Optimizing Kernel Summation

Chenhan D. Yu, University of Texas at Austin
George Biros, University of Texas at Austin

Abstract is currently empty.

Categories and Subject Descriptors: C.2.2 [**Computer-Communication Networks**]: Network Protocols

General Terms: Design, Algorithms, Performance

Additional Key Words and Phrases: Wireless sensor networks, media access control, multi-channel, radio interference, time synchronization

ACM Reference Format:
Chenhan D. Yu, 2015. Title here. *ACM Trans. Embedd. Comput. Syst.* 9, 4, Article 39 (March 2010), 20 pages.
DOI: http://dx.doi.org/10.1145/0000000.0000000

1. INTRODUCTION

Given a set of $n_{\mathcal{X}}$ points $\{x_i\}_{i=1}^{n_{\mathcal{X}}}$ in a k dimensional space, a kernel $\mathcal{K}(x_i, x_j)$ describes the pairwise interaction between two points. In the viewpoint of linear algebra, computing the kernel summantion u_i is to evaluate the overall interaction of x_i with all other points by a linear combination:

$$u_i = \sum_{j=1}^{n_{\mathcal{X}}} \mathcal{K}(x_i, x_j) w_j, 1 \leq i \leq n_{\mathcal{X}} \tag{1}$$

where $u \in \mathcal{R}^{n_{\mathcal{X}}}$ is overall interaction (potential), and $w \in \mathcal{R}^{n_{\mathcal{X}}}$ is the weight vector. $\mathcal{K}(x_i, x_j)$ is usually stored as a $n_{\mathcal{X}} \times n_{\mathcal{X}}$ dense matrix, and computing (1) for all points becomes a dense matrix-vector multiplication. For simplicity, we use the same notation $\mathcal{K} \in \mathcal{R}^{n_{\mathcal{X}} \times n_{\mathcal{X}}}$ to represent the kernel matrix in the rest of the article, where $\mathcal{K}(i, j) = \mathcal{K}(x_i, x_j)$

The practical challenge of computing (1) with a dense $\mathcal{K}$ is that the time and memory complexity $\mathcal{O}({n_{\mathcal{X}}}^2)$ soon become non-affordable with the increasing $n_{\mathcal{X}}$. To solve this problem, fast kernel summation methods approximate $\mathcal{K}w$ in $\mathcal{O}(n_{\mathcal{X}} \log(n_{\mathcal{X}}))$ or even $\mathcal{O}(n_{\mathcal{X}})$ by exploiting the low rank structure of $\mathcal{K}$, only partially computing $\mathcal{K}$ at runtime if necessary. In physics where $k = 2, 3$, treecodes [], fast multipoles methods [] can scale up to trillions of points by exploring the distribution with domain decomposition schemes. However, the scheme doesn't scale well with a larger k, since the complexity

This work is supported by the National Science Foundation, under grant CNS-0435060, grant CCR-0325197 and grant EN-CS-0329609.
Author's addresses: Chenhan D. Yu, Department of Computer Science, University of Texas at Austin, Austin, TX 78712; George Biros, Institute for Computational Engineering and Sciences, University of Texas at Austin, Austin, TX 78712.

DOI: http://dx.doi.org/10.1145/0000000.0000000

grows exponentially with k. For high dimensional data analysis problems such as kernel regression and kernel classification, schemes are usually linear or superlinear to k for the scalability. For example, ASKIT [] takes an nearest-neighbor pruning and a spatial tree to improve the scalability.

Runtime evaluating instead of reusing $\mathcal{K}$ significantly increases the time complexity especially when k is large. How to compute a dense $\mathcal{K}$ and it's summation $\mathcal{K}w$ efficiently is a new bottleneck for all applications that requires the operation. For example, the floating efficiency of LIBSVM [Chang and Lin 2011] doesn't scale with k, since the kernel value is computed element-wise without taking the advantage of the modern computer architecture. Without the insight of the computer architecture, most of the software written in pure interpreting or partially compiled language are usually running with in 3% of the CPU peak performance. Even a pure C/Fortran program, 10% is usually the average. One of the solution is to take the advantage of the high performance level 3 BLAS (Basic Linear Algebra Subroutines) to compute a submatrix of $\mathcal{K}$. $\mathcal{K}(x_i, x_j)$ is usually a function of the square distance $\|x_i - x_j\|_2^2$, and its expansion (2) is mainly a pairwise inner-product $x_i^T x_j$.

$$\|x_i - x_j\|_2^2 = \|x_i\|_2^2 + \|x_j\|_2^2 - 2x_i^T x_j \tag{2}$$

With expression (2), computing $\mathcal{K}$ can rely on a high performance matrix-matrix multiplication (GEMM) which can usually reach more than 80% peak performance on the modern CPU architectures with a large k. Level 3 BLAS routines such as GEMM are highly optimized by domain experts, and the interfaces are standardized to facilitate users. The kernel value can be accelerated by vectorized math functions (e.g. Intel Vectorized Math Library), and $\mathcal{K}w$ can be computed by GEMV (General Matrix-Vector Multiplication) which is also a level 2 subroutine of BLAS. CUKNN [Liang et al. 2009] computes pairwise distance with (2) to accelerate searching for k-nearest neighbors, and the same approach is taken in ASKIT [March et al. 2014] [March et al. 2015] a treecode approach for fast high dimensional kernel summation in both near-field and far-field evaluation.

Although the BLAS approach works reasonably well on large k, yet there are still several drawbacks remaining unsolved.

(1) The BLAS approach transform the low dimensional case of (1) from a computation bound problem to memory bound.
(2) Since GEMM is a memory bound problem for small k. Thus, GEMM needs a large enough k to reach high performance, yet most of the practical problems have $k \leq 32$.
(3) To compute a subproblem of (1), coordinates need to be collected to form dense $\mathcal{A}$ and $\mathcal{B}$ in order to use GEMM, requiring extra memory space.
(4) The intermediate results $-2x_i^T x_j$ need to be stored as a dense matrix $\mathcal{C}$ which requires extra memory space.
(5) The temporary spaces $\mathcal{A}$, $\mathcal{B}$ and $\mathcal{C}$ are also accompanied with extra memory access, suffering from a serious penalty.
(6) GEMV is a memory bound operation which can hardly reach the peak floating point performance.

To summarize the BLAS approach, the standardized interface of BLAS limits the possibility of combining different operations. A new BLAS like subroutine to compute the kernel summation is inspired by the drawbacks listed above, combining GEMM, GEMV and vectorized math functions together to exploit the modern CPU architecture.

contributions: we introduce GSKS (General Stride Kernel Summation), a kernel summation routine that takes general stride access of a k dimensional coordinates table as its input. We introduce the algorithm, implementation details of GSKS, modeling the performance, analyzing the difference between GSKS and the BLAS approach. In particular:

— we present a GSKS algorithm inspired by the BLIS framework which minimizes the architecture dependent part to increase the portability;
— we first standardize the interface of GSKS to support various storage formats;
— we model and analyze the performance of both GSKS and the BLAS approach to explain our experiment results;
— we provide two different parallel schemes to exploit the computational resources of a shared memory system;
— we derive the roundoff error bound of GSKS to prove the accuracy and the numerical stability.

Our key innovations are the new embedded micro-kernel that maximizes the memory reuse rate and a flexible packing routine that can directly pack the memory from the coordinates table to avoid duplicating the memory. Further more, we optimize the vectorized math functions which provides competent performance against vendor's implementation (Intel VML). In all, GSKS achieves a high arithmetic intensity, algorithmic optimality and portability.

The rest of the article is arranged as follow: in §2, we first define all the notations we are going to use in the algorithms, equations and analysis. We also give an algorithm that describes the BLAS approach in details; in §2.1, we present the algorithm design of GSKS; in §2.3, a analytic model is built to predict and compare the peak performance of GSKS and the BLAS approach; in §2.4, we give details about the implementation of the embedded micro-kernel, kernel function approximation and the parameters choosing; in §2.5, we give a task parallelism and a data parallelism scheme to compute GSKS efficiently on a shared memory system; in §2.6, we derive the error bound of GSKS, and discuss the numerical stability; in §3 and §4, we present the experiment setup and the result of the performance.

2. METHODS

Here we present the basic structure of GSKS, modeling the performance, analyzing the accuracy. First, we present the sequential algorithm in §2.1. Then a theoretical performance model will be introduced in §2.3 to compare different implementations. With the performance prediction, we later illustrate how to implement the kernel summation micro-kernel and choose the parameters to achieve a high performance rank-k update and a kernel function in §2.4. A data parallel and a task parallel implementations are presented in §2.5. The polynomial approximate accuracy and the numerical stability are discussed in §2.6.

