, mcdanel@fas.harvard.edu, steerapi@seas.harvard.edu

Abstract. We present a novel parallel approach, *parallel nearest neighbor unit* (PNNU), for finding the nearest member in a learned dictionary of high-dimensional features. This is a computation fundamental to machine learning and data analytics algorithms such as sparse coding for feature extraction. PNNU achieves high performance by using three techniques: (1) PNNU employs a novel fast table look up scheme to identify a small number of atoms as candidates from which the nearest neighbor of a query data vector can be found; (2) PNNU reduces computation cost by working with candidate atoms of reduced dimensionality; and (3) PNNU performs computations in parallel over multiple cores with low inter-core communication overheads. Based on efficient computation via techniques (1) and (2), technique (3) attains further speed up via parallel processing. We have implemented PNNU on multi-core machines. We demonstrate its superior performance on three application tasks in signal processing and computer vision. For an action recognition task, PNNU achieves 41x overall performance gains on a 16-core compute server against a conventional serial implementation of nearest neighbor computation. Our PNNU software is available online as open source.

Keywords: Nearest neighbor, NNU, PNNU, data analytics, sparse coding, learned dictionary, parallel processing, multi-core programming, speedup, matching pursuit, signal processing, computer vision, KTH, CIFAR

1 Introduction

In the era of big data, the need for high-performance solutions to support data-driven modeling and prediction has never been greater. In this paper, we consider parallel solutions to the nearest neighbor (NN) problem: given a set of data points and a query point in a high-dimensional vector space, find the data point that is nearest to the query point. NN is used in many data applications. For example, NN (or its extension of finding k nearest neighbors, kNN) is used to identify best-matched patterns in a set of templates [13]. NN also serves as an inner loop in popular feature-extraction algorithms such as matching pursuit (MP) [11] and orthogonal matching pursuit (OMP) [19].

A key operation in NN is the vector dot-product computation which computes the “closeness” of two vectors under cosine similarity. Exhaustive search of data points to find the largest dot-product value with the query point can quickly become prohibitively expensive as data size and dimensionality increase.

Developing efficient NN solutions for general data sets is known to be a challenging task. There is a vast amount of literature on this topic, including k-d trees [21], locality sensitive hashing [3], and nearest-neighbor methods in machine learning and computer vision [18]. For high-dimensional data, most methods in the literature usually do not outperform exhaustive NN search [6]. This is due to the fact that, in practical applications, the high-dimensional data space is commonly only sparsely populated. In our experiments, we find that this observation often holds for even a moderate dimensionality, such as 30.

In this paper, we consider parallel computing approaches to NN for applications in machine learning and data analytics. Particularly, we consider the problem of finding the nearest neighbor in a *dictionary* of atoms (features) learned from training data. We present a novel parallel scheme, *parallel nearest neighbor unit* (PNNU), offering a high-performance NN solution to this problem. By exploiting data characteristics associated with a learned dictionary, such as the dominance of a small number of principal components, PNNU realizes its high performance with three techniques:

T1. reducing the number of required dot-product computations,

T2. reducing the dimensionality in each dot-product computation, and

T3. parallel processing with low inter-core communication overheads.

For T1, we use a fast table look up scheme to identify a small subset of dictionary atoms as *candidates*. By carrying out dot-product computations only with these candidates, the query vector can quickly find its nearest neighbor or a close approximation. Our look-up tables are based on principal component analysis (PCA). For accurate candidates identification, we apply PCA to dictionary atoms rather than the original data set from which the dictionary is learned. The construction and usage of this fast table look up scheme is novel. For T2, we apply the same PCA technique to reduce dimensionality of the candidate atoms to lower the cost of computing their dot-products with the query vector. Finally, for T3, we show that multiple cores can each work on scalar projections of dictionary atoms on their respective dimensions independently without inter-core communication until the very end of the PNNU computation. At the very end, a simple and inexpensive reduction operation among multiple cores is carried out. The parallel processing enabled by T3 results in substantial speed-up gains on the already efficient computation brought by T1 and T2. Thus, PNNU does not suffer from a common drawback in parallel processing that good speedups are obtained only on more parallelizable but less efficient computations. We have implemented PNNU with these techniques in software for multicore computers, and our code is available as open source for public research use [10]. PNNU is written in C++ and contains language bindings and examples for Python and MATLAB making it simple to integrate into existing codebases.

2 Background: Learned Dictionaries and Spare Coding

A data-driven modeling and prediction task, such as those considered in this paper, generally involves two phases. The first phase is feature extraction, where we use clustering methods such as K-means and K-SVD [1] to learn a dictionary where atoms (features) are cluster centroids. These atoms are the most occurring, representative features of the data. The second phase is classification/regression, where we compute a sparse representation, via *sparse coding*, of an input data vector in the learned dictionary, and then based on the sparse representation perform classification/regression.

Mathematically, sparse coding is an optimization problem expressed as

$$\hat{\mathbf{y}} = \arg\min_{\mathbf{y}} \|\mathbf{x} - \mathbf{D}\mathbf{y}\|_2^2 + \lambda \cdot \psi(\mathbf{y}), \tag{1}$$

where $\mathbf{x}$ is an input data vector, $\mathbf{D}$ is a learned dictionary, $\hat{\mathbf{y}}$ is an sparse representation of $\mathbf{x}$, λ is certain constant and $\psi(\mathbf{y})$ is a sparsity constraint function. The choices of $\psi(y)$ are usually either the L_0-norm $\|\mathbf{y}\|_0$ or the L_1-norm $\|\mathbf{y}\|_1$.

Algorithms for sparse coding include those such as MP and OMP which greedily perform minimization under a L_0-norm constraint, and those such as Basis Pursuit [2] and LARS [4] which perform minimization under a L_1-norm constraint.

The inner loop in these algorithms is the NN problem for a learned dictionary: for a given input vector $\mathbf{x} \in \mathbb{R}^m$, find its nearest feature (atom) $\mathbf{d}_j$ in a $m \times n$ dictionary $\mathbf{D} = \begin{bmatrix}\mathbf{d}_1\ \mathbf{d}_2\ \ldots\ \mathbf{d}_n\end{bmatrix}$. In machine learning and data analytics applications, $\mathbf{D}$ is generally overcomplete with $m \ll n$, and that m and n can be large, e. g., $m = 100$ and $n = 4000$. In these cases, sparse coding is computationally demanding. The PNNU approach of this paper aims at alleviating this computational problem.

