Fast Implementation of String-Kernel-Based Support Vector Classifiers by GPU Computing

Yongquan Shi[1], Tao Ban[2,*], Shanqing Guo[1], Qiuliang Xu[1], and Youki Kadobayashi[2]

[1] Shandong University Jinan 250101, Shandong, China
[2] Information Security Research Center, National Institute of Information and Communications Technology Tokyo 184-8795, Japan

Abstract. Text categorization is widely used in applications such as spam filtering, identification of document genre, authorship attribution, and automated essay grading. The rapid growth in the amount of text data gives rise to the urgent need for fast text classification algorithms. In this paper, we propose a GPU based SVM solver for large scale text datasets. Using Platt's Sequential Minimal Optimization algorithm, we achieve a speedup of 5–40 times over LibSVM running on a high-end traditional processor. Prediction time based on the paralleled string kernel computing scheme shows 5–90 times faster performance than the CPU based implementation.

1 Introduction

Standard learning systems such as neural networks and decision trees operate on input data that are represented as feature vectors. There are many cases, however, where the input data cannot readily be represented as explicit feature vectors, e.g., bio-sequences, images, and text documents. An effective alternative to explicit feature extraction is provided by kernel methods (KM) [1]. The most well-known KM is the Support Vector Machine (SVM) [2][3], which implements the maximum margin principle by means of a convex optimization algorithm in the dual form. Other classic learning algorithms such as Perceptron, Principal Component Analysis (PCA), and Fisher Discriminant Analysis (FDA) can also presented in their dual forms [4][5][6] .

Recently, many string kernels which can incorporate domain-specific knowledge have been studied. String kernel based Support vector machines have shown competitive performance in tasks such as text classification [7] and protein homology detection [8]. A common feature of kernel based methods is their dependency on the kernel matrix computation: training an algorithm in the dual form usually invokes quadratic computation cost in terms of the number of kernel evaluations. For full-matrix based methods such as PCA and FDA, an $N \times N$ kernel matrix need to be computed and stored, and for sparse methods such as SVM and Perceptron, a $V \times N$ sub-matrix are needed, where V is usually in linear relation to N . Popular string kernels such as edit distance kernel and gap-weighted subsequences kernel are computed by dynamic programming and has a computational complexity of $O(|x||y|)$. The computational expense hinders the application of string-kernel based methods on large scale applications. Moreover, many text-related applications require real-time

K.W. Wong et al. (Eds.): ICONIP 2010, Part II, LNCS 6444, pp. 143–151, 2010.

system response, which also calls for an efficient implementation, computationally as well as economically, of string kernel computing algorithms.

The programmable Graphic Processor Unit (GPU) has already been used to implement many algorithms including computational geometry, image processing, as well as computer graphics [10][11]. Specifically, with regards to high-performance computing for pattern recognition, some previous works [12][13] show that general purpose numeric kernel functions including Linear kernel, Radial Basis Function kernel, Polynomial kernel, and Hyperbolic kernel can be paralleled easily with a high speedup compared with CPU based implementations.

Aiming at effectively treating with large scale text-based applications, this paper presents an implementation of string-kernel-based SVM on the CUDA C programming interface. Following the previous works in [12], we modify the Sequential Minimization Optimization algorithm [14] for nonlinear L1 soft-margin SVM algorithm and adopt a justifiable kernel sub-matrix caching method. Emphasis is given on how to parallelize and compute a group of string kernels simultaneously with highly parallelized GPU computing threads. Explored string kernels include p -spectrum kernel, gap-weighted subsequence kernel, and edit distance kernel. We verify the operation and evaluate the performance of the proposed integrated system using Reuters-21578 [15] and SpamAssassin Public Corpus [16]. Our gpuSKSVM achieves speedups of 5-90x over LibSVM running on a high-end traditional processor.

The organization of the paper is as follows. Section 2 describes the SVM and string kernels briefly. Section 3 gives an overview of the architectural and programming features of the GPU. Section 4 presents the details of implementation of the parallel string kernels on the GPU. Section 5 describes the experimental results. Section 6 concludes the paper.

2 Support Vector Machines and String Kernels

2.1 Support Vector Machines

We now focus on the standard two-class soft-margin SVM problem (C-SVM), which classifies a given data point $x \in R^n$ by assigning a label $y \in \{-1, 1\}$. The main task in training SVM is to solve the following quadratic optimization problem with respect to α :

$$\begin{aligned} \min_{\alpha} \quad & \frac{1}{2}\alpha^T Q\alpha - 1^T\alpha \\ \text{Subject to} \quad & y^T\alpha = 0 \\ & 0 \le \alpha_i \le C, i = 1,...,m \end{aligned}$$

where Q is the $N \times N$ positive semi-definite kernel matrix, $Q_{ij} = y_i y_j K(x_i, x_j)$; and $K(x_i, x_j) = (x_i)^T(x_j)$ is the kernel function; and 1 is a vector of all ones.

