TensorSVM: Accelerating Kernel Machines with Tensor Engine

Shaoshuai Zhang, Ruchi Shah, and Panruo Wu
{szhang36,rkshah5,pwu7}@uh.edu
Department of Computer Science
University of Houston, Texas USA

ABSTRACT

This paper explores the use of Tensor Engines to accelerate non-linear and linear SVM training. Support Vector Machine(SVM) is a classical machine learning model for classification and regression and remains to be the state-of-the-art model for some tasks such as text classification and bioinformatics. However large scale SVM training is still challenging because of its high computational complexity. This is especially severe for non-linear SVM with kernel tricks. On the other hand, the surging importance of neural networks fuels the emergence of specialized processors called Tensor Units (TensorCore in GPU and Tensor Processing Unit of Google) which are characterized by extreme efficiency and very limited precision and range. This paper proposes a TensorCore GPU based SVM algorithm and software system that is faster and more scalable than state-of-the-art SVM solvers. It includes a fast, accurate low-rank Gram matrix approximation that effectively utilizes the TensorCore in GPU and a primal-dual interior-point method to solve the quadratic program with a fast and predictable convergence rate. The random projection based Gram matrix approximation can be substantially accelerated by TensorCore on GPU.

This exploration ends up with a tale of randomized numerical linear algebra, convex optimization, and high performance computing on Tensor Engines. Particularly, this paper suggests that the emerging randomized numerical linear algebra algorithms and Tensor Engines are synergistic in opening up exciting new application areas that include statistical machine learning and the wider scientific/engineering computing.

CCS CONCEPTS

• **Computer systems organization → *Neural networks***; • **Theory of computation → Numeric approximation algorithms**; **Quadratic programming**; **Massively parallel algorithms**; **Random projections and metric embeddings**; **Support vector machines**; • **Mathematics of computing → Mathematical software performance**.

KEYWORDS

Support Vector Machine, SVM, kernel trick, large scale machine learning, randomized linear algebra, interior point method, quadratic program, GPU, TensorCore, HPC, kernel machine

ACM Reference Format:
Shaoshuai Zhang, Ruchi Shah, and Panruo Wu. 2020. TensorSVM: Accelerating Kernel Machines with Tensor Engine. In *2020 International Conference on Supercomputing (ICS '20), June 29-July 2, 2020, Barcelona, Spain.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3392717.3392770

1 INTRODUCTION

Driven by the ever expanding successful applications of large scale deep neural networks, hardware vendors are starting to offer specialized accelerator for neural network training and inference which we will call Tensor Engines in this paper. Among them are TensorCore from NVIDIA on its latest Volta and Turing GPUs, Google's Tensor Processing Unit (TPU)[1], and Intel's Cooper Lake and Sapphire Rapids Xeon processors, as well as its Nervana Neural Network Processor NNP-T 1000. These Tensor Engines are usually characterized by limited precision and range, and extremely efficient matrix-matrix multiplication like operations. For example, NVIDIA V100 boasts up to 125 "deep learning" teraFLOPS (125×10^{12} floating point operation per second) [27], which is basically half precision matrix multiplication accumulated in single precision. Google's TPU v3 claims 420 TeraFLOPS, also in doing half precision matrix-matrix multiplication. In contrast, V100 single precision peak performance is 14 TeraFLOPS, and double precision is 7TeraFLOPS.

Support Vector Machines (SVMs) is a classic statistical machine learning models for classification and regression. A survey conducted by Kaggle in 2017 shows that 26% of the data science practitioners use SVMs to solve their problems [33]. For datasets with smaller feature space, SVMs can employ kernel trick to map low-dimensional data to high-dimension space to obtain a non-linear model, which leads to a large and non-separable quadratic optimization problem with constraints. It involves a kernel matrix of size $n \times n$ for a data-set with n training examples. However, at large scale, it's infeasible to store this extremely large kernel matrix in memory and process it. There are two different approaches to deal with this challenge: one is to decompose the optimization problem into a sequence of smaller ones and the other is to solve an approximate optimization problem. The former approaches include the very successful and widely used LIBSVM package [17], and the latter approaches include deterministic low rank approximation of the kernel matrix ([2, 12]), and randomized approximation of the kernel matrix (Nystrom's method[9] for example).

[1]https://cloud.google.com/tpu/

Computationally, the SVM is a representative statistical machine learning problem thus inherits many of their challenges on a large scale. In this paper, we propose a novel randomized approximate kernel SVM training technique called TensorSVM that is accurate, scalable to large scale datasets and parallel processing units, and efficient on GPU with TensorCore units. Compared to the state-of-the-art approximate SVM solvers, TensorSVM is more accurate in approximation which yields uncompromising prediction accuracy, and possesses better data locality that allows it to exploit hierarchical memory system and specialized TensorCore on GPU. Compared to non-approximate kernel SVM solvers such as LIBSVM [17]/ThunderSVM [34], TensorSVM exposes much more parallelism, with lower asymptotic time complexity, and have predictable convergence rate and execution time.

The contributions of this paper are:

- **The demonstration of promising use of Tensor Engines in Randomized Low Rank Approximation.** We seem to be the first to discover the synergy between Tensor Units and Randomized Numerical Linear Algebra (RandNLA) [10]. This work suggests a new direction of much wider exploitation of emerging ubiquitous Tensor Engines outside of neural networks.
- **A novel approximate SVM training algorithm** The proposed TensorSVM algorithm combines randomized rank revealing matrix factorization with advanced primal-dual interior point method which converges very fast (typically converges in 100 iterations) and is insensitive to ill-conditioning.
- **Empirical Evaluation and Software** The TensorSVM has been released as open-source software and it's hosted at github[2]. Our implementation shows that TensorSVM is 1.8x-6.0x faster than state-of-the-art SVM solver on GPU with uncompromising accuracy. For linear SVM, the speedup is up to 125x without approximation. We also show that our implementation achieves >50% roofline model bounds based on arithmetic intensity.

The rest of this paper is organized as follows. Section 2 discusses the essential background regarding SVMs and GPUs to handle the challenges. Section 3 illustrates the details behind our low-rank approximation and primal-dual interior-point method and it also shows the reasoning of convergence, convergence rate, numerical issues, and approximation accuracy. It also shows the practical implementation side and performance and numerical optimizations. Section 4 provides experimental results on the accuracy, performance and various other characteristics of TensorSVM on benchmark data-sets. Section 5 concludes the paper, addresses limitations of TensorSVM, and includes some possible future research directions.

2 BACKGROUNDS AND CHALLENGES

2.1 Support Vector Machine

Support vector machine (SVM) [3] is widely used as supervised learning model for binary classification and regression analysis. It tries to find a hyperplane that separates the positive and the negative samples. The closest samples to the hyperplane are support vectors and the goal of SVM model training is to maximum the distance between the hyperplane and the support vectors named margin. The hyperplane H is characterized by a normal vector w and a offset b that $H = \{x : w^T x = b\}$ with a maximum margin. It could be modeled as a convex optimization problem as follows:

$$\min_{w,b} f(w) = \frac{1}{2} w^T w + C \sum_{i=1}^{n} \max\{0, 1 - y_i(w^T x_i + b)\}$$

Some data-sets are not linearly separable, thus kernel tricks are introduced to map the data to a much higher or infinity dimension so that the data-sets are more likely to be linearly separable in the high dimensional space (Reproducing Kernel Hilbert Space, RKHS). The original data points are transferred to a higher dimension with a function $\phi(\cdot)$ and the only change to the primal problem is to replace x_i with $\phi(x_i)$. Fortunately, in the dual problem, we can get rid of the inner product $\phi(x_i)^T \phi(x_j)$ of potentially infinite dimension inner product, and replace it with a kernel function $\kappa(x_i, x_j)$ that is easy to evaluate. The dual problem is:

$$\min_{\alpha \in \mathbb{R}^m} \quad g(\alpha) = \frac{1}{2} \sum_{i,j=1}^{m} y_i y_j \alpha_i \alpha_j \kappa(x_i, x_j) - \sum_{i=1}^{m} \alpha_i \tag{1}$$

$$\text{subject to} \quad 0 \le \alpha \le C, i = 1, \ldots, m \tag{2}$$

$$\sum_{i=1}^{m} \alpha_i y_i = 0 \tag{3}$$

The dual problem can be reshaped in a more compact way

$$g(\alpha) = \frac{1}{2} \alpha^T Q \alpha - e^T \alpha$$

where the labelled kernel matrix Q is a $m \times m$ matrix $Q_{i,j} = y_i y_j \kappa(x_i, x_j)$. The dual problem also avoids the non-differentiability of the hinge loss in the primal problem.