Notation: In the following, $\mathcal{X} \in \mathcal{R}^{k \times n_\mathcal{X}}$ is a k dimensional coordinate table containing $n_\mathcal{X}$ data points. $\mathcal{X}_A$ and $\mathcal{X}_B$ represent the target and the source coordinate tables which are subsets of $\mathcal{X}$. $x_i = \mathcal{X}_A(:, i)$ is the i-th target coordinate of $\mathcal{X}_A$, and $x_j = \mathcal{X}_B(:, j)$ is the j-th source coordinate of $\mathcal{X}_B$. $\|\mathcal{X}_A\|_2^2$ and $\|\mathcal{X}_B\|_2^2$ are the square 2-norm of each point in $\mathcal{X}_A$ and $\mathcal{X}_B$, where $\|\mathcal{X}_A\|_2^2(i) = \|x_i\|_2^2$ and $\|\mathcal{X}_B\|_2^2(j) = \|x_j\|_2^2$. We use $\mathcal{A}_s \in \mathcal{R}^{k \times m_s}$ and $\mathcal{B}_s \in \mathcal{R}^{k \times n_s}$ to describe the s-th subsets of $\mathcal{X}_A$ and $\mathcal{X}_B$. α_s and β_s are integer maps that describe the index relation where $\mathcal{A}_s(:, i) = \mathcal{X}_A(:, \alpha_s(i))$ and $\mathcal{B}_s(:, j) = \mathcal{X}_B(:, \beta_s(j))$. We also define the s-th subsets of w and u as w_s and u_s accordingly. $u_s = u(\alpha_s)$ shares the same map as $\mathcal{A}_s$, yet w_s doesn't necessary to share the

ALGORITHM 1: Kernel Summation with GEMM, VEXP and GEMV

$\mathcal{X}_A(:,\alpha) \rightarrow \mathcal{A}_s$, $\|\mathcal{X}_A\|_2^2(\alpha) \rightarrow \|\mathcal{A}_s\|_2^2$, $u(\alpha) \rightarrow u_s$;
$\mathcal{X}_B(:,\beta) \rightarrow \mathcal{B}_s$, $\|\mathcal{X}_B\|_2^2(\beta) \rightarrow \|\mathcal{B}_s\|_2^2$, $w(\omega) \rightarrow w_s$;
GEMM$(\mathcal{A}_s, -2.0, \mathcal{B}_s, 0.0, \mathcal{C}_s)$;
for *j=0:n-1* **do**
  for *i=0:m-1* **do**
    $\mathcal{C}_s(i,j) += \|\mathcal{A}_s\|_2^2(i) + \|\mathcal{B}_s\|_2^2(j)$;
    $\mathcal{C}_s(i,j) *= (-1/2h^2)$;
  end
end
VEXP$(\mathcal{C}_s)$;
GEMV$(\mathcal{C}_s, 1.0, w_s, 1.0, u_w)$;
$u_s \rightarrow u(\alpha)$;

same map as $\mathcal{B}_s$. For example, ASKIT creates skeleton weights $\tilde{w}$ for approximation; thus, here we use $w_s = w(\omega_s)$ to take care this special situation.

Given the notation above, (1) can be approximate by (3), and GSKS is designed a solve a dense kernel summation.

$$u = \sum_s u_s = \sum_s \mathcal{K}(\alpha_s, \beta_s) w(\omega_s) \tag{3}$$

Each elements of $\mathcal{K}(\alpha_s, \beta_s)$ is usually a function of the square distance $\|x_i - x_j\|_2^2$ where $x_i \in \mathcal{A}_s$ and $x_j \in \mathcal{B}_s$. To be more concise, we take Gaussian kernel as an example for the rest of the article, and the kernel function is written as:

$$\mathcal{K}(x_i, x_j) = \exp(-\|x_i - x_j\|_2^2/(2h^2)) \tag{4}$$

where h is the width of the kernel. The square distance can be evaluated directly or by (2). Precomputing $\|x_i\|_2^2 = \|\mathcal{X}_A\|_2^2(i)$ and reusing the results of $\|\mathcal{X}_A\|_2^2$ and $\|\mathcal{X}_B\|_2^2$ requires many fewer FMA (Fused Multiply Add) operations than evaluating $\|x_i - x_j\|_2^2$ directly. Moreover, $-2x_i^T x_j$, (4) and (3) can be computed by GEMM, GEMV and VEXP (vectorized exponential function) which provides an opportunity to achieve excellent performance on modern CPU architectures.

The GEMM, VEXP and GEMV combination approach is widely used in the modern kernel methods to achieve high a performance with the BLAS approach. We summarize the combination approach in Algorithm 1 by using the same notations we just defined. The drawback of this approach is that $\mathcal{A}_s$, $\mathcal{B}_s$ and $\mathcal{C}_s$ need to be formed explicitly in order to use GEMM, VEXP and GEMV due to the standardized BLAS interface. $\mathcal{A}_s$ and $\mathcal{B}_s$ are created to collect points from $\mathcal{X}_A$ and $\mathcal{X}_B$, since GEMM only takes contiguous or uniform stride inputs. $\mathcal{C}_s$ must be created to output the result of GEMM, and it's also required by VEXP and GEMV. These temporary spaces are redundant, and the extra memory accesses are accompanied. Inspired by the redundant memory operation, we develop a BLAS like subroutine which embeds Algorithm 1 into a micro-kernel which may avoid these redundant memory allocations and operations.

2.1. Sequencial General Stride Kernel Summation

We first present the pseudo-code of GSKS with Gaussian kernel in Algorithm 2 for computing a subproblem of (3). Other than GSKS, GEKS is a case of GSKS stands for the general storage version where $\alpha(i) = i$, $\beta(j) = j$ and $\omega(j) = j$. The algorithm contains 6 layers of loops which are corresponding to different partitioning of m, n and k. The partitioning scheme is similar to the GEMM implementation in BLIS [Van Zee and Van

De Geijn 2013] and GotoBLAS [Goto and Geijn 2008] which is designed for memory packing, reuse and alignment.

Starting from the outer most loop, the 6.th loop (j_c) partitions n loop with the block size n_c. The 5.th loop (p_c) partitions k loop with the block size k_c. The 4.th loop (i_c) partitions m loop with the block size m_c. Surrounded by three nested loops and several packing routines, the macro-kernel contains the 3.rd, 2.nd loops (j_r and i_r). The micro-kernel contains three parts: the 1.st loop (p_r) for the rank-k_c update, the kernel function and a $m_r \times n_r$ kernel summation.

A good way to understand the features of GSKS is to identify the scope of each function GEMM, VEXP and GEMV. By removing all statements inside the condition ($p_c + k_c \geq k$), the remaining parts of Algorithm 2 are a matrix-matrix multiplication. The only difference is that GSKS can take non-uniform stride inputs indicated by index maps α and β. On the other hand, VEXP and GEMV will only be executed in the last iteration of the 5.th loop (p_c), embedded inside the GEMM algorithm, taking the same partitioning on each dimension. Therefore, for different k, GSKS may be executed in two different modes. In the general case $k > k_c$ where k is partitioned into more than one block, GSKS will only perform rank-k_c update if it's not the last iteration of the 5.th loop. A temporary buffer C_c is required to accumulate the rank-k_c update which requires the same amount of memory operations as GEMM does. In the special case that $k \leq k_c$, no temporary buffer is required, so GSKS can achieve a lower memory complexity.

Other than $\mathcal{X}_A$, $\mathcal{X}_B$, $\|\mathcal{X}_A\|_2^2$, $\|\mathcal{X}_B\|_2^2$, u and w, Algorithm 2 creates $\mathcal{A}_c$, $\mathcal{B}_c$, $\|\mathcal{A}_c\|_2^2$, $\|\mathcal{B}_c\|_2^2$, u_c and w_c as temporary buffers which will partially duplicate and rearrange the data. The process is generally called memory packing, and here we use $\rightarrow$ to represent this memory operation. Starting from the 5.th loop, every n_c points will be pulled out from the coordinates table $\mathcal{X}_B$, and every time k_c dimensions will be packed into $\mathcal{B}_c$. Followed by the 4.th loop, every m_c points will be pulled out from $\mathcal{X}_A$, and the same k_c dimensions will be packed into $\mathcal{A}_c$. Similarly, $\|\mathcal{A}_c\|_2^2$ and u_c contain m_c entries of $\|\mathcal{X}_A\|_2^2$ and u. $\|\mathcal{B}_c\|_2^2$ and w_c contain n_c entries of $\|\mathcal{X}_B\|_2^2$ and w. Packing memory is not only designed for fitting the data into the fast memory but also for consecutive memory access and memory alignment inside the micro-kernel. Figure 1 shows that how coordinates are rearranged into several long "Z" shape micro-panels which are exactly how the data will be accessed in Algorithm 2. Despite that $\mathcal{C}_c$ is seldom packed in the modern GEMM implementation due to the overhead, $\mathcal{C}_c$ in GSKS is also stored in the "Z" shape to help stabilize the performance. However, we don't use the phrase "packed" on $\mathcal{C}_c$, since we are not reordering $\mathcal{C}_c$ from other patterns. In GSKS, coordinates are collected from the stride coordinate tables, directly packed into the desired format $\mathcal{A}_c$, $\mathcal{B}_c$. This general stride packing routine frees GSKS from storing $\mathcal{A}_s$ and $\mathcal{B}_s$ which are required in Algorithm 1.

