Efficient Parallel Algorithms for Linear RankSVM on GPU

Jing Jin, Xiaola Lin

Efficient Parallel Algorithms for Linear RankSVM on GPU

Jing Jin[1] and Xiaola Lin[1]

School of Information Science and Technology,
Sun Yat-sen University, Guangzhou, China
{jinj5@mail2,linxl@mail}.sysu.edu.cn

Abstract. Linear RankSVM is one of the widely used methods for learning to rank. Although using Order-Statistic Tree (OST) and Trust Region Newton Methods (TRON) are effective to train linear RankSVM on CPU, it becomes less effective when dealing with large-scale training data sets. Furthermore, linear RankSVM training with L2-loss contains quite amount of matrix manipulations in comparison with that with L1-loss, so it has great potential for achieving parallelism on GPU. In this paper, we design efficient parallel algorithms on GPU for the linear RankSVM training with L2-loss based on different queries. The experimental results show that, compared with the state-of-the-art algorithms for the linear RankSVM training with L2-loss on CPU, our proposed parallel algorithm not only can significantly enhance the training speed but also maintain the high prediction accuracy.

Keywords: Parallel Computing, GPU Computing, GPU sorting, Linear RankSVM, Learning to Rank

1 Introduction

As a promising parallel device for general-propose computing, Graphics Processing Unit (GPU) not only provides tens of thousands of threads for applications with data-level or task-level parallelism, but also shows superb computational performance on floating point operations in comparison with the current multicore CPUs [1]. Additionally, combining with Compute Unified Device Architecture (CUDA) programming model [2] released by NVIDA in 2007, quite a lot of existing applications can be conveniently programmed and ported to GPUs. Especially, the machines learning algorithms can be highly parallelizable on GPUs since they typically contain a large number of matrix manipulations [3].

According to the Chapelle et. al. [4], state of the art learning to rank models can be categorized into three types: *pointwise methods* such as [5], [6], *pairwise methods* such as [7], [8], [9], and *listwise methods* such as [10], [11]. Among these models, RankSVM, which can be consider as a special case of Support Vector Machine (SVM) [12], is a widely used pairwise approach for leaning to rank. There exists two types of RankSVMs: linear RankSVM [13], [14], [15], [16], [17]

and nonlinear RankSVM [18], [19]. Although both of them have been extensively studied, the lengthy training remains a challenging issue.

Given a set of training label-query-instance tuples $(y_i, q_i, \boldsymbol{x}_i)$, $y_i \in K \subset \mathbb{R}$, $q_i \in Q \subset \mathbb{Z}$, $\boldsymbol{x}_i \in \mathbb{R}^n$, $i = 1, \cdots, l$, where K is the set of possible relevance levels with $|K| = k$, Q is the set of queries with $|Q| = m$, l is the total number of training instances and n is the number of features for each training instance, as well as a defined set of *preference pairs*: $\mathcal{P} \equiv \{(i,j) \mid q_i = q_j, y_i > y_j\}$ with $p \equiv |\mathcal{P}|$, where (i,j) indicates $(\boldsymbol{x}_i, \boldsymbol{x}_j)$ for short, then the objective function $f(\boldsymbol{w})$ of linear RankSVM with L2-loss is presented by:

$$f(\boldsymbol{w}) = \min_{\boldsymbol{w} \in \mathbb{R}^n} \frac{1}{2}\boldsymbol{w}^T\boldsymbol{w} + C \sum_{(i,j)\in\mathcal{P}} \max(0, 1 - \boldsymbol{w}^T(\boldsymbol{x}_i - \boldsymbol{x}_j))^2 \tag{1}$$

where $\boldsymbol{w} \in \mathbb{R}^n$ is a vector of parameters, $C > 0$ is a regularization parameter. The goal of RankSVM is to learn $\boldsymbol{w}$ such that $\boldsymbol{w}^T\boldsymbol{x}_i > \boldsymbol{w}^T\boldsymbol{x}_j$ if $(i,j) \in \mathcal{P}$.

Although there have been many serial algorithms for linear RankSVM, however, there is no empirical research exists on this issue for achieving linear RankSVM on some parallel systems. Moreover, on the one hand, although the linear RankSVM training using TRust regiON Newton methods (TRON) [20] instead of Cutting Plane Method (CPM) may obtain more quick convergence speed, it becomes less effective when dealing with the large-scale training data sets; on the other hand, the linear RankSVM with L2-loss contains more matrix-matrix or matrix-vector operations over that with L1-loss, so training linear RankSVM with L2-loss can be accelerated effectively on GPU. This motivates us to design efficient GPU algorithms to train linear RankSVM with L2-loss.

The main contributions of this paper can be summarized as follows: (1) We define a new rule of how to determine the preference pairs in terms of the different queries (Please see Definition 1); (2) Based on the new rule, we propose a parallel algorithm P-SWX for linear RankSVM training with L2-loss; (3) We propose an efficient GPU sorting algorithm, GPU-quicksorting, that can sort multiple sequences within a single GPU kernel. Meanwhile, we conduct extensive comparison experiments to prove the effectiveness of our proposed algorithms. To the best of our knowledge, this is the first work that achieves RankSVM training on GPU.

The rest of the paper is organized as follows: In Section 2, we briefly introduce the basic principle of linear RankSVM with L2-loss we are interest in solving, as well as its effective solution TRON. Section 3 is mainly devoted to designing parallel algorithms P-SWX and GPU-quicksorting on GPU to accelerate training speed of the linear RankSVM with L2-loss. Experiments, which indicate the performance of our proposed algorithms, are given and analysed in Section 4. Finally, Section 5 summarizes the conclusion of this project and points the future research work.

2 Linear RankSVM Training with L2-loss

This section briefly introduces the linear RankSVM training with L2-loss by using TRON.

