Performance Optimization for the K Nearest-Neighbor Kernel on x86 Architectures

Chenhan D. Yu[*†]
chenhan@cs.utexas.edu

Jianyu Huang[*]
jianyu@cs.utexas.edu

Woody Austin[*†]
austinwn@utexas.edu

Bo Xiao[†]
bo@ices.utexas.edu

George Biros[†]
gbiros@acm.org

[*]Department of Computer Science
[†]Institute for Computational Engineering and Sciences
The University of Texas at Austin
Austin, TX 78712

ABSTRACT

Nearest neighbor search is a cornerstone problem in computational geometry, non-parametric statistics, and machine learning. For N points, exhaustive search requires $\mathcal{O}(N^2)$ work. There are many fast algorithms that reduce this complexity to $\mathcal{O}(N \log N)$ both for exact and approximate searches. The common kernel (the kNN kernel) in all these algorithms solves many small-size ($\ll N$) problems exactly using exhaustive search.

We propose an efficient implementation and performance analysis for the kNN kernel on x86 architectures. Currently, the most efficient method is to first compute all the pairwise distances using (highly optimized) matrix-matrix multiplication and then use a max heap to select the neighbors. We propose a different approach. By fusing the distance calculation with the neighbor selection, we are able to utilize memory throughput. We present an analysis of the algorithm and explain parameter selection. We perform an experimental study varying the size of the problem, the dimension of the dataset, and the number of nearest neighbors. Overall we observe significant speedups. For example, when searching for 16 neighbors in a point dataset with 1.6 million points in 64 dimensions, our kernel is over 4× faster than existing methods.

1. INTRODUCTION

Problem Definition: Given a set of N *reference* points $\mathcal{X} := \{x_j\}_{j=1}^N$ and a *query* point x in a d-dimensional space, the nearest neighbor problem aims to find the set $\mathcal{N}_x$ of the k nearest neighbors of x. That is, $\mathcal{N}_x$ is a set of points with cardinality k such that $\forall x_p \in \mathcal{N}_x$ we have $\|x-x_j\|_2 \geq \|x-x_p\|_2$, $\forall x_j \in \mathcal{X}\backslash\mathcal{N}_x$. When we want to compute the nearest neighbors of all points $x_j \in \mathcal{X}$, the problem is called the all-nearest-neighbor problem. In many applications, (e.g., image datasets, streaming datasets) there are frequent updates of $\mathcal{X}$ and computing all nearest-neighbors fast efficiently is time-critical.

Significance: The all-nearest-neighbor problem is widely used in non-parametric statistics and machine learning. It is used in cross-validation studies in supervised learning, construction of nearest-neighbor graphs for manifold learning, hierarchical clustering, kernel machines, and many other applications [9]. For very large N, the all-nearest-neighbor problem is prohibitively expensive since it requires $\mathcal{O}(N^2)$ distance evaluations. In low dimensions (say $d < 10$), regular spatial decompositions like quadtrees, octrees, or KD-trees can solve the kNN problem using $\mathcal{O}(N)$ distance evaluations [25]. But in higher dimensions it is known that tree-based algorithms end up having quadratic complexity [30]. To circumvent this problem, we must abandon the concept of exact searches and settle for approximate searches. State-of-the art methods for the kNN problem in high dimensions use randomization methods, for example, tree-based methods [6, 15, 21, 31] or hashing based methods [1, 2].

The kNN kernel: The gist of all these approximate search methods is the following. Given $\mathcal{X}$ of size N, we partition it into N/m groups of size $m \times n$ (with m query points and n reference points—both from $\mathcal{X}$) and solve N/m exact search kNN problems for the m query points in each group, where $n = \mathcal{O}(m)$ depends on the details of the approximate search method. We iterate using a different grouping and update the nearest neighbor lists until convergence. The kNN kernel is the exact search of the k nearest points of m query points given n reference points (where both sets of points are selected from $\mathcal{X}$). The kNN kernel appears not only in the approximate search algorithms, but also in the lower-dimensional exact search algorithms.

State-of-the-art for computing the kNN kernel: The kernel can be split into two parts: (1) the distance computation between the m query points and the n reference points and (2) the neighbor selection. State-of-the art implementations of the kNN kernel use this decomposition. In general, the majority of kNN implementations use the Euclidean distance metric (i.e. ℓ_2 norm). In that case, the pairwise distance can be computed using the `GEMM` kernel from any BLAS library [7]. This is done by expanding $\|x_j - x_i\|_2^2$

SC15 November 15–20, 2015, Austin, Texas, USA

k	Method	d=16	d=64	d=256	d=1024
16	`ref`	50	53	72	149
	GSKNN	5	9	28	93
512	`ref`	136	141	125	234
	GSKNN	69	74	60	159
2048	`ref`	306	310	326	402
	GSKNN	267	268	285	358

Table 1 *Performance gains using* GSKNN. *As a highlight, we report the 8-node execution time (seconds) using a MPI-based randomized KD-tree code and demonstrate the performance improvements observed when switching from a* GEMM*-based scheme to* GSKNN. *The outer solver that calls* GSKNN *is described in [31]. Each random KD-tree has m points per leaf. The reference results "*`ref`*" correspond to the* GEMM*-based implementation. We used $N = 1,600,000$ and $m = 8,192$ and varying values of d and k. The time spent in the* GSKNN *kernel is over 90% of the overall time.*

as follows.

$$\|x_i - x_j\|_2^2 = \|x_i\|_2^2 + \|x_j\|_2^2 - 2x_i^T x_j \quad (1)$$

Computing $\|x_i\|_2^2$ scales as $\mathcal{O}(m+n)$. The inner product terms $x_i^T x_j$ can be computed by a GEMM call that, depending on the problem size, can achieve near peak performance; the complexity of this calculation is $\mathcal{O}(mnd)$. Once the distances have been computed, we must perform the nearest-neighbor search for each query point. The best known algorithm to select the k nearest neighbors utilizes a maximum heap selection, which requires $\mathcal{O}(n)$ in the best case and $\mathcal{O}(n \log k)$ in the worst case. Thus, the overall time complexity of a kNN base case is $\mathcal{O}(mnd + mn \log k)$. However the constants in front of these two terms can vary greatly depending on the floating point operation capability, the specific GEMM implementation, and the memory latency. As we discuss in §2, the kNN search can be memory bound, depending on the sizes of m, n, d and k. When d is large (say $d > 512$) the current practice of using GEMM is optimal. But when d is small (say $d < 512$), using GEMM for the kNN can be suboptimal. The dimensionality of many datasets used in practice (e.g., [18]) ranges in the $\mathcal{O}(1)$–$\mathcal{O}(1000)$ interval. In addition, the performance of the kernel crucially depends on the number of nearest neighbors k; if k is relatively small (as is in many applications) GEMM may not be optimal. We can do much better than just using GEMM.

Contributions: Our key insight is that kNN only requires a portion of the calculated pairwise distances to be stored; most of the pairwise distances can be immediately discarded once we are confident that they are not one of the k nearest neighbors. In the best case, we get all of the kNN in the first k distance evaluations and we only need to store an $m \times k$ matrix rather than an $m \times n$ matrix. We have developed a new kernel for x86 architectures that exploits this observation.

- We introduce GSKNN[1] (General Stride k Nearest Neighbors), a kNN kernel that can be directly coupled to both exact and approximate search algorithms to compute exact searches for a number of small (compared to N) $m \times n$ problems (§2.3). GSKNN embeds the neighbor searching, square distance evaluation and coordinate collection inside different levels of GEMM to achieve greater efficiency. The kernel employs vectorization (§2.4) and multithreading (§2.5).
- We explore performance optimization and examine the memory access at the different stages of our kernel; we propose a performance model (§2.6) that can be used to explain the result and to select tuning parameters. We demonstrate the performance of the kernel for a wide range of m, d and k (§4).

Inspired by the BLIS library [28, 29], GSKNN improves the cache performance by blocking and memory packing. Additionally, instead of doing the heap selection after the GEMM call, the selection can be moved inside GSKNN's distance calculation kernel, which has six layers of loops allowing for multiple ways to combine distance calculations and neighbor selection in a single step. We have also implemented a vectorized max heap [17] that further improves the performance of the kernel. Overall, GSKNN achieves high arithmetic intensity. In Table 1 we show an example of the difference GSKNN can make for the all-nearest-neighbor problem. Note that whereas the GEMM-based algorithm is limited to the Euclidean and cosine distances, the new GSKNN kernel applies to any ℓ_p norm $0 < p \leq \infty$ (§2.4).