Figure 2 provides the insights of how an expert will design GSKS to match the memory hierarchy in the modern CPU architectures. The partitioning scheme slices the data to fit into different memory hierarchies, and the order of the loops is designed to maximize data reuse in caches based on the LRU (Least Recent Used) replacing policy. For example, the size of a micro-panel of $\mathcal{B}_c$ is $n_r \times k_c$, where k_c is typically chosen to fit the micro-panel into the L1 cache. The reuse rates of $\mathcal{A}_c$ and $\mathcal{B}_c$ are mostly decided by the order of the 2.nd loop and the 3.rd loop in the macro-kernel. The $\mathcal{B}_c$ micro-panel is reloaded at the 3.rd loop which is designed to have a longer lifetime than the $\mathcal{A}_c$ micro-panel. This makes sure that each time reloading $\mathcal{A}_c$ in the 2.nd loop will replace the original $\mathcal{A}_c$ micro-panel but keep the $\mathcal{B}_c$ micro-panel inside the L1 cache.

ALGORITHM 2: Blocked kernel summation with a general stride storage (GSKS)

```
                                                  blocked partitioning (4, 5, 6th loops, i_c, p_c and j_c)
for j_c = 0 : n_c : n − 1 do
    for p_c = 0 : k_c : k − 1 do
        X_B(p_c : p_c + k_c − 1, β(j_c : j_c + m_c − 1)) → B_c;
        if p_c + k_c ≥ k then
            ‖X_B‖_2^2(β(j_c : j_c + m_c − 1)) → ‖B_c‖_2^2;
            w(ω(j_c : j_c + m_c − 1)) → w_c;
        end
        for i_c = 0 : m_c : m − 1 do
            X_A(p_c : p_c + k_c − 1, α(i_c : i_c + m_c − 1)) → A_c;
            if p_c + k_c ≥ k then
                ‖X_A‖_2^2(α(i_c : i_c + m_c − 1)) → ‖A_c‖_2^2;
                u(α(i_c : i_c + m_c − 1)) → u_c;
            end
                                                  macro-kernel (2.nd, 3.rd loops, i_r and j_r)
            for j_r = 0 : n_r : n_c − 1 do
                for i_r = 0 : m_r : m_c − 1 do
                                                  micro-kernel (1.st loop, p_r)
                    for p_r = 0 : k_c − 1 do
                        #Loop unrolling m_r-by-n_r;
                        for j = 0 : n_r − 1 do
                            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) −= 2A_c(i_r + i, p_r)B_c(j_r + j, p_r);
                                C_c(i_c + i_r + i, j_c + j_r + j) = C_r(i, j);
                            end
                        end
                    end
                    if p_c + k_c ≥ k then
                        #Loop unrolling m_r-by-n_r;
                        for j = 0 : n_r − 1 do
                            for i = 0 : m_r − 1 do
                                C_r(i, j) += ‖A_c‖_2^2(i_r + i) + ‖B_c‖_2^2(j_r + j);
                                C_r(i, j) = exp(C_r(i, j)/(2h^2));
                                u_c(i_r + i) += C_r(i, j)w_c(j_r + j);
                            end
                        end
                    end
                end
            end
            if p_c + k_c ≥ k then
                u_c → u(α(i_c : i_c + m_c − 1));
            end
        end
    end
end
```

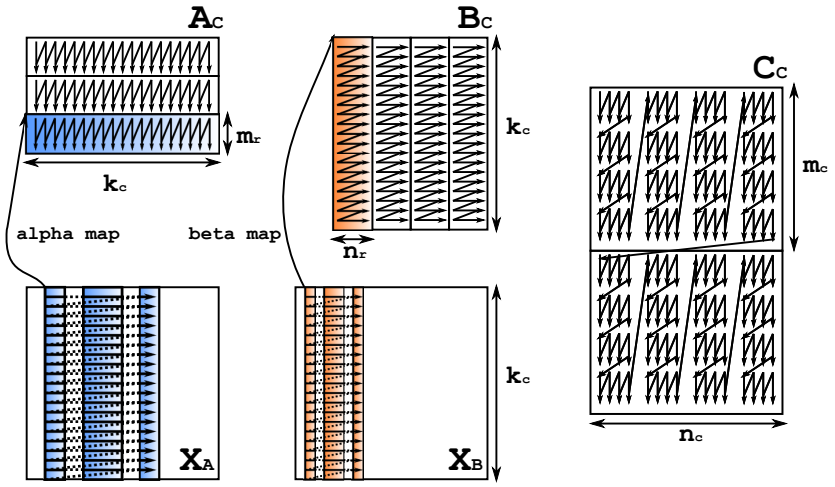

Fig. 1: GSKS *stride packing scheme: data in the stride coordinate table is accessed via* α *and* β *maps.* $\mathcal{A}_c$ *is composed by several* $m_r \times k_c$ *folding micro-panels, and* $\mathcal{B}_c$ *is composed by several* $n_r \times k_c$ *micro-panels. The arrow indicates the contiguous memory direction which is also the sequential accessing order of the micro-panel.*

Inside the micro-kernel, an $m_r \times n_r$ tile of rank-k_c update or a kernel summation will be computed each time depending on whether this is the last iteration of the 5.th loop (p_c). The rank-k_c update, kernel functions and the matrix-vector multiplication are all vectorized in order to take the advantage of the SIMD (Single Instruction Multiple Data) architecture. We will explain how the kernel function ($\exp()$ as an example) can be vectorized by a polynomial approximation, and the implementing details of the micro-kernel will be discussed in §2.4.

2.2. Remez Polynomial Approximation

Despite the cost of computing the kernel value is constant, yet it can be significant if k is small. Fast math functions including polynomial approximation and table lookup are widely used instead of those standard or intrinsic math functions with a high accuracy, since fast methods are usually easier to be vectorized. Although fast math functions are not bitwise accurate as the intrinsic math functions, yet the precision is usually acceptable in the N-body problems and data analysis. In GSKS, we choose a polynomial approximation which allows us to vectorize the $\exp()$ function. Instead of Taylor expansion, we choose Remez exchange algorithm to compute the polynomial coefficients which usually requires a lower polynomial order to reach the same error criteria.

Remez exchange algorithm is an iterative method that try to minimize the maximum absolute difference between the polynomial function and the target function. In GSKS especially, we are looking for an order 11 polynomial $P_{11}(x)$ to approximate the $\exp()$ function where $x \in [0 \log(2)]$. This order is high enough for IEEE 754 double precision, and the absolute error is controlled with in the magnitude of the machine epsilon (2.22E-16) . $\exp(x)$ is evaluated in the form of $\exp(a \ln 2 + b) = 2^a \times \exp(b)$, where a is an integer and $b \in [0, \ln 2]$. In this case, we only need to approximate $\exp(b)$, and 2^a can be computed by shifting. The algorithm usually starts from $(11 + 2)$ Chebyshev nodes of the interval as sample points. In each iteration, a 13×13 linear system is solved to get the coefficients, and a new set of points of absolute local maximum error $|P_{11}(b) - \exp(b)|$ is found with the coefficients. If all local maximum errors meet the