2.1 Linear RankSVM Traing with L2-loss by Using Trust Region Newton Method

Typically, TRON can be viewed as an effective Newton method to solve the optimization problem $f(\boldsymbol{w})$, the primary goal of which, at the d-th iteration, is to find a $\boldsymbol{w}^{d+1}$ so that $f(\boldsymbol{w}^{d+1})$ is less than $f(\boldsymbol{w}^d)$. To update $\boldsymbol{w}^d$ by $\boldsymbol{w}^{d+1} = \boldsymbol{w}^d + \boldsymbol{v}$, TRON takes an improved Conjugate Gradient (CG) method to find an optimal direction $\boldsymbol{v} \in \mathbb{R}^n$ by iteratively minimizing $F_d(\boldsymbol{v})$ which is the second-order Taylor approximation of $f(\boldsymbol{w}^{d+1}) - f(\boldsymbol{w}^d)$.

$$\begin{aligned} &F_d(\boldsymbol{v}) \equiv \nabla f(\boldsymbol{w}^d)^T \boldsymbol{v} + \frac{1}{2}\boldsymbol{v}^T \nabla^2 f(\boldsymbol{w}^d)\boldsymbol{v} \\ &\min_{\boldsymbol{v}} \quad F_d(\boldsymbol{v}) \quad \text{subject to} \quad \|\boldsymbol{v}\| \leq \Delta_d \end{aligned} \tag{2}$$

where Δ_d is the size of the trust region, and $\nabla f(\boldsymbol{w}^d)$ and $\nabla^2 f(\boldsymbol{w}^d)$ are indicated the first and second order differential function of $f(\boldsymbol{w})$, respectively. Apparently, TRON contains two levels iterations, inner iterations and outer iterations. The inner one is the CG iterations which are used to find an optimal $\boldsymbol{v}$ within the trust region iteratively for updating $\boldsymbol{w}$, while the outer one is Newton Method which is applied to generate a more optimal $\boldsymbol{w}$ for $f(\boldsymbol{w})$. The whole framework of TRON can be clearly presented by Algorithm 1. Our setting for updating Δ_d follows the work done by Lin et al. [21]. But for the stopping condition, we follow that of TRON in the package LIBLINEAR[1] [22] to check if the gradient is small enough compared with an initial gradient shown as follows.

$$\|\nabla f(\boldsymbol{w}^d)\|_2 \leq \epsilon_s \|\nabla f(\boldsymbol{w}^0)\|_2 \tag{3}$$

where $\boldsymbol{w}^0$ is the initial iteration and ϵ_s is the stopping tolerate given by users.

Algorithm 1 Trust Region Newton Method

Input $\boldsymbol{w}^0 \leftarrow \mathbf{0}$, maximum outer iterations N
Output $\boldsymbol{w}^d$

Initialize $\Delta_0 \leftarrow 0$ and $d \leftarrow 0$
while $d \leq N$ **do**
    //The *while*-loop indicates the whole outer iterations.
    if $\|\nabla f(\boldsymbol{w}^d)\|_2 \leq \epsilon_s \|\nabla f(\boldsymbol{w}^0)\|$ **then**
        return $\boldsymbol{w}^d$
    else
        Apply CG iterations (inner iterations) until subproblem (2) is solved or $\boldsymbol{v}$ reaches the trust-region boundary.
        Update $\boldsymbol{w}^d$ and Δ_d to $\boldsymbol{w}^{d+1}$ and Δ_{d+1} respectively.
        $d \leftarrow d + 1$
    end if
end while

[1] http://www.csie.ntu.edu.tw/ cjlin/liblinear/liblinear-1.94.tar.gz

Optimizing $f(\boldsymbol{w})$ by TRON refers to computing $\nabla f(\boldsymbol{w})$ and $\nabla^2 f(\boldsymbol{w})$. However, the $\nabla^2 f(\boldsymbol{w})$ doesn't exist because $\nabla f(\boldsymbol{w})$ is not differentiable. To derive a faster method to calculate $\nabla^2 f(\boldsymbol{w})\boldsymbol{v}$, Lee et.al. [16] has explored the structure of $\nabla^2 f(\boldsymbol{w})\boldsymbol{v}$ by defining some expressions as follows.

$$\begin{aligned}
&\mathrm{SV}(\boldsymbol{w}) \equiv \{(i,j) \mid (i,j) \in \mathcal{P}, 1 - \boldsymbol{w}^T(\boldsymbol{x}_i - \boldsymbol{x}_j) > 0\} \\
&\mathrm{SV}_i^+ \equiv \{\boldsymbol{x}_j \mid (j,i) \in \mathrm{SV}(\boldsymbol{w})\} \\
&\mathrm{SV}_i^- \equiv \{\boldsymbol{x}_j \mid (i,j) \in \mathrm{SV}(\boldsymbol{w})\} \\
&p_w \equiv |\mathrm{SV}(\boldsymbol{w})|
\end{aligned} \tag{4}$$

$$\begin{aligned}
\beta_i^+ \equiv \left|\mathrm{SV}_i^+\right|, \alpha_i^+ \equiv \sum_{\boldsymbol{x}_j \in \mathrm{SV}_i^+} \boldsymbol{x}_j^T \boldsymbol{v}, \gamma_i^+ \equiv \sum_{\boldsymbol{x}_j \in \mathrm{SV}_i^+} \boldsymbol{w}^T \boldsymbol{x}_j \\
\beta_i^- \equiv \left|\mathrm{SV}_i^-\right|, \alpha_i^- \equiv \sum_{\boldsymbol{x}_j \in \mathrm{SV}_i^-} \boldsymbol{x}_j^T \boldsymbol{v}, \gamma_i^- \equiv \sum_{\boldsymbol{x}_j \in \mathrm{SV}_i^-} \boldsymbol{w}^T \boldsymbol{x}_j
\end{aligned} \tag{5}$$

Following the above definitions, Lee et.al. [16] converted $f(\boldsymbol{w})$ and $\nabla f(\boldsymbol{w})$ and $\nabla^2 f(\boldsymbol{w})\boldsymbol{v}$ into following expressions.

$$\begin{aligned}
f(\boldsymbol{w}) &= \frac{1}{2}\boldsymbol{w}^T\boldsymbol{w} + C(A_w X\boldsymbol{w} - \boldsymbol{e}_w)^T(A_w X\boldsymbol{w} - \boldsymbol{e}_w) \\
&= \frac{1}{2}\boldsymbol{w}^T\boldsymbol{w} + C(\boldsymbol{w}^T X^T((A_w^T A_w X\boldsymbol{w}) - (2A_w^T\boldsymbol{e}_w)) + p_w)
\end{aligned} \tag{6}$$

$$\nabla f(\boldsymbol{w}) = \boldsymbol{w} + 2CX^T((A_w^T A_w X\boldsymbol{w}) - (A_w^T\boldsymbol{e}_w)) \tag{7}$$

$$\nabla^2 f(\boldsymbol{w})\boldsymbol{v} = \boldsymbol{v} + 2CX^T(A_w^T A_w X\boldsymbol{v}) \tag{8}$$

where X indicates $[\boldsymbol{x}_1, \cdots, \boldsymbol{x}_l]^T$, $A_w \in \mathbb{R}^{p_w \times l}$ is a matrix indicated by:

$$A_w \equiv \begin{array}{c} \\ \vdots \\ (i,j) \\ \vdots \end{array} \begin{array}{c} \begin{array}{ccccccc} \cdots & & i & \cdots & & j & \cdots \end{array} \\ \begin{bmatrix} & & & & & & \\ 0\cdots 0 & +1 & 0\cdots 0 & -1 & 0\cdots 0 & & \\ & & & & & & \end{bmatrix} \end{array}$$