Related work: We do not review the vast literature on algorithms for the all-nearest-neighbor problem since the kNN kernel is a plugin for all these methods. We focus our attention on the kNN kernel and its use in existing nearest neighbor packages. To the best of our knowledge, there are no other studies on high dimensional brute force kNN without using GEMM as it is defined in BLAS. There is work on GPU architectures but not on x86 platforms. Earlier work involving CUDA kNN [10] was shown to not be competitive with the GEMM-based cuKNN-toolbox [27]. On x86 platforms, the only package we found that uses the GEMM-based kernel is our version of the randomized trees and locality sensitive hashing [31]. Other packages such as FLANN (Fast Library for Approximate Nearest Neighbors) [23], ANN (Approximate Nearest Neighbors) [22] and MLPACK [5] compute the pairwise distances per query point using a single loop over all reference points.

2. METHOD

In this section we present the methods we used, model their performance, and give implementation details. In §2.1, we first demonstrate the GEMM approach to solving the kNN kernel. In §2.2, we discuss various selection algorithms and point out the necessary features of a suitable algorithm. In §2.3, we present variants of GSKNN and explain the general logic of their design. In §2.4, we illustrate how to implement an architecture-dependent kNN micro-kernel and select appropriate blocking parameters. In §2.5, we present data- and task-parallel schemes for the GSKNN algorithm; both are useful depending on the input parameters. We then introduce a theoretical performance model in §2.6 to compare different implementations of the kNN kernel and provide an analytical method for tuning GSKNN.

2.1 GEMM Approach to k Nearest Neighbors

Given the notation in Table 2, we start by reviewing the kNN kernel in an arbitrary d dimensional space. Given a set of $q$uery points $Q(1{:}d,1{:}m)$ and a set of $r$eference points $R(1{:}d,1{:}n)$ which are subsets of $\mathcal{X}(1{:}d,1{:}N)$, Algorithm 2.1 summarizes the method of updating the neighbor lists $<\mathcal{N},\mathcal{D}>$ of the queries Q. By using expansion (1), the pairwise square distance C can be computed in two parts: (1) $C = Q^T R$ (GEMM) and (2) Q_2 and R_2. For each row

[1] https://github.com/ChenhanYu/rnn

Notation	Size	Description
N		problem size
d		dimension
m		number of query points
n		number of reference points
k		neighbor size
p		number of processors
$\mathcal{X}(:,i)$	$\mathbb{R}^{d\times 1}$	$\mathcal{X}(:,i) = x_i \in \mathbb{R}^d$
$\mathcal{X}$	$\mathbb{R}^{d\times N}$	point dataset
$\mathcal{X}_2(i)$	$\mathbb{R}^N$	$\mathcal{X}_2(i) = \|x_i\|_2^2$
q	$\mathbb{N}^m$	a subset of $\mathcal{X}$ ids
r	$\mathbb{N}^n$	a subset of $\mathcal{X}$ ids
Q	$\mathbb{R}^{d\times m}$	$Q(:,i) = \mathcal{X}(:,q(i))$
Q_2	$\mathbb{R}^m$	$Q_2(i) = \mathcal{X}_2(q(i))$
R	$\mathbb{R}^{d\times n}$	$R(:,j) = \mathcal{X}(:,r(j))$
R_2	$\mathbb{R}^n$	$R_2(j) = \mathcal{X}_2(r(j))$
C	$\mathbb{R}^{m\times n}$	$C(i,j) = \|Q(:,i) - R(:,j)\|_2^2$
$\mathcal{N}(i,:)$	$\mathbb{N}^{m\times k}$	kNN ids of $q(i)$
$\mathcal{D}(i,:)$	$\mathbb{R}^{m\times k}$	kNN square distance of $q(i)$

Table 2 *Notation used with MATLAB style matrix representations*

$C(i,:)$, the kNN search selects the k smallest values and updates its corresponding neighbor list $<\mathcal{N}(i,:),\mathcal{D}(i,:)>$ with $<r(j),C(i,j)>$ where $r(j)$ is the global index of reference $R(:,j)$.

Algorithm 2.1 `GEMM` Approach to k-Nearest Neighbor

$C = -2Q^T R$; `//GEMM(-2,`Q^T`,R,0,C);`
for each query i and reference j **do**
  $C(i,j) += Q_2(i) + R_2(j)$; `//`$-2x_i^T x_j + \|x_i\|_2^2 + \|x_j\|_2^2$`;`
end for
for each query i **do**
  Update $<\mathcal{N}(i,:),\mathcal{D}(i,:)>$ with $<r(:),C(i,:)>$
end for

Storage: In approximate nearest-neighbor algorithms, Q and R are gathered from the global coordinate table $\mathcal{X}$. When Q and R are distributed non-uniformly in $\mathcal{X}$, we need to first collect the data points and pack them into a contiguous dense matrix format, since `GEMM` can only deal with regular stride inputs. The collection process can be written as $Q(:,i) = \mathcal{X}(:,q(i))$ and $R(:,j) = \mathcal{X}(:,r(j))$ where $q(i)$ indicates the global index of the i-th point in Q, and $r(j)$ indicates the global index of the j-th point in R. The square 2-norms Q_2 and R_2 can be collected from $\mathcal{X}_2$ along with $<\mathcal{N},\mathcal{D}>$. Another issue is that the output C of `GEMM` is traditionally stored in column major ordering. In this case, the neighbor search on $C(i,:)$ will be stride accessed, but not contiguous. A simple fix is to compute the transpose form $C^T = R^T Q$ which yields a row major storage of C.

Algorithm 2.1, using `GEMM` as a building block, provides good efficiency for high dimensional kNN. The `GEMM` routine itself can also be parallelized efficiently with a data-parallel scheme. The remaining part of Algorithm 2.1 is embarrassingly parallel, since each query can be processed independently. However, `GEMM` is also the bottleneck of the method due to its standardized interface, which disables further optimization across functions. For example, the square distance evaluation and the neighbor selection algorithms we discuss in §2.2 can be fused with `GEMM` to filter out the unlikely neighbor candidates. But this requires a custom implementation of the `GEMM`.

2.2 Neighbor Search Algorithms

Method	Best case	Worst case	Average
Heap Select	n	$n\log(k)$	$n\log(k)$
Quick Select	$n+k$	$(n+k)^2$	$n+k$
Merge Sort	$n\log(k)$	$n\log(k)$	$n\log(k)$

Table 3 *Complexity of selection algorithms [4]*

`GSKNN` seeks to embed the neighbor search in the `GEMM` kernel, where only a small portion of the n candidates ($\ll k$) will be updated each time. Table 3 illustrates possible solutions to the neighbor search problem and their corresponding complexity. It is important that the selection algorithm has $\mathcal{O}(n)$ best case complexity which reduces the probability of degrading the efficiency of `GEMM`. Thus, `GSKNN` uses maximum heap selection, because array storage provides good memory locality, and it can be vectorized by adjusting the number of children per node. Next, we explain our decision through the discussion of possible selection algorithms.

Quick select: In [14] the n candidates are partitioned recursively by a pivot value and then one selects the kth smallest number. This method has $\mathcal{O}(n)$ average complexity. Unfortunately, given that the average complexity has a relatively large coefficient, the sizes of n and k that we are interested in the kNN kernel are typically not large enough to take advantage of linear average complexity. To update a given neighbor list, we first concatenate the list with n candidates and find the new kth element to split the $n+k$ array. This leads to $\mathcal{O}(n+k)$ best case complexity when updating the neighbor list. This property makes it not suitable for embedding within the `GEMM` kernel when n is small.

Merge sort: This variant of merge sort divides the length n array into $\frac{n}{k}$ chunks, each of which has length k. We perform merge sort for each chunk, which leads to $\frac{n}{k} \times k\log(k)$ complexity. Each chunk is then merged updating the neighbor list, and at each merge step we only keep the first k elements. This algorithm achieves $n\log(k)$ complexity in both the best and worst case, and it guarantees contiguous memory access, which can be highly vectorized with a bitonic merge [3]. However, the extra required memory space and fixed complexity still make merge sort selection not competitive with other methods. Another drawback is that the algorithm has $\mathcal{O}(\log(k))$ complexity for updating the given neighbor list, which is too expensive when n is small.