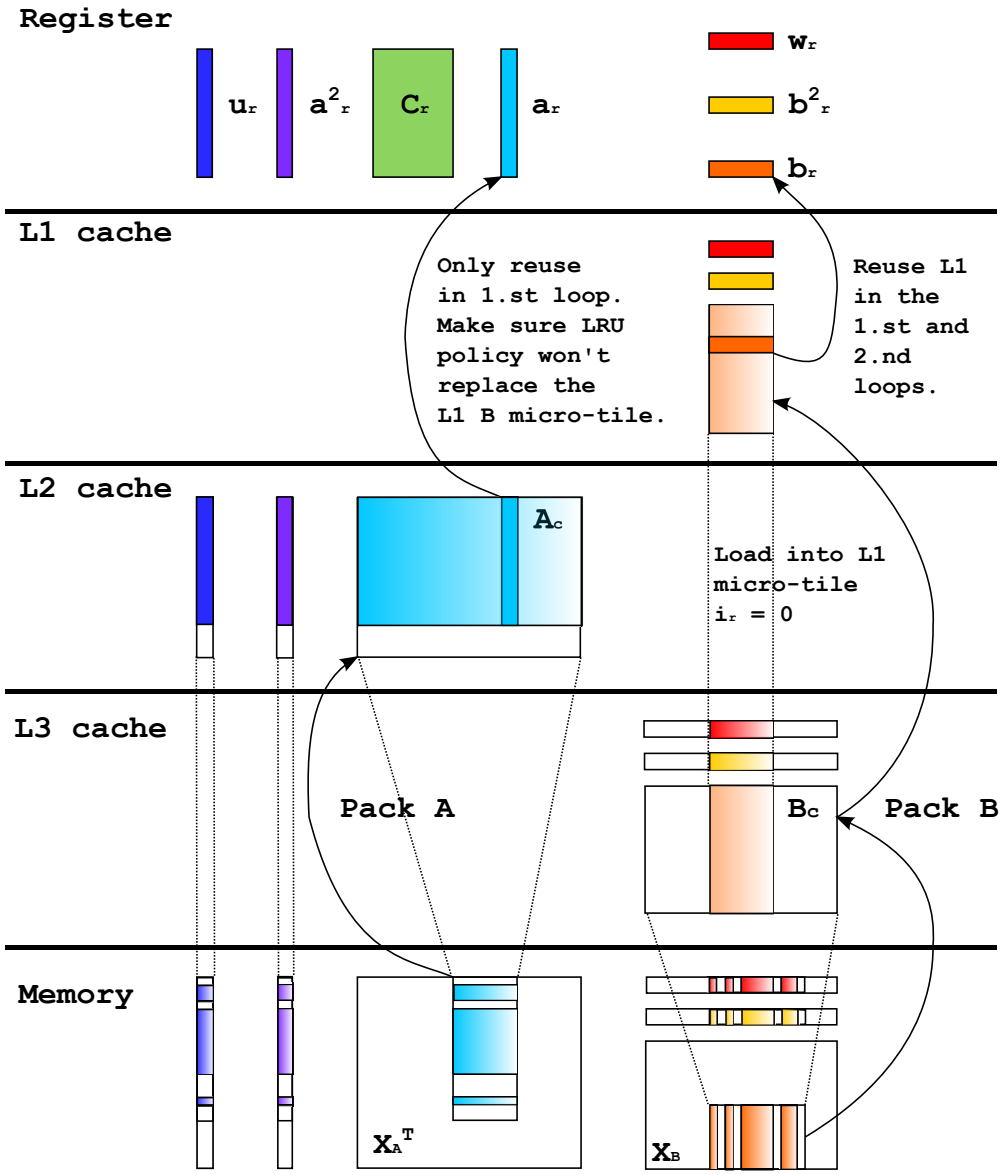

Fig. 2: *The figure maps* GSKS (2) *to the memory hierarchy. From the bottom, $\mathcal{X}_A$ and $\mathcal{X}_B$ are packed from the memory directly into $\mathcal{A}_c$ and $\mathcal{B}_c$. According to the reuse rate, one or two micro-panels of $\mathcal{B}_c$ will be preserved in the L1 cache, the whole $\mathcal{A}_c$ in the L2 cache, and the whole $\mathcal{B}_c$ will be preserved in the L3 cache.*

magnitude of the stopping criteria (usually the magnitude of the machine epsilon) and alternate in sign, then the process stops. Otherwise, the new set of points will replace the old one in the new iteration. In GSKS, we use Lol Remez Exchange Toolbox (http://lolengine.net/wiki/oss/lolremez), a high precision implementation of Remez algorithm, to find coefficients of the order 11 polynomial, and the output coefficients precision is 20 digits in decimal.

	type	Algorithm 1	Algorithm 2 ($k \leq k_c$)	Algorithm 2 ($k > k_c$)
T_f	-	$\frac{1}{\tau}(2k+35)m_s n_s$	$\frac{1}{\tau}(2k+35)m_s n_s$	$\frac{1}{\tau}(2k+35)m_s n_s$
T_o	-	$\frac{1}{\tau}30m_s n_s$	$\frac{1}{\tau}30m_s n_s$	$\frac{1}{\tau}30m_s n_s$
$T_d^{\mathcal{X}_A}$	r	$\phi k\lceil\frac{n_s}{n_c}\rceil m_s$	$\phi k\lceil\frac{n_s}{n_c}\rceil m_s$	$\phi k\lceil\frac{n_s}{n_c}\rceil m_s$
$T_d^{\mathcal{A}_c}$	w	$\phi k\lceil\frac{n_s}{n_c}\rceil m_s$	$\phi k\lceil\frac{n_s}{n_c}\rceil m_s$	$\phi k\lceil\frac{n_s}{n_c}\rceil m_s$
$T_d^{\|\mathcal{X}_A\|_2^2}$	r	$\phi\lceil\frac{n_s}{n_c}\rceil m_s$	$\phi\lceil\frac{n_s}{n_c}\rceil m_s$	$\phi\lceil\frac{n_s}{n_c}\rceil m_s$
$T_d^{\|\mathcal{A}_c\|_2^2}$	w	$\phi\lceil\frac{n_s}{n_c}\rceil m_s$	$\phi\lceil\frac{n_s}{n_c}\rceil m_s$	$\phi\lceil\frac{n_s}{n_c}\rceil m_s$
T_d^{α}	r	$2\phi\lceil\frac{n_s}{n_c}\rceil m_s$	$2\phi\lceil\frac{n_s}{n_c}\rceil m_s$	$2\phi\lceil\frac{n_s}{n_c}\rceil m_s$
T_d^{u}	r/w	$2\phi\lceil\frac{n_s}{n_c}\rceil m_s$	$2\phi\lceil\frac{n_s}{n_c}\rceil m_s$	$2\phi\lceil\frac{n_s}{n_c}\rceil m_s$
$T_d^{u_c}$	r/w	$2\phi\lceil\frac{n_s}{n_c}\rceil m_s$	$2\phi\lceil\frac{n_s}{n_c}\rceil m_s$	$2\phi\lceil\frac{n_s}{n_c}\rceil m_s$
$T_d^{\mathcal{X}_B}$	r	$\phi k n_s$	$\phi k n_s$	$\phi k n_s$
$T_d^{\mathcal{B}_c}$	w	$\phi k n_s$	$\phi k n_s$	$\phi k n_s$
$T_d^{\|\mathcal{X}_B\|_2^2}$	r	ϕn_s	ϕn_s	ϕn_s
$T_d^{\|\mathcal{B}_c\|_2^2}$	w	ϕn_s	ϕn_s	ϕn_s
T_d^{β}	r	ϕn_s	ϕn_s	ϕn_s
T_d^{w}	r	ϕn_s	ϕn_s	ϕn_s
$T_d^{w_c}$	w	ϕn_s	ϕn_s	ϕn_s
T_d^{ω}	r	ϕn_s	ϕn_s	ϕn_s
$T_d^{C_c}$	r/w	$2\phi(\lceil\frac{k}{k_c}\rceil - 1)m_s n_s$	-	$2\phi(\lceil\frac{k}{k_c}\rceil - 1)m_s n_s$
$T_d^{A_s}$	r/w	$2\phi k m_s$	-	-
$T_d^{B_s}$	r/w	$2\phi k n_s$	-	-
$T_d^{C_s}$	r/w	$4\phi m_s n_s$	-	-

Table I: *Theoretical breakdown analysis of Algorithm 1 and Algorithm 2*

2.3. Performance Model

We present a model to predict the execution time T and the peak floating point operation efficiency (Gflops) of Algorithm 2. The theoretical performance model can be used for performance debugging, optima scheduling in the task parallelism scheme, and it can also be used to contrast the difference between Algorithm 1 and Algorithm 2. The model assumes the system with a small fast on-chip memory and a big slow off-chip memory. All data are originally residing in the slow memory, and any floating point operation can only be performed on those data inside the fast memory. The cost of moving the data from the slow memory to the fast memory is counted, but the cost of accessing the fast memory is neglected in this model. Fast memory access can be overlapped by ALU (arithmetic logic unit) operations which can be achieved by inserting the prefetch instruction properly in the modern CPU architectures. However, the latency generated by loading the slow memory can't be hidden, and these memory operations are presented explicitly in Algorithm 2 as several packing routines. Based on the assumption above, the whole sequential computation time is mainly contributed by a serial of slow memory operations and the floating point operations.

Notation: Except for the notations introduced in §2, we further define τ, ϕ, T, T_f, T_d and T_o in the model. τ is the maximum amount of floating point operations that the system can perform in one second, and ϕ denotes the time (second) of a unit data movement between the slow memory and the fast memory. T_f and T_d are the time used in floating operations and data movement, and T_o is the total time spending on other instructions, and $T = T_f + T_d + T_o$ is the total time.

In Table I, we list the theoretical breakdown cost of each component. Starting from the floating point operation time T_f, we give details about T_o and each individual term of T_d. The floating point operation time of the s-th subproblem in (3) can be computed

by the following formula:

$$T_f = \frac{1}{\tau}(2k + 35)m_s n_s \tag{5}$$

where $2m_s n_s k$ is the complexity of the rank-k update, and $35m_s n_s$ is the floating point operations required by (2) and the order 11 polynomial approximation of (4). There are some non-floating point operations required by the rank-k update and the polynomial approximation. For example, the butterfly permutation inside the rank-k kernel requires `permute`, `blend`, `shuffle` instructions, and the exponent shifting and scaling inside the vertorized $\exp()$ function requires `max`, `min`, `cmp`, `slli` and other instructions. We use (6) to estimate the total time spending on these instructions in Algorithm 2, and we assume that Algorithm 1 requires the same amount of T_o, since we may not know how `GEMM`, `VEXP` and `GEMV` are implemented in Algorithm 1.

$$T_o = \frac{1}{\tau}30m_s n_s \tag{6}$$

The times in (5) and (6) are computed by dividing the total floating point operation count (or instruction throughput count) with the theoretical peak floating point operation throughput (τ) per second.

