GPU Accelerated Support Vector Machines for Mining High-Throughput Screening Data

Quan Liao,[†] Jibo Wang,[‡] Yue Webster,[‡] and Ian A. Watson*,[‡]

ChemExplorer Co. Ltd., 965 Halei Road, Shanghai 201203, People's Republic of China, and Lilly Research Laboratories, Eli Lilly and Company, Lilly Corporate Center, Indianapolis, Indiana 46285

Received September 4, 2009

Support Vector Machine (SVM), one of the most promising tools in chemical informatics, is time-consuming for mining large high-throughput screening (HTS) data sets. Here, we describe a parallelization of SVM-light algorithm on a graphic processor unit (GPU), using molecular fingerprints as descriptors and the Tanimoto index as kernel function. Comparison experiments based on six PubChem Bioassay data sets show that the GPU version is 43−104× faster than SVM-light for building classification models and 112−212× over SVM-light for building regression models.

INTRODUCTION

High-throughput screening (HTS) is commonly used in drug discovery. A typical HTS often screens hundreds of thousands to millions of compounds and generates large quantities of data. Effectively analyzing this data is obviously vital to the success of any HTS.[1] Various data mining techniques, from simple selection strategies (such as nearest-neighbor and substructure searches) to more complicated machine learning methods (such as Neural Networks[2] and Decision Trees[3]), are now increasingly applied to HTS data analysis.

Support Vector Machine (SVM) is one of the most promising tools in data mining and also has been well recognized in chemical informatics area.[4] Geppert et al.[5] performed systematic database search experiments using five different fingerprints and found that the SVM-based ranking scheme consistently outperformed conventional methods including nearest neighbor and centroid approaches, even if only very small training sets were used for SVM learning. Glick et al.[6] compared the SVM, Laplacian-modified Naïve Bayes, and Recursive Partitioning on four HTS data sets and found SVM outperformed the other two methods in capturing active compounds and scaffolds in the top 1%.

Many SVM models were based on training sets of small or medium size,[7,8] while, on the other hand, more and more SVM models were built with more than 10 000 molecules. Fang et al.[9] applied SVM to a type I methionine aminopeptidases (MetAPs) inhibition assay. The data set contained 43 736 molecules and was randomly split into a training set and a test set with 3:1 ratio. The SVM models built on training data increased the hit rates significantly; 50% of the active compounds could be covered by screening only 7% of the test set. The authors also found that the training set size is very important to the quality of derived models. In that study, at least 10 000 compounds were needed to build a model with reasonable predictive power. Liew et al.[10] built an SVM model for the identification of potential Lymphocyte-specific protein tyrosine kinase (Lck) inhibitors. The model was trained with 820 active compounds and 70 inactive compounds. By adding 65 142 generated putative negatives to the training set, the false positive rate was reduced from 61.2% to 0.52% when validated on the MDDR database. This was not surprising because the model's applicability domain was enlarged by adding diverse putative negatives. Li et al.[11] used SVM to establish a drug-likeness filter by separating the World Drug Index from the Available Chemical Directory and examined the model performance with 5-fold cross-validation (80% for training and 20% for testing). When using a subset of 9208 compounds, the classification accuracy was 87.71%. Yet if using a much larger data set with 341 610 compounds, the classification accuracy increased to 92.73%. Kawai et al.[12] carried out multilabel SVM classification on 100 different drug classes. A data set with 98 634 MDDR compounds was used, 60% as training set, 30% as validation set, and the remaining 10% as testing set. The collective "one against the rest" models could correctly predict 80% of the biological activities.

Training SVM models can be time-consuming because it requires solving very large quadratic programming (QP) optimization problems. A great deal of research has been done to reduce this training time. SVM-light[13] is based on a decomposition idea by Osuna et al.[14] and introduces some efficient optimizations such as shrinking and kernel caching. LibSVM[15] applies Platt's sequential minimal optimization (SMO) method[16] and uses a smart working set selection heuristic. SVM-perf[17] solves the SVM training problem in linear time but can only be used with a linear kernel.