SVM predict the class label of a new data point x to be classified by the following decision function:

$$f(x) = sign(\sum_{i=1}^{m} y_i \alpha_i K(x_i, x) + b)$$

where b is a bias constant.

2.2 String Kernels

Regular kernels for SVM work mainly on numerical data, which is unsuitable for text-based applications. To extend SVM for text data analysis, we implemented the following string kernels algorithms in our experiments.

2.2.1 Gap-Weighted Subsequence Kernel

Gap Weighted Subsequence (GWS) kernel is to compare strings by way of the common subsequences they share – the more common subsequences and the less gaps (the degree of contiguity between the subsequences), the more the contribute to the inter-string similarity. A decay factor λ is used to weight the presence of a gap in a string. We weight the occurrence of the subsequence u with the exponentially decaying weight $\lambda^{l(i)}$. The feature space for the gap-weighted subsequences kernel of length p is indexed by I = ∑p, with the mapping given by

$$\varphi_u^p(s) = \sum_{i:u=s(i)} \lambda^{l(i)}, u \in \sum^p$$

where i is an index sequence identifying the occurrence of a subsequence $u = s(i)$ in string s and $l(i)$ is the length of s. The corresponding kernel is defined as

$$k_p(s,t) = \langle \varphi^p(s), \varphi^p(t) \rangle = \sum_{u \in \sum^p} \varphi_u^p(s) \varphi_u^p(t)$$

We consider computing an intermediate dynamic programming table $\mathrm{DP_p}$. The complexity of the computation required to compute the table $\mathrm{DP_p}$ is clearly $\mathrm{O}\left(\mathrm{p}\,|\mathrm{s}||\mathrm{t}|\right)$ making the overall complexity of $\mathrm{k_p}\,(\mathrm{s},\,\mathrm{t})$ equal to $\mathrm{O}\left(\mathrm{p}\,|\mathrm{s}||\mathrm{t}|\right)$ [1].

2.2.2 Edit Distance Kernel

Edit distance (or Levenshtein Distance, abbreviated as LD hereinafter) is a measure of the similarity between two strings. More specifically, LD between two strings is the number of insertions, deletions, and substitutions required to transforms a source string s into a target string t. Clearly, the greater the LD, the more different the strings are. For s and t with respective length of n and m, the calculation of LD is a recursive procedure. First set $d(i,0)$ to i for $i = 0, \ldots, \mathrm{n}$, and $d(0, j)$ for $j = 0, \ldots, \mathrm{m}$, . Then, for other pairs i, j we have

$$d(i,j) = \min(d(i-1,j)+1, d(i,j-1)+1, d(i-1,j-1)+r(s(i),t(j)))$$

where for characters a and b, $r(a,b) = 0$ if $a = b$ and $r(a,b) = 1$, otherwise.

Similar to the above GWS kernel, the computational complexity of LD is $\mathrm{O}\,(|\mathrm{s}|\,|\mathrm{t}|)$.

2.2.3 P-spectrum Kernel

p -spectrum transforms strings into high dimensional feature vectors where each feature corresponds to a contiguous substring. Comparing the p -spectra of two strings can give important information about their similarity in applications where contiguity plays an important role. The p -spectrum kernel is defined as the inner product of their p-spectra.

We adopt an idea for p-spectrum kernel from the open source software Shogun [18], which is a large scale machine learning toolbox. This idea for p-spectrum kernel has a time complexity of $O(|s|+|t|)$ after the pretreatment on the input string data.

3 Graphics Processors

Now GPUs are mainly used to accelerate specific parts of an application, being attached to a host CPU that performs most of the control-dominant computation. In this study, the algorithm is implemented on GeForce 9800GT GPU, which is widely available on the market. Related parameters are listed in Table 1.

Table 1. NVIDIA GeForce 9800GT Parameters

Number of multiprocessors	14
Multiprocessor width	8
Multiprocessor share memory size	16KB
Number of stream processors	112
Clock rate	1.57 GHZ
Memory capacity	1024 MB
Memory bandwidth	57.6 GB/S
Compute capacity	1.1

4 Kernels on GPU

We use the SMO, which is first proposed by Platt [14], with the improved first-order variable selection heuristic proposed by Keerthi [9]. In one iteration step of SMO we need to calculate kernel values between a single string and all the training strings. For large data sets, the amount of kernel calculations is very large. Obviously we can use the inherent features of GPU to simplify these calculations, which is the idea used in our study. Although parallelization of a specific kernel algorithm may result in better speedup, we are more interested in developing a generic methodology that can be applicable to string kernels with different functional characteristics. This is especially useful where parallelization of the kernel algorithm is knotty, e.g., LD and subsequence kernels which require dynamic programming.