Convolutional neural networks (CNN) and convolutional sparse coding (CSC) have become popular due to their success in many machine learning tasks [12, 9]. Interestingly, PNNU can help accelerate CSC. Convolution in CNN with Fast Fourier Transform has a complexity of $O(nm\log(m))$ as compared to $O(nm^2)$ for CSC. With PNNU, CSC's complexity cost is reduced to $O(\alpha\beta m^2)$ with a penalty to accuracy, for small α and β, which is discussed in detail in Sec. 5.

3 Parallel Nearest Neighbor Unit (PNNU)

In this section, we describe *parallel nearest neighbor unit* (PNNU) for a learned dictionary $\mathbf{D}$. The three subsections describe three techniques that make up the PNNU algorithm. The first technique T1 uses the Nearest Neighbor Unit (NNU) to reduce the number of dot-product computations. The second technique T2 reduces the cost of each dot-product computation via dimensionality reduction. The third technique T3 parallelizes NNU. These three techniques work in conjunction for high-performance nearest neighbor computation. That is, the first two techniques improves computation efficiency by reducing total cost of dot-product computations while the last technique further reduces the processing time via parallel processing.

3.1 Technique T1 (NNU): Identification of Candidates for Reducing Dot-product Computations

Technique T1 concerns a novel table look-up method for identifying a small number of candidate atoms in $\mathbf{D}$ from which the nearest neighbor of a query data vector or a close approximation can be found. We call this the *nearest neighbor unit* or NNU. As Fig. 1 depicts, the naive exhaustive search involves $O(n)$ dot-product computations while NNU's candidate approach reduces this number to $O(m)$. This saving is significant for overcomplete dictionaries with $m \ll n$. As described below, the technique is divided into two steps: offline table preparation and online candidates identification.

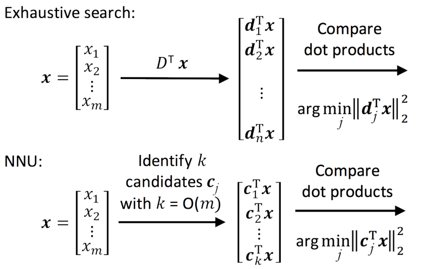

Exhaustive search:
$x = [x_1, x_2, \vdots, x_m]$
$D^T x$
$[d_1^T x, d_2^T x, \vdots, d_n^T x]$
Compare dot products
$\arg\min_j \|d_j^T x\|_2^2$
NNU:
Identify k candidates c_j with $k = O(m)$
$[c_1^T x, c_2^T x, \vdots, c_k^T x]$
Compare dot products
$\arg\min_j \|c_j^T x\|_2^2$

Fig. 1. A contrast between the naive exhaustive search and the NNU's candidates approach in the number of dot-product computations. The k candidates are a subset of D which are selected by NNU. Increasing the α and β parameters in NNU increases k, where $k \leq \alpha \cdot \beta$.

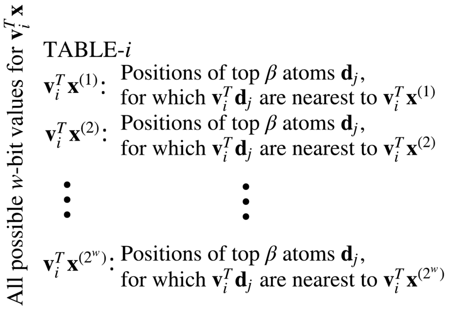

All possible w-bit values for $\mathbf{v}_i^T\mathbf{x}$
TABLE-i
$\mathbf{v}_i^T\mathbf{x}^{(1)}$: Positions of top β atoms $\mathbf{d}_j$, for which $\mathbf{v}_i^T\mathbf{d}_j$ are nearest to $\mathbf{v}_i^T\mathbf{x}^{(1)}$
$\mathbf{v}_i^T\mathbf{x}^{(2)}$: Positions of top β atoms $\mathbf{d}_j$, for which $\mathbf{v}_i^T\mathbf{d}_j$ are nearest to $\mathbf{v}_i^T\mathbf{x}^{(2)}$
$\mathbf{v}_i^T\mathbf{x}^{(2^w)}$: Positions of top β atoms $\mathbf{d}_j$, for which $\mathbf{v}_i^T\mathbf{d}_j$ are nearest to $\mathbf{v}_i^T\mathbf{x}^{(2^w)}$

Fig. 2. Offline table preparation of content for TABLE-i associated with the top principal component $\mathbf{v}_i$ of $\mathbf{D}$ for $i = 1, 2, \ldots, \alpha$. For each possible w-bit value W for $\mathbf{v}_i^T\mathbf{x}$ the dictionary positions of the β atoms for which their scalar projections on $\mathbf{v}_i$ are nearest to W are stored at table location W.

NNULookup Table Preparation We first compute principal components $\mathbf{V}$ for $\mathbf{D}$ by performing PCA [7] on $\mathbf{D}$, that is, $\mathbf{D}\mathbf{D}^T = \mathbf{V}\boldsymbol{\Sigma}\mathbf{V}^T$ for a diagonal $\boldsymbol{\Sigma}$. We then form a sub-matrix $\mathbf{V}_\alpha$ of $\mathbf{V}$ by including the top α principal components for some $\alpha = O(m)$, which together explain the majority of data variations in $\mathbf{D}$, that is, $\mathbf{V}_\alpha^T = \left[\mathbf{v}_1^T, \mathbf{v}_2^T, \ldots, \mathbf{v}_m^T\right]^T$.

Based on $\mathbf{D}$ and $\mathbf{V}_\alpha$, we prepare content for α tables using $\mathbf{V}_\alpha^T\mathbf{D}$. As depicted in Fig. 2, for TABLE-i corresponding to $\mathbf{v}_i$, $i = 1, \ldots, \alpha$, we map each possible w-bit value of $\mathbf{v}_i^T\mathbf{x}$ to the dictionary positions of the β atoms $\mathbf{d}_j$, for which $\mathbf{v}_i^T\mathbf{d}_j$ are nearest to the $\mathbf{v}_i^T\mathbf{x}$ value.