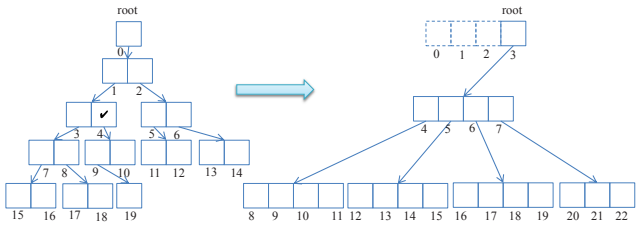

Figure 1 *Binary and d-heap data structure for heap selection*

Maximum heap select: A max heap maintains the k nearest neighbors in a complete binary tree (Figure 1) via an array, where the root (index 0) always contains the largest value. To update the heap with a new neighbor candidate that is smaller than the root, replace the root with the candidate and perform heap adjustment such that the heap property is preserved. If the heap is full, the worst case complexity of the update is $\mathcal{O}(n\log(k))$ for n candidates. The best case only takes $\mathcal{O}(n)$ by filtering out all the candi-

dates larger than the root. If the heap is empty, the first k candidates only take $\mathcal{O}(k)$ [8] to build the heap. The key to attaining a linear best case complexity is the capability to access and delete the largest value in $\mathcal{O}(1)$ time, which reduces the damage to the original GEMM design. However, the worst case complexity is still very bad for large k. In such cases, the memory access pattern also becomes random, destroying locality. To reduce the random access penalty, an implicit d-heap [16] yields better practical performance in modern processors. By having more children for each node (Figure 1), a longer cache line is utilized. In §2.4, we show how a d-heap[2] can further be optimized by memory padding and vectorization.

Given the above considerations, we decided to use heap selection algorithm for GSKNN. Although heap selection has a higher average complexity than the quick select, it has much better memory access patterns. It's important that the neighbor search algorithm does not add too much extra burden to the GEMM kernel. Heap selection is the ideal method due to its array storage and cheap update cost for small n. Given that the worst case complexity of this selection still grows with k, it is unclear how to optimally place it withing the GEMM loop hierarchy. This will be further discussed in §2.3.

2.3 General Stride k-Nearest Neighbors

We present a new computational kernel GSKNN, that refactors the kNN problem by fusing the GEMM kernel with the distance calculations and the nearest-neighbor search. We expose the possible benefits of reusing the output C (square distances) immediately after it's computed such that less memory latency is suffered. Heap selection can be performed right after a small block of C has been computed, which increases the reuse rate of the data because C is still in the L1 cache. By doing the heap selection early, C can be discarded without writing back to memory. To accomplish this we require a transparent GEMM implementation and a light weight neighbor search algorithm such as those discussed in §2.2.

Algorithm 2.2 GSKNN Approach to k-Nearest Neighbor

```
for j_c = 0 : n_c : n − 1 do                                   /* 6th */
  for p_c = 0 : d_c : d − 1 do                                 /* 5th */
    X(p_c : p_c + d_c − 1, r(j_c : j_c + n_c − 1)) → R^c
    if p_c + d_c ≥ d then
      X_2(r(j_c : j_c + n_c − 1)) → R_2^c
    end if
    for i_c = 0 : m_c : m − 1 do                               /* 4th */
      X(p_c : p_c + d_c − 1, q(i_c : i_c + m_c − 1)) → Q^c
      if p_c + d_c ≥ d then
        X_2(q(i_c : i_c + m_c − 1)) → Q_2^c
      end if
      for j_r = 0 : n_r : n_c − 1 do                           /* 3rd */
        for i_r = 0 : m_r : m_c − 1 do                         /* 2nd */
          micro-kernel() in Algorithm 2.3
        end for
      end for                                  /* end macro-kernel */
    end for
  end for
end for
```

[2]The d here is not related to the d in Table 2. d-heap is the conventional name of the d-ary heap.

We first present pseudo-code of GSKNN in Algorithm 2.2 and Algorithm 2.3. We later refactor GSKNN to show how square distance evaluations and heap selections in Algorithm 2.1 can be embedded into the GEMM kernel. While Algorithm 2.3 seeks to embed the heap selection in the earliest stage, there are other placements of heap selection which can be explored. We present six variants which expose all possible placements of the selection at the end of this section and discuss their pros and cons.

The algorithm contains six layers of loops, each of which corresponds to different partitioning of the m, n and d dimensions. The partition scheme is identical to that of the Goto algorithm in BLIS [29] and GotoBLAS [11], which efficiently amortizes memory packing, reuse, and alignment operations to achieve high computational efficiency. Beginning with the outer most loop (6th loop; indexed by j_c), the n dimension ($R(:, 1:n)$) is partitioned with block size n_c. The 5th loop (indexed by p_c) partitions the d dimension with block size d_c. The 4th loop (indexed by i_c) partitions the m dimension ($Q(:, 1:n)$) with block size m_c. The macro-kernel contains the 3rd and the 2nd loops (indexed by j_r and i_r) which further partition Q and R into smaller blocks. The micro-kernel (Algorithm 2.3) contains the 1st loop (indexed by p_r) that calculates square distances for a small block of C with size $m_r \times n_r$. We also place the heap selection within this loop.

Algorithm 2.3 GSKNN micro-kernel (1st loop)

```
for p_r = 0 : d_c − 1 do                                       /* 1st */
  for j = 0 : n_r − 1 do                       /* unroll m_r-by-n_r */
    for i = 0 : m_r − 1 do
      C^r(i, j) = C^c(i_c + i_r + i, j_c + j_r + j)
      C^r(i, j) −= 2Q^c(i_r + i, p_r)R^c(j_r + j, p_r)
      C^c(i_c + i_r + i, j_c + j_r + j) = C^r(i, j)     //−2x_i^T x_j
    end for
  end for
end for                                    /* end rank-d_c update */
if p_c + d_c ≥ d then
  for j = 0 : n_r − 1 do                       /* unroll m_r-by-n_r */
    for i = 0 : m_r − 1 do
      C^r(i, j) += Q_2^c(i_r + i) + R_2^c(j_r + j)   //‖x_i − x_j‖_2^2
    end for
  end for
  for i = i_c + i_r : i_c + i_r + m_r − 1 do
    Update <N(i, :),D(i, :)> with <β(j_c + j_r : j_c + j_r +
    n_r − 1), C^r>
  end for
end if
```

Refactoring: A good way to understand the features of GSKNN is to identify the scope of GEMM and the heap selection in Algorithm 2.1. If we remove all statements inside the **if** control flow ($p_c + d_c \geq d$), the remaining parts of Algorithm 2.2 and Algorithm 2.3 are almost exactly the original Goto algorithm [11]. The only difference is that GSKNN can take non-uniform stride inputs indicated by the global indices $q(:)$ and $r(:)$. The reason for the support of this feature is explained in the packing paragraph. The heap selection and the square 2-norm accumulation in Algorithm 2.1 are all moved into the 2nd loop (in Algorithm 2.3); these computations will *only* be executed in the last p_c iteration after the 2nd loop of GEMM has completed. If p_c is not in its last

iteration, then GSKNN will only perform the rank-d_c update:

$$C+ = Q(p_c : p_c + d_c - 1, :)^T R(p_c : p_c + d_c - 1, :) \qquad (2)$$

where the temporary rank-d_c update result will be accumulated in the C^c buffer. Otherwise, the square distance will be computed and heap selection will be performed.

Packing: According to the partitioning of each dimension, Algorithm 2.2 creates temporary buffers $Q^c(m_c, d_c)$, $R^c(d_c, n_c)$, $Q_2^c(m_c, 1)$, $R_2^c(1, n_c)$ which duplicate and rearrange relevant pieces of data. This process is called memory packing [11, 29] and is used within typical implementations of GEMM. Each buffer is designed to fit into different portions of the memory hierarchy in Figure 2. Each region is packed into a "Z" shape to guarantee a contiguous access pattern in the macro- and micro-kernel. Another reason that we must perform packing is to align the memory to a specific address offset which is essential for vectorization. The SIMD (Single Instruction Multiple Data) instructions we use in the micro-kernel (see §2.4) require the memory to be aligned to reduce the memory latency. Given the fact that the GEMM routine always repacks the memory, Algorithm 2.2 skips the phase of collecting Q and R, choosing to pack directly from $\mathcal{X}$ to Q^c and R^c, avoiding redundant memory operations and saving buffer space.

Var#1: There are six variants of GSKNN labeled according to when we choose to perform heap selection. We label Algorithm 2.2 and Algorithm 2.3 as Var#1. In this variant, the heap selection is performed at the earliest possible location in the code (immediately after the 1.st loop). To help visualize the algorithm design, we illustrate its data flow in Figure 2. After the partitioning of the reference set in the 6th loop, we identify the portion of the those points (orange) in $\mathcal{X}$ which are located in the main memory. In the 5th loop, we collect the first d_c elements of the orange portion with the global indices $r(:)$ and pack the reference points into R^c. In the 4th loop, we identify and collect the query points (light blue) from $\mathcal{X}$ with the global indices $q(:)$, packing them into Q^c. At this moment, R^c will be evicted to the L3 cache since Q^c will occupy all of L1 and part of L2. R^c and Q^c will be reused inside the macro-kernel (3rd and 2nd loops). For each micro-kernel call, a panel of R^c and Q^c will be promoted to the L1 cache for the rank-d_c update, but only the R^c panel will stay in L1 since the 2nd loop will access different panels of Q^c and reuse the same R^c panel. Inside the micro-kernel, the result of the rank-d_c update will be accumulated inside the registers C^r (green). In the last p_c iteration, Q_2^c (purple) and R_2^c (yellow) will also be collected from $\mathcal{X}_2$ which will reside in the same level as Q^c and R^c. After the 1st loop, Q_2^r and R_2^r are loaded into the registers, and the square distances are computed in C^r. The heap (blue) is promoted all the way from memory to registers to complete the heap selection on C^r, and we can immediately discard C^r without storing it back. The drawback of Var#1 is that the heap selection may evict Q^c and R^c from the L1 and L2 caches if k is too large. To ensure the proper cache behavior, we may need to move the heap selection to another loop depending on the size of k.