Despite these efforts, SVM training time can still be significant for larger training sets. There have been several previous attempts to further speed up the SVM training process on different parallel hardware architectures.[18,19] Others have implemented SVM codes on high-performance graphical processing units (GPUs). Do et al.[20] implemented a linear SVM solver on GPU and achieved 45× speedup over a CPU implementation. Recently, Catanzaro et al.[21]

* Corresponding author phone: (317) 277-6704; fax: (317) 276-6545; e-mail: watson_ian_a@lilly.com.
[†] ChemExplorer Co. Ltd.
[‡] Eli Lilly and Co.

reported an SVM solver based on the SMO algorithm and achieved speedups of 9−35× over LibSVM.

In this Article, we propose a parallelization of SVM training on GPU hardware. There are several major differences between our work and Catanzaro's work:[21]

(1) Our work is based on SVM-light algorithm, while Catanzaro's work is based on LibSVM's SMO algorithm.

(2) Catanzaro's work uses a dense format to store the input vectors, while our work uses sparse format, which is more suitable for large sparse data sets.

(3) We do not use the standard linear, polynomial, or Gaussian kernel function, but parallelize a Tanimoto kernel that is especially useful in molecular similarity comparisons.

Our implementation is not for general purpose use, but specially adapted for analyzing HTS data, including both classification and regression problems. While there are already many GPU applications for computational chemistry and molecule modeling,[22] our work demonstrates another example where GPU technology will be useful for drug discovery.

METHODS

Support Vector Machines. Given a labeled training set consisting of a set of data points $\mathbf{x}_i$, $i \in \{1, ..., l\}$ with their accompanying labels y_i, $i \in \{1, ..., l\}$, the SVM training problem can be written as the following quadratic program (QP):

minimize

$$W(\boldsymbol{\alpha}) = -\sum_{i=1}^{l} \alpha_i + \frac{1}{2}\boldsymbol{\alpha}^T \mathbf{Q}\boldsymbol{\alpha} \tag{1}$$

subject to

$$\sum_{i=1}^{l} y_i \alpha_i = 0 \tag{2}$$

$$\forall_i : 0 \le \alpha_i \le C \tag{3}$$

where $\boldsymbol{\alpha}$ is a vector of weights; each α_i corresponds to a training example ($\mathbf{x}_i$,y_i), which are being optimized to determine the SVM solution. C is a parameter, which trades classifier generality for accuracy on the training set. $\boldsymbol{Q}$ is a matrix with $Q_{ij} = y_i y_j k(\mathbf{x}_i,\mathbf{x}_j)$, where $k(\mathbf{x}_i,\mathbf{x}_j)$ is a kernel function.

The size of the optimization problem depends on the number of training examples l. Because the size of $\boldsymbol{Q}$ is l^2, it is often impractical to store the whole $\boldsymbol{Q}$ matrix in memory for large training sets with more than 10 000 examples. SVM-light uses a decomposition strategy that splits the variable $\boldsymbol{\alpha}$ into two categories: (1) the set B of free variables, (2) the set N of fixed variables. The set B is usually referred to as "working set", which has a constant size q much smaller than l. Arrange the arrays $\boldsymbol{\alpha}$, $\mathbf{y}$, and $\boldsymbol{Q}$ with respect to B and N,

$$\boldsymbol{\alpha} = \begin{bmatrix}\alpha_B \\ \alpha_N\end{bmatrix}, \mathbf{y} = \begin{bmatrix}\mathbf{y}_B \\ \mathbf{y}_N\end{bmatrix}, Q = [\begin{matrix}Q_{BB}\, Q_{BN} \\ Q_{NB}\, Q_{NN}\end{matrix}] \tag{4}$$

The algorithm works as follows:

(1) While $\boldsymbol{\alpha}$ does not satisfy the optimality conditions.

(1.1) Select working set B with the steepest feasible direction of descent.

(1.2) Solve a QP-subproblem to optimize the $W(\boldsymbol{\alpha})$ on working set B.

(2) Terminate and return $\boldsymbol{\alpha}$.

The implementation of SVM-light is very elegant, introducing an important middle variable $\mathbf{s}$:

$$s_i = \sum_{j=1}^{l} \alpha_j y_j K(\mathbf{x}_i, \mathbf{x}_j) \tag{5}$$

which is commonly needed in working set selection, subproblem optimization, and optimality checking. When $\boldsymbol{\alpha}$ changes from $\boldsymbol{\alpha}^t$ to $\boldsymbol{\alpha}^{t+1}$ at each iteration, s can be updated efficiently using:

$$s_i^{t+1} = s_i^t + \sum_{j \in B} (\alpha_j^{t+1} - \alpha_j^t) y_j K(\mathbf{x}_i, \mathbf{x}_j) \tag{6}$$

Furthermore, SVM-light implements various efficiency optimizations to reduce the computational cost: for example, a caching strategy to reuse previously calculated kernel values, providing a good trade-off between memory consumption and training time; and a shrinking strategy to reduce the size of the problem by removing certain examples that are unlikely to end up as support vectors or likely will be bounded support vectors.