$\boldsymbol{e}_w \in \mathbb{R}^{p_w \times l}$ is a vector of ones. Four of $X^T A_w^T A_w X\boldsymbol{v}$, $A_w^T\boldsymbol{e}_w$, p_w and $A_w^T A_w X\boldsymbol{w}$, according to the derivation done by Lee et al. [16], can be computed by:

$$X^T A_w^T A_w X\boldsymbol{v} = X^T \begin{bmatrix} (\beta_1^+ + \beta_1^-)\boldsymbol{x}_1^T\boldsymbol{v} - (\alpha_1^+ + \alpha_1^-) \\ \vdots \\ (\beta_l^+ + \beta_l^-)\boldsymbol{x}_l^T\boldsymbol{v} - (\alpha_l^+ + \alpha_l^-) \end{bmatrix}$$

$$A_w^T\boldsymbol{e}_w = \begin{bmatrix} \beta_1^- - \beta_1^+ \\ \vdots \\ \beta_l^- - \beta_l^+ \end{bmatrix}, \; p_w = \sum_{i=1}^{l} \beta_i^+ = \sum_{i=1}^{l} \beta_i^-$$

$$A_w^T A_w X \boldsymbol{w} = \begin{bmatrix} (\beta_1^+ + \beta_1^-)\boldsymbol{w}^T \boldsymbol{x}_1 - (\gamma_1^+ + \gamma_1^-) \\ \vdots \\ (\beta_l^+ + \beta_l^-)\boldsymbol{w}^T \boldsymbol{x}_l - (\gamma_l^+ + \gamma_l^-) \end{bmatrix}$$

If all β_i^+, β_i^-, α_i^+, α_i^-, γ_i^+ and γ_i^- are already calculated, then computing $\nabla^2 f(\boldsymbol{w})\boldsymbol{v}$ in terms of (8) would cost $O(ln+n)$, where $O(ln)$ is for computing $X^T A_w^T A_w X \boldsymbol{v}$ and $O(n)$ is for vector addition. Similarly, the computations of $\nabla f(\boldsymbol{w})$ and $f(\boldsymbol{w})$ both cost $O(ln+n)$ if all β_i^+, β_i^-, γ_i^+ and γ_i^- are computed already. Furthermore, $\nabla^2 f(\boldsymbol{w})\boldsymbol{v}$ can be viewed as the computational bottlenecks since it refers to not only the CG iteration but also the outer iteration of TRON. According to the definitions of SV_i^+ and SV_i^-, computing all parameter variables in (5) requires to determine whether $1 - \boldsymbol{w}^T(\boldsymbol{x}_i - \boldsymbol{x}_j)$ or $1 - \boldsymbol{w}^T(\boldsymbol{x}_j - \boldsymbol{x}_i)$ is greater than zero. So sorting all $\boldsymbol{w}^T\boldsymbol{x}_i$ before CG iterations of TRON must be a reasonable way to do a quick decision. If all $\boldsymbol{w}^T\boldsymbol{x}_i$ is sorted already, then computing the all parameter variables in (5) by DCM may cost $O(lk)$ [23]. If taking advantage of favourable searching performance of OST, then the $O(lk)$ term is would be reduced to $O(l\log(k))$ [15]. Therefore, by using TRON along with OST, the total computation complexity of linear RankSVM training with L2-loss is equal to $(O(l\log l) + O(ln + l\log(k) + n) \times$ average #CG iterations$) \times$ #outer iterations, where the $O(l\log l)$ term is the cost of sorting all $\boldsymbol{w}^T\boldsymbol{x}_i$.

3 Novel Parallel Algorithms for Linear RankSVM Training with L2-loss on Graphic Processing Units

In this section, we devote to designing efficient parallel algorithms for training linear RankSVM with L2-loss on GPU.

3.1 Efficient Parallel Algorithm for Computing Hessian-vector Product on Graphic Processing Units

As shown in (5), each $\boldsymbol{x}_i$ has one-to-one relationship with the parameters β_i^+, β_i^-, α_i^+, α_i^-, γ_i^+ and γ_i^-. So we can assign a thread to calculate these variables that correspond to $\boldsymbol{x}_i$. Although this rough parallel method can effectively achieve data-level parallelism on GPU, each assigned thread has to execute $O(l)$ steps, which is less effective over DCM ($O(lk)$) or OST ($O(l\log(k))$) if k is small.

However, according to definition of $\mathcal{P}$, $\boldsymbol{x}_i$ and $\boldsymbol{x}_j$ can combine into a preference pair if and only if $q_i = q_j$ holds true. Hence, all $\boldsymbol{x}_i$ (or all y_i) can be divided into m subsets because of existing m different queries in Q. We assume that for a query $Q(t) \in Q$, where $t = 1, \cdots, m$, $X_t = [\boldsymbol{x}_{t1}, \cdots, \boldsymbol{x}_{t|X_t|}]$ and $Y_t = [y_{t1}, \cdots, y_{t|Y_t|}]$ indicate the corresponding subsets of the training instances and labels respectively.

Theorem 1. *If all $\boldsymbol{x}_i$ (or all y_i) are divided into m subsets X_t (or Y_t) in terms of m different queries $Q(t), t = 1, \cdots, m$, then any two of X_t (or Y_t) are independent of each other.*

Proof. According to SV_i^+ and SV_i^-, computing the parameter variables corresponding to $\boldsymbol{x}_i$ (or y_i) only refers to all $\boldsymbol{x}_j$ (or all y_j) satisfying $q_j = q_i$. Therefore, it implies that any two of X_t (Y_t) are independent of each other.

According to Theorem 1, if $\boldsymbol{x}_i \in X_t$ and $y_i \in Y_t$, then computing the parameters β_i^+, β_i^-, α_i^+, α_i^-, γ_i^+ and γ_i^- corresponding to $\boldsymbol{x}_i$ should go through only the subsets X_t and Y_t but not all $\boldsymbol{x}_i$ and all y_i , which can reduce the computation complexity significantly. Meanwhile, we should sort all subsets $w^T X_t = [\boldsymbol{w}^T \boldsymbol{x}_{t1}, \cdots, \boldsymbol{w}^T \boldsymbol{x}_{t|X_t|}]$ independently, instead of all $\boldsymbol{w}^T \boldsymbol{x}_i$. Moreover, if all X_t (or all Y_t) are obtained already, then computing $f(\boldsymbol{w})$, $\nabla f(\boldsymbol{w})$, and $\nabla^2 f(\boldsymbol{w})\boldsymbol{v}$ based on $\mathrm{SV}(\boldsymbol{w})$ is not suitable any more because it refers to all $\boldsymbol{x}_i$ and all y_i.