Note that if the m is very large(e.g. *SUSY* has 5 million records), we will need 100TB to store Q in memory. Besides, the floating point operations per second (FLOPS) to factorize Q is nearly $m^3 \approx 1.25 \times 10^{20}$ (125 ExaFLOPs) to solve a single linear system, which is only possible on a flagship supercomputer on the top500.org list.

2.2 Optimization Algorithms

As discussed in the previous subsection, the SVM training problem is formulated as a quadratic program (1), with simple bounds inequality constraints (2) and a single linear equality constraint (3). There are many optimization algorithms that apply to this problem; however few of them can handle millions of variables and constraints efficiently. The most successful and popular one is a decomposition based coordinate descent like algorithm called sequential minimal optimization (SMO), and its subsequent improvements incorporated in software package LIBSVM [1, 17] and ThunderSVM [34]. In its original scheme, SMO picks two variables and fix the rest, and optimize the two-variable quadratic program using an analytical solution. Which variables to pick is integral to the success of this algorithm, and heuristics are often used to guide the selection of the two variables according to second order information. The resulting algorithm and software (LIBSVM/ThunderSVM) is elegant, robust, and fast. However, there are two undesirable properties of this line of algorithms: 1) it's hard to parallelize and scale, since SMO decomposes into a sequence of extremely small

[2]https://github.com/robbwu/tensorsvm

problems; 2) the convergence speed vary greatly and is very sensitive to the data matrix conditioning and the hyperparameters C, γ.

Another broad class of algorithms—second order, or Newton like algorithms—seem to be attractive at large scale, primarily due to its quadratic local convergence rate (correct digits double in every iteration) and often superlinear global convergence rate, and its insensitivity to the conditioning of the quadratic program. Newton like algorithms often converge in a few dozen iterations, but each iteration involves a linear solve with the Gram matrix, therefore entailing $O(n^3)$ operations. For n in the millions, $O(n^3)$ is ExaFlops which is extremely expensive. However, if the Gram matrix is of low rank $k \ll n$, then the linear system can be solved in $O(nk^2)$ time, which is the motivation of *approximating* the Gram matrix with a low rank matrix (see the subsection 3.1.1 for details).

The specific second order methods that solve quadratic program with equality and inequality constraints are called barrier methods or interior point methods. The basic idea is to turn the inequality constraint into a soft-barrier with increasingly high penalty when it approaches violation of the constraints. With the barrier, the program is approximated as a quadratic program without constraints which can be solved using Newton's method. A more effective, and easier to use the interior point method is called primal-dual interior point method, which solves the perturbed KKT conditions using Newton's method. We will discuss the basics of the interior point methods we use in more detail in subsection 2.4.

2.3 Half Precision Arithmetic and TensorCore GPU

The dual problem with kernel tricks involves many matrix matrix multiplications and if we use RBF(Radial Basis Function), we must handle plenty of exponential function evaluations as well. Both of the two kinds of calculations could be executed much faster on Graphics Processing Unis(GPUs) than CPU. The GPU comes with dedicated Special Function Units (SFU) that support common transcendental functions with very high throughput [21].

Additionally, Nvidia recently introduced a specialized unit called TensorCore from their Volta architecture and they claim that TensorCore could reach 120 TFLOPS for mixed half/single precision matrix-matrix multiplication. Compared to singe/double precision, Tensor Core matrix matrix multiplication is 7x/16x faster respectively, at the cost of a potential loss of precision and robustness. Apparently, TensorCore is driven by the need for neural network training and inference which is quite tolerant of lower precision matrix matrix multiplication, however its use outside of neural networks is only emerging [4, 13, 16]. The key challenge is to strategically use TensorCore low precision for the bulk of computations that are not sensitive to numerical precision, while having critical sections of computations running at high precision to safeguard the accuracy and stability. Alternatively, one could employ some kind of iterative refinement procedure to improve accuracy.

There are several methods to program the TensorCore. The easiest way to use it is by calling the CUBLAS routine provided by CUDA. One can also program TensorCore through the CUTLASS template library or directly call the WMMA intrinsic which is more flexible. In this paper, we use TensorCore through the CUBLAS library.

2.4 Mehrotra's Predictor-Corrector Interior Point Method

The specific second order method we use is one variant of primal-dual interior point method due to Mehrotra [26], and we will call Mehrotra Predictor-Corrector (MPC) algorithm. Hereafter we will rename the α to x and y to a for better conforming to standard convex optimization literature convention [12, 37]. We rephrase the nonlinear SVM (dual) problem as follows:

$$\begin{aligned} \min_{x} \quad & \frac{1}{2}x^T Qx - e^T x \\ \text{s.t.} \quad & a^T x = 0 \\ & 0 \leq x \leq c \end{aligned} \tag{4}$$

The (perturbed) KKT conditions of this problem is

$$\begin{aligned} Xs &= \sigma\mu e \\ (C - X)\xi &= \sigma\mu e \\ a^T x &= 0 \\ -Qx + ay + s - \xi &= -e \end{aligned} \tag{5}$$

$$0 \leq x \leq c, s \geq 0, \xi \geq 0 \tag{6}$$

where x is the primal variable and $X = \text{diag}(x)$, and y, s, ξ are dual variables, $\sigma \in [0, 1]$ is the centering parameter, and μ called duality gap. Primal-dual Interior Point Methods (PD-IPM) typically solves the constrained non-linear KKT system using Newton's method. To avoid prematurely reaching the boundaries which significantly limits later maneuver thus slowing convergence, the path-following method solves a slightly perturbed version of the KKT condition. The path-following algorithms solve successive perturbed KKT with $\mu \to 0$, thus following a central path while staying strictly in the interior ($c > x > 0, s > 0, \xi > 0$). Mehrotra's Predictor-Corrector (MPC) algorithm is particularly effective and convenient in practice, as it has a good heuristic for a parameter μ. Each iteration in MPC consists of two steps: a predictor step (small σ) that aims for large Newton step towards optimal solution, and a corrector step (large σ) that brings the iterate back to along the central path to allow longer steps in subsequent iterations. Both steps solve a linear equation derived by linearizing the perturbed KKT conditions (5):

$$\begin{bmatrix} -Q & a & I & -I \\ a^T & 0 & 0 & 0 \\ S & 0 & X & 0 \\ -\Xi & 0 & 0 & C - X \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta y \\ \Delta s \\ \Delta \xi \end{bmatrix} = \text{RHS} = \begin{bmatrix} r_1 \\ r_2 \\ r_3 \\ r_4 \end{bmatrix} \tag{7}$$

with different right hand side (RHS). For the predictor step, the RHS is $[-a^T x; -e + Qx - ay - s + \xi; -Xs; -(C - X)\xi]$; for the corrector step, the RHS is $[-a^T x; -e + Qx - ay - s + \xi; -Xs + \sigma\mu - dXds; -(C - X)\xi + \sigma\mu + dXd\xi]$, where the $dX, ds, d\xi$ are the $\Delta X, \Delta s, \Delta\xi$ from the predictor step.

To solve the linear systems (7) in the predictor and correction steps, we first use the last two (block) rows to eliminate $\Delta s, \Delta\xi$ from the equations, and then eliminate Δx from the first two equations, at which time we are left with one equation involving Δy.

3 METHODS

The TensorSVM consists of two parts: the low rank approximation and the interior point method to solve the approximate quadratic program.

3.1 Algorithms

There are two phases of the TensorSVM algorithm: phase I computes a low rank approximation for the kernel matrix G using a method adapted from randomized rank-revealing QR factorization [25], and phase II solves the quadratic program from the dual SVM optimization problem using the approximated kernel matrix and primal-dual interior point method. We describe the two phases in sequence.

3.1.1 *Efficient and Accurate Low Rank Approximation of the Gram Matrix on GPU.*

When training on large scale datasets the kernel matrix $G \in \mathbb{R}^{n \times n}$ can be of tremendous size and renders solving linear equation with it prohibitively expensive. Fortunately for a lot of kernel functions and parameters, the kernel matrix can be very well approximated by a low rank matrix, as it has rapidly decaying eigenvalues [32, 36]. Our low rank approximate is based on the random projection method [25]. The idea is that we want to find an orthonormal matrix $Q \in \mathbb{R}^{n \times k}$ with the small number k columns such that the range of Q covers much of range of the matrix G we wish to approximate: range$(Q) \approx$ range(G). It turns out that for a matrix $\Omega \in \mathbb{R}^{n \times k}$ with i.i.d random Gaussian entries with mean 0 and standard deviation 1, the product range$(G\Omega) \approx$ range(G) in the expectation, and can be enhanced with extremely low variance with power refinement and slight over-sampling. The trick is that $G\Omega$ has a smaller size than G so we can find an orthonormal basis from this smaller matrix $G\Omega$ faster: $Q =$ ortho$(G\Omega)$. In practice, a QR factorization can be used to find an orthonormal basis of $G\Omega$. See Algorithm 1 for our adapted algorithm which is simplified [25], and a pictorial illustration in Figure 1. It's interesting to note that almost all of the operations are level 3 BLAS which can be executed in the maximum speed of GPU in terms of FLOPs per second.