The efficient implementation relies on one key factor of GPU programming – the trade off between memory access and the number of computing threads running in parallel. The shared memory of a Streaming Multiprocessor (SM), is much faster than local and global memory. However, there is a inherent limit on the volume of shared memory. Unlike the numerical input data, string data take up much more space. Employing the shared memory monotonously will result in very few active threads in an SM. To express thousands of threads making use of the hardware capabilities effectively, we have to employ GPU global memory for input string data and intermediate data. In general, one thread is created for computing $k(x_i, x_j)$, where x_i and x_j stand for two input strings. If some input operand resides in off-chip memory, the latency is much higher. As a result, reducing the number of global memory access and accessing global memory efficiently are two factors for speeding up string kernel.

4.1 p-spectrum Kernel

We adopt the idea for p -spectrum kernel from Shogun. This idea has a time complexity of $O(|s| + |t|)$ after some pretreatment on the input string data.

The pretreatment consists of four steps. The first step is to extract p -mers from input strings. All threads of one block load current input string s into the shared memory concurrently. After s is loaded into shared memory, all the threads of the block extract p -mers of s independently. The extracted p -mers are stored into the shared memory. The second step is to sort the p -mers for sting s . It is probable that one input string contains the same p -mers, so the third step is removing repeated p -mers from the sorted p -mer list of input string s . When detecting a unique p -mer in the sorted p -mer list, occurrence number of the p -mer is attached to the p -mer value. During the repeatability statistics the self kernel $k(s,s)$ of s can be computed. The last step is storing the new sorted p -mer lists for the subsequent kernel evaluation.

In the SMO iteration when a kernel function call is launched, the kernel value $k(x_s, x_i)$ where $i \in [1, N]$ are to be evaluated. Here N stands for the total number of source strings and x_s stands for the common source string between the N kernel value evaluations. We load x_s into the GPU's per SM shared memory. This is the key to the performance improvement, since accessing the shared memory is orders of magnitude faster than accessing the global memory. In order to make use of GPU resources as far as possible every thread is responsible for single kernel value, which means the other source string of each thread is distinct. Because of the limit on the shared memory, x_i for each thread is loaded from global memory during the kernel value computation. Now the p -spectrum kernel value can be achieved easily by traversal of the two p -mer lists. However, the traversal pace of each thread is inconsistent in nature, which destroys the coalesced memory access. This thread branch problem may cause one-third decline in performance, which is a very serious bottleneck.

In consideration of the nature characteristics of desynchronization, it is almost impossible to solve thread branch and memory access latency problem together. We mainly focus on the memory access latency due to its great impact on efficiency. At the beginning all threads in a warp access global memory in coalesced way. Each thread has an independent index variable indicating the current shifting position in shared p -mer list. After one thread gets its global p -mer we let the index of this thread moves forward as far as possible, which means examining the shared p -mer continuously until the next shared p -mer is equal or bigger than the loaded global p -mer element. Thus, each global memory access can be carried out in coalesced way. In view of the above improvements the p -spectrum kernel evaluation on GPU can be much faster than that on CPU.

4.2 LD Kernel and Gap-Weighted Subsequence Kernel

LD kernel and gap-weighted Subsequence Kernel are both dynamic programming applications. A dynamic programming application solves an optimization problem by storing and reusing the results of its sub-problem solutions.

The computational complexity of LD is $O(|s||t|)$. For two strings s and t, the shared string s among all the threads is loaded into shared memory by threads belonging to one block and one $(n+1)(m+1)$ matrix D is required to evaluate the LD $L(s,t)$ between s and t. The potential pairs of sequences are organized in this 2D matrix. The algorithm fills the matrix from top left to bottom right, step-by-step. The value of each data element depends on the values of its northwest-, north- and west-adjacent elements. Matrix row is filled in turn, so only the previous line and the current line of the matrix D is useful. Using this feature one space of length n is enough for storing matrix D, which saves a lot of memory space. Even though we have this shortcut, the intermediate space of each thread is still too much to be saved in the shared memory. In this realization we encounter no thread branch problem. Without loss of generality we compute $D(i, j)$ using $D(i-1, j-1)$, $D(i, j-1)$, and $D(i-1, j)$ from a previous step. Next we compute $D(i, j+1)$, which in turn depends on $D(i-1, j)$, $D(i, j)$, and $D(i-1, j+1)$. Now we find that there is no need to load $D(i-1, j)$ and $D(i, j)$ used by $D(i, j+1)$ from global memory for computing $D(i, j+1)$ after computing $D(i, j)$. This finding halves the memory access operation. The GPU implement of LD kernel is fully compliance with CUDA's parallel constraint. It stands to reason that GPU LD may achieve great speed up.