The total time of the data movement not only varies with the problem size m, n and k but also relates to the block size m_c, k_c and n_c. Table I contains all data movement cost, yet in the model we consider the read/write overlapping capability that a independent read and write can be overlapped. Thus, the readers may find some of the terms are only half to the value shown in Table I in the following data movement calculation. Notice that in the beginning of the section, we only consider the cost of loading the data from the slow memory to the fast memory, yet storing to slow memory is only considered when it can't be overlapped. According to the order of the 4.th and the 6.th loop in Algorithm 2, $\mathcal{A}_c$, $\|\mathcal{A}_c\|_2^2$ and u_c must be reloaded from the slow memory $\lceil \frac{n}{n_c} \rceil$ times due to the 6.th loop (j_c). On the other hand, $\mathcal{B}_c$, $\|\mathcal{B}_c\|_2^2$ and w_c will only be loaded once, reused in the 4.th loop (i_c) for $\lceil \frac{m}{m_c} \rceil$ times. Whether or not Algorithm 2 needs to load and store C_c depends on the size of k. We separate the data movement time T_d into two cases: $T_d^{k \le k_c}$ and $T_d^{k > k_c}$ which result in one term difference on $\mathcal{C}_c$. We count the total amount of memory we pack and unpack in Algorithm 2, yet an concurrent load and store will only be counted once. The data movement cost for the s-th subproblem can be computed by the following two formula:

$$T_d^{k \le k_c} = \phi(n_s k + 4n_s + m_s k \left\lceil \frac{n_s}{n_c} \right\rceil + 5m_s \left\lceil \frac{n_s}{n_c} \right\rceil) \tag{7}$$

$$T_d^{k > k_c} = \phi(n_s k + 4n_s + m_s k \left\lceil \frac{n_s}{n_c} \right\rceil + 5m_s \left\lceil \frac{n_s}{n_c} \right\rceil + m_s n_s \left\lceil \frac{k}{k_c} \right\rceil) \tag{8}$$

The first part $n_s k + 4n_s$ in (7) and (8) is contributed by packing $\mathcal{B}_c$, $\|\mathcal{B}_c\|_2^2$ and w_c. $n_s k$ is resulted from loading data from $\mathcal{X}_B$ and storing to $\mathcal{B}_c$ which can be overlapped. $4n_s$ includes loading β, ω, $\|\mathcal{X}_B\|_2^2$, w and storing $\|\mathcal{B}_c\|_2^2$, w_c. Loading $\|\mathcal{X}_B\|_2^2$ can be overlapped with storing $\|\mathcal{B}_c\|_2^2$, and loading w can be overlapped with storing w_c, The second part $m_s k \left\lceil \frac{n_s}{n_c} \right\rceil + 5m_s \left\lceil \frac{n_s}{n_c} \right\rceil$ is contributed by packing and unpacking $\mathcal{A}_c$, $\|\mathcal{A}_c\|_2^2$ and u_c which need to be redone in the 6.th loop. $m_s k \left\lceil \frac{n_s}{n_c} \right\rceil$ is resulted from loading data from $\mathcal{X}_A$ and storing to $\mathcal{A}_c$. $5m_s \left\lceil \frac{n_s}{n_c} \right\rceil$ includes loading α twice, $\|\mathcal{X}_A\|_2^2$, u, u_c and storing $\|\mathcal{A}_c\|_2^2$,

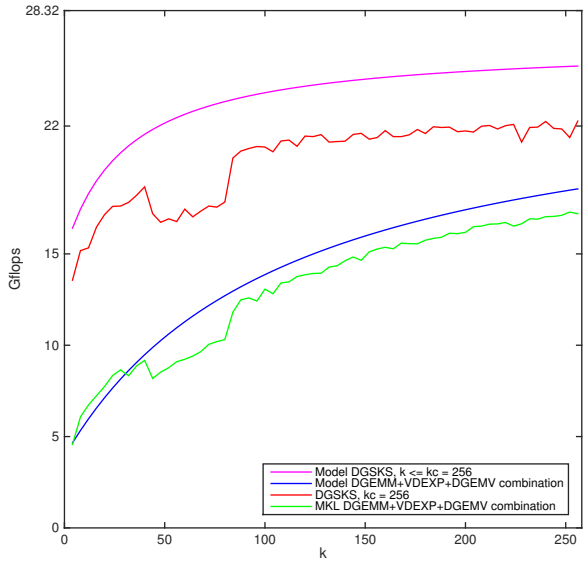

Fig. 3: *We compare the modeling performance with the experimental performance of Algorithm 1 and the special case of Algorithm 2 where $k \leq k_c$. The problem is fixed with $m = 4096$, $n = 512$ and differed in k from 4, 8 to 256. The two smooth curves are the predicted performance using $\tau = 3.54 \times 8$ and $\phi = 2.2$, and the plots with tooth are experimental performance collected from a Intel Xeon E5-2680 v2 (Ivy-bridge) processor with DDR3-1866 (13-13-13) DRAM.*

u, u_c. $\|\mathcal{X}_A\|_2^2$ can be overlapped with $\|\mathcal{A}_c\|_2^2$, and u is overlapped with u_c. If the case of $k > k_c$, an additional term $\phi(m_s n_s \lceil \frac{k}{k_c} \rceil)$ is introduced as an additional cost of loading and storing $\mathcal{C}_c$. Here we only also consider storing $\mathcal{C}_c$ can be overlapped with loading $\mathcal{C}_c$; thus we only count it once. The total time $T = T_f + T_d + T_o$ can be computed by summing up (5), (6) and (7) or (8). And the floating point operation capability per second can be computed by the following formula:

$$\frac{m_s n_s (2k + 35)}{T} \tag{9}$$

Algorithm 1 has the same floating point operations complexity, yet the memory complexity is higher. By assuming that `GEMM`, `VEXP` and `GEMV` in Algorithm 1 are using the same partitioning size as Algorithm 2 does, the data movement time can be predicted with the base term T_d with some extra memory operations summarized in (10).

$$T_d^{Algorithm\ 1} = T_d^{k>k_c} + \phi(m_s k + n_s k + 2 m_s n_s) \tag{10}$$

The additional part $\phi(m_s k + n_s k)$ is contributed by reading $\mathcal{A}_s$ and $\mathcal{B}_s$ from $\mathcal{X}_A$ and $\mathcal{X}_B$, and $\phi(2 m_s n_s)$ is the cost of accessing the intermediate results $\mathcal{C}_s$. `GEMM` will first store $\mathcal{C}_s$ back to the memory. `VEXP` will load and store $\mathcal{C}_s$ once, and `GEMV` needs to load $\mathcal{C}_s$ to complete the kernel summation. We consider the load and store of $\mathcal{A}_s$, $\mathcal{B}_s$ and $\mathcal{C}_s$ are all concurrent; thus, the time is only a half to the time shown in the table.

Figure 3 compares the floating point operations efficiency between `GSKS` and the MKL combination with different $k < k_c$. The purple line is the predicted efficiency of

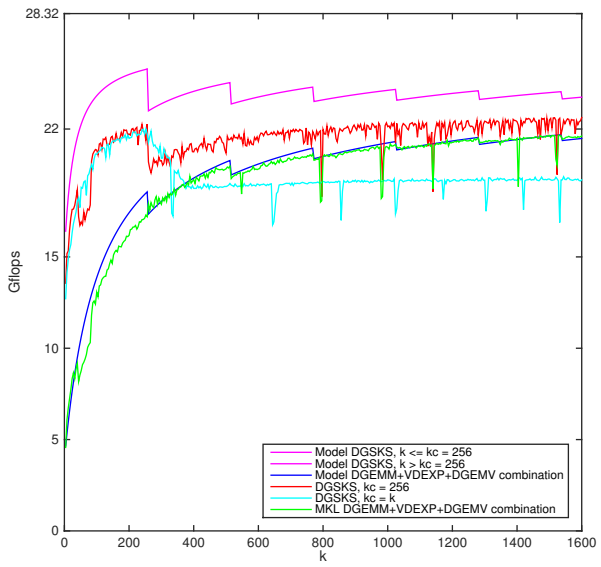

Fig. 4: *In this figure we keep the same configuration chosen in Figure 3 and expand k to 1600.*

GSKS on an Intel Ivy-bridge processor using (9), and the blue line is the predicted efficiency of Algorithm 1 using MKL and VML. The ceiling of Figure 3 is labeled with 28.32 which is the theoretical peak performance of this Intel processor, and this number is derived by multiplying it's stable clockrate (3.54 GHz) with 8 which is the flops of a length-4 vectorized FMA (VFMA) instruction. In the real experiment shown with the red green lines, we show that the prediction is usually overestimating the efficiency, but we confirm that the saving on the memory complexity is beneficial. The model suggests that the efficiency of GSKS should dominate Algorithm 1, since the additional memory operations resulted from the intermediate result $\mathcal{C}_s$ is unrelated to k which leads to a relatively huge performance penalty if k is small. Explicitly forming $\mathcal{A}_s$, $\mathcal{B}_s$ and $\mathcal{C}_s$ is also not practical, since the memory requirement may become non-affordable. To briefly summarize the performance model, we can find that the BLAS approach is actually a memory bound operation for small k, yet GSKS is still a computation bound operation.