Definition 1. *For a query $Q(t)$, the rule of how to determine the preference pairs $(i,j), 1 \le i \le |X_t|, 1 \le j \le |X_t|, i \ne j$, is defined as $\mathrm{SV}_t(\boldsymbol{w}) \equiv \{(i,j) \mid 1 - \boldsymbol{w}^T(\boldsymbol{x}_{ti} - \boldsymbol{x}_{tj}) > 0\}$ with $p_t \equiv |\mathrm{SV}_t(\boldsymbol{w})|$.*

The $\mathrm{SV}_t(\boldsymbol{w})$ is similar but essentially different from $\mathrm{SV}(\boldsymbol{w})$ because it only involves the data information related to $Q(t)$. So if all X_t (or all Y_t) are obtained already, then the expressions of (4) and (5) should be transformed into (9) and (10) respectively.

$$\begin{aligned} \mathrm{SV}_{ti}^+ &= \{\boldsymbol{x}_{tj} \mid y_{tj} > y_{ti}, 1 - \boldsymbol{w}^T(\boldsymbol{x}_{tj} - \boldsymbol{x}_{ti}) > 0\} \\ \mathrm{SV}_{ti}^- &= \{\boldsymbol{x}_{tj} \mid y_{tj} < y_{ti}, 1 - \boldsymbol{w}^T(\boldsymbol{x}_{ti} - \boldsymbol{x}_{tj}) > 0\} \end{aligned} \tag{9}$$

$$\begin{aligned} \beta_{ti}^+ = \left|\mathrm{SV}_{ti}^+\right|, \alpha_{ti}^+ &= \sum_{\boldsymbol{x}_{tj} \in \mathrm{SV}_{ti}^+} \boldsymbol{x}_{tj}^T \boldsymbol{v}, \gamma_{ti}^+ = \sum_{\boldsymbol{x}_{tj} \in \mathrm{SV}_{ti}^+} \boldsymbol{w}^T \boldsymbol{x}_{tj} \\ \beta_{ti}^- = \left|\mathrm{SV}_{ti}^-\right|, \alpha_{ti}^- &= \sum_{\boldsymbol{x}_{tj} \in \mathrm{SV}_{ti}^-} \boldsymbol{x}_{tj}^T \boldsymbol{v}, \gamma_{ti}^- = \sum_{\boldsymbol{x}_{tj} \in \mathrm{SV}_{ti}^-} \boldsymbol{w}^T \boldsymbol{x}_{tj} \end{aligned} \tag{10}$$

Accordingly, $f(\boldsymbol{w})\boldsymbol{v}$, $\nabla f(\boldsymbol{w})$ and $\nabla^2 f(\boldsymbol{w})$ have to be converted into:

$$\begin{aligned} f(\boldsymbol{w}) &= \frac{1}{2}\boldsymbol{w}^T\boldsymbol{w} + C(\tilde{A}_w\tilde{X}\boldsymbol{w} - \boldsymbol{e}_w)^T(\tilde{A}_w\tilde{X}\boldsymbol{w} - \boldsymbol{e}_w) \\ &= \frac{1}{2}\boldsymbol{w}^T\boldsymbol{w} + C(\boldsymbol{w}^T\tilde{X}^T((\tilde{A}_w^T\tilde{A}_w\tilde{X}\boldsymbol{w}) - 2(\tilde{A}_w^T\boldsymbol{e}_w)) + \sum_{t=1}^{m} p_t) \end{aligned} \tag{11}$$

$$\nabla f(\boldsymbol{w}) = \boldsymbol{w} + 2C\tilde{X}^T((\tilde{A}_w^T\tilde{A}_w\tilde{X}\boldsymbol{w}) - (\tilde{A}_w^T\boldsymbol{e}_w)) \tag{12}$$

$$\nabla^2 f(\boldsymbol{w})\boldsymbol{v} = \boldsymbol{v} + 2C\tilde{X}^T(\tilde{A}_w^T\tilde{A}_w\tilde{X}\boldsymbol{v}) \tag{13}$$

where $\tilde{X} = [X_1, \cdots, X_m]^T$, $\tilde{Y} = [Y_1, \cdots, Y_m]$, $\tilde{A}_w \in \mathbb{R}^{(\sum_{t=1}^{m} p_t) \times l}$ is as similar as A_w, $p_t = \sum_{i=1}^{|X_t|} \beta_{ti}^+ = \sum_{i=1}^{|X_t|} \beta_{ti}^-$. Of course, combined with (9) and (10), $\tilde{A}_w^T\tilde{A}_w\tilde{X}\boldsymbol{v}$, $\tilde{A}_w^T\tilde{A}_w\tilde{X}\boldsymbol{w}$ and $\tilde{A}_w^T\boldsymbol{e}_w$ can be computed by:

$$\tilde{A}_w^T\tilde{A}_w\tilde{X}\boldsymbol{v} = \begin{bmatrix} (\beta_{11}^+ + \beta_{11}^-)\boldsymbol{x}_{11}^T\boldsymbol{v} - (\alpha_{11}^+ + \alpha_{11}^-) \\ \vdots \\ (\beta_{m|X_m|}^+ + \beta_{m|X_m|}^-)\boldsymbol{x}_{m|X_m|}^T\boldsymbol{v} - (\alpha_{m|X_m|}^+ + \alpha_{m|X_m|}^-) \end{bmatrix}$$

$$\tilde{A}_w^T\tilde{A}_w\tilde{X}\boldsymbol{w} = \begin{bmatrix} (\beta_{11}^+ + \beta_{11}^-)\boldsymbol{w}^T\boldsymbol{x}_{11} - (\gamma_{11}^+ + \gamma_{11}^-) \\ \vdots \\ (\beta_{m|X_m|}^+ + \beta_{m|X_m|}^-)\boldsymbol{w}^T\boldsymbol{x}_{m|X_m|} - (\gamma_{m|X_m|}^+ + \gamma_{m|X_m|}^-) \end{bmatrix}$$

$$\tilde{A}_w^T\boldsymbol{e}_w = \begin{bmatrix} \beta_{11}^- - \beta_{11}^+ \\ \vdots \\ \beta_{m|X_m|}^- - \beta_{m|X_m|}^+ \end{bmatrix}$$