The GWS kernel is a little bit more complicated than the LD kernel. It involves two intermediate matrices. All the skills employed to compute the LD kernel are applied to compute the GWS kernel. One difference is that there is a simple thread branch in GWS kernel, which affects the parallelism. Due to more time-consuming global memory operations than the LD kernel, speedup on GWS kernel algorithm are not expected to be as significant as the LD kernel.

5 Experiment Results

The current gpuSKSVM system is tested on GeForce 9800 GT. Our gpuSKSVM's performance is compared with LibSVM, which also employs SMO. Experiments are done on three text datasets. Detailed information on these datasets are listed in Table 2. LibSVM was run on an Intel Core 2 Duo 2.33 GHz processor and given a cache size of 800M, which is almost the same as the memory volume limit of our gpuSKSVM. File I/O time was not included for all the solvers. Computation time of gpuSKSVM includes data transfer between GPU and CPU memory.

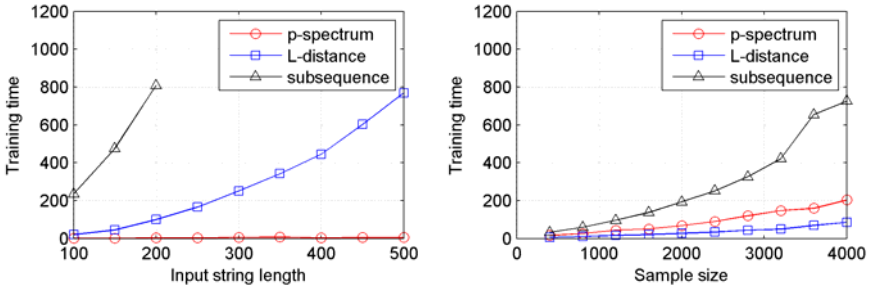
Table 2. Text datasets for benchmarking

DATASET	# POINTS	# LENGTH
Reuters-21578 Earn vs Acq	6295	100
Reuters-21578 Earn vs Rest	10753	100
SpamAssassin Public Corpus	6047	100

Table 3 presents training performance results for the two solvers. We can see that the GPU based implementation in all cases achieves a speedup from 5-90 times compared with the CPU based implementation. Although for p -spectrum kernel and

Table 3. Comparison of GPU and LibSVM on computation time and accuracy

			Training Time(SEC)			Test Time(SEC)			Accuracy	
		Parameters	**CPU**	**GPU**	**Speedup**	**CPU**	**GPU**	**Speedup**	**CPU**	**GPU**
Spam	P-spec.	C=10, L=100,p=4	12.30	2.48	4.96	2.2	0.07	31.43	95.2	95.2
	LD	C=10, L=100,g=0.005	354.66	18.70	18.97	66.42	1.78	37.31	93.7	91.7
	GWS	C=10, L=100,p=5	5748	149.45	38.46	670	44.56	15.04	97.5	95.9
E.V.A	P-spec.	C=10, L=100,p=4	12.60	1.45	8.69	4.80	0.13	36.92	98.06	98.06
	LD	C=10, L=100,g=0.005	317.99	19.87	16.00	100.41	5.89	17.05	96.39	97.50
	GWS	C=10, L=100,p=5	3035	234.56	12.94	805	8.94	90.10	90.22	92.00
E.V.R.	P-spec.	C=10, L=100,p=4	31.70	3.15	10.06	11.50	0.34	33.82	98.87	98.87
	LD	C=10, L=100,g=0.005	761.8	59.15	12.88	215.18	11.97	17.98	94.63	94.82
	GWS	C=10, L=100,p=5	7838	639.61	12.25	2012	359	5.59	98.74	98.8



(a) (b)

Fig. 1. Computation times with different sample and input string length

subsequence kernel, GPU based implementation can only use single-precision float, further detailed inspection on the results shows that there is little difference with respect to the number of iterations until converge. The number of support vector obtained by the two implementations are almost the same. This indicates that the computation cost for prediction are comparable -- recall that the computation cost of prediction heavily depend on kernel computation between support vectors. So the speedup on the prediction can also be deemed as speed improvement on the kernel function evaluations. For p -spectrum kernel, we achieve a speedup on kernel computation with factor of around 30; for GWS kernel, the factor is from 5 to 90, depend on the nature of the problem; for the LD kernel, the CPU algorithm happens to not converge for dataset with sample size over 4,000, and we are inspecting the reason of this problem, and the numerical result will be reported.