Algorithm 1 Phase I: low rank approximation (LRA) of Gram matrix using randomized Gaussian projection

```
Goal: Given a Gram matrix $G \in \mathbb{R}^{n \times n}$ (symmetric positive definite) and a target rank $k < n$, find a low rank approximator $U \in \mathbb{R}^{n \times k}$ such that $G \approx UU^T$.
Step 1: Generate a Gaussian random matrix $\Omega \in \mathbb{R}^{n \times k}$;
Step 2: Compute matrix multiplication $Y = G\Omega$.
Step 3: Find the orthonormal basis of range($Y$): $Q =$ ortho($Y$), by taking the Q factor of QR factorization of $Y = QR$
Step 4: Form $Y = GQ$
Step 5: Form $C = Q^T Y$
Step 6: Eigen-decompose $C = X\Sigma X^T$;
Step 7: Form the low rank approximator $U = QX\Sigma^{\frac{1}{2}}$
```

3.1.2 *Interior Point Method.*

The interior point method is a second order optimization algorithm to solve convex optimization problems with constraints.

The dual problem of the non-linear SVM is a constrained quadratic program for which the Newton method based dual-primal interior point method is highly effective. The structure we used is similar in [12] but with two main differences: 1) we use the phase I for the low rank approximation; 2) we use Sherman-Morrison-Woodbury formula as linear solver. In contrast, Fine et. al [12] combines the low-rank approximation and linear solver with a single partial factored Cholesky factorization which is more stable numerically but has higher approximation error than ours. The overall optimization algorithm is the predictor-corrector interior-point method attributed to Mehrotra [26] and we used a variant that is close to [37] (Chapter 10, Algorithm MPC). We describe the full algorithm in Algorithm 2.

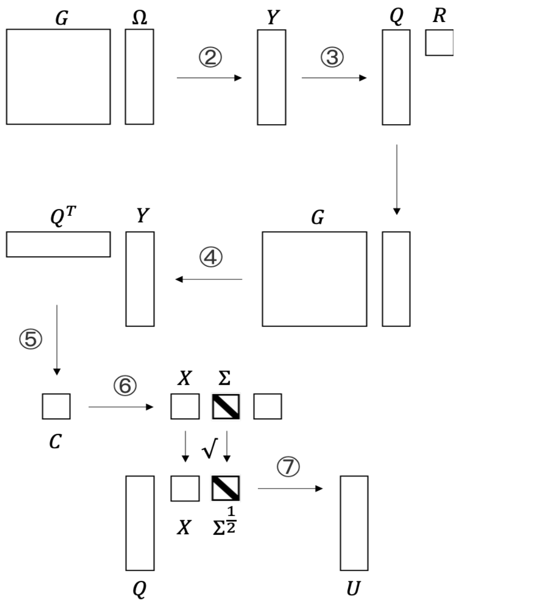


Figure 1: An illustration of the randomized low rank approximation of Algorithm 1. The circled number indicates the step in the algorithm.

3.1.3 *Various Algorithmic Issues. Linear SVM training:*

For a special case, the TensorSVM can also train linear SVM models very efficiently *without approximation*, if the number feature is not large (<2000). It's sufficient to note that in the linear SVM model, the Gram matrix Q is the product of the data matrix, $Q = ZZ^T$, thus already a low rank form suitable as an input of Algorithm 2.

Loss of symmetry and positive semi-definiteness: When the rank is set to be too high, the low rank approximation may generate a non-positive definite and non-symmetric matrix C, thus failing step 7 in Algorithm 1 and Figure 1, where the square roots of the eigenvalues of C are taken. One mitigation is to rerun the training program with a lower k. To mitigate the loss of symmetry, we asymmetric C: $C = (C + C^T)/2$. For the loss of positive semi-definiteness, we filter out the negative eigenvalues in step 7, taking them to be 0. In practice, the negative eigenvalues are very small, and it's likely to be perturbed zero eigenvalues caused by the randomization process and floating point roundoff errors.

Algorithm 2 Phase II: Mehrotra's primal-dual interior point method on the quadratic program from dual SVM problem. Notation: $\odot$ stands for Hadamard product, or element wise vector/matrix product; $X = \text{diag}(x), S = \text{diag}(s)$

```
Goal: Given a symmetric positive semi-definite matrix $A \in \mathbb{R}^{n\times n}$, vector $a \in \mathbb{R}^n$, and scalar $C$, this algorithm finds the minimizer $x^*$ of the quadratic form
    $$\frac{1}{2}x^T Qx - x^T e$$
subject to constraints
    $$a^T x = 0, \quad 0 \le x_i \le C, \text{for } i = 1, \dots, n$$
Initialize the primal variables $x \in \mathbb{R}^n$ and dual variables $y \in \mathbb{R}, s \in \mathbb{R}^n, \xi \in \mathbb{R}^n$.
While the KKT conditions are not satisfied, do
Step 0: Compute: $\mu = (x^T s + (C - x)^T \xi)/(2n)$ and
    $$\begin{bmatrix} dx \\ dy \\ ds \\ d\xi \end{bmatrix} = \begin{bmatrix} Qx - ya - s + \xi - 1 \\ -a^T x \\ -s \odot x \\ -\xi \odot (C - x) \end{bmatrix}$$
Step 1: (Predictor) Solve the Newton update direction $[\Delta x_1, \Delta y_1, \Delta s_1, \Delta\xi_1]$ from linear system using Algorithm 3
    $$\begin{bmatrix} -Q & a & I & -I \\ a^T & 0 & 0 & 0 \\ S & 0 & X & 0 \\ -\Xi & 0 & 0 & C-X \end{bmatrix} \begin{bmatrix} \Delta x_1 \\ \Delta y_1 \\ \Delta s_1 \\ \Delta\xi_1 \end{bmatrix} = \begin{bmatrix} dx \\ dy \\ ds \\ d\xi \end{bmatrix}$$
Step 2: Find a large step size $\alpha$ such that the new updates will still be within the constraints: $0 \le x + \alpha\Delta x_1 \le C, s + \alpha\Delta s_1 \ge 0, \xi + \alpha\Delta\xi_1 \ge 0$.
Step 3: Compute the centering parameter $\sigma =$
    $$\left(\frac{(x + \alpha\Delta x_1)^T (s + \alpha\Delta s_1) + (C - x - \alpha\Delta x_1)^T (\xi + \alpha\Delta\xi_1)}{x^T s + (C - x)^T \xi}\right)^3$$
Step 4: (Corrector) Solve the Newton update direction $[\Delta x_2, \Delta y_2, \Delta s_2, \Delta\xi_2]$ from linear system using Algorithm 3
    $$\begin{bmatrix} -Q & a & I & -I \\ a^T & 0 & 0 & 0 \\ S & 0 & X & 0 \\ -\Xi & 0 & 0 & C-X \end{bmatrix} \begin{bmatrix} \Delta x_2 \\ \Delta y_2 \\ \Delta s_2 \\ \Delta\xi_2 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ \sigma\mu - dx \odot ds \\ \sigma\mu + dx \odot d\xi \end{bmatrix}$$
Step 5: Compute
$[\Delta x, \Delta y, \Delta s, \Delta\xi] = [\Delta x_1, \Delta y_1, \Delta s_1, \Delta\xi_1] + [\Delta x_2, \Delta y_2, \Delta s_2, \Delta\xi_2]$, and find a large step size $\alpha$ such that the new updates will still be within the constraints: $0 \le x + \alpha\Delta x \le C, s + \alpha\Delta s \ge 0, \xi + \alpha\Delta\xi \ge 0$.
Step 6: Update the primal and dual variables:
$x = x + \alpha\Delta x, s = s + \alpha\Delta s, \xi = \xi + \alpha\Delta\xi$
End Loop
```

Algorithm 3 Linear solve for Newton step linear system (7)

```
Require: $Q = ZZ^T \in \mathbb{R}^{n\times n}, Z \in \mathbb{R}^{n\times k}, a \in \mathbb{R}^n, x \in \mathbb{R}^n, s \in \mathbb{R}^n, \xi \in \mathbb{R}^n, C \in \mathbb{R}$ , right hand side $[r1, r2, r3, r4]$, $X = \text{diag}(x)$.
Ensure: Solution to (7) overwrites $[r1, r2, r3, r4]$
1: $r5 = r1 - r3 + (C - X)^{-1} r4$
2: $b = (ZZ^T + D)^{-1} r5$ {For this and following inverse $(ZZ^T + D)^{-1} b$ see Algorithm 4}
3: $r6 = r2 + a^T b$
4: $r2 = r6/a^T (ZZ^T + D)^{-1} a$
5: $r1 = ar2 - r5$
6: $r1 = (ZZ^T + D)^{-1} r1$
7: $r3 = X^{-1}(r3 - s \odot r1)$
8: $r4 = (C - X)^{-1}(r4 + \xi \odot r1)$
```

Algorithm 4 Sherman-Woodbury-Morrison formula to solve linear system $(ZZ^T + D)^{-1} b$, where $Z \in \mathbb{R}^{n\times k}$ is tall skinny and D is diagonal.

```
1: Return $D^{-1}b - D^{-1}Z(I + Z^T D^{-1} Z)^{-1} Z^T D^{-1} b$ (evaluate from right to left).
```

3.2 Algorithmic Complexity and Performance Model

The execution time of TensorSVM is quite predictable, in contrast to the highly unpredictable ThunderSVM/LIBSVM. The reason is the observed superlinear convergence rate of interior point method, which in practice never exceeds 100 iterations. In this subsection, we characterize the time complexity and execution rate of TensorSVM.