According to Theorem 1, it needs to assign m thread blocks to compute all parameter variables shown in (10) in parallel on GPU. Moreover, each $\boldsymbol{x}_{ti}$ corresponds to β_{ti}^+, β_{ti}^-, α_{ti}^+, α_{ti}^-, γ_{ti}^+ and γ_{ti}^-, so do such computation can effectively achieve data-level parallelism on GPU in terms of the Definition 1. Assume that, for the t-th query $Q(t)$, $\boldsymbol{w}^T X_t'$ indicates the sorted $\boldsymbol{w}^T X_t$, i.e., $\boldsymbol{w}^T X_t' = [\boldsymbol{w}^T\boldsymbol{x}_{t\pi(1)}, \cdots, \boldsymbol{w}^T\boldsymbol{x}_{t\pi(|X_t|)}]$ satisfying $\boldsymbol{w}^T\boldsymbol{x}_{t\pi(1)} \leq, \cdots, \leq \boldsymbol{w}^T\boldsymbol{x}_{t\pi(|X_t|)}$, $X_t'\boldsymbol{v} = [\boldsymbol{x}_{t\pi(1)}^T\boldsymbol{v}, \cdots, \boldsymbol{x}_{t\pi(|X_t|)}^T\boldsymbol{v}]$ and $Y_t' = [y_{t\pi(1)}, \cdots, y_{t\pi(|X_t|)}]$. Then, we map three of $\boldsymbol{w}^T X_t'$, $X_t'\boldsymbol{v}$ and Y_t' into the t-th thread block of GPU jointly for parallel computing.

Based on the above discussions, we propose an efficient parallel algorithm, P-SWX, to compute parameter variables shown in (10) on GPU. The specific steps of P-SWX are clearly shown in Algorithm 2 in which the threads in the t-th thread block should execute at most $O(|X_t|)$ steps. Let $l_L \ll l$ indicates the largest $|X_t|$, then the threads on GPU should execute at most $O(l_L)$ steps.

However, to get more favourable training speed, the matrix operations, including matrix-matrix products, matrix-vector products and vector additions, should be calculated by respectively adopting *Segmm*, *Sgemv* and *Saxay* subroutines in CUBLAS [24]. So if all parameter variables shown in (10) are calculated already, then computing $\nabla^2 f(\boldsymbol{w})\boldsymbol{v}$, $\nabla f(\boldsymbol{w})$ and $f(\boldsymbol{w})$ on GPU by invoking CUBALS may be a more reasonable choice. Although P-SWX and CUBLAS may effectively improve the training speed of linear RankSVM with L2-loss, the sorting costs on CPU, $O(l\log l)$ term, is still high when addressing large-scale training data sets.

3.2 Efficient GPU Sorting for Linear RankSVM Training with L2-loss

As discussed in 3.1, we should assigning m thread blocks to sort all $\boldsymbol{w}^T X_t$ concurrently on GPU. As the subscript i of each $\boldsymbol{w}^T\boldsymbol{x}_{ti}$ needs to be applied in the

Algorithm 2 P-SWX: m thread blocks should be assigned on GPU.

Input $Y_t' \in \mathbb{R}^{|X_t|}$, $\boldsymbol{w}^T X_t' \in \mathbb{R}^{|X_t|}$, $X_t'\boldsymbol{v} \in \mathbb{R}^{|X_t|}$ and $t = 1, \cdots, m$
Output $\beta^+_{t\pi(i)}$, $\beta^-_{t\pi(i)}$, $\alpha^+_{t\pi(i)}$, $\alpha^-_{t\pi(i)}$, $\gamma^+_{t\pi(i)}$ and $\gamma^-_{t\pi(i)}$ ($t = 1, \cdots, m$ and $\pi(i) = 1, \cdots, |X_t|$).
Initialize:$\beta^+_{t\pi(i)} \leftarrow 0$, $\beta^-_{t\pi(i)} \leftarrow 0$, $\alpha^+_{t\pi(i)} \leftarrow 0$, $\alpha^-_{t\pi(i)} \leftarrow 0$, $\gamma^+_{t\pi(i)} \leftarrow 0$ and $\gamma^-_{t\pi(i)} \leftarrow 0$, ($t = 1, \cdots, m$ and $\pi(i) = 1, \cdots, |X_t|$)
$j \leftarrow 1$
while $j \leq |X_t|$, $\boldsymbol{x}_{t\pi(j)} \in \mathrm{SV}^+_{t\pi(i)}$, $\pi(i) = 1, \cdots, |X_t|$ and $t = 1, \cdots, m$ **do**
    $\alpha^+_{t\pi(i)} \leftarrow \alpha^+_{t\pi(i)} + \boldsymbol{x}^T_{t\pi(j)}\boldsymbol{v}$
    $\beta^+_{t\pi(i)} \leftarrow \beta^+_{t\pi(i)} + 1$
    $\gamma^+_{t\pi(i)} \leftarrow \gamma^+_{t\pi(i)} + \boldsymbol{w}^T\boldsymbol{x}_{t\pi(j)}$
    $j \leftarrow j + 1$
end while
$j \leftarrow |X_t|$
while $j \geq 1$, $\boldsymbol{x}_{t\pi(j)} \in \mathrm{SV}^-_{t\pi(i)}$, $\pi(i) = 1, \cdots, |X_t|$ and $t = 1, \cdots, m$ **do**
    $\alpha^-_{t\pi(i)} \leftarrow \alpha^-_{t\pi(i)} + \boldsymbol{x}^T_{t\pi(j)}\boldsymbol{v}$
    $\beta^-_{t\pi(i)} \leftarrow \beta^-_{t\pi(i)} + 1$
    $\gamma^-_{t\pi(i)} \leftarrow \gamma^-_{t\pi(i)} + \boldsymbol{w}^T\boldsymbol{x}_{t\pi(j)}$
    $j \leftarrow j - 1$
end while

next operations such as the operations in Algorithm 2, we should keep the subscript i of each $\boldsymbol{w}^T\boldsymbol{x}_{ti}$. Thus, we should convert each $\boldsymbol{w}^T\boldsymbol{x}_{ti}$ into a corresponding *struct* node that contains two elements *value* and *id*. Taking a $\boldsymbol{w}^T\boldsymbol{x}_{ti}$ for example, the *value* and *id* of its corresponding *struct* node store the value of $\boldsymbol{w}^T\boldsymbol{x}_{ti}$ and i, respectively.