To contain the table size, we aim for a small bit width w in representing $\mathbf{v}_i^T\mathbf{x}$. Specifically, we use the 16-bit IEEE 754 half-precision floating-point data type for all of our experimental results. Empirically, we have found that for many practical applications such as object classification for tens or hundreds of classes, $w = 16$ is sufficient. In this case, the tables can be easily fit in the main memory or even the L3 cache (4-8 MB) of today's laptops. However, this is no inherent restriction on the data type stored in the table and w can be increased when higher precision is required.

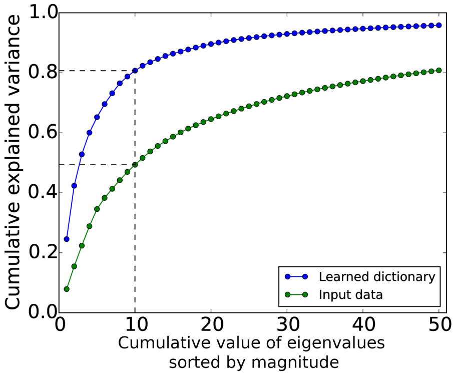


Fig. 3. Cumulative variance explained by PCA applied to the learned dictionary and raw input data for the action recognition task described in Sec. 5.1. The eigenvalues are sorted by magnitude and cumulatively summed to show total explained variance.

Table 1. Accuracy results of PNNU(α,β), for different α and β configurations, for the action recognition task described in Sec. 5.1 when applying PCA on a learned dictionary (PCA-D) versus applying PCA on the raw data (PCA-X).

	PCA-X	PCA-D
PNNU(1,1)	64.20%	82.70%
PNNU(1,5)	79.20%	87.30%
PNNU(1,10)	80.30%	89.60%
PNNU(5,1)	78.60%	87.90%
PNNU(5,5)	83.20%	92.50%
PNNU(5,10)	86.70%	90.80%
PNNU(10,1)	79.80%	86.70%
PNNU(10,5)	87.30%	90.20%
PNNU(10,10)	89.00%	90.80%

Note that our use of PCA here departs from the conventional application of PCA where principal components are computed from the raw data set, rather than the dictionary learned from this data set. Since dictionary atoms are cluster centroids learned by clustering methods such as K-means, they are denoised representation of the data. As a result, when PCA is applied to dictionary atoms, a smaller percentage of principal components can capture most of variations in the data, as compared to PCA applied to the raw data directly. This is illustrated by Fig. 3. The top 10 eigenvalues of the learned dictionary explain over 80.7% of the variance, compared to 49.3% for the raw data. Moreover, as shown in Table 1, NNU with applying PCA on a learned dictionary rather than the raw data gives results of substantially higher accuracy for an action recognition task. The use of PCA in this way, using the projection between $\mathbf{V}_\alpha$ and an input vector $\mathbf{x}$ to build a fast look up table, is novel and one of the largest contributions of this paper. (We note a similar use of PCA in [5] for a different purpose of preserving pairwise dot products of sparse code under dimensionality reduction.)

NNULookup Algorithm Given an input vector $\mathbf{x}$ we are interested in finding its nearest atom in $\mathbf{D}$. We first prepare search keys for $\mathbf{x}$, that is, $\mathbf{V}_\alpha^T\mathbf{x} = \left[\mathbf{v}_1^T\mathbf{x}, \mathbf{v}_2^T\mathbf{x}, \dots \mathbf{v}_m^T\mathbf{x}\right]^T$. Next, for $i = 1, 2, \dots, \alpha$, we use a w-bit representation of $\mathbf{v}_i^T\mathbf{x}$ as a key into TABLE-i, as depicted in Fig. 4. Note that these α table look ups can be done independently in parallel, enabling straightforward parallelization (see Sec. 3.3). Finally, we identify candidates for the nearest neighbor of $\mathbf{x}$ by taking the union of the results from all α tables as illustrated in Fig. 5 for $\alpha = 3$. Note that taking a union with the “OR” operator is amenable to efficient hardware and software implementations.

For a given α and β, our table-lookup method will yield at most $\alpha\beta$ candidates. Increasing α and β will raise the probability that identified candidate atoms will include the nearest neighbor. In Sec. 4 we show that this probability approaches 1 as α and β increase. Since tables can be accessed in parallel (see Sec. 3.3 for PNNU), increasing α does not incur additional look up time beyond the final low-cost reduction step. Additionally, since each look up produces β neighbors at the same time from each table, increasing β does not incur additional look up time beyond the cost of outputting β values for the union operation of Fig. 5.

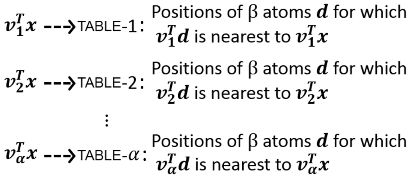
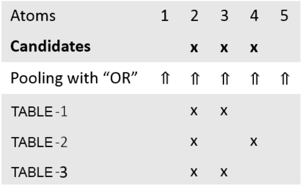

$v_1^T x$ ---> TABLE-1: Positions of β atoms d for which $v_1^T d$ is nearest to $v_1^T x$
$v_2^T x$ ---> TABLE-2: Positions of β atoms d for which $v_2^T d$ is nearest to $v_2^T x$
⋮
$v_\alpha^T x$ ---> TABLE-α: Positions of β atoms d for which $v_\alpha^T d$ is nearest to $v_\alpha^T x$

Fig. 4. Online retrieval of content from tables.



Atoms	1	2	3	4	5
Candidates		**x**	**x**	**x**	
Pooling with "OR"	⇑	⇑	⇑	⇑	⇑
TABLE-1		x	x		
TABLE-2		x		x	
TABLE-3		x	x		

Fig. 5. The union operation: pooling results from 3 tables with the "OR" operator.