The low rank approximation phase (Algorithm 1 and Figure 1) consists of the following operations: 1) (step 1&4) generating the Gram matrix $G \in \mathbb{R}^{n\times n}$ where $G_{ij} = y_i y_j \exp(-\gamma||x_i - x_j||^2)$ and $x_i \in \mathbb{R}^d, i = 1, \dots, n$ are training samples; 2) (step 2,4,7) matrix multiplication; 3) (step 3) QR factorization; 4) (step 6) eigenvalue decomposition. Add them together we get the time complexity of the low rank approximation:

$$T_1 = O(n^2 d) + O(n^2 k) + O(nk^2) + O(k^3)$$

The MPC phase (Algorithm 2) are iterative in nature but for all practical purposes the number of iterations to convergence is constant (<100) due to its superlinear convergence rate. In each iteration, step 1 and step 4 that solve a linear system is the dominant computations, which in turn is dominated by the Sherman-Woodbury-Morrison formula in Algorithm 4. The time complexity is, therefore:

$$T_2 = O(nk^2)$$

Adding this together we get the asymptotic time complexity:

$$T = O(n^2(d + k)) + O(nk^2) + O(k^3)$$

Note that all the dominant computations are dense matrix-matrix multiplications or matrix factorization thus can be expected to be executed at a very high rate on GPU.

3.3 Streaming Computations

Since the matrices involved is completely beyond the capacity GPU memory and sometimes the host CPU memory, the TensorSVM takes advantage of the pass-efficient algorithms and implement the TensorSVM with out-of-core processing, on-demand generation of the Kernel matrix, and overlapping communication with computation.

Out of core implicit Kernel Matrix Multiplication To support training on millions of samples, the lack of GPU device memory must be addressed because not only the Gram matrix (size $n \times n$) cannot fit into GPU memory, the low rank factor U alone may not fit. As such TensorSVM supports out of GPU memory (out of core) processing by streaming the matrices to GPU in chunks. For example, the step 1 and 4 in Algorithm 1 and Figure 1 invoke kernel matrix multiplication. We divide the kernel matrix into 100,000 by 100,000 chunks and stream them to the GPU. The Gram matrix does not even fit in CPU memory so we will re-generate the kernel matrix on-the-fly whenever it is in the pipeline to be streamed to the GPU.

$$\begin{bmatrix} A_1 \\ A_2 \\ A_3 \\ A_4 \end{bmatrix} \overset{\textcircled{1}}{=} \begin{bmatrix} Q_{11}R_1 \\ Q_{12}R_2 \\ Q_{13}R_3 \\ Q_{14}R_4 \end{bmatrix} \overset{\textcircled{2}}{=} \begin{bmatrix} Q_{11} & & & \\ & Q_{12} & & \\ & & Q_{13} & \\ & & & Q_{14} \end{bmatrix} \begin{bmatrix} R_1 \\ R_2 \\ R_3 \\ R_4 \end{bmatrix}$$

$$\overset{\textcircled{3}}{=} \begin{bmatrix} Q_{11} & & & \\ & Q_{12} & & \\ & & Q_{13} & \\ & & & Q_{14} \end{bmatrix} \begin{bmatrix} Q_{21} \\ Q_{22} \\ Q_{23} \\ Q_{24} \end{bmatrix} R$$

$$\overset{\textcircled{4}}{=} \begin{bmatrix} Q_{11}Q_{21} \\ Q_{12}Q_{22} \\ Q_{13}Q_{23} \\ Q_{14}Q_{24} \end{bmatrix} R \overset{\textcircled{5}}{=} QR$$

Figure 2: Communication Avoidance QR: two steps version

Out of core QR factorization. Due to the limitation of GPU memory, processing whole QR factorization for matrix $A = QR$ is not possible if the size of the matrix A is too large. In this paper, we use a QR algorithm named Communication Avoidance QR(CA-QR) [6] which allows us to divide the matrix into smaller blocks and stream them to GPU for processing, and then reassemble them to complete the QR factorization. Figure 2 gives an intuitive explanation on CA-QR. There are 5 steps indicated by the number over the equality sign. In the ① step, we divide a tall matrix A into 4 smaller matrices (still tall, more rows than columns), and QR factorize them independently. In step ② we stack the R factors vertically. Note that the number of rows of the R factors is less than the number of rows of the original A. In step ③, we factorize the vertically stacked Rs (potentially carry this process recursively). In ④, we do 4 matrix-matrix multiplications for the 4 corresponding Q factors. In ⑤ we reinterpret the result as the QR factors of the original A. The reason Q is orthogonal, is that in step ④ the 4 matrix-matrix multiplication is equivalent to the product of two orthogonal matrices (second line), and therefore is orthogonal.

Performing QR factorization on GPU becomes possible through these steps. We first copy A_1 from CPU to GPU and perform QR factorization on A_1 to get Q_1 and R_1. Then we send Q_1 back to CPU memory and save R_1 on the device and we recursively factorize other blocks, finally, we have R_1 to R_4 on the device. Afterwards, we factorize this part on GPU and generate $Q_{21}, Q_{22}, Q_{23}, Q_{24}$. Similarly, we also use out of core matrix multiplication in step ④. Note that we create two buffers for moving data and computation respectively to overlap the data movement between CPU and GPU.

3.4 TensorCore Acceleration

The emergence of TensorCore in NVIDIA GPUs and other tensor engines in accelerators or CPUs provides tremendous potential to greatly accelerate numerical applications with high energy efficiency. However, outside of applications of neural networks, effective and robust use of TensorCores is only beginning. Two challenges must be properly handled: 1) how to provide sufficient data locality to allow TensorCore to accelerate meaningfully? 2) how to mitigate the loss of accuracy and stability due to the limited precision/range in the 16 bits floating point format? In this subsection, we consider carefully the use of TensorCore to accelerate TensorSVM. We show that the low rank approximation (phase I) is conducive to TensorCore acceleration while the second order optimization (phase II) is susceptible to numerical instability.

TensorCore is accurate and stable for low rank approximation (phase I, Algorithm 1) . The reasons are three folds. First, the Gram matrix G is well scaled by construction, and the right factors (random Ω, ortho-normal Q) are well conditioned and scaled. This ensures that the matrix multiplication is unlikely to have overflow which results in ∞ or completely incorrect result. Second, the low rank approximation error is dominated by the truncation due to the low rank constraint. Compared to a typical truncation error, the roundoff errors are insignificant and thus can be ignored. Third, some potential instability is handled at low rank approximatino algorithm level as discussed in section 3.1.3. To give an example, let's directly examine the low rank approximation error of Algorithm 1 with and without TensorCore in figure 3. From the figure, we can see the error of TensorCore Randomized LRA closely tracks the error of single precision Randomized LRA up to around relative error level 10^{-4} and then the TensorCore LRA error stops improving. This is to be expected because the half precision arithmetic TensorCore uses has unit roundoff error 9.8×10^{-4}. It seems that

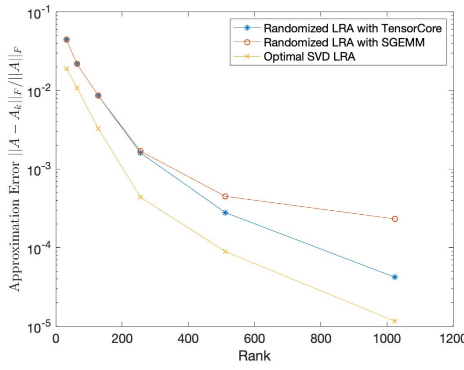


Figure 3: The Low Rank Approximation (LRA) error of the kernel matrix of the first 4096 data points from the *ijcnn*1 dataset. First two lines are using Algorithm 1 with and without TensorCore; the last line is the optimal low rank approximation given by SVD.