GPU Hardware and CUDA API. The graphic processor unit was initially used as a specialized accelerator for 3D graphics and has evolved into a many-core processor with tremendous computational power and very high memory bandwidth. Current GPU processors can provide up to an order of magnitude greater floating-point capability than their CPU counterparts because the GPU core is specialized for compute-intensive, highly parallel computation, and therefore designed so that more resources are devoted to data processing than data caching or flow control. As a result, achieving high GPU performance is largely dependent on finding high degrees of parallelism to effectively utilize the hardware capabilities. A typical computation running on a GPU often uses thousands of threads.

In this work, we used one newly developed NVIDIA GeForce GTX 280 GPU, which has 1 GB DDR3 RAM (global memory) and 30 single multiprocessors (SMs). Each SM is equipped with 8 scalar cores, 16 384 registers (32-bit), and 16 KB of high-bandwidth low latency memory (shared memory). It supports vectorized single and double precision floating-point operations and is theoretically capable of 1 Tflop computational power.

NVIDIA provides a standard C-like language interface to develop the GPU program, called the Compute Unified Device Architecture (CUDA). In a CUDA program, the CPU code and GPU code are handled in a master-slave model. The CPU is the master node, which calls one or more GPUs as slave nodes. When the CPU code invokes a GPU calculation (called grid), the grid is organized as a set of thread blocks. Each block contains a maximum of 512 threads that execute together, sharing local memories and synchronizing at specified barriers. In contrast, threads belonging to different blocks are not able to communicate efficiently nor to synchronize their execution. Thus, inter-block communication must be avoided in an efficient GPU implementation. The CUDA framework assigns a unique identifier to each block in a grid, and each thread in a block. This makes it possible for each thread to identify its share of the work in a Single Program Multiple Data (SPMD) application.

The processors on the GPU are not able to access CPU memory directly. Therefore, before a GPU calculation is executed, the CPU must copy required data from the CPU memory to the GPU's global memory using CUDA functions, and copy the results back to CPU after a GPU call is finished. Because of the relatively slow transfer speeds between the CPU and GPU, it is important to avoid CPU−GPU data transfer as much as possible. Accessing the GPU global memory is orders of magnitude slower than accessing GPU registers. If threads in a warp access data elements in consecutive global memory locations, the access will separate into two or more transactions. So it is important to ensure coalesced access to global memory. Shared memory is as fast as registers as long as there is no bank conflict. So it is often helpful to copy data from the GPU's global memory to shared memory to decrease references to global memory. More details of the NVIDIA GeForce GPU architecture and the CUDA API are provided in the CUDA Programming Guide.[23]

Tanimoto Kernel Calculation on GPU. Kernel (similarity) is an important concept in SVM, as well as other kernel methods. The default kernel functions in SVM-light include linear, polynomial, radial basis function (RBF), and sigmoid tanh. However, we are most interested in another kernel called Tanimoto coefficients (or Minmax coefficient), which has long been used in chemoinformatics to measure the similarity between molecules and is typically applied to molecular fingerprints. A detailed comparison of a large number of similarity coefficients demonstrates that the Tanimoto coefficient remains the method of choice for the computation of fingerprint-based similarity, despite possessing some inherent biases related to the sizes of the molecules that are being sought.[24] Two comparative studies[25,26] on 22 binary similarity indexes applied to similarity searches conclude that the Tanimoto index may be the single best measure of similarity and that no combinations with other indexes result in consistent improvement over its use. When used as kernel function for SVM, the Tanimoto index also shows its advantage over other general purpose kernels.[27−29] Wale et. al[29] compared the Tanimoto-based kernels and the widely used RBF-based kernels on 28 different classification problems from 18 different bioactivity data sets and found that the classification performance obtained by the Tanimoto-based kernels is in general higher than that obtained by the RBF-based kernels. Molecular fingerprints can be binary presence values or integer counts. Both can be stored in dense format (Figure 1a) or sparse format (Figure 1b). Considering that many fingerprints are very sparse, it is often better to store them in sparse format. For example, the widely used ECFP[30] is often hashed into 32-bit integers, but dense storage of such fingerprints would be impractical due to the need to store 2^32 values. In this study, we used an in-house extended connectivity fingerprint implementation of ECFP. Fingerprint bits are counted and stored in sparse format. We only discuss this kind of fingerprint here.