3.2 Technique T2: Dimension Reduction for Minimizing the Cost of Each Dot-product Computation

By technique T1, we can identify a set of candidate atoms that have a high likelihood of containing the nearest neighbor of an input vector $\mathbf{x}$. Among these candidate atoms, we will find the closest one to $\mathbf{x}$. The straightforward approach is to compute the dot product between $\mathbf{x}$ and each candidate atom. In this subsection, we describe technique T2 based on dimension reduction using the same PCA on $\mathbf{D}$ as in technique T1, now for the purpose of lowering the cost of each dot-product computation. For example, suppose that the original atoms are of dimensionality 500, and after PCA we keep only their scalar projections onto the top 10 principal components. Then a dot-product computation would now incur only 10 multiplications and 9 additions, rather than the original 500 multiplications and 499 additions. Note that it is also possible to apply PCA on raw data $\mathbf{X}$, but applying PCA on $\mathbf{D}$ is more natural to our approach, and produces superior results on application accuracy as we demonstrate in Sec. 5.

Since PCA dimensionality reduction is a lossy operation, it is inevitable that dot-products over reduced-dimension vectors will lower the accuracy of the application result. In practice, we keep the top principal components whose eigenvalues can contribute to over 80% of the total for all eigenvalues. In this case, as results in Sec. 5 demonstrate, the impact on accuracy loss is expected to be acceptable for typical applications we are interested in.

Note that in the preceding subsection, we use PCA to identify candidates. In this subsection, we use the same PCA to reduce dimensionality. These are two different usages of PCA. The former usage is novel in its role of supporting fast table look up for NNU, while the latter usage is conventional.

3.3 Technique T3: Parallel Processing with Low Inter-core Communication Overheads

This subsection describes the third technique making up PNNU. The NNU algorithm of technique T1 leads naturally to parallel processing. We can perform table-lookup operations for α dimensions in parallel on a multi-core machine. That is, for $i = 1, 2, \ldots, \alpha$, core i performs the following operations for an input data vector $\mathbf{x}$: (1) compute $\mathbf{v}_i^T\mathbf{x}$, (2) look up β values from table i based on $\mathbf{v}_i^T\mathbf{x}$, (3) compute β dot-product computations or reduced-dimension dot-product computations between the candidate dictionary atoms and $\mathbf{x}$, and (4) output the candidate atom which yields the maximum dot-product value on the i^{th} dimension.

The final reduction step is performed across all cores (dimensions) to find the dictionary atom which yields the maximum dot-product value. We note that the table look-ups from multiple tables are carried out in parallel, so are the corresponding dot-product computations or reduced-dimension dot-product computations. We also note that this parallel scheme incurs little to no inter-core communication overhead, except at the final reduction step where α candidate atoms are reduced to a single atom that has the maximum dot-product value with $\mathbf{x}$. In Sec. 5, experiments show that this low communication overhead leads to large parallel speedups.

4 Probabilistic Analysis of PNNU

In this section, we analyze the probability P that for a given query vector $\mathbf{x}$, the PNNU algorithm finds the nearest neighbor $\mathbf{d}$ in a dictionary $\mathbf{D}$. Let $\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_\alpha$ be the α top principal components of $\mathbf{D}$. We show that the probability P approaches 1 as α and β increase, satisfying a certain condition.

For a given $\epsilon \in (0, 1)$, let β_i be the least number of the nearest neighbors of $\mathbf{v}_i^T\mathbf{x}$ such that the probability that $\mathbf{v}_i^T\mathbf{d}$ is not any of the β_i nearest neighbors of $\mathbf{v}_i^T\mathbf{x}$ is less than or equal to ϵ. Given an α, for $i = 1, \ldots, \alpha$, let Y_i be an event that $\mathbf{v}_i^T\mathbf{d}$ is not any of the β nearest neighbors of $\mathbf{v}_i^T\mathbf{x}$, where $\beta = \max_{1\le i\le\alpha} \beta_i$. Therefore, $\Pr(Y_i) \le \epsilon$. Assume that Y_i are mutually independent. Then, we have $P = 1 - \Pr\left(\bigcap_{i=1}^{\alpha} Y_i\right) = 1 - \prod_{i=1}^{\alpha} \Pr(Y_i) \ge 1 - \epsilon^\alpha$. Thus, as α increases, and also β increases accordingly, ϵ^α decreases toward 0 and P approaches 1.

Consider using the parallel processing T3 technique of PNNU. Since we have low inter-core communication overheads, increasing α (the number of cores) does not impact the processing time significantly. Therefore, for a particular application, we can pick an ϵ and keep increasing α, and also β accordingly, until the probability $\Pr(A)$ is high enough.

To simplify the analysis, we have assumed that Y_i are mutually independent. Experimentally, we have found that this assumption holds well. For all experiments reported in this paper, $\Pr\left(\bigcap_{i=1}^{\alpha} Y_i\right)$ and $\prod_{i=1}^{\alpha} \Pr(Y_i)$ are reasonably close empirically. For example, in one experiment, these two numbers are 0.72 and 0.71 and in another experiment, they are 0.47 and 0.45.

5 Experimental Results of PNNU on Three Applications

In this section, we provide empirical performance results for PNNU on three applications: action recognition, object classification and image denoising. All three applications require the nearest neighbor computation. We replace the nearest neighbor computation with PNNU(α,β), where α, β denote different parameter configurations of PNNU. All experiments are run on a compute server using two Intel Xeon E5-2680 CPUs, with a total of 16 physical cores.