10^{-4} level of low rank approximation error is quite adequate for the SVM, as supported by the observations in Section 4, where enabling TensorCore in phase I maintains prediction accuracy. To summarize, our particular low rank approximation is conducive to TensorCore acceleration and resistant to instability caused by low precision and range of FP16.

TensorCore is unstable in Interior Point Method linear solver (phase II). Blindly applying TensorCore to all matrix-matrix multiplication may quickly lead to instability. In fact, in the second phase of TensorSVM, divergence results (algorithm 4). The reason is that as the iteration in phase II goes on, the linear system $(I + Z^T D^{-1} Z)$ becomes increasingly ill-conditioned ($\kappa \to \infty$ at the solution); moreover the SWM based algorithm exhibits excellent data locality but is not backward stable. When D becomes ill-conditioned, computing $Z^T D^{-1} Z$ on TensorCore will deviate from the solution $(ZZ^T + D)^{-1} b$ so much that the IPM iteration almost always diverge. To summarize, unless the strongly backward stable algorithm is used, TensorCore will likely lead to cause instability. More research is required to exploit TensorCore in phase II, which this paper does not explore.

3.5 Comparison to Other SVM Solvers and Low Rank Approximators

As TensorSVM consists of both innovations from algorithms and parallel computing, the performance advantage of TensorSVM in comparison with other approximate/non-approximate solvers may come from contributions from both. However, the two contributing aspects (algorithmic and technical) are naturally intertwined, because the algorithms are designed to better exploit the underlying architecture we are targeting (multiple GPU accelerators with neural engine units), and the technical implementation takes advantage of the opportunities to accelerate and parallelize. Nevertheless, we attempt to compare TensorSVM to other non-approximate and approximate SVM solvers with vastly different algorithms and implementation and comment on the performance aspect.

Low Rank Approximation. Traditional deterministic low rank approximation such as Singular Value Decomposition and truncated rank-revealing QR factorization are inefficient in our case because: 1) they are not *pass-efficient*, meaning that they require $O(n)$ passes over the Gram matrix which is very slow when the matrix does not fit GPU memory or even the CPU memory; 2) only half of the arithmetic is spent on matrix-matrix multiplication so that TensorCore can at most accelerate them by 1.75x. A more suitable deterministic low rank approximation is based on incomplete symmetric pivoted Cholesky factorization [2, 12]. This method features low complexity $O(nk^2)$ and can be made pass-efficient, however, it suffers from low data locality and low approximation accuracy (see Table 4 PSVM column). On the other hand, randomized low rank approximation including Nystrom's method [9, 38], and the Algorithm 1 are more promising in that they are pass-efficient and arithmetic intensive to benefit from TensorCore. Nystrom's method randomly *sample* k columns of the Gram matrix to form $C \in \mathbb{R}^{n\times k}$, and approximate the Gram matrix G with $G \approx CW^{+}C^{T}$. Algorithm 1 randomly *projects* onto a subspace of range of G, in a more similar fashion as randomized SVD or QR [24]. It seems that the Nystrom's method approximation accuracy is inferior than Algorithm 1; see Table 4 LLSVM column. To summarize, the proposed Algorithm 1 is accurate even with small rank, pass-efficient (only 2 passes), and can be substantially accelerated by Tensor Units.

Structured Low Rank Linear Solver. In the interior point method with low rank matrix, each iteration involves solving a dense structured low rank linear system $D + ZZ^T$ where D is diagonal and $Z \in \mathbb{R}^{n\times k}$. There are two efficient algorithms that solve such system in $O(nk^2)$ time rather than $O(n^3)$: Sherman-Woodbury-Morrison (SWM) inversion formula in Algorithm 4, and factored form Cholesky (FFC) factorization [12]. The former has excellent data locality but sensitive to ill-conditioning, whereas the latter has poor data locality but is stable. We chose the former as structured low rank linear solver for its much better speed on GPU and avoid the instability by terminating the optimization when the matrix D becomes too ill-conditioned, and fine tuning the initial guess.

Numerical Optimization Algorithms: Second Order vs. First Order. To make use of the Tensor Units the algorithm must be of very high arithmetic intensity, otherwise, the operations are memory bandwidth bound. Furthermore, the algorithm must be rich in matrix-matrix multiplication like operations (the only operations supported by TensorCore). First-order optimization methods (gradient descent, coordinate descent, etc) operate with the gradient which means vector-vector operations hence low arithmetic intensity. Therefore LIBSVM/ThunderSVM does **not** benefit from the presence of TensorCore. Furthermore, first order methods can converge really slow depending on the condition of the optimization problem, which in turn depends on the input data and hyperparameters C, γ. On the other hand, the second order Interior Point Method used in TensorSVM is observed to have superlinear convergence rate and converge in 100 iterations in all cases we tested. For example, see figure 4 for comparison on the same data set with different hyperparameters. From the figure we can see that TensorSVM execution time is insensitive to the different conditioning ($\kappa(G)$) induced by the hyperparameter, whereas the execution time of ThunderSVM can vary by more than 100 times.

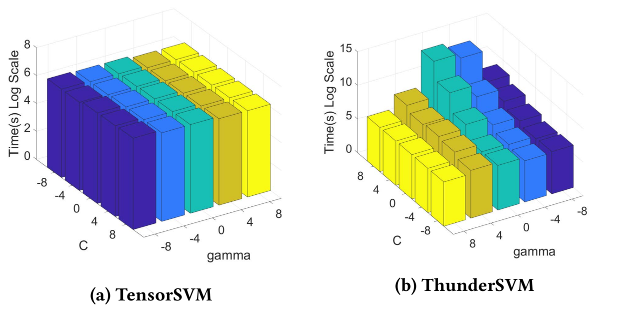


Figure 4: Rough grid search performance comparison for TensorSVM and ThunderSVM with $C = [2^{-8}, 2^{-4}, 2^{0}, 2^{4}, 2^{8}]$ and $\gamma = [2^{-8}, 2^{-4}, 2^{0}, 2^{4}, 2^{8}]$. Time(s) is plotted in log scale with base 2.

4 EXPERIMENTAL EVALUATION

In this section, we conduct a comprehensive empirical evaluation of the speed and accuracy of TensorSVM, in comparison with other state-of-the-art SVM solvers. We investigate the characteristics of our algorithmic design and implementation using the roofline model to evaluate the efficiency of TensorSVM in exploiting the GPU hardware. We also empirically evaluate the impact of TensorCore on the accuracy and speed aspects of TensorCore.

The experiments were performed on a typical HPC cluster. The CPU on the machine is 24 cores Intel Xeon E5-2680v4(2.40GHZ) and the GPU is Nvidia Volta V100 (PCIe) with CUDA version 10.0.130. The operating system is Linux version 3.10.0-957.12.2.el7.x86_64 and the compiler version is GCC 6.4.0. We implemented the TensorSVM in C++ with CUDA. We make use of cuBLAS/cuSOLVER and MKL 2017.3.196 for basic linear algebra operations. Our source code runs 2663 lines long.

The dataset we test against are listed in table 1. We obtain all our datasets from the LIBSVM Dataset repository[3]. For the *Epsilon* dataset we used the training/testing data split from this source. For

[3]http://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets

dataset	# training samples	# testing samples	# features
mushrooms	7,200	924	112
w8a	42,240	7,509	300
*ijcnn*1	81,920	9,781	22
covtype	522,874	58,138	54
SUSY	4,608,000	392,000	18
Epsilon	400,000	100,000	2,000
Higgs	10,000,000	1,000,000	28

Table 1: Data set statistics.

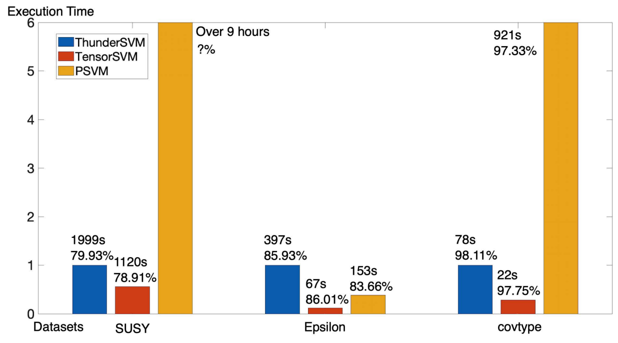


Figure 5: Performance and Accuracy of three state-of-the art SVM solvers on three large scale datasets. The execution time is normalized against ThunderSVM and also annotated along the bars. The percentage annotation refers the prediction accuracy on testing data, using the best model obtained from the same grid search on hyperparameters.

other datasets that do not already have the split, we randomly split the datasets into training and testing sets.