In computer memory, $\boldsymbol{w}^T\tilde{X}^T$ is always stored instead of all $\boldsymbol{w}^T X_t$, thus $\boldsymbol{w}^T\tilde{X}^T$ should be converted into a *struct* sequence d^{pri}, and each $\boldsymbol{w}^T X_t$ corresponds to a subsequence d^{pri}_t of d^{pri}. Apparently, how to locate the boundaries of each d^{pri}_t in d^{pri} is crucial to achieve the sorting in parallel on GPU. Thus, we define a *struct* parameter *workset* with two elements *beg* and *end* to record the boundaries of each subsequence in d^{pri}, where *beg* and *end* store starting position and ending position of a subsequence respectively. Taking the t-th subsequence d^{pri}_t for example, both *beg* and *end* of $workset(t)$ can be calculated by using following expression.

$$workset(t) = \begin{cases} beg = 1 + |X_1| + \cdots + |X_{t-1}| \\ end = |X_1| + |X_2| + \cdots + |X_t| \end{cases} \tag{14}$$

According to the above discussions and work done by D. Cederman et al. [25], we propose an efficient GPU sorting, GPU-quicksorting, by using an auxiliary buffer d^{aux} which is as large as d^{pri}. The basic principle of such a GPU sorting are primarily broken down into two steps. The first one is that if the $|X_t|$, in the t-th thread block, is larger than a user-defined $minsize$, then the d^{pri}_t would be partitioned into two sub-sequences by a randomly selected *pivot*; if not, the

d_t^{pri} would be sorted directly by *bitornic sorting* [26]. The second one is that the each divided subsequence, generated in the first step, with the size $\leq minsize$ would be sorted by *bitornic sorting*, while those/that with the large size should be further divided by the first step until the size of each new divided subsequence has been $\leq minsize$. The specific steps of GPU-quicksorting are presented in Algorithm 3.

Theoretically, our proposed GPU-quicksorting would be an efficient GPU sorting algorithm since it can make full use of the unique properties of GPU to sort multiple sequences in parallel within a single *GPU Kernel*. Consequently, by using Algorithm 1, Algorithm 2, Algorithm 3 and CUBLAS, we can design an efficient GPU implementation P-SWXRankSVM to train linear RankSVM with L2-loss on GPU.

4 Performance Evaluation

In this section, we set two state-of-the-art applications OSTRankSVM[2] and DCMRankSVM as the comparison tools to evaluate P-SWXRankSVM in our experimental tests. The OSTRankSVM is an effective method that solves the linear RankSVM training with L2-loss by TRON along with OST, while DCM-RankSVM is another effective method that solves the linear RankSVM training with L2-loss by TRON along with DCM.

There are six real world training data sets, the size of each of which is clearly presented in Table 1, that are used in our experimental tests. We set C, ϵ_s and $minsize$ to be 1, 10^{-5} and 64 respectively. It is unclear yet if it is the best option, but certainly we would like to try custom setting first. The measurements are carried out in a single server with four Intel(R) Xeon(R) E5620 2.40GHz four-core CPUs and 32GB of RAM running Ubuntu 9.04(64 bit). The graphics processor used is a NVIDA Tesla C2050 card with 448 CUDA cores, and the frequency of each CUDA core is 1.15 GHz. The card has 3GB GDDR5 memory and a memory bandwidth of 144 GB/s. Besides, the CUDA driver and runtime versions used in our experiments are both 5.0, and only one Tesla C2050 card is used in all benchmark tests.

Table 1: Training Data Sets

Data Set	l	n	k	$\|Q\|$	p	l_L
MQ2007-list	743,790	46	1,268	1,017	285,943,893	1,268
MQ2008-list	540,679	46	1,831	471	323,151,792	1,831
MSLR-WEB10K	723,421	136	2	6,000	31,783,391	809
MSLR-WEB30K	2,270,296	136	5	18,919	101,312,036	1,251
MQ2007	42,158	46	2	1017	246,015	147
MQ2008	9,630	46	3	471	52,325	121

[2] http://www.csie.ntu.edu.tw/ cjlin/papers/ranksvm/ranksvml2_exp-1.3.tgz

Algorithm 3 GPU-quicksorting(GPU kernel)

Input $workset$, m, d^{pri} and d^{aux}

1: $bx \leftarrow xblockid$ //$xblockid$: the ID number of thread block
2: **if** $bx \leq m$ **then**
3: $beg, end \leftarrow workset(bx).beg, workset(bx).end$
4: **if** $end - beg < minsize$ **then**
5: $d^{aux}(beg \rightarrow end) \leftarrow d^{pri}(beg \rightarrow end)$
6: $bitonic(d^{aux}(beg \rightarrow end), d^{pri}(beg \rightarrow end))$ //Sort the data.
7: **else**
8: push both beg and end to $workstack$
9: **while** $workstack \neq \emptyset$ **do**
10: $d^{aux}(beg \rightarrow end) \leftarrow d^{pri}(beg \rightarrow end)$
11: $pivot \leftarrow \text{random}(d^{aux}(beg \rightarrow end))$ //Select a $pivot$ randomly.
12: $lt_{threadid}, gt_{threadid} \leftarrow 0, 0$ //$threadid$: the ID number of threads
13: **for** $i \leftarrow beg + threadid, i \leq end, i \leftarrow i + threadcount$ **do**
14: **if** $d^{aux}(i).value \leq pivot.value$ and $pivot.id \neq d^{aux}(i).id$ **then**
15: $lt_{threadid} \leftarrow lt_{threadid} + 1$
16: **else**
17: $gt_{threadid} \leftarrow gt_{threadid} + 1$
18: **end if**
19: **end for**
20: $Cumulative\ \ Sum$: $lt_0, lt_1, lt_2, \cdots, lt_{sum} \leftarrow 0, lt_0, lt_0 + lt_1, \cdots, \sum_{i=0}^{threadcount} lt_i$
21: $Cumulative\ \ Sum$: $gt_0, gt_1, gt_2, \cdots, gt_{sum} \leftarrow 0, gt_0, gt_0 + gt_1, \cdots, \sum_{i=0}^{threadcount} gt_i$
22: $lp, gp \leftarrow beg + lt_{threadid}, end - gt_{threadid+1}$
23: **for** $i \leftarrow beg + threadid, i \leq end, i \leftarrow i + threadcount$ **do**
24: **if** $d^{aux}(i).value \leq pivot.value$ and $pivot.id \neq d^{aux}(i).id$ **then**
25: $d^{pri}(lp) \leftarrow d^{aux}(i).value$, $lp \leftarrow lp + 1$ //Write the data to the left of $pivot$.
26: **else**
27: $d^{pri}(gp) \leftarrow d^{aux}(i).value$, $gp \leftarrow gp - 1$ //Write the data to the right of $pivot$.
28: **end if**
29: **end for**
30: **for** $i \leftarrow beg + lt_{sum} + threadid, i < end - gt_{sum}, i \leftarrow i + threadcount$ **do**
31: $d^{pri}(i) \leftarrow pivot$
32: **end for**
33: pop both beg and end from $workstack$
34: **if** $|d^{pri}(beg \rightarrow (beg + lt_{sum}))| \leq minisize$ **then**
35: $d^{aux}(beg \rightarrow (beg + lt_{sum})) \leftarrow d^{pri}(beg \rightarrow (beg + lt_{sum}))$
36: $bitonic(d^{aux}(beg \rightarrow (beg + lt_{sum})), d^{pri}(beg \rightarrow (beg + lt_{sum})))$
37: **else**
38: push both beg and $beg + lt_{sum}$ to $workstack$
39: $beg, end \leftarrow beg, (beg + lt_{sum})$
40: **end if**
41: **if** $|d^{pri}((end - gt_{sum}) \rightarrow end)| \leq minisize$ **then**
42: $d^{aux}((end - gt_{sum}) \rightarrow end) \leftarrow d^{pri}((end - gt_{sum}) \rightarrow end)$
43: $bitonic(d^{aux}((end - gt_{sum}) \rightarrow end), d^{pri}((end - gt_{sum}) \rightarrow end))$
44: **else**
45: push both $(end - gt_{sum})$ and end to $workstack$
46: $beg, end \leftarrow (end - gt_{sum}), end$
47: **end if**
48: **end while**
49: **end if**
50: **end if**