Algorithms to Compare. We consider both PNNU and PNNU without technique T2 (PNNU-no-T2). The latter involves more dot-product computations, but yields better application accuracy. We compare PNNU and PNNU-no-T2 (both serial and parallel implementations) with three other algorithms:

1. Straightforward method (S). This is the straightforward exhaustive search algorithm to find the nearest neighbor in terms of the cosine distance. If the input data vector is $\mathbf{x}$ and candidate atoms are the columns of $\mathbf{D}$, we compute $\mathbf{D}^T\mathbf{x}$. We call its serial implementation S. This method is the only algorithm in the comparison that is guaranteed to find the nearest neighbor of $\mathbf{x}$ in $\mathbf{D}$.
2. PCA-dimensional-reduction-on-dictionary (PCAonD(α)). For dimensionality reduction, we first perform PCA on $\mathbf{D}$ to get its top α principal components $\mathbf{V}_D^T$, that is, $\mathbf{D}\mathbf{D}^T = \mathbf{V}_D \boldsymbol{\Sigma} \mathbf{V}_{\mathbf{D}}{}^T$ for some diagonal $\boldsymbol{\Sigma}$. Then during computation, instead of computing $\mathbf{D}^T\mathbf{x}$, we compute dot products $(\mathbf{V}_D^T\mathbf{D})^T(\mathbf{V}_D^T\mathbf{x})$ of reduced dimensionality. Note the parameter α specifies dimensionality of dot-product computations after PCA dimension reduction. In these experiments, we use $\alpha = 10$.
3. PCA-dimensional-reduction-on-data (PCAonX(α)). This is the same as the previous algorithm, but instead we compute PCA on the input data $\mathbf{X}$. Let $\mathbf{V}_X^T$ contain the top α principal components. We compute $(\mathbf{V}_X^T\mathbf{D})^T(\mathbf{V}_X^T\mathbf{x})$. Note the parameter α specifies the dimensionality of dot-product computations after PCA dimension reduction. We use $\alpha = 10$.

Performance Measures. We compare algorithms in terms of the following performance related measures, where an algorithm Y can be S, PCAonD, PCAonX, PNNU or PNNU-no-T2:

N: The number of arithmetic operations per query vector. This is the number of addition and multiplication operations each algorithm performs for a single query vector. For S, a dot-product between a query vector $\mathbf{x} \in \mathbb{R}^m$ and a dictionary $\mathbf{D} \in \mathbb{R}^{m \times n}$ incurs $n(2m-1)$ arithmetic operations (nm for the multiplication and $n(m-1)$ for the addition). For PCAonD(α) and PCAonX(α), it is $n(2\alpha-1)$. For PNNU(α,β), it is bounded above by $\alpha\beta(2\alpha-1)$. For PNNU-no-T2(α,β), it is bounded above by $\alpha\beta(2m-1)$.

G: Efficiency gain. For an algorithm Y, its efficiency gain is the number of arithmetic operations of the straightforward method (N_{S}) over that of the algorithm Y (N_{Y}): $^{N_{\mathrm{S}}}/_{N_{\mathrm{Y}}}$.

$\mathbf{T_s}$: Serial processing wall clock time in seconds. This is the time it takes for the serial implementation of the algorithm to run.

$\mathbf{U_s}$: Serial speedup of an algorithm Y over the serial straightforward method. It is the wall clock serial execution time of the straightforward method over that of algorithm Y: $^{T_{s_{\mathrm{S}}}}/_{T_{s_{\mathrm{Y}}}}$. This is a run-time realization of the theoretical efficiency gain G.

$\mathbf{T_p}$: Parallel processing wall clock time in seconds. This is the time it takes for the parallel implementation of the algorithm to run.

$\mathbf{U_p}$: Parallel-over-serial speedup. This is the parallel scaling performance of the algorithm. It is $^{T_s}/_{T_p}$.

$\mathbf{U_t}$: Total performance gain of an algorithm Y over the serial implementation of the straightforward method: $^{T_{s_{\mathrm{S}}}}/_{T_{p_{\mathrm{Y}}}} = U_s \times U_p$.

Q: Quality metric which is defined per application. For action recognition and object classification, we report the recognition/classification accuracy on the test set, i.e., the percentage of times the algorithm predicts the correct class labels. For image denoising, we report the peak signal-to-noise ratio (PSNR).

Performance Highlights. For each application, we will highlight the following points in our performance analysis:

1. A comparison of how PNNU performs compared to the simple PCA methods (PCAonX and PCAonD).
2. The algorithm and setting with the best quality metric (Q) compared to the straightforward method.
3. The algorithm and setting with the best total performance gain (U_t).

In the following we will explicitly mention these highlighted points for each application, and mark them with bold faces in the tables which report experiment results.

5.1 Application A1: Action Recognition

For the action recognition task we use a standard benchmark dataset, the KTH dataset [17], which is a video dataset consisting of 25 subjects where in each video a single subject is performing one of six actions (walking, jogging, running, boxing, hand waving and hand clapping). The dataset is split on subjects into a training and testing set. Features are extracted from each video using the same method as described in [20]. Features from each video consist of a variable number of columns, where each column is a 150-long feature vector. K-means is then performed on the training set to learn a dictionary of size 1000. Finally, each column from every video is then encoded with the learned dictionary using either conventional dot product or our PNNU approach. Each column is given a single atom assignment, and for a given video these column assignments are aggregated using a bag-of-words model. An SVM classifier with chi-squared kernel is then trained on the bag-of-words representation in order to obtain prediction results.

Table 2. The experiment results for the KTH dataset.

Algorithm	N	G	T_s	U_s	T_p	U_p	U_t	Q
S	299,000	1	692.89	1.00	108.48	6.39	6.39	**94.20%**
PCAonX(10)	19,000	16	129.25	5.36	13.15	9.83	52.69	**77.50%**
PCAonD(10)	19,000	16	128.40	5.40	13.24	9.70	52.34	**77.50%**
PNNU-no-T2(1,1)	299	1,000	7.39	93.75	9.80	0.75	70.71	82.70%
PNNU-no-T2(1,10)	2,990	100	28.80	24.06	20.41	1.41	33.94	89.60%
PNNU-no-T2(5,1)	1,495	200	22.91	30.24	12.44	1.84	55.71	87.90%
PNNU-no-T2(5,5)	7,475	**40**	75.30	**9.20**	16.73	**4.50**	**41.41**	**92.50%**
PNNU-no-T2(5,10)	14,950	20	140.23	4.94	22.90	6.12	30.26	90.80%
PNNU-no-T2(10,1)	2,990	100	19.24	36.01	10.44	1.84	66.35	86.70%
PNNU-no-T2(10,10)	29,900	10	260.30	2.66	24.99	10.42	27.73	90.80%
PNNU(1,1)	1	299,000	6.36	108.96	5.73	1.11	**120.95**	**82.70%**
PNNU(1,10)	10	29,900	15.75	43.99	6.91	2.28	100.29	78.00%
PNNU(5,1)	45	6,644	15.56	44.54	8.05	1.93	86.08	85.50%
PNNU(5,5)	225	1,329	44.64	15.52	8.16	5.47	84.95	83.80%
PNNU(5,10)	450	664	80.87	8.57	8.98	9.01	77.16	84.40%
PNNU(10,1)	190	1,574	27.33	25.35	9.53	2.87	72.69	83.80%
PNNU(10,10)	1,900	157	162.95	4.25	10.69	15.24	64.79	87.30%