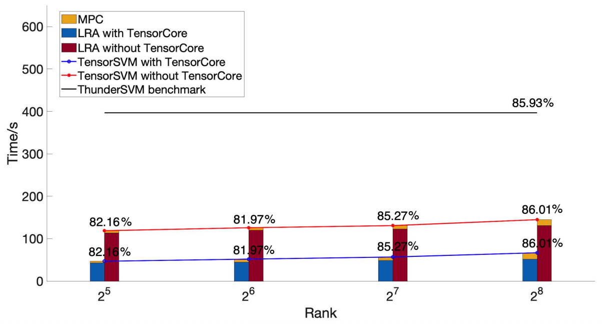
4.1 Speed and Accuracy

In this section, we pick three large datasets in the LIBSVM datasets repository to test the speed and accuracy of the proposed TensorSVM. To put the performance and accuracy in perspective, we consider LIBSVM to be the gold standard in terms of accuracy, and ThunderSVM as the baseline for performance. In fact, ThunderSVM implements a similar algorithm as LIBSVM on GPU with more than 100x speedup, hence we choose ThunderSVM as the baseline for accuracy as well. In fact, ThunderSVM is the fastest SVM solver to achieve near optimal accuracy in all cases we have tested. Another interesting baseline is the PSVM which is closer to TensorSVM in terms of algorithm. PSVM is a distributed memory CPU approximate SVM solver with deterministic low rank approximation to kernel matrix using incomplete Cholesky factorization. In this section we use 512 CPU cores for PSVM.

From figure 5 we can see that for these three large scale datasets, the TensorSVM is **1.8x/6.0x/3.5x** faster than ThunderSVM with almost the same optimal accuracy. On the other hand, PSVM on 512 cores failed to complete in 9 hours for *SUSY*, manage to converge in faster than ThunderSVM on *Epsilon* with significantly inferior accuracy (83.66% vs. 86%), and converges 12x slower than ThunderSVM on *covtype*. These set of experiments seem to suggest that TensorSVM is the fastest among the three solvers, with uncompromising prediction accuracy due to its accurate low rank approximation.

4.2 The Impact of TensorCore and Rank

To reveal the impact of TensorCore on the performance and accuracy of TensorSVM, we conduct further experiments on the *Epsilon* and in figure 6, by comparing TensorSVM with TensorCore enabled and disabled. To provide reference we include the performance and accuracy of ThunderSVM as a baseline. From figure 6a we can see that enabling TensorCore substantially accelerates the TensorSVM, specifically the low rank approximation phase I. Also, we can see that enabling TensorCore does not reduce the prediction accuracy, confirming the analysis in section 3.4. Furthermore, increasing the rank generally increases the low rank approximation accuracy which subsequently increases the prediction accuracy, at the cost of more computations. To summarize, the TensorCore is able to accelerate TensorSVM solver significantly without compromising its accuracy.



(a) *Epsilon*, **TensorSVM**($C = 0.01, \gamma = 0.06$), **ThunderSVM**($C = 0.01, \gamma = 1$)

Figure 6: The training time and prediction accuracy (annotated by the numbers beside the points) differences between TensorSVM(inducing enabling TensorCore and disabling TensorCore) and ThunderSVM for ***epsilon***

4.3 The Empirical Time Complexity Comparison between TensorSVM and ThunderSVM

Theoretically, the time complexity of TensorSVM is $O(n^2k)$. For ThunderSVM, it is unclear because the number of iterations depend on datasets and hyperparameters. So we conduct an empirical study on the asymptotic time complexity of both. The Y-axis in Figure 7 is a log scale of total training time for *SUSY* and the X-axis is the number of training examples. ThunderSVM shows a steeper curve than TensorSVM which is consistent with our assumption. Roughly, the slope of ThunderSVM is 2.5 while TensorSVM is 2 which indicates that TensorSVM has a better asymptotic time complexity - ($O(n^2)$ vs. $O(n^{2.5})$). Therefore, it suggests that TensorSVM is much more scalable to large datasets.

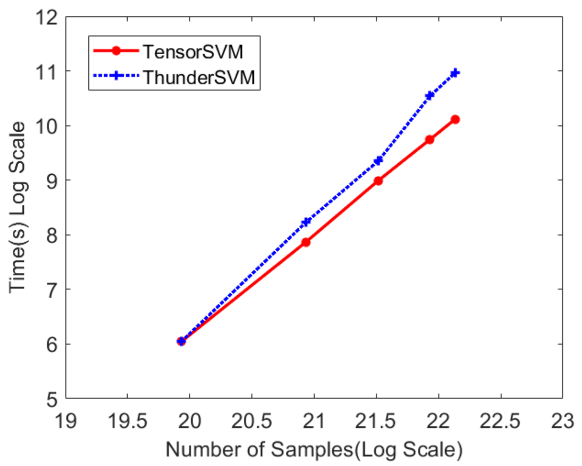


Figure 7: log-log plot of training time comparison for ThunderSVM and TensorSVM with different size of training examples(log base is 2). The slope indicates the power of n in time complexity.

4.4 Roofline Model and Rate of Execution

To understand the detailed execution profile of TensorSVM, we study the three most time consuming components of the TensorSVM training: Kernel Generation (KerGen) and Low Rank Approximation (LRA) in phase I, and Linear Solve (LinSolve) in phase II; see table 2. We calculate the number of FLOPs in each component and its arithmetic intensity, and from which we derive the peak execution rate bounded by the roofline model [35]. If the arithmetic intensity (FLOP/Byte) is larger than the critical intensity we categorize the component as compute bound; otherwise, as memory bandwidth bound. Either way, we derive the peak execution rate based on the arithmetic intensity and roofline model. The actual execution rate is measured. From table 2 we see that our implementation achieved more than 50% of peak execution rate overall, which means that a better implementation cannot improve our speed by 2x. We also note that we achieved up to 48.7 TFLOPS in the LRA operation using TensorCore.

4.5 Non-approximate Linear SVM

It turns out that TensorSVM is also an extremely fast (and the fastest as we know) linear SVM training on datasets with millions of data samples with relatively few features. In this case the kernel matrix (linear kernel XX^T is already the product of low rank factors so can directly feed into Algorithm 2. Note that in this mode TensorSVM is non-approximate, and should give the same results as LIBSVM/ThunderSVM; see Table 3. From the table, we can see that TensorSVM is 55x-125x faster than ThunderSVM for the large scale datasets and parameters we set.

4.6 Accuracy Validation and Comparison with Other Approximate SVM Tools

To validate that our low rank approximation doesn't compromise prediction accuracy on a broader set of benchmarks, we add three more testing data-set with a smaller size in Table 4. In the table, we also include other approximate SVM solvers, such as PSVM, DC-SVM[4], and LLSVM [5], and TensorSVM gives the most consistent and accurate models. Furthermore, we also compare the performance of them on the largest three datasets shown in table 5. We see that TensorSVM is the fastest by a fairly large margin in most cases, except for LLSVM in $SUSY$. However, LLSVM is inaccurate as compared to TensorSVM (accuracy 72% vs. 79%). We see that TensorSVM is able to deliver both great speed and optimal accuracy consistently.

5 RELATED WORK

PSVM [2] is a MPI based parallel implementation of low rank Gram matrix approximation proposed in [12]. It is perhaps the closest to this paper in that they both try to approximate the Gram matrix using a low rank approximate to reduce computation burden, and also use primal dual interior point method for solving the approximated quadratic program. The most salient difference is the low rank approximation algorithm and the linear solver in the Newton steps. In PSVM, the low-rank approximation is accomplished by a pivoted incomplete Cholesky factorization in factored form. It boasts amazing time complexity $O(nk^2)$ just by looking at all elements in the Gram matrix once will incur $O(n^2)$ complexity and numerical robustness in the Newton linear solver, because of the incomplete Cholesky factorization. However, PSVM has two major disadvantage compared to TensorSVM: 1) the low rank approximation quality is significantly lower, thus usually requiring a much larger rank k to obtain enough accuracy to get decent prediction accuracy; 2) the operation in the incomplete Cholesky factorization with pivoting is essentially BLAS2 operation with low arithmetic intensity, which can only be executed at less than 1% of the rate as TensorCore on GPU. We are not aware of GPU implementations of PSVM algorithm. It'll be interesting to further research the prospect of PSVM on GPU and how it compares with TensorSVM and ThunderSVM.