Figure 4 provides an efficiency overview of GSKS. Once k goes over k_c, Algorithm 2 will start to load and store the intermediate rank-k_c update results in $\mathcal{C}_c$ which significantly increases the memory complexity shown in (8). In the model prediction, we can observe the spike in every k_c stride on the purple and the blue lines, since the memory operations on $\mathcal{C}_c$ are increasing every k_c. We can also observe the importance of blocking k_c from the performance drop shown by the light blue line. The light blue line is a special version of Algorithm 2 that sets $k_c = k$. Although, this may free the method from accessing $\mathcal{C}_c$, yet the L1 cache will start to spill the memory to L2 due to the oversize micro-panels of $\mathcal{A}_c$ and $\mathcal{B}_c$. The red line shows that despite storing the intermediate results $\mathcal{C}_c$ may lead a constant overhead, yet it secures the performance for a larger k.

To summarize the analysis we present in this section, the GSKS model shows that the embedded solution Algorithm 2 dominates Algorithm 1 by minimizing the memory complexity. The benefit shows that the saving is significant on low-dimension problems, and the algorithm is also memory space efficient which makes GSKS more robust in practical applications.

2.4. Micro-Kenerl Implemetation and Parameters choosing

The most important feature of the micro-kernel structure is to minimize the architecture dependent part of the implementation. The idea of the micro-kernel first came from the BLIS framework. Comparing to GotoBLAS implementations where the 1.st to 3.rd loops (macro-kernel or inner-kernel) are all written in assembly, the micro-kernel structure only implements the 1.st loop in SIMD assembly or intrinsic. The 2.nd to 6.th loops are usually implemented in C which may generate some small overheads, but the design can significantly reduce the developing time for portability.

The kernel summation micro-kernel computes a $m_r \times n_r$ kernel matrix and the matrix-vector multiplication in four steps: rank-k_c update, computing square distances, computing kernel values by polynomial approximation and eventually computing the matrix-vector multiplication. Different implementations may be varied from architectures to architectures; thus, here we only illustrate the idea of manipulating vector registers without explaining the details. The intermediate results $\mathcal{C}_r$ are created and maintained inside the vector registers, and the rest of the vector registers are used by a_r, b_r, $\|\mathcal{A}_r\|_2^2$, $\|\mathcal{B}_r\|_2^2$, u_r, w_r and other coefficients required by the polynomial approximation. To overlap the memory operation on loading a_r and b_r, double buffering is used by unrolling the k_c loop twice or four times. Next a_r ($a + m_r$) and b_r ($b + n_r$) are preloaded with the rank-1 update concurrently. At the same time, the next micro-panel of $\mathcal{B}_c$ and the current $\mathcal{C}_c$ can be preloaded and overlapped with the whole rank-k_c update.

The way that an expert implements a rank-k_c update can be found in [], and the vectorized math functions can be found in []. With a system that supports vectorized VLOAD/VSTORE, VADD and VMUL (or VFMA), an efficient way is to load a vector b_r and a_r and apply an VFMA which gives us the diagonal of $\mathcal{C}_r$. By permuting (shuffling) a_r, b_r and recomputing the VFMA, we will end up getting a permuted $\mathcal{C}_r$. After accumulating all rank-k_c updates to $\mathcal{C}_r$, eventually, we permute $\mathcal{C}_r$ back to the right order to complete the update. The scheme is usually called the butterfly permutation which is also well known in Fast Fourier Transform. After the scaled square distances have been updated in $\mathcal{C}_r$, the vectorized $\exp()$ is computed by in thr form of $\exp(a \ln 2 + b) = 2^a \times \exp(b)$. A vector of $\mathcal{C}_r$ will be scaled by a VMUL with $\frac{1}{\ln 2}$, and the multiple a can be computed by type converting (floating to integer). By shifting a multiple of $\ln 2$, the remainder $b \in [0, \ln 2]$ is suitable for polynomial approximation. The order 11 polynomial is evaluated in the nested form such that there are 12 VFMA in total, and 2^a can simply be computed by integer shifting and type converting back to the floating point format.

The optimal m_r and n_r are chosen based on the latency and the dependencies of the VADD and VMUL (or a VFMA if supported). The intermediate results are accumulated in $\mathcal{C}_r$ which will be updated successively by the 1.st loop, and the polynomial approximation will also take $\mathcal{C}_r$ as the intermediate space. Ways to selected parameters for GEMM or other BLAS subroutines can be classified into two different approaches: tuning or theoretical analysis. Here, we follow the [LOW et al. 2014] approach and take a similar analysis to briefly describe how we choose the parameters for GSKS. To fully exploit the CPU computation resources, one must first consider to prevent the ALU pipeline

from stalling. Since there are no branches inside the micro-kernel, the vectorized instruction latency is the only thing we concern. Registers can't be reused by another instruction with in the latency; thus $m_r \times n_r$ needs to be large enough to avoid the ALU from stalling. Here we use L_{VFMA} to indicate the cycles that two VFMA instructions can be applied to the same register in $\mathcal{C}_r$, and N_{VFMA} is the throughput of VFMA in cycles. To avoid the ALU from stalling, at least $L_{VFMA}N_{VFMA}$ VFMAs need to be issued to fill the pipeline. We only need to consider the case of VFMA, since that VFMA is usually the instruction with the largest latency times throughput. This is typically true for the rank-k update, and it's also true here for the polynomial approximation. According to [LOW et al. 2014], given the vector length N_{VEC},

$$m_r \times n_r \geq N_{VEC}L_{VFMA}N_{VFMA} \tag{11}$$

must be satisfied. Other than this lower bound, $m_r \times n_r$ is bounded by the maximum number of the registers. A large $m_r \times n_r$ may lead to registers spilling; thus, usually we choose $m_r \times n_r$ to be the half of the maximum number of the registers. The rest of the registers should provide enough space for the rank-k_c update and the polynomial approximation.

The optimal k_c is chosen based the size of L1 cache and the desired replacement behavior of the micro-panel of $\mathcal{A}_c$ and $\mathcal{B}_c$. In the case of 8-way fully associate L1 cache, the size of a micro-panel is usually designed to be the multiple of a set. This make sure that the micro-panel of $\mathcal{B}_c$ will remain in the L1 through out the 2.nd loop, and the micro-panel of $\mathcal{A}_c$ will be replaced entirely by the next micro-panel. Thus, k_c is usually set to let micro-panels take 6 sets of the L1 out of 8, and the 2 remaining sets are preserved for other use such as $\|\mathcal{A}_c\|_2^2$, $\|\mathcal{B}_c\|_2^2$, u_c, w_c and $\mathcal{C}_c$. For `GSKS`, k_c is also the threshold of storing $\mathcal{C}_c$. A slightly relax to get a bigger k_c may help smooth the performance. As we have seen that there are possible spike drops on the efficiency around the multiple of k_c in Figure 4, some tolerance can be made to slightly increase k_c to avoid an additional iteration on the 5.th loop (p_c). In BLIS for example, a k_c^{max} can be provided by the user to smooth the performance from having spikes, yet currently for simplicity we are not going to further discuss the k_c^{max} in `GSKS`.

The optimal m_c and n_c are chosen based on the size and the configuration of L2 and L3 cache. $\mathcal{A}_c$ is designed to be fitted in L2; thus, its size $m_c \times k_c$ must be smaller than the L2 cache size. m_c is usually chosen to occupy half or $\frac{3}{4}$ L2 cache, and the rest of the space will be used for the micro-panel of $\mathcal{B}_c$ and other packed memory to float through. At last, $\mathcal{B}_c$ with size $n_c \times k_c$ should be fitted into the L3 cache, and n_c is usually chosen to fill $\frac{1}{3}$ of the cache space.

2.5. Shared Memory Parallel Scheme

We present two schemes to parallelize (1) and `GSKS` on a shared memory system. Parallelizing (1) can achieve a perfect task parallelism except for the possible concurrent write on u which may be resolved by privatizing u. `GSKS` itself can also be parallelized on different loops which are similar to parallelizing `GEMM` except for the possible concurrent writes on u of the n and k loops. For the task parallelism, we take a greedy first termination strategy to create a static schedule, and for the data parallelism we present a simple scheme that perfectly parallelize the 4.th (i_c) loop without any concurrent write issue.