Other variants: The other variants from Var#2 to Var#6 are defined by inserting the heap selection after the appropriate loop. The index of the variant reveals the loop in which we perform heap selections. As the index of the loop increases, so does the update size of that selection. Var#1 updates the nearest neighbors after a small tile of square

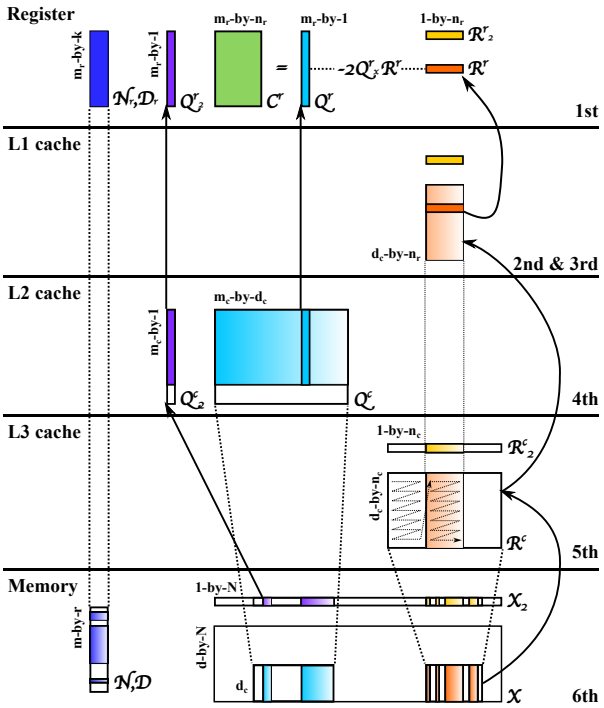

Figure 2 *maps* GSKNN *to a 5-level memory hierarchy (3-level cache). Q (light blue), Q_2 (purple), R (orange), R_2 (yellow) and $<\mathcal{N},\mathcal{D}>$ (blue) will be packed and then reside in different levels of cache. A piece of memory associated with the micro-kernel operation is shown in the higher levels if pieces from those levels will be used within the micro-kernel. C^r (green) is temporarily created at the register level and is immediately discarded after heap selection.*

distances C^r have been evaluated; a higher numbered variant will update its neighbors with a larger square distance matrix. For example, Var#6 performs heap selection after the 6th loop, just like in Algorithm 2.1. Var#2 and Var#3 are not preferred because of two reasons. (1) When k is small, they need to store a larger square distance matrix which leads to a higher memory overhead than Var#1. (2) When k is large, they lead to high reuse rate of the heap such that the heap won't be evicted from L1 and L2. Recall from Figure 2 that L1 and L2 are designed to accommodate the panels of R^c and Q^c. If the heap occupies L1 and L2 as occurs with a high reuse rate, GSKNN needs to repeatedly load R^c and Q^c from L3 which will degrade its efficiency. Therefore, these two variants are usually slower than Var#6 when k is large. Var#4 is not viable, since the 5th loop blocks in the d dimension where the square distances are not fully computed. Var#5 is only preferred when there is not enough memory to store all the square distances. After the 5th loop, $m \times n_c$ of square distances will be computed. Heap selections must load the square distances from memory, since $m \times n_c$ can be much larger than the L3 cache. Thus, Var#5 and Var#6 both suffer the DRAM latency, and the only difference is that Var#5 gets to store only $m \times n_c$ distances instead of $m \times n$. Still, Var#5 is not preferred even it can save some memory space, since all heaps may need to be reloaded from memory $\frac{n}{n_c}$ times which doubles the memory latency or even worst.

Overall, Var#1 gives the greatest opportunity for memory reuse without disturbing the proper cache behavior of GEMM, and Var#6 corresponds to the typical approach. In §2.6 we will briefly discuss how to switch between these two variants

based on different sizes of k and d.

2.4 Micro-Kernel Implementation and Parameter Selection

We have illustrated the principles that go into designing the kNN kernel, and now we will discuss architecture dependent implementation details. We need to understand how to implement a micro-kernel such that the rank-d_c update is competitive to `GEMM`. We also need to know which blocking parameters to use and when to switch between variants of `GSKNN` to maintain efficiency. In the rest of this section, we briefly illustrate the micro-kernel design logic and present an analytical method for selecting parameters.

The idea of exposing the micro-kernel first came from the BLIS framework which aims to identify the minimum portion of `GEMM` calculations that are architecture dependent. GotoBLAS [12] (predecessor of OpenBLAS [24]) implements the 1st to 3rd loops (macro-kernel or inner-kernel) in assembly, but the BLIS framework only implements the 1st loop (micro-kernel) in SIMD assembly or vector intrinsics. The 2nd through 6th loops are usually implemented in `C` which generates some overhead, but programmability and portability are significantly increased. Following the same design philosophy, `GSKNN` maintains the micro-kernel structure and uses outer loops (2nd-6th) written in `C`. We include the rank-d_c update (the inner-most loop), the square distance evaluations and the heap selection (Var#1) in the micro-kernel.

The Var#1 micro-kernel (Algorithm 2.3) computes an $m_r \times n_r$ square distance matrix C^r and updates the neighbor lists in four steps: (1) rank-d_c update, (2) computing square distances, (3) conditionally storing the square distances, and (4) heap selections. Different implementations may vary between architectures. Here we only illustrate the idea of manipulating vector registers and pipelining the memory operations.

Rank-d_c update and square distances: The intermediate results of the rank-d_c update, C^r, are created and maintained inside vector registers. Remaining vector registers are used by Q^r, R^r, Q_2^r, R_2^r, and other values required in the square distance calculations. To overlap the memory operations when loading Q^r and R^r, we double buffer by unrolling the 1st (d_c) loop either two or four times. The idea is to use auxiliary registers to preload the data from memory so that the data is already present when it is required, which prevents the ALU (Arithmetic Logic Unit) from stalling. For example, the next Q^r (Q^c pointer + m_r) and R^r (R^c pointer + n_r) are preloaded by two auxiliary registers concurrently with the rank-1 update ($C^r += Q^r \times R^r$). An even larger scale memory optimization can be accomplished by using the prefetch instruction. This promotes memory to a faster level (e.g. L1 cache) in the hierarchy without actually loading it into registers. For example, the next required micro-panel of R^c, the current C^c and the root of the heap can be prefetched and overlapped with the current rank-d_c update. These two optimization schemes are called the rank-d_c update pipeline.

In addition to the memory pipeline, the instruction count can be further optimized if the system supports vectorized instructions (SIMD). Given that `VLOAD`, `VSTORE`, `VADD` and `VMUL` (or `VFMA`) are supported, an efficient way to complete the rank-1 update in the rank-d_c update pipeline is to load vectors Q^r and R^r and perform a series of `VFMA` instructions interleaved with register permutation. Figure 3 shows that a

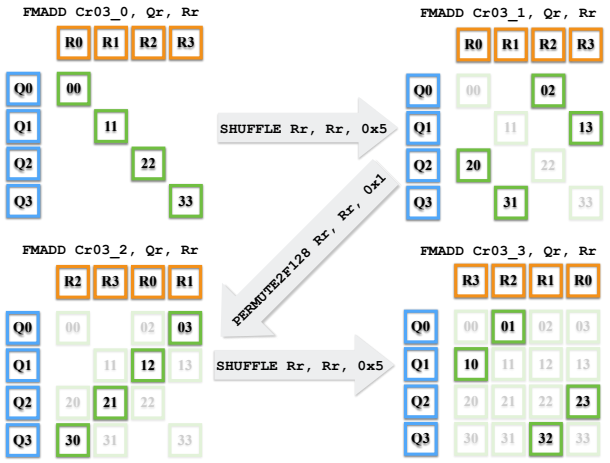

Figure 3 `AVX` 4×4 *rank-1 update. Given Q^r (blue) and R^r (orange), we compute the 4×4 outer-product C^r (green) by 4* `VFMA`*s interleaved with vectorized shuffling operations. The 3rd operands (0x5, 0x1) indicate the shuffling (permutation) type.*

4×4 C^r can be computed by executing the `VFMA` instruction on Q^r and different permutations of R^r four times. Each time a permutation occurs, a `VFMA` computes a diagonal of the current C^r. At the end of the rank-d_c updates, we permute C^r back to original order. The -2 from Equation (1) is scaled at the end of the rank-d update, and the square distances can be computed by adding A_2^r and B_2^r to C^r.