As large scale kernel SVM is difficult to train, various approximation methods are developed to make the training time manageable, usually at the cost of prediction accuracy. One approach that is close to ours is to approximate the Gram matrix with some low-rank one; see Nystrom's method for approximating Gram Matrix [9], Memory efficient Kernel Approximation (MEKA) [31], QR-SVM (Jaas 2017) [5] for L2 SVM (from the standpoint of optimization, an easier problem than L1 SVM that we use), and LLSVM [38] which uses low rank approximation to approximate a non-linear SVM with a linear SVM similar to Nystrom's method. Randomized approximation of the Gram matrix: LLSVM: Nystrom method for nonlinear SVM. FastFood: random Fourier features to approximate the kernel function. See [18] for a comprehensive comparison between many approximate SVM training algorithms. Note that the best of them are usually 10-20x faster than LIBSVM with slightly less prediction accuracy. In contrast, the TensorSVM can often achieve 400x speedup than LIBSVM for kernel SVM training and more than 10000x in linear SVM training (we did not try LIBSVM because it takes too long; our calculation is based on that TensorSVM can be

[4]https://rdrr.io/github/hetong007/SwarmSVM/man/dcSVM.html
[5]https://astro.temple.edu/ tuc17157/budgetedSVM/algorithms.html

	Procedure	# FLOP	Exec. Rate	Peak Rate	Efficiency	Bottleneck	Proportion
epsilon	KerGen	$4n^2$	382.54GFLOPS	659GFLOPS	58.0%	Memory Bandwidth	9.2%
	LRA	$2n^2d + 2n^2k$	48.7TFLOPS	92.1TFLOPS	52.9%	Compute	82.9%
	LinSolve	# iters $\times nk^2$	1.28TFLOPS	6.4TFLOPS	20%	Compute	7.9%
SUSY	KerGen	$4n^2$	386.07GFLOPS	659GFLOPS	58.6%	Memory Bandwidth	33.1%
	LRA	$2n^2d + 2n^2k$	36.47TFLOPS	45.1TFLOPS	80%	Memory Bandwidth	66.2%
	LinSolve	# iters $\times nk^2$	0.879TFLOPS	6.4TFLOPS	13.7%	Compute	0.7%
covtype	KerGen	$4n^2$	383.85GFLOPS	659GFLOPS	58.2%	Memory Bandwidth	31.6%
	LRA	$2n^2d + 2n^2k$	28.32TFLOPS	51.1TFLOPS	55.5%	Memory Bandwidth	63.2%
	LinSolve	# iters $\times nk^2$	0.326TFLOPS	6.4TFLOPS	5.1%	Compute	5.2%

Table 2: Main components of TensorSVM and their efficiency.

	TensorSVM		ThunderSVM		
dataset	time(s)	accuracy	time(s)	accuracy	speedup
covtype	25	76%	1123	76%	44x
SUSY	211	78.6%	26400	78.8%	125x
Higgs	526	61.9%	>43200	?	>82x

Table 3: Linear SVM performance and prediction accuracy for TensorSVM and ThunderSVM with hyperparameter $C = 32$

Dataset	TensorSVM	PSVM	DC-SVM	LLSVM
mushrooms	100%	100%	100%	100%
w8a	100%	97.17%	99.45%	97.66%
*ijcnn*1	98.67%	99.43%	99.73%	97.67%
covtype	97.75%	97.33%	97.15%	66.34%
epsilon	86.01%	83.66%	?	82.62%
SUSY	78.91%	?	?	72.42%

Table 4: Accuracy validation and comparison with PSVM, DC-SVM and LLSVM. The ? fields represent termination due to 9 hours time limit.

Dataset	TensorSVM	PSVM	DC-SVM	LLSVM
covtype	22s	921s	13750s	505s
epsilon	57s	153s	>9 hours	2002s
SUSY	1120s	>9 hours	>9 hours	603s

Table 5: Performance comparison with PSVM, DC-SVM and LLSVM on large datasets.

more than 100x faster than ThunderSVM, which in turn is consistently more than 100x faster than LIBSVM).

Non-approximation SVM and software: LIBSVM [1] is probably the most popular software package that supports linear and non-linear L1-SVM training. The optimization method used is a highly customized variant of coordinate gradient descent method (similar to sequential minimal optimization, SMO [28]). LIBLINEAR [11] include a trust region Newton method for TRON [22] for L2-SVM only because of the indifferentiability of the hinge loss function of L1-SVM. Lee and Roth [20] described a distributed box-constrained quadratic optimization for the dual linear SVM by using a block-diagonal approximation of the Hessian. ThunderSVM [34] is a new improved implementation of LibSVM that utilizes multiple cores and GPUs to accelerate the training and inference. The supported models are the same as LibSVM; the optimization technique used is a variant of the popular SMO. BudgetedSVM [7] is a toolbox in C++ that implements four algorithms for training non-linear classifier for Large-scale data: Adaptive Multi-hyperplane Machines (AMM) (trained by stochastic gradient descent), LLSVM [38] (dual coordinate descent), Budgeted Stochastic Gradient Descent (BSGD), and Primal Estimated sub-Gradient Solver for SVM (Pegasos) [30] SVM-QP [29] proposes and implements an active set algorithm to solve the quadratic programming problem in the kernel SVM dual problem.

NVIDIA introduced TensorCore technology with its Volta architecture [27] in 2017. Resources about NVIDIA TensorCore include detailed micro-architecture analysis and benchmarking [19], an early report on the programmability, performance, and precision [23]. In [4] important parallel primitives reduction and scan are accelerated with TensorCore. In [13–15] TensorCore was used for accelerating linear system solvers in the framework of hybrid CPU/GPU linear algebra package MAGMA [8]. There are numerous use cases of half precision or even lower precision in the application of neural networks.

6 CONCLUSION

In this paper, we set out to make use of the emerging Tensor Engines to accelerate statistical machine learning. In this endeavor, we discovered the great synergy between Tensor Units and Randomized Low Rank Approximation where the limited precision/range of half precision format are well tolerated and compensated. Together with an interior point method, we propose a novel approximate non-linear SVM solver which appears to be consistently and considerably faster and/or accurate than the well researched state-of-the-art. We conducted detailed experiments to validate the efficiency and accuracy of the proposed system. The findings of the paper should not be limited in the SVM solver. Two future directions seem to be profitable: 1) extending the highly efficient kernel matrix approximation on Tensor Units to other kernel machines such as Kernel Logistic Regression, and Gaussian Process. 2) extending and refining the use the Tensor Units on randomized numerical linear algebra to achieve great speedup that deterministic algorithms cannot.

ACKNOWLEDGEMENTS

We are very grateful for the valuable suggestions from the anonymous reviewers and their time and efforts in reviewing the manuscript.

We would like to thank the University of Houston and the state of Texas for providing the startup funding in supporting this work. We acknowledge the use of the Sabine Cluster and the advanced support from the Research Computing Data Core at the University of Houston to carry out the research presented here. We are also grateful for the generous of hardware donation of NVIDIA Corporation. We gratefully acknowledge the Texas Advanced Computing Center (TACC) at The University of Texas at Austin for providing HPC resources that have contributed to the research results reported within this paper.

REFERENCES

[1] Chih-chung Chang, Chih-jen Lin, and Tijmen Tieleman. 2008. LIBSVM : A Library for Support Vector Machines. *ACM Transactions on Intelligent Systems and Technology (TIST)* 307 (2008), 1–39. https://doi.org/10.1145/1961189.1961199 arXiv: 0-387-31073-8 ISBN: 2157-6904.

[2] Edward Y. Chang, Kaihua Zhu, Hao Wang, and Hongjie Bai. 2008. PSVM: Parallelizing Support Vector Machines on Distributed Computers. In *NIPS*. https://doi.org/10.1016/S0016-5085(11)62303-2 ISSN: 0016-5085.

[3] Corinna Cortes and Vladimir Vapnik. 1995. Support-vector networks. *Machine Learning* (1995), 273–297.

[4] Abdul Dakkak, Cheng Li, Isaac Gelado, Jinjun Xiong, and Wen-mei Hwu. 2019. Accelerating Reduction and Scan Using Tensor Core Units. *Proceedings of the ACM International Conference on Supercomputing - ICS '19* (2019), 46–57. https://doi.org/10.1145/3330345.3331057 arXiv: 1811.09736.

[5] Jyotikrishna Dass, V. N.S.Prithvi Sakuru, Vivek Sarin, and Rabi N. Mahapatra. 2017. Distributed QR Decomposition Framework for Training Support Vector Machines. *Proceedings - International Conference on Distributed Computing Systems* (2017), 753–763. https://doi.org/10.1109/ICDCS.2017.222 ISBN: 9781538617915.

[6] James Demmel, Laura Grigori, Mark Hoemmen, and Julien Langou. 2012. Communication-optimal Parallel and Sequential QR and LU Factorizations. *SIAM Journal on Scientific Computing* 34, 1 (Jan. 2012), A206–A239. https://doi.org/10.1137/080731992

[7] Nemanja Djuric, Liang Lan, Slobodan Vucetic, and Zhuang Wang. 2013. BudgetedSVM: A Toolbox for Scalable SVM Approximations. *Journal of Machine Learning Research* 14 (2013), 3813–3817. http://jmlr.org/papers/v14/djuric13a.html ISBN: 1532-4435.