Table 2 shows the experiment results for the KTH dataset. The straightforward method, denoted as S, achieves the highest accuracy (Q) of 94.20%. PCAonX(10) and PCAonD(10) both achieve accuracy (Q) of 77.50%, which is in general substantially lower than PNNU configurations. Additionally, many PNNU configurations are strictly better in terms of both quality (Q) and total performance gain (U_t).

PNNU-no-T2(5,5) has an accuracy of 92.50%, the closest to that of S, with an efficiency gain (G) of 40. This translates into a serial speedup (U_s) of 9.20x (the difference between G and U_s is due to both run-time overhead and G only counting arithmetic operations). The parallel speedup (U_p) is 4.50x, for a total performance gain (U_t) of 41.41x over the serial implementation of S.

Notably, PNNU(1,1) achieves the highest total performance gain (U_t) of 120.95x with accuracy (Q) of 82.70%. This trade-off is good for applications that can accept a small reduction in quality in order to significantly reduce running time. As expected, PNNU-no-T2 achieves higher accuracy than PNNU at the expense of increased running time. We note this trend in other applications as well.

Though in general increasing α and β improves Q, it is not always the case. For instance, we observe a drop of 1.7% in Q when going from PNNU-no-T2(5,5) to PNNU-no-T2(5,10). The reason for this is explained in the following example. Suppose given an input sample $\mathbf{x}$, the nearest atom to $\mathbf{x}$ is $\mathbf{d}^*$. Increasing β from 5 to 10 leads to finding the candidate atom $\mathbf{d}_{\beta=10}$ that is nearer to $\mathbf{x}$ than the candidate atom $\mathbf{d}_{\beta=5}$. Nonetheless, there is a chance that $\mathbf{d}_{\beta=10}$ is further away from $\mathbf{d}^*$ than $\mathbf{d}_{\beta=5}$. This results in the drop in Q. In general, when $\mathbf{x}$ is already close to $\mathbf{d}^*$, this phenomenon is unlikely to happen.

5.2 Matching Pursuit Algorithm with PNNU

The object classification and image denoising tasks rely on computing sparse codes. Before going into those applications, we introduce MP (Algorithm 1), the sparse coding algorithm that we use to compute sparse representations for these tasks. We modify the nearest neighbor computation section of MP to use PNNU and obtain MP-PNNU (Algorithm 2). For comparison with other algorithms, we just replace PNNU routine with other algorithms' routines of finding the nearest neighbor.

Algorithm 1. MP

```
1: Input: data vector $\mathbf{x}$, dictionary $\mathbf{D} = [\mathbf{d}_i, \dots, \mathbf{d}_n]$, and the number of iterations $L$
2: Output: sparse code $\mathbf{y}$
3: $\mathbf{r} \leftarrow \mathbf{x}$
4: for $t = 1$ to $L$ do
5:     $i \leftarrow \arg\max |\mathbf{D}^T \mathbf{r}|$
6:     $y_i \leftarrow \mathbf{d}_i^T \mathbf{r}$
7:     $\mathbf{r} \leftarrow \mathbf{r} - y_i \mathbf{d}_i$
8: end for
```

Algorithm 2. MP-PNNU

```
1:  Input: data vector $\mathbf{x}$, dictionary $\mathbf{D} = [\mathbf{d}_i, \dots, \mathbf{d}_n]$, orthonormal basis $\mathbf{V}$, the number of iterations $L$, and PNNU
2:  Output: sparse code $\mathbf{y}$
3:  $\mathbf{r} \leftarrow \mathbf{x}$
4:  for $t = 1$ to $L$ do
5:      $\mathbf{v} \leftarrow \mathbf{V}^T \mathbf{r}$
6:      $\mathbf{C} \leftarrow \mathrm{PNNU}(\mathbf{v})$
7:      $j \leftarrow \arg\max |\mathbf{C}^T \mathbf{r}|$
8:      $i \leftarrow i$ s.t. $\mathbf{d}_i = \mathbf{c}_j$
9:      $y_i \leftarrow \mathbf{d}_i^T \mathbf{r}$
10:     $\mathbf{r} \leftarrow \mathbf{r} - y_i \mathbf{d}_i$
11: end for
```

The MP algorithm finds the column $\mathbf{d}_j$ in the dictionary $\mathbf{D}$ which is best aligned with data vector $\mathbf{x}$. Then, the scalar projection y_j along this $\mathbf{d}_j$ direction is removed from $\mathbf{x}$ and the residual $\mathbf{r} = \mathbf{x} - y_j \mathbf{d}_j$ is obtained. The algorithm proceeds in each iteration by choosing the next column $\mathbf{d}_j$ that is best matched with the residual $\mathbf{r}$ until the desired number of iterations is performed. We note that for each iteration, line 5 is the most costly nearest neighbor step. As we noted previously, for a $m \times n$ dictionary $\mathbf{D}$, exhaustive search will incur a cost of $O(mn)$ and thus can become prohibitively expensive when m and n are large. The MP-PNNU algorithm can mitigate this problem. MP-PNNU has the same overall structure as MP, except that in finding the best matched column $\mathbf{d}_j$, it uses the PNNU approach as described in Sec. 3.