Although the general optimal scheduling problem is an NP complete problem, yet it becomes relatively easy while there is no dependency between each task. On a shared memory homogeneous parallel system (e.g. multi-core CPU), the optimal schedule of (3) can be solved by a first termination list scheduling where the list is presorted in the

descending order of the cost (time) of each task. The complexity of finding the schedule is dominated by the sorting complexity $\mathcal{O}(n_{\mathcal{X}} \log(n_{\mathcal{X}}) + pn_{\mathcal{X}})$ where p is the number of the processors which is smaller than $n_{\mathcal{X}}$. Each processor maintains a individual workload counter and a job queue. For the s-th subproblem of (3), the cost can be estimated by the model time T in §2.3, or a rougher cost $m_s n_s$ is also enough due to the fixed k. The s-th task will be assigned to the processor that has the minimum workload, and the workload will increase with the amount of the cost. After the all jobs have been enqueued, all workers will start to dequeue and accumulated u_s to it's private buffer. The reduction of each private u_s can also be paralleled element-wise to secure the parallelism.

Although the task parallelism scheme usually works pretty well while the number of tasks is large enough to balance the workload, yet the worst case scenario (the runtime of the optimal schedule) is still depending on the most costly task. To solve this problem, a subproblem must be parallelized inside, and the most common data parallelism scheme is to parallelize Algorithm 2 on the 4.th loop (i_c). The 1.st and the 5.th loops are seldom selected to be parallelized due to the reduction data dependency on $\mathcal{C}_s$, and the 3.rd and the 6.th also has the reduction data dependency on u_s. Despite the data dependency can be resolved by privatizing u_s, parallelizing the 6.th loop requires to maintain several $\mathcal{B}_s$ buffers inside the L3 cache which may disturb the cache behavior, since the L3 cache is usually shared by many processors. We usually don't consider the 2.nd loop, since it's bounded by m_c which doesn't provide enough parallelism. Thus, the 4.th loop is the best choice to parallelize amount all the possible solutions. Parallelizing the 4.th loop requires privatizing $\mathcal{A}_s$, yet the $\mathcal{B}_s$ packed inside the 5.th loop will be reused by all processors. In parallel system that has a private L2 and a shared L3 cache, the private $\mathcal{A}_s$ will be reused inside the private L2, and the shared $\mathcal{B}_s$ will be reused inside the shared L3.

2.6. Numerical Stability

The numerical stability of `GSKS` is discussed in two parts: the pairwise distances and the polynomial approximation. We show that computing the pairwise distance using expansion (2) and the rank-k update has a similar roundoff error bound as the direct computing scheme, and both these two schemes are backward stable. The kernel function is approximate by a order 11 polynomial approximation where the coefficients are computed by Remez exchange algorithm. The approximation is both forward and backward stable.

We first derive the roundoff error introduced by directly computing $\|x_i - x_j\|_2^2$. For each dimension, $r = x_i - x_j$ is computed first, followed by the square operation $r^2 = r \times r$, and r^2 is accumulated to the final result. `GSKS` expands the square pairwise distance into $\sum_{p=1}^{k} x_i^2 + \sum_{p=1}^{k} x_j^2 - 2\sum_{p=1}^{k} x_i x_j$. For the both scheme, the accumulated roundoff error is bounded by $(1+\epsilon)^{k+2}\|x_i - x_j\|_2^2$. , where ϵ is the machine epsilon of the current precision. Similar to all inner-product base operations, the square distance is not forward stable. The square distance can be zero in (12) which may lead to an unbounded relative error.

$$\frac{(1+\epsilon)^{k+2}\|x_i - x_j\|_2^2 - \|x_i - x_j\|_2^2}{\|x_i - x_j\|_2^2} \tag{12}$$

To derive the backward stability, we move the error term into the square operation. $(1+\epsilon)^{k+2}\|x_i - x_j\|_2^2 = \|(1+\epsilon)^{\frac{k+2}{2}}x_i - (1+\epsilon)^{\frac{k+2}{2}}x_j\|_2^2$.

$$\frac{\|(1+\epsilon)^{\frac{k+2}{2}}x_i - x_i\|_2}{\|x_i\|_2} = (1+\epsilon)^{k+2} - 1 \leq O(k\epsilon) \tag{13}$$

The expansion computes the square pairwise distance more efficiently by reusing the result of $\|\mathcal{X}_A\|_2^2$ and $\|\mathcal{X}_B\|_2^2$. Similar to the direct evaluation, both schemes are backward stable, and the round-off error is also the same.

In the polynomial approximation part, the roundoff error mainly comes from the nested polynomial evaluation.

$$P_{11}(x) = c_{11} + (... + (c_5 + (c_4 + (c_3 + (c_2 + (c_1 + c_0 x)x)x)x)x)x...)x \tag{14}$$

The roundoff error of the order-n ($n \geq 1$) polynomial summation has the following closed form: $c_n x^n(1+\epsilon)^{2n} + \sum_{i=n-1}^{0} c_i x^i (1+\epsilon)^{2i+1}$. This polynomial approximation is forward stable, since $\exp(b) \geq 1$ for $b \in in[0, \ln 2]$. The forward stability is derived in (15) and (16).

$$\frac{|c_n x^n[(1+\epsilon)^{2n} - 1] + \sum_{i=n-1}^{0} c_i x^i[(1+\epsilon)^{2i+1} - 1]|}{|\sum_{i=n}^{0} c_i x^i|} \leq \tag{15}$$

$$\frac{|2n\epsilon||\sum_{i=n}^{0} c_i x^i| + O(\epsilon^2)}{|\sum_{i=n}^{0} c_i x^i|} = O(n\epsilon) \tag{16}$$

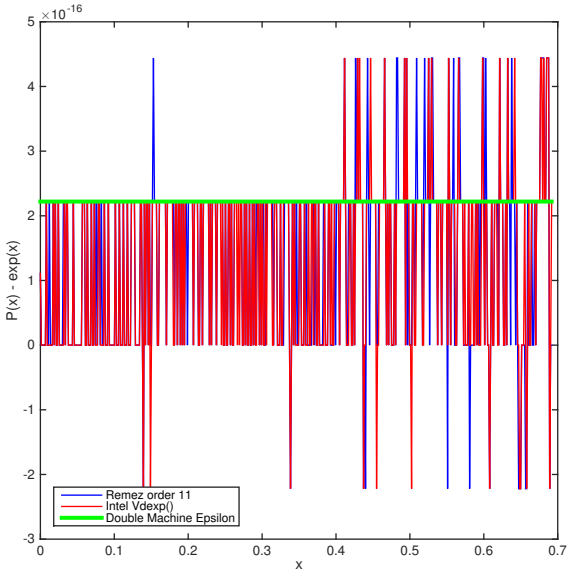

Fig. 5: *Error comparison between Remez order eleven polynomial approximation and Intel vdExp() function: the polynomial is chosen to fit the exponential function with* $[0, \log(2)]$*, and the converging criteria for the Remez exchange algorithm is the double machine epsilon (2.22E-16).*

3. EXPERIMENTAL SETUP

In this we give details on the experimental setup we used to test our methods. GSKS comes with default double precision x86-64 micro-kernels which is designed for Intel Sandy-Bridge/Ivy-Bridge architectures.

Implementation and hardware: GSKS routine is implemented in C, SSE2 and AVX intrinsic, and the task parallelism scheme is written in C++. Except the default micro-kernel is written in SSE2 and AVX, all other parts are written in pure C. The code is compiled with Intel C compiler version 14.0.1.106, and the compiler optimization level is -O3. We carry out runtime experiments on a single NUMA(Non-Uniform Memory Access) node of the Maverick system at TACC which has two ten-core CPUs. The dual-CPUs in each node are Intel Xeon E5-2680 v2 (Ivy Bridge) processors (2.8GHz/3.6GHz) with 12.8Gb/core of memory and a three-level cache: 32K L1 data cache, 256K L2 cache and 25600K L3 cache.

GSKS **parameters:** we have briefly discussed how to choose parameters in §2.4. Here we present parameters chosen for the Intel Ivy Bridge processor on Maverick. The micro-kernel is a $m_r \times n_r$ rank-k_c update where $m_r = 8$, $n_r = 4$ and $k_c = 256$. $m_c = 96$ and $n_c = 4096$ which make the size of $\mathcal{A}_c$ 216 KB and the size of $\mathcal{B}_c$ 8192 KB.

4. RESULTS

The experiment results are presented in two parts. In the first part, we show the floating point operation efficiency of the sequential GSKS, and we compare the results with the BLAS approach using MKL GEMM, GEMV and VML VEXP. In the second part, we show the stalability of both the data parallelism scheme and the task parallelism on a single NUMA node of Maverick.