[8] Jack Dongarra, Mark Gates, Azzam Haidar, Jakub Kurzak, Piotr Luszczek, Stanimire Tomov, and Ichitaro Yamazaki. 2014. Accelerating numerical dense linear algebra calculations with GPUs. *Numerical Computations with GPUs* (2014), 3–28. https://doi.org/10.1007/978-3-319-06548-9_1 ISBN: 9783319065489.

[9] Petros Drineas and Michael W. Mahoney. 2005. Approximating a Gram Matrix for Improved Kernel-Based Learning. *Journal ofMachine Learning Research* 6 (2005), 2153–2175. https://doi.org/10.1007/11503415_22 ISBN: 3540265562.

[10] Petros Drineas and Michael W Mahoney. 2016. RandNLA: randomized numerical linear algebra. *Commun. ACM* 59, 6 (2016), 80–90.

[11] Rong-En Fan, Kai-Wei Chang, Cho-Jui Hsieh, Xiang-Rui Wang, and Chih-Jen Lin. 2008. LIBLINEAR: A Library for Large Linear Classification. *Journal of Machine Learning Research* 9 (2008), 1871–1874.

[12] Shai Fine and Katya Scheinberg. 2001. Efficient SVM Training Using Low-Rank Kernel Representations. *Journal of Machine Learning Research* 2 (2001), 243–264. https://doi.org/10.1162/15324430260185619 ISBN: 0048-9697.

[13] Azzam Haidar, Ahmad Abdelfattah, Mawussi Zounon, Panruo Wu, Srikara Pranesh, Stanimire Tomov, and Jack J. Dongarra. 2018. The Design of Fast and Energy-Efficient Linear Solvers: On the Potential of Half-Precision Arithmetic and Iterative Refinement Techniques. In *Computational Science - ICCS 2018 - 18th International Conference, Wuxi, China, June 11-13, 2018, Proceedings, Part I*. 586–600. https://doi.org/10.1007/978-3-319-93698-7_45

[14] Azzam Haidar, Stanimire Tomov, Jack Dongarra, and Nicholas J Higham. 2018. Harnessing GPU Tensor Cores for Fast FP16 Arithmetic to Speed up Mixed-Precision Iterative Refinement Solvers. In *SC*.

[15] Azzam Haidar, Panruo Wu, Stanimire Tomov, and Jack Dongarra. 2017. Investigating Half Precision Arithmetic to Accelerate Dense Linear System Solvers. In *8th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems*.

[16] Azzam Haidar, Panruo Wu, Stanimire Tomov, and Jack J. Dongarra. 2017. Investigating half precision arithmetic to accelerate dense linear system solvers. In *Proceedings of the 8th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems, ScalA@SC 2017, Denver, CO, USA, November 13, 2017*. 10:1–10:8. https://doi.org/10.1145/3148226.3148237

[17] Cho-Jui Hsieh, Kai-Wei Chang, Chih-Jen Lin, S. Sathiya Keerthi, and S. Sundararajan. 2008. A dual coordinate descent method for large-scale linear SVM. *Proceedings of the 25th international conference on Machine learning - ICML '08* 2 (2008), 408–415. https://doi.org/10.1145/1390156.1390208 ISBN: 9781605582054.

[18] Cho-Jui Hsieh and Inderjit S Dhillon. 2014. A Divide-and-Conquer Solver for Kernel Support Vector Machines. (2014), 16.

[19] Zhe Jia, Marco Maggioni, Benjamin Staiger, and Daniele P. Scarpazza. 2018. Dissecting the NVIDIA Volta GPU Architecture via Microbenchmarking. (2018). http://arxiv.org/abs/1804.06826 arXiv: 1804.06826.

[20] Ching-Pei Lee and Dan Roth. 2015. Distributed Box-Constrained Quadratic Optimization for Dual Linear SVM. *Proceedings of the 32nd International Conference on Machine Learning* 37 (2015), 987–996. http://proceedings.mlr.press/v37/leea15.html ISBN: 9781510810587.

[21] Ang Li, Shuaiwen Leon Song, Mark Wijtvliet, Akash Kumar, and Henk Corporaal. 2016. SFU-Driven Transparent Approximation Acceleration on GPUs. In *Proceedings of the 2016 International Conference on Supercomputing - ICS '16*. ACM Press, Istanbul, Turkey, 1–14. https://doi.org/10.1145/2925426.2926255

[22] Chih-jen Jen Lin, Ruby C Weng, and Sathiya Sathiya Keerthi. 2008. Trust Region Newton Method for Logistic Regression. *Journal of Machine Learning Research* 9 (2008), 627–650. https://doi.org/10.1145/1273496.1273567 ISBN: 9781595937933.

[23] Stefano Markidis, Steven Wei Der Chien, Erwin Laure, Ivy Bo Peng, and Jeffrey S. Vetter. 2018. NVIDIA tensor core programmability, performance & precision. *Proceedings - 2018 IEEE 32nd International Parallel and Distributed Processing Symposium Workshops, IPDPSW 2018* (2018), 522–531. https://doi.org/10.1109/IPDPSW.2018.00091 arXiv: 1803.04014 ISBN: 9781538655559.

[24] Per-Gunnar Martinsson. 2016. Randomized methods for matrix computations. *arXiv:1607.01649 [math]* (July 2016). http://arxiv.org/abs/1607.01649 arXiv: 1607.01649.

[25] Per-Gunnar Martinsson and Sergey Voronin. 2016. A randomized blocked algorithm for efficiently computing rank-revealing factorizations of matrices. *SIAM J. Sci. Comput.* 38, 5 (2016), 485–507. https://doi.org/10.1137/15M1026080 arXiv: 1503.07157.

[26] Sanjay Mehrotra. 1992. On the Implementation of a Primal-Dual Interior Point Method. *SIAM Journal on Optimization* 2, 4 (1992), 575–601. https://doi.org/10.1137/0802028 ISBN: 9783540354451.

[27] Nvidia. 2017. *NVIDIA TESLA V100 GPU ARCHITECTURE*. Technical Report. 53 pages. Issue: v1.1.

[28] John C Platt. 1998. *Sequential Minimal Optimization : A Fast Algorithm for Training Support Vector Machines*. Technical Report. Microsoft Research. 1–21 pages.

[29] K Scheinberg. 2006. An efficient implementation of an active set method for SVMs. *Journal of Machine Learning Research* 7 (2006), 2237–2257. ISBN: 1532-4435.

[30] Shai Shalev-Shwartz, Yoram Singer, Nathan Srebro, and Andrew Cotter. 2011. Pegasos: Primal estimated sub-gradient solver for SVM. *Mathematical Programming* 127, 1 (2011), 3–30. https://doi.org/10.1007/s10107-010-0420-4 ISBN: 9781595937933.

[31] Si Si, Cho-Jui Hsieh, and Inderjit Dhillon. 2017. Memory Efficient Kernel Approximation. *Journal of Machine Learning Research* 18 (2017). https://doi.org/10.1063/1.474497 ISBN: 9781634393973.

[32] Alexander J Smola and Bernhard Holkopf. 2000. Sparse Greedy Matrix Approximation for Machine Learning. *Proceedings of the Seventeenth International Conference on Machine Learning* (2000), 911–918. https://doi.org/1558607072 arXiv: 1558607072 ISBN: 1-55860-707-2.

[33] Amber Thomas. 2017. 2017 The State of Data Science & Machine Learning. https://www.kaggle.com/surveys/2017

[34] Zeyi Wen, Jiashuai Shi, Bingsheng He, Qinbin Li, Jian Chen, Kevin Murphy, and Bernhard Schölkopf. 2018. ThunderSVM: A Fast SVM Library on GPUs and CPUs. *Journal of Machine Learning Research* 1, 201x (2018), 1–5.

[35] Samuel Williams, Andrew Waterman, and David Patterson. 2009. Roofline: an insightful visual performance model for multicore architectures. *Commun. ACM* 52, 4 (2009), 65–76.

[36] Robert C Williamson, Alex J Smola, and Bernhard Schölkopf. 2001. Generalization Performance of Regularization Networks and Support Vector Machines Via Entropy Numbers of Compact Operators. 47, 6 (2001), 2516–2532.

[37] Stephen J Wright. 1997. *Primal-dual interior-point methods*. Vol. 54. Siam.

[38] Kai Zhang, Liang Lan, Zhuang Wang, and Fabian Moerchen. 2012. Scaling Up Kernel SVM on Limited Resources: A Low-Rank Linearization Approach. *Proceedings of the 15th International Conference on Artificial Intelligence and Statistics (AISTAT)* (2012), 1425–1434. https://doi.org/10.1109/TNNLS.2018.2838140