5.3 Application A2: Object Classification

For the image object classification task we use the CIFAR-10 image dataset [8], an image dataset of 10 object classes. We randomly select 4,000 images from the training set and evaluate on 1,000 images from the test set (we ensure that the same number of samples are selected from each class). For each image, all 6 by 6 3-color-channel (RGB) patches are extracted sliding by one pixel, and therefore, each vector is 108 dimension long. We learn a 3,000-atom dictionary using K-SVD [1], a generalization of K-means, on the training patches. For encoding, we compare the classic MP (Algorithm 1) with MP-PNNU (Algorithm 2), setting $k = 5$ (number of coefficients) for both algorithms. Finally, we perform a maximum pooling operation over each image to obtain a feature vector. A linear SVM classifier is trained on the obtained training set feature vectors and testing set accuracy results are reported.

Table 3. The experiment results for the CIFAR-10 dataset.

Algorithm	N	G	T_s	U_s	T_p	U_p	U_t	Q
S	645,000	1	3,815.89	1.00	890.37	4.29	4.29	**51.90%**
PCAonX(10)	57,000	11	1,492.36	2.56	187.27	7.97	20.38	**30.40%**
PCAonD(10)	57,000	11	1,600.88	2.38	185.38	8.64	20.58	**33.10%**
PNNU-no-T2(1,1)	215	3,000	38.2375	99.79	76.1259	0.50	50.13	33.90%
PNNU-no-T2(1,10)	2,150	300	69.9699	54.54	86.6791	0.81	44.02	41.70%
PNNU-no-T2(5,1)	1,075	600	65.3232	58.42	80.3086	0.81	47.52	40.20%
PNNU-no-T2(5,5)	5,375	120	143.894	26.52	93.466	1.54	40.83	42.30%
PNNU-no-T2(5,10)	10,750	60	241.786	15.78	113.46	2.13	33.63	45.10%
PNNU-no-T2(10,1)	2,150	300	199.547	19.12	153.835	1.30	24.81	39.40%
PNNU-no-T2(10,10)	21,500	**30**	971.899	**3.93**	115.558	**8.41**	**33.02**	**46.60%**
PNNU(1,1)	1	645,000	76.8262	49.67	68.934	1.11	55.36	33.10%
PNNU(1,10)	10	64,500	113.847	33.52	72.8819	1.56	52.36	34.10%
PNNU(5,1)	45	14,333	114.627	33.29	65.21	1.76	**58.52**	**37.30%**
PNNU(5,5)	225	2,867	227.224	16.79	85.5631	2.66	44.60	37.30%
PNNU(5,10)	450	1,433	367.583	10.38	95.8492	3.84	39.81	36.30%
PNNU(10,1)	190	3,395	165.41	23.07	121.995	1.36	31.28	35.80%
PNNU(10,10)	1,900	5	724.173	5.27	108.528	6.67	35.16	39.10%

Table 3 shows the experiment results for the CIFAR-10 dataset. The straightforward method S achieves the highest accuracy (Q) of 51.90%. (This multi-class classification task is known to be difficult, so the relatively low 51.90% achieved accuracy is expected for a simple algorithm like this [14].) PCAonX(10) and PCAonD(10) achieve accuracy of 30.40% and 33.10%, respectively. Once again, we see that many PNNU configurations are strictly better in terms of both quality (Q) and total performance gain (U_t). PNNU-no-T2(10,10) has an accuracy of 46.60%, the closest to that of S, with an efficiency gain (G) of 30. This translates into a serial speedup (U_s) of 3.93x, a parallel speedup (U_p) of 8.41x, for a total performance gain (U_t) of 33.02x over the serial implementation of S. PNNU(5,1) achieves the highest total performance gain (U_t) of 58.52x with accuracy (Q) of 37.30%.

5.4 Application A3: Image Denoising

In the previous subsections, we have shown that PNNU works well for classification problems. In this subsection, we showcase its performance at reconstruction, specifically, removing noise from an image of Lena [15]. First, a noisy version of the Lena image is generated by adding Gaussian noise with zero mean and standard deviation 0.1. This noisy image is then patched in the same manner as described in the previous subsection, using 8 by 8 grayscale patches, creating 64-dimensional vectors. These patches (roughly 250,000) are then used to learn a dictionary of 3,000 atoms using K-SVD with the number of sparse coefficients set to 5. The denoising process consists of encoding each patch with either MP or MP-PNNU. After encoding, each patch is represented as a sparse feature vector (sparse representation). To recover a denoised version of the input signal, the dot-product between the sparse vectors and learned dictionary is computed. Finally, the recovered patches are each averaged over a local area to form the denoised image. For our quality measure (Q), we report the peak signal-to-noise ratio (PSNR). A PSNR for a 8-bit per pixel image that is acceptable to human perception ranges between 20dB and 40dB [16].

Table 4 shows the experiment results for denoising the Lena image. From the table, we see that S achieves the highest PSNR (Q) of 32.34. PCAonX(10) and PCAonD(10) achieve similar PSNR of 31.18 and 31.20 respectively. In contrast with the other two applications, both algorithms perform reasonably well for this application. PNNU-no-T2(10,10) has PNSR (Q) of 32.19, the closest to that of S, with a G of 30, translating into a 12.49x speedup (U_s). Its parallel implementation (U_p) adds another 4.18x speedup, for a total performance gain (U_t) of 52.16x. Notably, PNNU(1,1) achieves the highest total performance gain (U_t) of 80.34x with PSNR (Q) of 25.71. This is good for scenarios where a rougher denoising result is acceptable for a significant gain in performance.

6 Conclusion

In this paper, we have described how nearest-neighbor (NN) is a key function for data analytics computations such as sparse coding. To enhance the performance of the NN computation, we have taken three orthogonal techniques: (T1) reduce the number of required dot-product operations; (T2) lower the cost of each dot-product computation by reducing dimensionality; and (T3) perform parallel computations over multiple cores. Noting that the gains from (T1), (T2) and

Table 4. The experiment results for denoising the Lena image.