In the second group of experiments, we examine how computation time of gpuSKSVM changes with the sample size and input string length. Generally the prediction accuracy increases as the input strings increase in length, because of the increment of presented information. However, as we know, the computation complexity of the kernels also increases along with the string length. In Figure 1(a)

we show the influence of the substring length on the computation time. Theoretically the p-spectrum kernel increase in linear proportion to, and the LD kernel and subsequence kernel increase in quadratic to the length of the input strings, the curves in Figure 1(a) clearly verifies these properties. Moreover, the figure also indicates that the complexity of this learning algorithm does not significantly increase with the input string length -- the number of invoked iteration before convergence remain almost the same when the string length varies. Generally as shown in Figure 1(b), the training time should always increase as the sample size increases. This is more about the nature of the SMO algorithm rather than a feature of its GPU implementation. However, by Figure 1(b), we wan to show that, GPU programming based implementation will speedup the training time even for datasets with considerably small sample size.

6 Conclusion and Future Work

In this paper we presented a high-speed SVM learning algorithm based on CUDA. For the implemented string kernel we achieve considerable speedup and expect better performance by exploiting more effective methodologies. First, we have not yet implemented all the optimizations possible for this problem. For example, LibSVM uses a second order heuristic for picking the pair of samples to optimize at a single iteration of QP optimization, while our GPU implementation uses a first order heuristic, which in most cases leads to more iterations than LibSVM. Second, there could be further optimization of the algorithm exploiting fast speed texture memory.

Acknowledgement. Supported by Specialized Research Fund for the Doctoral Program of Higher Education (Grant No. 20090131120009), Natural Science Foundation of Shandong Province for Youths (Grant No. Q2008G01), Outstanding Young Scientists Foundation Grant of Shandong Province(Grant No. BS2009DX018), Independent Innovation Foundation of Shandong University(Grant No. 2009TS031), The Key Science-Technology Project of Shandong Province of Shandong(Grant No. 2010GGX10117).

References

[1] Shawe-Taylor, J., Cristianini, N.: Kernel Methods for Pattern Analysis. Cambridge University Press, Cambridge (2004)
[2] Vapnik, V.: The Nature of Statistical Learning Theory. Springer, Heidelberg (1995)
[3] Cristianini, N., Shawe-Taylor, J.: An introduction to Support Vector Machines. Cambridge University Press, Cambridge (2000)
[4] Oei, C., Friedland, G., Janin, A.: Parallel Training of a Multi-Layer Perceptron on a GPU. ICSI Technical Report (2009)
[5] Andrecut, M.: Parallel GPU Implementation of Iterative PCA Algorithms. Journal of Computational Biology 16(11), 1593–1599 (2009)
[6] Hall, J.D., Hart, J.C.: GPU Acceleration of Iterative Clustering (2004)
[7] Lodhi, H., Saunders, C., Shawe-Taylor, J., Cristianini, N., Watkins, C.: Text classification using string kernels. Journal of Machine Learning Research (2), 419–444 (2002)

[8] Saigo, H., Vert, J., Ueda, N., Akutsu, T.: Protein homology detection using string alignment kernels. Oxford University Press, Oxford (2004)
[9] Keerthi, S.S., Shevade, S.K., Bhattacharyya, C., Murthy, K.R.K.: Improvements to Platt's SMO Algorithm for SVM Classifier Design. Neural Comput. 13, 637–649 (2001)
[10] Moravanszky, A.: Linear algebra on the GPU. In: Engel, W.F. (ed.) Shader X 2. Wordware Publishing, Texas (2003)
[11] Manocha, D.: Interactive geometric & scientific computations using graphics hardware. In: SIGGRAPH, Tutorial Course#11 (2003)
[12] Catanzaro, B., Sundaram, N., Keutzer, K.: Fast support vector machine training and classification on graphics processors. In: ICML 2008: Proceedings of the 25th International Conference on Machine Learning, pp. 104–111. ACM, New York (2008)
[13] Carpenter, A.: CUSVM: A cuda implementation of support vector classification and regression (2009)
[14] Platt, J.C.: Fast training of support vector machines using sequential minimal optimization. In: Advances in Kernel Methods: Support Vector Learning, pp. 185–208. MIT Press, Cambridge (1999)
[15] http://www.cs.umb.edu/~smimarog/textmining/datasets/index.html
[16] http://spamassassin.apache.org/publiccorpus/