General ℓ_p norm: We briefly discuss how to implement other kinds of norms within the micro-kernel structure. ℓ_1 and ℓ_∞ can be implemented by replacing each `VFMA` with `VMAX`, `VMIN`, `VSUB` and `VADD` (`VMAX` if ℓ_{inf}). The 3-norm and other p-norms can either be expanded or approximated by a `VPOW`[3] dependent upon instruction counts.

Heap selection: Before the heap selection, we have a chance to decide whether or not to store the square distance, C^r, currently in register memory back to slower memory by checking whether the square distance is smaller than the root of the max heap. This comparison can also be vectorized by broadcasting the root value and using a `VCMP`. In the best case scenario, all of C^r can be discarded which provides `GSKNN` a very large performance benefit. In practice, we use either a binary or a 4-heap, depending on the magnitude of k. By padding the root with three empty space (Figure 1), four children will fall in the same cache line (say 256 bytes). Accessing all children only needs one cache line which decreases the latency. The kind of heap used for selection depends on how many cache lines are present and how many extra instructions are needed in the worst case. Selection on a 4-heap requires extra comparisons to find the maximum child, but the depth of the heap ($\log_4(k)$) is smaller than the binary heap. Vectorizing the maximum child search requires 2 `VMAX`, 1 `PERMUTE`, 1 `PERMUTE2F128` and 1 `VTESTQ`, but these operations all work on the same cache line. In contrast, a binary heap can find its maximum child with a single comparison, but it may require accessing more cache lines due to $\log_2(k)$ height. In `GSKNN`, Var#1 uses a binary heap to deal with small k, and Var#6 uses a 4-heap for large k.

Selecting parameters: The parameters m_c, n_c, d_c, m_r and n_r are all architecture dependent. To choose the optimal

[3] `VPOW` is not part of `AVX`, but it is supported in Intel SVML. To implement `VPOW`, the convention is to use a minimax polynomial approximation.

combination of parameters for a specific architecture, there are two primary approaches: tuning by exhaustive search or tuning by modeling. Following [19], we use the Intel Ivy-Bridge architecture as an example to demonstrate how to choose these parameters.

- m_r and n_r are chosen based on the latency of a `VFMA` instruction. For Ivy-Bridge (no `VFMA`), the latency of an equivalent `VMUL` and `VADD` is 8 cycles. To prevent the pipeline from stalling, more than 8 `VFMA`s must be issued. In total, 8×4 registers are required.

- d_c is chosen to utilize the L1 cache, which corresponds to the size of R^c and Q^c micro-panels. To fit the micro-panels in L1, d_c is chosen such that $m_r \times d_c + n_r \times d_c$ is about $\frac{3}{4}$ of the L1 cache size. For Ivy-Bridge, $d_c = 256$. This preserves $\frac{1}{4}$ of the L1 cache for other memory to stream through.

- m_c and n_c are chosen based on the L2 and L3 cache sizes. In the single-core case, we choose $m_c = 96$ so that $\frac{3}{4}$ of the L2 cache contains Q^c, and we choose $n_c = 4096$ to fit R^c in the L3 cache. In the case of multi-core execution, m_c will be dynamically determined depending on the number of processors and the problem size m. This helps achieve better load balancing across multiple processors.

Switching between variants: Since we have eliminated other variants in §2.3, we only have to choose between Var#1 and Var#6. Var#1 avoids storing C^c but suffers a large memory penalty when k is large. Var#6 maintains the efficiency of the rank-d_c update for large k, but does not gain the reuse benefits of Var#1. A two dimensional threshold can be set on the (d,k) space, and which variant is used is determined based on the threshold. A tuning based decision table would need to search the whole (d,k) space which can be time consuming. By understanding the trade off between these two variants, we later introduce a runtime prediction model in §2.6 which can quickly produce a smaller search space for fine tuning.

2.5 Parallel k-Nearest Neighbors Search

kNN can be naturally parallelized by processing each query independently. In approximate nearest-neighbor, all kNN kernels can be solved independently which is called task-parallelism. Computation in a kNN kernel can also be parallelized by exploiting data parallelism. This approach is more challenging. We first tackle the load balancing problem inherent to task-parallelism, and then we explain the data-parallel scheme by exploring different options in the six layers of loops.

The task-parallel scheme is a good choice when a large number of small kNN kernels need to be parallelized. A small kernel usually doesn't provide enough parallelism to exploit all resources, but the number of tasks can be sufficient to grant speedup. Each kernel is assigned to a processor which creates its own schedule. Generally, optimal scheduling is an NP-complete problem [26], but it is relatively easy if there are no dependencies between each task. On a homogeneous parallel system, an optimal static schedule can be found by a greedy first-termination list scheduling on a sorted task-list (a special case of [13]). All kNN kernels are first sorted in descending order according to their estimated runtime (see §2.6), and each task is assigned to the processor with the smallest accumulated runtime.

	type	coeff	Algorithm 2.1	`GSKNN`
T_f	-	$\frac{1}{\tau_f}$	$(2d+3)mn$	
T_o	-	$\frac{1}{\tau_f}$	$24mn + 24mk\log(k)$	
$T_m^{\mathcal{X}}$	`r`	τ_b	$dn + dm\lceil\frac{n}{n_c}\rceil$	$dn + dm\lceil\frac{n}{n_c}\rceil$
$T_m^{\mathcal{X}_2}$	`r`	τ_b	$n + m\lceil\frac{n}{n_c}\rceil$	$n + m\lceil\frac{n}{n_c}\rceil$
$T_m^{Q^c}$	`w`	τ_b	$dm\lceil\frac{n}{n_c}\rceil$	$dm\lceil\frac{n}{n_c}\rceil$
$T_m^{Q_2^c}$	`w`	τ_b	$m\lceil\frac{n}{n_c}\rceil$	$m\lceil\frac{n}{n_c}\rceil$
T_m^{q}	`r`	τ_b	$m\lceil\frac{n}{n_c}\rceil$	$m\lceil\frac{n}{n_c}\rceil$
$T_m^{R^c}$	`w`	τ_b	dn	dn
$T_m^{R_2^c}$	`w`	τ_b	n	n
T_m^{r}	`r`	τ_b	n	n
$T_m^{C_c}$	`r/w`	τ_b	$2(\lceil\frac{d}{d_c}\rceil - 1)mn$	$2(\lceil\frac{d}{d_c}\rceil - 1)mn$
$T_m^{\mathcal{D}}$	`r/w`	τ_ℓ	$2\epsilon mk\log(k)$	$2\epsilon mk\log(k)$
$T_m^{\mathcal{N}}$	`r/w`	τ_ℓ	$2\epsilon mk\log(k)$	$2\epsilon mk\log(k)$
T_m^{Q}	`r/w`	τ_b	$2dm$	-
T_m^{R}	`r/w`	τ_b	$2dn$	-
T_m^{C}	`r/w`	τ_b	$4mn$	$(mn*)$

Table 4 *Theoretical runtime breakdown analysis of* `GSKNN` *and Algorithm 2.1. Note that only Var#6 has the* T_m^C *term.*

When m and n are large enough, Algorithm 2.2 reveals more parallelism that can be exploited. Exploiting parallelism inside different layers of loops is called data-parallelism. Our goal is to choose the loop that provides the most benefit in reusing the cache that is shared by all the processors. Following [28], we aim to parallelize the 4th loop surrounding the micro-kernel. Every m_c query will be assigned to a processor with a periodic schedule (OpenMP parallel loop pragma with a static schedule). Given enough parallelism in the 4th loop, this parallel scheme is the best choice on Sandy-Bridge/Ivy-Bridge processors, because the memory packing scheme in `GSKNN` is naturally portable to multi-core architectures with separated L2 caches and a shared L3 cache. Each processor will create a private Q^c and preserve it in its private L2. R^c is shared and preserved in the L3 cache, making it available to all processors. The only drawback of this scheme is that load balancing issues may arise when m is not a multiple of $m_c \times p$. This problem can be solved by dynamically deciding m_c to better fit the given p and m. Other choices may be suitable for parallelization in different architectures. For example, the 6th loop is a good candidate for separated L3 caches such as on a NUMA (Non Uniform Memory Access) node which has more than one CPU. The 3rd loop is parallelized in [28] on Intel Xeon Phi due to the large $\frac{n_c}{n_r}$ ratio. However, the 3rd and 6th loop are not suitable for `GSKNN`, since parallelization on the reference side may lead to a potential race condition when updating the same neighbor list.