We present the floating point operation efficiency of GSKS in three experiments which will increase either n, m or k at a time. For all plots, the y-axis indicates the floating point operation efficiency in Gflops per second where the range is from zero to the theoretical peak performance of the target CPU (28.32 Gflops/sec @ 3.54 GHz). The experiment that increases k and fixes n and m has been presented in Figure 3 and Figure 4 together with the predicted efficiency. We show that GSKS with the Ivy-Bridge kernel can achieve 50% of peak performance even with a low dimension $k = 4$ while the BLAS approach can only reach about 20% due to the memory overhead. In the experiment that varies n and m but fixes k, Figure 6 and Figure 7 shows that GSKS in general requires a smaller n and m to reach a 50% peak performance. For a normal problem size $m = 4096$, $n = 512$ and $k = 256$, GSKS can achieve an 80% peak performance, yet the BLAS approach requires a much bigger problem size to reach the same efficiency which at the same time requires much more memory space for buffering.

The task parallelism experiment is carried out by solving (3) with a $k \times n_{\mathcal{X}}$ random coordinate table. We randomly generate 500 subproblems, and each of the subproblem has a random m_s, n_s, α_s, β_s and ω_s. Each m_s is randomly assigned with in $[256, 4352]$, and each n_s is randomly assigned with in $[256, 512]$. Each α_s and βs is randomly sampled from $[0 n_{\mathcal{X}} - 1]$ without replacement. Given the same problem of (3), we measure the floating point operation efficiency of the task parallelism scheme with different number of processors. In Figure 8, we show the strong scaling property of the task parallelism scheme which is able to reach a 66% peak performance.

Finally, we present the data parallelism scheme which parallelizes GSKS on the 4.th loop. The experiment is done with $m = 16384$, $n = 512$ and $k = 256, 257$ using different number of processors. Figure 9 shows the strong scaling property of the data parallelism scheme on a single ten-core CPU, and the parallel GSKS can reach a 60% peak performance while $k = k_c$ and a 50% peak performance while $k = k_c + 1$.

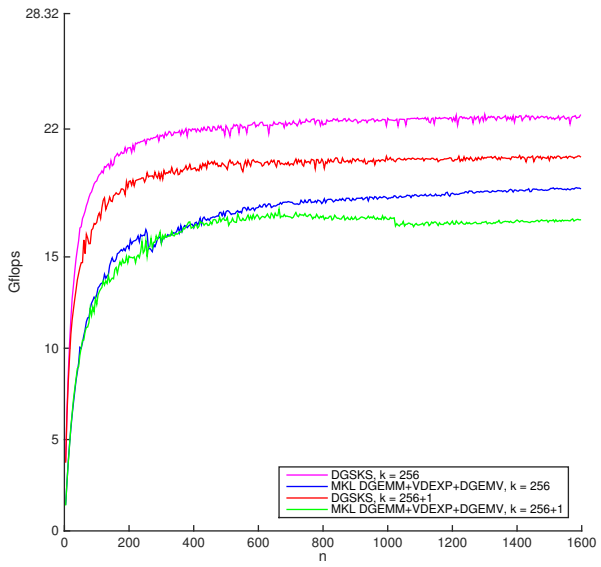

Fig. 6: *Sequential* GSKS *floating point operation efficiency (Gflops) with fixed* $m = 4096$, $k = 256$, $k = 257$ *and different* n *from 4 to 1600.*

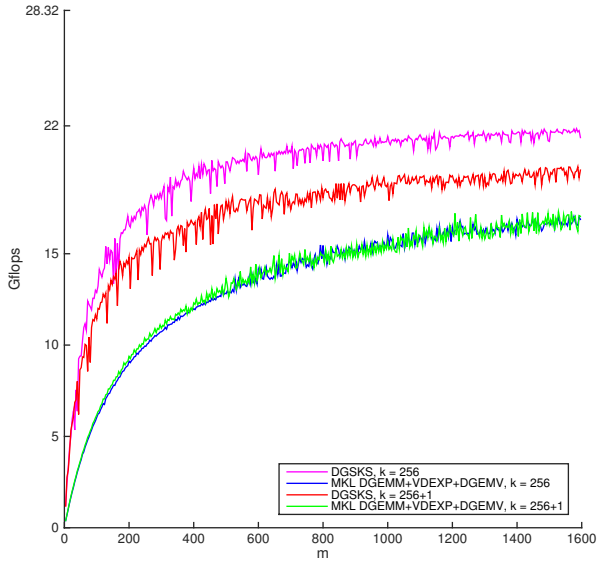

Fig. 7: *Sequential* GSKS *floating point operation efficiency (Gflops) with fixed* $n = 512$, $k = 256$, $k = 257$ *and different* m *from 4 to 1600.*

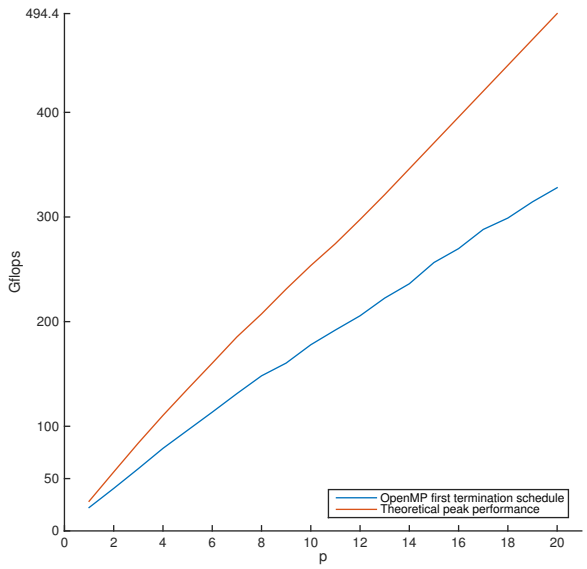

Fig. 8: *Task parallelism on sequential* `GSKS`*: the floating point operation efficiency (blue) is measure in Gflops, and the theoretical peak performance is computed by* $8 \times p$ *times of the measured clockrate on a single core.*

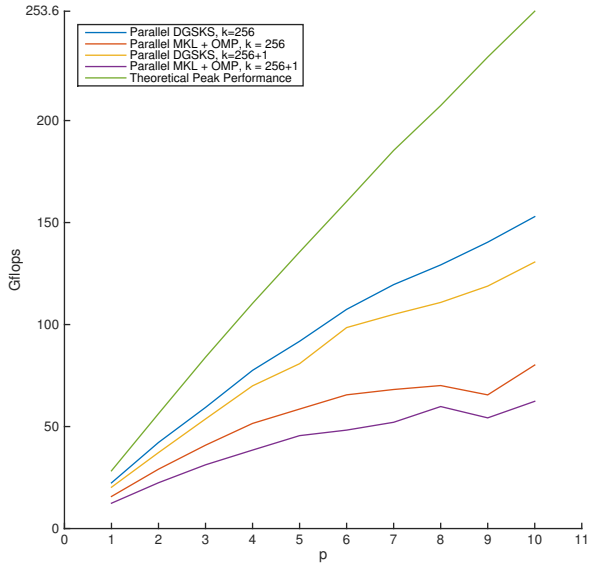

Fig. 9: *Parallel* `GSKS` *floating point operation efficiency (Gflops) with a fixed* $m = 16384$*,* $n = 512$*,* $k = 256$*,* $k = 257$ *and different number of processors. The theoretical peak performance is computed by* $8 \times p$ *times of the measured clockrate on a single core.*

5. CONCLUSION

ACKNOWLEDGMENTS

I want to thank Field Van Zee for spending two hours with me hacking BLIS to prove my idea on `GSKS` despite that the original goal is to hack it within fifteen minutes.

REFERENCES

Chih-Chung Chang and Chih-Jen Lin. 2011. LIBSVM: a library for support vector machines. *ACM Transactions on Intelligent Systems and Technology (TIST)* 2, 3 (2011), 27.

Kazushige Goto and Robert A Geijn. 2008. Anatomy of high-performance matrix multiplication. *ACM Transactions on Mathematical Software (TOMS)* 34, 3 (2008), 12.

Shenshen Liang, Cheng Wang, Ying Liu, and Liheng Jian. 2009. CUKNN: A parallel implementation of K-nearest neighbor on CUDA-enabled GPU. In *Information, Computing and Telecommunication, 2009. YC-ICT'09. IEEE Youth Conference on*. IEEE, 415–418.

TZE MENG LOW, FRANCISCO D IGUAL, TYLER SMITH, and ENRIQUE S QUINTANA-ORTI. 2014. Analytical Modeling is Enough for High Performance BLIS. *ACM under-reviewing* (2014).

William B March, Bo Xiao, and George Biros. 2014. ASKIT: Approximate Skeletonization Kernel-Independent Treecode in High Dimensions. *arXiv preprint arXiv:1410.0260* (2014).

William B March, Bo Xiao, Chenhan D Yu, and George Biros. 2015. An algebraic parallel treecode in arbitrary dimensions. *IPDPS15 accepted* (2015).

Field G Van Zee and Robert A Van De Geijn. 2013. BLIS: A framework for rapidly instantiating BLAS functionality. *ACM Transactions on Mathematical Software (TOMS). To appear* (2013).

Received February 2015; revised March 2009; accepted June 2009