4.1 Performance Evaluation for P-SWXRankSVM

The specific performance comparisons among DCMRankSVM, OSTRankSVM and P-SWXRankSVM with respect to the different training data sets are presented in Table 2. As shown in the table, the OSTRankSVM performs better than OSTRankSVM, which mainly relies on the superior property of OST ($O(l\log(k))$) in reducing the computation complexity compared to DCM ($O(lk)$). As expected, the P-SWXRankSVM has greater speedup performance over both of DCMRankSVM and OSTRankSVM when addressing the large-scale training data sets. Apparently, such efficient speedup for P-SWXRankSVM mainly depends on that P-SWXRankSVM can make full use of the great computation power of GPU based on our designing.

Table 2: Performance Comparison among DCMRankSVM, OSTRankSVM, P-SWXRankSVM

Data Set	DCMRankSVM Training Time(s)	OSTRankSVM Training Time(s)	P-SWXRankSVM Training Time(s)	P-SWXRankSVM Speedup
MQ2007-list	380.51	203.02	22.46	16.94x/9.04x
MQ2008-list	493.12	276.46	37.17	11.81x/7.44x
MSLR-WEB10K	2791.22	2481.35	276.11	10.11x/8.99x
MSLR-WEB30K	19449.65	17019.16	939.38	20.70x/18.12x
MQ2007	12.67	10.18	5.90	2.18x/1.73x
MQ2008	0.67	0.48	1.37	0.49x/0.35x

Moreover, to analyse the convergence performance of DCMRankSVM, OSTRankSVM and P-SWXRankSVM in more detail, we investigate the relative difference η to the optimal function value shown as:

$$\eta = \left| \frac{f(\boldsymbol{w}) - f(\boldsymbol{w}^*)}{f(\boldsymbol{w}^*)} \right| \tag{15}$$

where the $\boldsymbol{w}^*$ is the optimum of (1), and the ϵ_s is also set to be 10^{-5}.

The measured results involving convergence speed with respect to the different training data sets are clearly illustrated in Figure 1. From the figure, we can observe that the OSTRankSVM converges faster than DCMRankSVM as training time goes, but the convergence speed of OSTRankSVM is not marked enough over that of DCMRankSVM as training time goes if the data sets have a small k. This may be because OST becomes more efficient over DCM if k is large enough. As expected, P-SWXRankSVM can converge much faster than both of DCMRankSVM and OSTRankSVM. However, it is special for "MQ2008". The reason for this case relies on that if the data sets with small size can't effectively utilize the great computation power of GPU, so invoking P-SWXRankSVM to train the small data set, such as "MQ2008", would cost too much time in launching GPU kernels and communicating between CPU and GPU, rather than in computing.

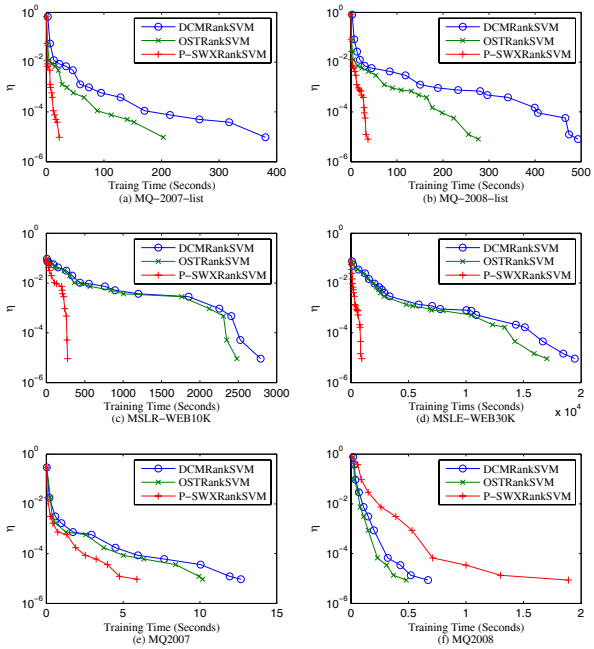

Fig. 1: A Convergence speed comparison among DCMRankSVM, OSTRankSVM, P-SWXRankSVM

4.2 Prediction Performance Evaluation for P-SWXRankSVM

If an optimum $\boldsymbol{w}^*$ of (1) is obtained, then we need to evaluate that such a $\boldsymbol{w}^*$ is whether good or not for doing prediction. In general, checking Pairwise Accuracy (PA) [27] is very suitable for measuring the prediction performance of pairwise approach such as RankSVM. Thus, we choose PA as the measurement in here.

$$\text{PA} \equiv \frac{\mid \{(i,j) \mid \in \mathcal{P}, \boldsymbol{w}^{*T}\boldsymbol{x}_i > \boldsymbol{w}^{*T}\boldsymbol{x}_j\} \mid}{p} \tag{16}$$

The specific measured results are presented in Table 3 clearly. As shown in the table, training linear RankSVM with L2-loss by using anyone of these implementations would result in almost same prediction performance, which effectively proves that P-SWXRankSVM not only can accelerate the linear RankSVM training with L2-loss significantly but also guarantee the prediction accuracy. However, please note that such a case can be explained as follows: In essence, four of DCM, OST, P-SWX and GPU-quicksorting are devoted to improving the training speed of the linear RankSVM with L2-loss, thus they, theoretically, couldn't influence the prediction accuracy.