2.6 Performance Model

We derive a model to predict the execution time T and the floating point operation efficiency (GFLOPS) of Algorithm 2.2. These theoretical predictions can be used for performance debugging and helping us understand possible bottlenecks of the `GEMM` approach. The estimated time T can also be used in task scheduling on both heterogeneous and homogeneous parallel systems. T can also help to decide between the variants of `GSKNN`.

Assumption: For simplicity, the model assumes that the underlying architecture has a modern cache hierarchy,

which contains a small piece of fast memory (cache) and a large chunk of slow memory (DRAM). We further assume that accessing the fast memory can be overlapped with ALU (Arithmetic Logic Unit) operations with proper preloading or prefetching instructions. In our model the latency of loading slow memory cannot be hidden. For memory store operations, we assume a lazy write-back policy, which won't write through the slow memory if it's not necessary. We assume that the write-through time can be overlapped as well. The slow memory operations in Algorithm 2.2 are presented as three parts (1) memory packing (2) reading/writing C^c and (3) heap selections. Based on the assumptions above, these slow memory operations followed by fast floating point operations comprise the majority of the sequential computation time.

Notation: In addition to the notation introduced in §2, we further define τ_f, τ_b, τ_ℓ, ϵ, T, T_f, T_m and T_o in the model. τ_f is the number of floating point operations that the system can perform in one second, τ_b (bandwidth related) denotes the average time (in seconds) of a unit of contiguous data movement from slow memory to fast memory; and τ_ℓ (latency related) denotes the time (in seconds) of a random memory access. We use $\epsilon \in [0, 1]$ to predict the expected cost of heap selection. T_f and T_m represent the time consumed by floating operations and memory operations respectively. T_o is the total time spent on other instructions. By summing the three terms together, $T = T_f + T_m + T_o$, we get the total time.

Floating point and other operations: In Table 4, we break down the theoretical cost of each component, starting with the floating point operation time T_f, followed by T_o, and then by each individual term of T_m. T_f and T_o can be computed using the following formula:

$$T_f + T_o = \frac{1}{\tau_f}(2d+3)mn + \frac{24}{\tau_f}(mn + \epsilon mk\log(k)) \qquad (3)$$

where $2dmn$ is the number of operations in the rank-d update, and $3mn$ is the number of floating point operations required in Equation (1). Notice that heap selection doesn't require any floating point operations but still contributes to the runtime. T_o is the time required for the selection. Each heap selection requires `MAX`, `MIN`, `CMP` and pointer calculations for swapping $<\mathcal{N}, \mathcal{D}>$. In the model, we estimate the cost optimistically by assuming that each heap will only require $\epsilon k\log(k)$ adjustments, and each adjustment takes 12 instructions (about 24 floating points operations). The expected cost can be adjusted by setting ϵ to reflect the expected complexity of heap selection. The times in Equation (3) are computed by dividing the total floating point operation count (or instruction throughput count) with the theoretical peak floating point operation throughput (τ_f) per second.

Memory operations: The total data movement time varies with both problem size m, n, d and k and block sizes m_c, n_c and d_c. Despite being able to calculate complexity in advance, choosing the proper coefficient to accurately model the time taken by data movement remains difficult. The details of data movement between registers and caches is hidden by the compiler, and the movement between caches and DRAM is hidden by the cache coherence protocol of the architecture. Fortunately, the square distance evaluations in Algorithm 2.2 are designed based on the understanding of memory movement, which makes cache behavior more predictable. The remaining difficulty comes from the heap selections, which can vary in complexity from the best case to the worst case. To make matters worse, memory access can be random during the heap adjustment for large k.

All memory movement costs in Table 4 are labeled by subindices T_m. Each term is characterized by its read/write type and the amount of memory involved in the movement. Following the model assumptions, the time for memory movement in Var#1 can be computed using the following formula:

$$\begin{aligned} T_m^{Var\#1} &= \tau_b(nd+2n) + \tau_b(dm+2m)\left\lceil \frac{n}{n_c} \right\rceil) + \\ &\quad \tau_b(\left\lceil \frac{d}{d_c} \right\rceil - 1)mn + \tau_\ell(2\epsilon mk\log(k)) \end{aligned}$$

All the write operations are omitted; we only sum the read operations. Operations will be repeated multiple times depending on which loop they reside in. For example, $\tau_b(\left\lceil \frac{d}{d_c} \right\rceil - 1)mn$ is the cost of reading/writing the intermediate result C^c, which increases with d as a step function because C^c is used to accumulate the rank-d_c (to be reloaded in the 5th loop). The cost of the heap selection, $\epsilon\tau_\ell 2mk\log(k)$, has a different unit cost than τ_b since the memory access may be random–especially when k is large. For a binary heap, τ_ℓ is roughly $2\tau_b$ depending on the target DRAM's column access strobe (CAS) latency and the memory cycle time. For a 4-heap where four children will be accessed together, τ_ℓ will be roughly equal to τ_b. We can also derive an estimate for $T_m^{Var\#6}$ in (4) which has an additional term mn due to the cost of storing C.

$$T_m^{Var\#6} = T_m^{Var\#1} + \tau_b mn. \qquad (4)$$

If we further assume that `GEMM` is implemented as a variant of the Goto algorithm, the memory movement cost of Algorithm 2.1 can be estimated by $T_m^{Var\#1}$ with 3 additional terms A, B and C.

$$T_m^{Algorithm\ 2.1} = T_m^{Var\#1} + \tau_b(dm + dn + 2mn). \qquad (5)$$

The cost of these three additional terms from the `GEMM` interface can be calculated. The coordinates in $\mathcal{X}$ need to be collected in A and B which leads to $dm + dn$. The standard `GEMM` output C takes mn. The square distance accumulation will read/write C which leads to another factor of mn cost in (5).

Comparison: Because T_f and T_o are the same between `GSKNN` and Algorithm 2.1, we can show that Var#1 has smaller memory complexity (in both space and memory operations) by not explicitly forming A, B and C. This effect is significant in low d, since $T_m^C = 2\tau_b mn$ doesn't decrease with d, which makes `GEMM` a memory bound operation. In Figure 4, we compare the floating point efficiency ($\frac{(2d+3)mn}{T}$) between the model prediction and experimental results. We evaluate the average runtime of three consecutive kNN kernels. The prediction always overestimates the efficiency because we omit the cost of accessing the fast memory. We also find that the prediction is too optimistic in low d, because the model doesn't capture the fact that the CPU pipeline is not fully occupied during the ramp up and ramp down phase. The model captures the performance difference between methods for large d exactly, which reflects the memory savings of `GSKNN`.

The model can further be used to select between variants, helping reduce the tuning time. Figure 5 plots the GFLOPS

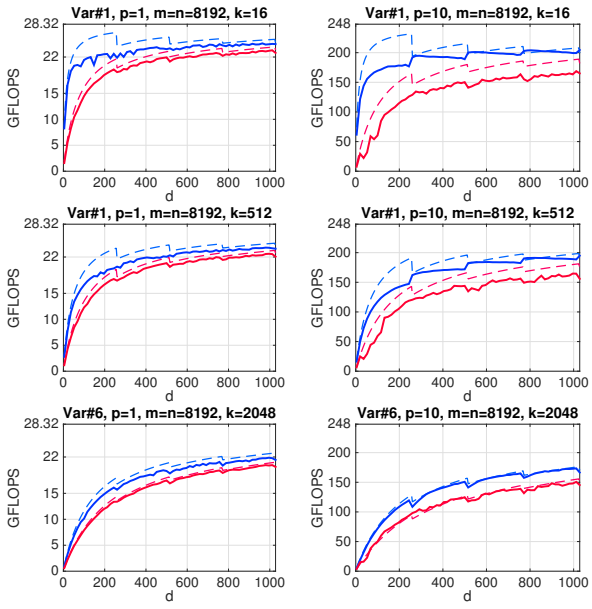

Figure 4 *Predicted floating point efficiency (GFLOPS). Dashed lines are the predicted efficiency by* (4) *and* (5)*. The solid blue line is the experimental efficiency of* GSKNN*, and the solid red line is the reference kernel with* MKL GEMM *and an* STL *max heap. Parameters:* $m = n = 8192$, $k = 16, 512, 2048$, $\tau_f = 8\times3.54$, $\tau_b = 2.2\times10^{-9}$, $\tau_\ell = 13.91\times10^{-9}$, $\epsilon = 0.5$. *For the 10-thread result,* $\tau_f = 10\times8\times3.10$, τ_b *and* τ_ℓ *are* $\frac{1}{5}$ *the original value.*

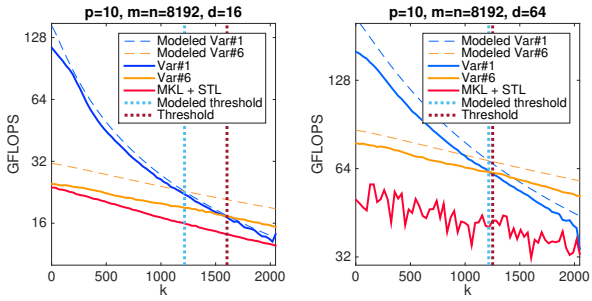

Figure 5 *Predicted 10-core floating point efficiency (GFLOPS) for different* k*. The predicted threshold (light blue dotted line) crosses the intersection of the two dashed lines, and the experimental threshold (purple dotted line) crosses the intersection of the solid blue and yellow lines.*

along with k which provides a predicted threshold for the variant selection. The predicted threshold (light blue dotted line) is close to the real experimental threshold (purple dotted line) when the dimension grows. This prediction can help quickly narrow down a small region for fine tuning and prevent an exhaustive search.