Algorithm	N	G	T_s	U_s	T_p	U_p	U_t	Q
S	381,000	1	392.92	1.00	39.24	10.01	10.01	**32.34**
PCAonX(10)	57,000	7	48.27	8.14	13.59	3.55	28.90	**31.18**
PCAonD(10)	57,000	7	59.23	6.63	16.78	3.53	23.42	**31.20**
PNNU-no-T2(1,1)	127	3,000	5.98	65.68	5.42	1.10	72.51	25.88
PNNU-no-T2(1,10)	1,270	300	6.79	57.89	8.29	0.82	47.40	27.36
PNNU-no-T2(5,1)	635	600	11.25	34.91	8.66	1.30	45.35	29.05
PNNU-no-T2(5,5)	3,175	120	22.95	17.12	8.06	2.85	48.74	30.95
PNNU-no-T2(5,10)	6,350	60	35.85	10.96	10.47	3.42	37.53	31.58
PNNU-no-T2(10,1)	1,270	300	8.22	47.82	7.55	1.09	52.03	29.84
PNNU-no-T2(10,10)	12,700	**30**	31.46	**12.49**	7.53	**4.18**	**52.16**	**32.19**
PNNU(1,1)	1	381,000	4.55	86.33	4.89	0.93	**80.34**	**25.71**
PNNU(1,10)	10	38,100	5.64	69.61	5.28	1.07	74.41	25.80
PNNU(5,1)	45	8,467	6.77	58.07	5.42	1.25	72.45	28.71
PNNU(5,5)	225	1,693	11.14	35.28	5.49	2.03	71.60	29.87
PNNU(5,10)	450	847	14.88	26.41	5.68	2.62	69.23	30.17
PNNU(10,1)	190	2,005	6.76	58.11	5.26	1.29	74.75	29.67
PNNU(10,10)	1,900	201	25.96	15.13	5.96	4.36	65.95	31.64

(T3) complement each other, we have proposed a *parallel nearest neighbor unit* (PNNU) algorithm which uses a novel fast table look up, parallelized over multiple dimensions, to identify a relatively small number of dictionary atoms as candidates. Only these candidates are used to perform reduced-dimension dot products. PNNU allows the dot-product computations for these candidates to be carried out in parallel. As noted in Sec. 3.1, a key to the success of the PNNU approach is our application of PCA to dictionary atoms, rather than raw data vectors as in conventional PCA applications. This use of PCA to build a table lookup for the purpose of identifying the nearest candidate atom is novel.

We have validated the PNNU approach on multi-core computers with several application tasks including action recognition, image classification and image denoising. Substantial total performance gains (e.g., 41x) are achieved by software implementations of PNNU without compromising the accuracy required by the applications.

Other potential applications for PNNU are abundant. For example, large-scale data-driven deep learning can benefit from reduced dot product requirements in its computation. Mobile computing can benefit from speed and energy efficient implementation of sparse coding resulting from PNNU to allow sophisticated learning on client devices. In the future, we expect to implement PNNU as a hardware accelerator which can further speed up NN computations. In addition, we will explore integrated use of PNNU in conjunction with GPU accelerators.

Acknowledgments: This work is supported in part by gifts from the Intel Corporation and in part by the Naval Postgraduate School Agreement No. N00244-15-0050 awarded by the Naval Supply Systems Command.

References

1. Aharon, M., Elad, M., Bruckstein, A.: K-svd: An algorithm for designing overcomplete dictionaries for sparse representation. IEEE Transactions on Signal Processing 54(11), 4311–4322 (2006).
2. Chen, S.S., Donoho, D.L., Saunders, M.A.: Atomic decomposition by basis pursuit. SIAM journal on scientific computing 20(1), 33–61 (1998).
3. 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, pp. 253–262 (2004).
4. Efron, B., Hastie, T., Johnstone, I., Tibshirani, R.: Least angle regression. The Annals of statistics 32(2), 407–499 (2004).
5. Gkioulekas, I.A., Zickler, T.: Dimensionality reduction using the sparse linear model. In: Advances in Neural Information Processing Systems, pp. 271–279 (2011).
6. Indyk, P.: Nearest neighbors in high-dimensional spaces. (2004).
7. Jolliffe, I.: *Principal component analysis.* Wiley Online Library (2002).
8. Krizhevsky, A., Hinton, G.: Learning multiple layers of features from tiny images. Computer Science Department, University of Toronto, Tech. Rep (2009).
9. Krizhevsky, A., Sutskever, I., Hinton, G.E.: Imagenet classification with deep convolutional neural networks. In: Advances in neural information processing systems, pp. 1097–1105 (2012).
10. Kung, H., McDanel, B., Teerapittayanon, S. *NNU Source Repository.* Available at https://gitlab.com/steerapi/nnu.
11. Mallat, S.G., Zhang, Z.: Matching pursuits with time-frequency dictionaries. IEEE Transactions on Signal Processing 41(12), 3397–3415 (1993).
12. Mathieu, M., Henaff, M., LeCun, Y.: Fast training of convolutional networks through ffts. arXiv preprint arXiv:1312.5851 (2013).
13. Peterson, L.E.: K-nearest neighbor. Scholarpedia 4(2), 1883 (2009).
14. Rifai, S., Muller, X., Glorot, X., Mesnil, G., Bengio, Y., Vincent, P.: Learning invariant features through local space contraction. arXiv preprint arXiv:1104.4153 (2011).
15. Roberts, L.: Picture coding using pseudo-random noise. Information Theory, IRE Transactions on 8(2), 145–154 (1962).
16. Saha, S.: Image compression-from DCT to wavelets: a review. Crossroads 6(3), 12–21 (2000).
17. Schuldt, C., Laptev, I., Caputo, B.: Recognizing human actions: a local SVM approach. In: Proceedings of the 17th International Conference on Pattern Recognition, pp. 32–36 (2004).
18. Shakhnarovich, G., Indyk, P., Darrell, T.: *Nearest-neighbor methods in learning and vision: theory and practice.* (2006).
19. Tropp, J.A., Gilbert, A.C.: Signal recovery from random measurements via orthogonal matching pursuit. IEEE Transactions on Information Theory 53(12), 4655–4666 (2007).
20. Wang, H., Klaser, A., Schmid, C., Liu, C.-L.: Action recognition by dense trajectories. In: IEEE Conference on Computer Vision and Pattern Recognition, pp. 3169–3176 (2011).
21. Wess, S., Althoff, K.-D., Derwand, G.: *Using kd trees to improve the retrieval step in case-based reasoning.* Springer (1994).