Table 3: Measured Pairwise Accuracy of DCMRankSVM, OSTRankSVM, P-SWXRankSVM

Data Set	Pairwise Accuracy (PA)		
	DCMRankSVM(%)	OSTRankSVM(%)	P-SWXRankSVM(%)
MQ2007-list	81.11	81.11	81.11
MQ2008-list	82.11	82.11	80.72
MSLR-WEB10K	61.25	61.04	60.43
MSLR-WEB30K	60.96	60.79	60.45
MQ2007	70.59	70.59	70.60
MQ2008	80.24	80.24	80.24

5 Conclusion and Future Work

In this paper, we have proposed two efficient parallel algorithms P-SWX and GPU-quicksorting to accelerate the linear RankSVM training with L2-loss on GPU. To sum up, to design efficient parallel algorithms for the linear RankSVM training with L2-loss, we not only divide all training instances $\boldsymbol{x}_i$ and labels y_i into several independent subsets in terms of different queries, but also redefine a new rule of how to determine the preference pairs (i, j). Just because of this, our proposed parallel algorithms can achieve both task-level and data-level parallelism effectively on GPU. Since this is the initial work to design GPU implementations for training linear RankSVM with L2-loss on a single GPU, there are still many challenges that should to be addressed to further explore their multiple GPU implementations. So, the next extension of this work will use multiple GPU devices to solve even larger training problem in parallel fashion.

Acknowledgement

We would like to thank all anonymous referees for their valuable comments. This research is partially supported by the National Natural Science Foundation of China under Grants No. 60773199, U0735001 and 61073055.

References

1. Owens, J.D., Houston, M., Luebke, D., Green, S., Stone, J.E., Phillips, J.C.: Gpu computing. Proceedings of the IEEE **96**(5) (2008) 879–899
2. Nvidia, C.: Compute unified device architecture programming guide. (2007)
3. Steinkraus, D., Buck, I., Simard, P.: Using gpus for machine learning algorithms. In: Document Analysis and Recognition, 2005. Proceedings. Eighth International Conference on, IEEE (2005) 1115–1120
4. Chapelle, O., Chang, Y.: Yahoo! learning to rank challenge overview. Journal of Machine Learning Research-Proceedings Track **14** (2011) 1–24
5. Fuhr, N.: Optimum polynomial retrieval functions based on the probability ranking principle. ACM Transactions on Information Systems (TOIS) **7**(3) (1989) 183–204

6. Cooper, W.S., Gey, F.C., Dabney, D.P.: Probabilistic retrieval based on staged logistic regression. In: Proceedings of the 15th annual international ACM SIGIR conference on Research and development in information retrieval, ACM (1992) 198–210
7. Freund, Y., Iyer, R., Schapire, R.E., Singer, Y.: An efficient boosting algorithm for combining preferences. The Journal of machine learning research **4** (2003) 933–969
8. Burges, C., Shaked, T., Renshaw, E., Lazier, A., Deeds, M., Hamilton, N., Hullender, G.: Learning to rank using gradient descent. In: Proceedings of the 22nd international conference on Machine learning, ACM (2005) 89–96
9. Quoc, C., Le, V.: Learning to rank with nonsmooth cost functions. Proceedings of the Advances in Neural Information Processing Systems **19** (2007) 193–200
10. Wu, Q., Burges, C.J., Svore, K.M., Gao, J.: Ranking, boosting, and model adaptation. Tecnical Report, MSR-TR-2008-109 (2008)
11. Valizadegan, H., Jin, R., Zhang, R., Mao, J.: Learning to rank by optimizing ndcg measure. In: Advances in neural information processing systems. (2009) 1883–1891
12. Cortes, C., Vapnik, V.: Support-vector networks. Machine learning **20**(3) (1995) 273–297
13. Joachims, T., Finley, T., Yu, C.N.J.: Cutting-plane training of structural svms. Machine Learning **77**(1) (2009) 27–59
14. Sculley, D.: Large scale learning to rank. In: NIPS Workshop on Advances in Ranking. (2009) 1–6
15. Airola, A., Pahikkala, T., Salakoski, T.: Training linear ranking svms in linearithmic time using red–black trees. Pattern Recognition Letters **32**(9) (2011) 1328–1336
16. Lee, C.P., Lin, C.J.: Large-scale linear ranksvm. Neural Computation (2014) 1–37
17. Chapelle, O., Keerthi, S.S.: Efficient algorithms for ranking with svms. Information Retrieval **13**(3) (2010) 201–215
18. Yu, H., Kim, Y., Hwang, S.: Rv-svm: An efficient method for learning ranking svm. In: Advances in Knowledge Discovery and Data Mining. Springer (2009) 426–438
19. Kuo, T.M., Lee, C.P., Lin, C.J.: Large-scale kernel ranksvm
20. Conn, A.R., Gould, N.I., Toint, P.L.: Trust region methods. Number 1. Siam (2000)
21. Lin, C.J., Weng, R.C., Keerthi, S.S.: Trust region newton method for logistic regression. The Journal of Machine Learning Research **9** (2008) 627–650
22. Fan, R.E., Chang, K.W., Hsieh, C.J., Wang, X.R., Lin, C.J.: Liblinear: A library for large linear classification. The Journal of Machine Learning Research **9** (2008) 1871–1874
23. Joachims, T.: A support vector method for multivariate performance measures. In: Proceedings of the 22nd international conference on Machine learning, ACM (2005) 377–384
24. NVIDIA, C.: Cublas library programming guide. NVIDIA Corporation. edit **1** (2007)
25. Cederman, D., Tsigas, P.: A practical quicksort algorithm for graphics processors. In: Algorithms-ESA 2008. Springer (2008) 246–258
26. Batcher, K.E.: Sorting networks and their applications. In: Proceedings of the April 30–May 2, 1968, spring joint computer conference, ACM (1968) 307–314
27. Richardson, M., Prakash, A., Brill, E.: Beyond pagerank: machine learning for static ranking. In: Proceedings of the 15th international conference on World Wide Web, ACM (2006) 707–715