3. EXPERIMENTAL SETUP

Here we give details on the experimental setup used to test our methods. The current version of GSKNN contains double precision x86-64 micro-kernels designed for Intel Sandy-Bridge/Ivy-Bridge architectures. In all experiments, an AVX assembly micro-kernel is used to compute the rank-d_c update, and an AVX intrinsics micro-kernel is used to compute the case when d is not a multiple of d_c. Our GSKNN kernel has been integrated with our implementations approximate all-nearest-neighbor implementations of randomized KD-trees and locality sensitive hashing [20, 31]. These implementations use MPI and OpenMP parallelism and originally used the GEMM approach for the kNN kernel.

Implementation and hardware: GSKNN is implemented in C, SSE2 and AVX intrinsics and assembly. Everything except the micro-kernel is written in C. The parallel randomized KD-tree kNN is written in C++. The code is compiled with the Intel C compiler version 14.0.1.106 and mvapich2 version 2.0b with the -O3 optimization flag. We carry out runtime experiments on the Maverick system at TACC with dual-socket Intel Xeon E5-2680 v2 (Ivy Bridge) processors. The stable CPU clockrate is 3.54GHz/3.10GHz for 1/10 core experiments. To make sure the computation and the memory allocation all reside on the same socket, we use a compact KMP_AFFINITY. The pure OpenMP results use a single socket of a NUMA node, and the results in Table 1 use 8 NUMA nodes (16 sockets).

GSKNN parameters: We select blocking parameters as discussed in §2.4 with $m_r = 8$, $n_r = 4$, $k_c = 256$, $m_c = 104$ and $n_c = 4096$, which make the size of Q^c 208 KB and the size of R^c 8192 KB. For all experiments with $k \leq 512$, we use Var#1. Otherwise, we use Var#6.

Dataset: In the integrated experiment in Table 1, we use a 10 dimensional Gaussian distribution generator and embed the sample point to a high dimensional space ($d = 16, 64, 256, 1024$). For the remaining experiments we sample from a uniform $[0, 1]^d$ distribution to create synthetic dataset for our experiments.

4. RESULTS

We have already presented the integrated runtime results of the parallel MPI randomized KD-tree approximation to the all-nearest-neighbor search in §1. Here we report two sets of results to illustrate the performance of GSKNN.

- We use a runtime breakdown to better show how individual terms affect the overall runtime. The results show that early reuse of C can significantly reduce memory overhead.
- The 10-core efficiency comparison examines different magnitudes of m, n, d and k, providing a performance overview of GSKNN on a shared memory system. We show that GSKNN overcomes the memory bottleneck of GEMM in low d. GSKNN can achieve 80% of theoretical peak performance for $k \leq 128$ and $d \geq 512$.

Breakdown analysis: Table 5 breaks down T_{total} into $T_{coll} + T_{\text{GEMM}} + T_{sq2d} + T_{heap}$, which represents gathering data from $\mathcal{X}$, evaluating GEMM, evaluating the square distances, and heap selections respectively. For GSKNN, it is difficult to measure the time spent on each phase since a timer call would lead to a serious overhead inside the 2nd loop. Thus, we only report the total time for GSKNN and estimate the time spent on heap selection (see Table 5). For large $d \geq 256$, T_{GEMM} dominates the runtime, and the remaining $T_{coll} + T_{sq2d} + T_{heap}$ are minor (10%). The integrated square distance kernel in GSKNN is slightly faster than T_{GEMM} because the time GEMM spends on the $T_m^{C^c}$ term is amortized by its rank-d update. However, the saving of $T_m^{C^c}$ is significant (40% to 60%) if d is small ($d = 16, 64$). This reflects the fact that the GEMM-based nearest neighbor kernel is a memory bound operation in low d. According to the table, Var#1 does have a smaller latency during the heap selection. Given that the complexity is the same, the latency costs can be up to 10 time smaller by reusing C at the earliest opportunity. For Var#6 ($k = 2048$), we show that the 4-heap is 30% to 50% more efficient than the binary heap,

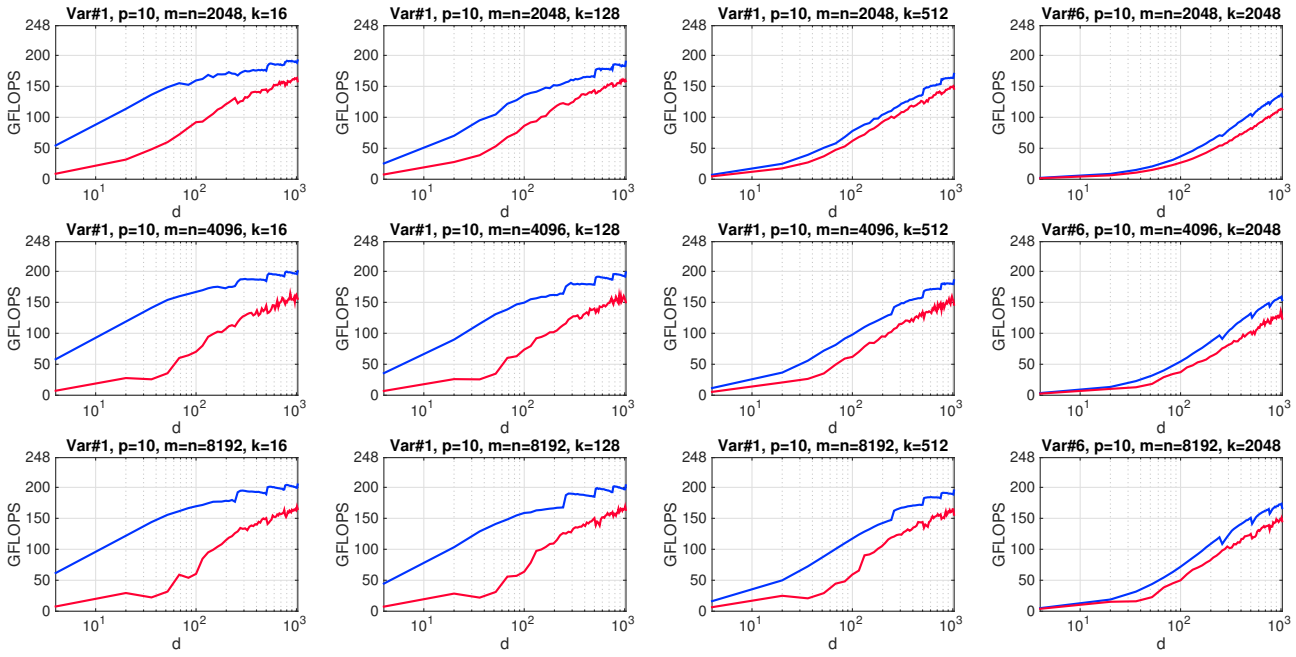

Figure 6 *10-core floating point efficiency comparison: m and n from top to bottom are 2048, 4096 and 8192. k from left to right are 16, 128, 512 and 2048. The X-axis is the dimension size from 4 to 1028, and the Y-axis is GFLOPS where the theoretical peak performance is 248 GFLOPS ($8 \times 3.1 \times 10$). For $k = 16, 128, 512$, Var#1 is used, and for $k = 2048$ Var#6 is used instead.*

MKL+STL / GSKNN		$m = n = 8192$, $d = 16$	
k	$T_{coll} + T_{\texttt{GEMM}} + T_{sq2d}$	T_{heap}	T_{total}
16	0 + 55 + 24 / 20	13 / 1	92 / 21
128	0 + 55 + 24 / 20	16 / 5	95 / 25
512	0 + 55 + 24 / 20	30 / 33	109 / 53
2048	0 + 55 + 24 / 76	52 / 34	131 / 110
MKL+STL / GSKNN		$m = n = 8192$, $d = 64$	
16	1 + 117 + 24 / 52	13 / 1	155 / 53
128	1 + 122 + 24 / 52	15 / 6	162 / 58
512	1 + 113 + 24 / 52	30 / 35	168 / 87
2048	1 + 126 + 24 / 94	52 / 34	203 / 128
MKL+STL / GSKNN		$m = n = 8192$, $d = 256$	
16	3 + 210 + 24 / 186	13 / 2	250 / 188
128	3 + 209 + 24 / 186	15 / 13	251 / 199
512	3 + 211 + 24 / 186	30 / 38	268 / 224
2048	3 + 213 + 24 / 202	52 / 34	292 / 236
MKL+STL / GSKNN		$m = n = 8192$, $d = 1024$	
16	9 + 702 + 24 / 665	13 / 0	748 / 665
128	9 + 734 + 24 / 665	15 / 11	782 / 676
512	9 + 728 + 24 / 665	30 / 40	791 / 705
2048	9 + 735 + 24 / 673	51 / 34	819 / 707

Table 5 *Runtime breakdown analysis (ms). Due to the timer call overhead, T_{heap} can't be measured accurately in Var#1 ($k = 16, 128, 512$); thus, T_{heap} is estimated by the total time difference with the $k = 1$ case. Taking the first row ($k = 16$) of Table 5 as an example,* GSKNN *spends* 21 *ms in total. The estimated heap selection time is computed by $21 - 20 = 1$ where* 20 *is the total execution time of the case $k = 1$. For var#6 ($k = 2048$), T_{heap} is measured separately.*

and that the savings on the square distance evaluations are consistent with the term $T_m^{C^c}$ in our model.

10-core efficiency overview: In Figure 6, we plot the floating point efficiency (GFLOPS) as a function of the problem size m, n, k, and d (Notice the logarithmic scale for the horizontal coordinate). Performance (GFLOPS) increases with problem size m, n and dimension d but degrades with k. This is because larger m, n and d provide a higher parallelism, but neighbor search only contributes to the runtime without performing any floating point operations. To summarize, GSKNN outperforms the GEMM kernel. For m large enough (which the case in approximate search methods), 80% of theoretical peak performance can be achieved in high d for $k \leq 128$. For $k = 2048$, 65% of peak performance can be achieved. GSKNN performs especially well for small $k \leq 128$, where it is up to 5× more efficient than the GEMM kernel for $d \in [10, 100]$, which is commonly found in applications.

5. CONCLUSION

We demonstrate that by fusing the GEMM kernel with the neighbor search we can significantly optimize the performance of the kNN kernel. We discuss blocking, loop reordering, and parameter selection that results in six possible algorithmic variants, which we analyze to derive the best combination. To increase portability, GSKNN follows the BLIS design logic which reduces the architecture dependent part of the algorithm to the micro-kernel level. This makes GSKNN's portability to future x86 architectures less challenging since it only requires changing the block size and rewriting the micro kernel. Our analytical model not only helps explain the results but also enables high-level scheduling and tuning. Taken together, we show that GSKNN can achieve a high efficiency on modern x86 architectures. Ongoing work includes extension to non-Euclidean distance metrics, extensions to GPU architectures, and integration with other higher-level algorithms for clustering and learning.

6. REFERENCES

[1] D. Aiger, E. Kokiopoulou, and E. Rivlin, *Random grids: Fast approximate nearest neighbors and range searching for image search*, in Computer Vision (ICCV), 2013 IEEE International Conference on, IEEE, 2013, pp. 3471–3478.

[2] A. Andoni and P. Indyk, *Near-Optimal Hashing Algorithms for Approximate Nearest Neighbor in High Dimensions*, COMMUNICATIONS OF THE ACM, 51 (2008), pp. 117–122.

[3] J. Chhugani, A. D. Nguyen, V. W. Lee, W. Macy, M. Hagog, Y.-K. Chen, A. Baransi, S. Kumar, and P. Dubey, *Efficient implementation of sorting on multi-core simd cpu architecture*, Proceedings of the VLDB Endowment, 1 (2008), pp. 1313–1324.

[4] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, et al., *Introduction to algorithms*, vol. 2, MIT press Cambridge, 2001.

[5] R. R. Curtin, J. R. Cline, N. P. Slagle, W. B. March, P. Ram, N. A. Mehta, and A. G. Gray, *MLPACK: A scalable C++ machine learning library*, Journal of Machine Learning Research, 14 (2013), pp. 801–805.

[6] S. Dasgupta and Y. Freund, *Random projection trees and low dimensional manifolds*, in Proceedings of the 40th annual ACM symposium on Theory of computing, ACM, 2008, pp. 537–546.

[7] J. J. Dongarra, J. Du Croz, S. Hammarling, and I. S. Duff, *A set of level 3 basic linear algebra subprograms*, ACM Transactions on Mathematical Software (TOMS), 16 (1990), pp. 1–17.

[8] R. W. Floyd, *Algorithm 245: Treesort*, Communications of the ACM, 7 (1964), p. 701.

[9] J. Friedman, T. Hastie, and R. Tibshirani, *The elements of statistical learning*, vol. 1, Springer Series in Statistics, 2001.

[10] V. Garcia, E. Debreuve, and M. Barlaud, *Fast k nearest neighbor search using gpu*, in Computer Vision and Pattern Recognition Workshops, 2008. CVPRW'08. IEEE Computer Society Conference on, IEEE, 2008, pp. 1–6.

[11] K. Goto and R. A. Geijn, *Anatomy of high-performance matrix multiplication*, ACM Transactions on Mathematical Software (TOMS), 34 (2008), p. 12.

[12] GotoBLAS. https://www.tacc.utexas.edu/research-development/tacc-software/gotoblas2.

[13] R. L. Graham, *Bounds for certain multiprocessing anomalies*, Bell System Technical Journal, 45 (1966), pp. 1563–1581.

[14] C. A. R. Hoare, *Algorithm 65: find*, Communications of the ACM, 4 (1961), pp. 321–322.

[15] P. Jones, A. Osipov, and V. Rokhlin, *Randomized approximate nearest neighbors algorithm*, Proceedings of the National Academy of Sciences, 108 (2011), pp. 15679–15686.

[16] A. LaMarca and R. Ladner, *The influence of caches on the performance of heaps*, Journal of Experimental Algorithmics (JEA), 1 (1996), p. 4.

[17] D. Larkin, S. Sen, and R. E. Tarjan, *A back-to-basics empirical study of priority queues.*, in ALENEX, SIAM, 2014, pp. 61–72.

[18] M. Lichman, *UCI machine learning repository*, 2013.

[19] T. M. Low, F. D. Igual, T. Smith, and E. S. Quintana-Orti, *Analytical modeling is enough for high performance BLIS*, ACM under-reviewing, (2014).

[20] L. Moon, D. Long, S. Joshi, V. Tripath, B. Xiao, and G. Biros, *Parallel algorithms for clustering and nearest neighbor search problems in high dimensions*, in 2011 ACM/IEEE conference on Supercomputing, Poster Session, Piscataway, NJ, USA, 2011, IEEE Press.

[21] D. Mount and S. Arya, *ANN: A library for approximate nearest neighbor searching*, in CGC 2nd Annual Fall Workshop on Computational Geometry, 1997. www.cs.umd.edu/~mount/ANN/.

[22] D. M. Mount and S. Arya, *Ann: Library for approximate nearest neighbour searching*, (1998).

[23] M. Muja and D. Lowe, *Scalable nearest neighbour algorithms for high dimensional data*, (2014).

[24] OpenBLAS. http://www.openblas.net/.

[25] H. Samet, *Foundations of multidimensional and metric data structures*, Morgan Kaufmann, 2006.

[26] O. Sinnen, *Task scheduling for parallel systems*, vol. 60, John Wiley & Sons, 2007.

[27] N. Sismanis, N. Pitsianis, and X. Sun, *Parallel search of k-nearest neighbors with synchronous operations*, 2012. http://autogpu.ee.auth.gr/doku.php?id=cuknns:gpu_accelerated_k-nearest_neighbor_library.

[28] T. M. Smith, R. v. d. Geijn, M. Smelyanskiy, J. R. Hammond, and F. G. V. Zee, *Anatomy of high-performance many-threaded matrix multiplication*, in Parallel and Distributed Processing Symposium, 2014 IEEE 28th International, IEEE, 2014, pp. 1049–1059.

[29] F. G. Van Zee and R. A. Van De Geijn, *Blis: A framework for rapidly instantiating blas functionality*, ACM Trans. Math. Softw, (2013).

[30] R. Weber, H. Schek, and S. Blott, *A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces*, in Proceedings of the International Conference on Very Large Data Bases, IEEE, 1998, pp. 194–205.

[31] B. Xiao, *Parallel algorithms for the generalized n-body problem in high dimensions and their applications for Bayesian inference and image analysis*, PhD thesis, Georgia Institute of Technology, Atlanta, GA, USA, 8 2014.