Given two fingerprints fp_1 and fp_2, the Tanimoto index (T) can be calculated as follows:

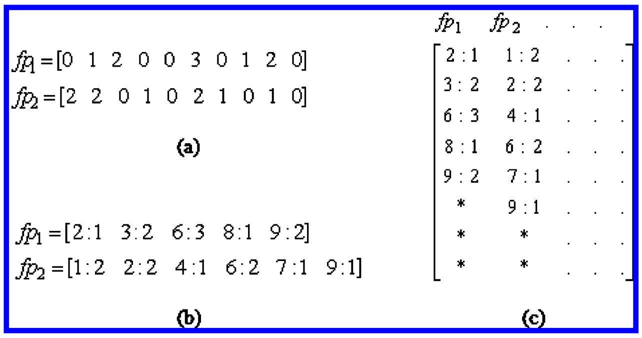
$$T = \frac{c}{a + b - c} \tag{7}$$

where c is the total number of bits common to both in fp_1 and fp_2, a is total bit count in fp_1, and b is total bit count in fp_2. Because the a and b values can be precalculated for each fingerprint, the main workload in Tanimoto calculation is to loop through all bits and sum up the common items c.

Because GPUs need a large number of threads to efficiently exploit parallelism, our GPU code for the Tanimoto calculation is based on a coarse-grained parallelism by computing one row of Tanimoto similarities at each GPU call. Given a query molecular fingerprint fp_q, and a pool of fingerprints fp_i, $i \in \{1, ..., l\}$ with size l, the calculation of one row of Tanimoto similarities is spread into l threads; each thread will calculate one single Tanimoto similarity between fp_q and fp_i. This design is scalable for increasingly large data sets and future GPUs with more cores.

Before the GPU calculation, the whole fingerprint pool should be copied from CPU RAM to the GPU's global memory. To ensure each thread can access the corresponding fingerprint in a coalesced way, the fingerprints are transferred to ALLPACK/ITPACK[31] format and stored in column-major order[32] (Figure 1c). For a data set with l fingerprints and a maxim k nonzero bits per fingerprints, the nonzero values are stored in a dense $l*k$ matrix; each element is an *int2* (CUDA data type for 2 integers) for an index-value pair. The indices for each pair in the same column are already in ascending order. Fingerprints with fewer than k nonzero bits are padded with sentinel index-value pair "*M:0*", where M is the maximal possible bit index in the fingerprint.

Because the query fingerprint fp_q is shared by all threads in the Tanimoto calculation, we copied it into the GPU's shared memory. The copy is needed for every block but can be parallelized by all threads in the same block. With the query fingerprint fp_q in shared memory, and the pool fingerprint fp_i still in global memory, Figure 2 shows the major part of the GPU code for Tanimoto calculation, that is, the calculation of common counts c in eq 7. The code is the same for all threads, but each thread is assigned different fingerprint data specified by thread number i that is calculated from each thread's inherent block and thread identifiers. In the algorithm, (1) is a major for-loop, where (1.1) concurrently reads each row of fp_i from global memory, (1.2) search the same index in fp_q, and (1.3) capture the common counts. Because different threads have different bit indices, the search procedure (1.2) and (1.3) will lead to divergent execution flow. Fortunately, these divergent codes only involve fast


fp_1 = [0 1 2 0 0 3 0 1 2 0]
fp_2 = [2 2 0 1 0 2 1 0 1 0]

(a)

fp_1 = [2:1 3:2 6:3 8:1 9:2]
fp_2 = [1:2 2:2 4:1 6:2 7:1 9:1]

(b)

$$\begin{array}{ccc} fp_1 & fp_2 & \cdots \\ \hline 2:1 & 1:2 & \cdots \\ 3:2 & 2:2 & \cdots \\ 6:3 & 4:1 & \cdots \\ 8:1 & 6:2 & \cdots \\ 9:2 & 7:1 & \cdots \\ * & 9:1 & \cdots \\ * & * & \cdots \\ * & * & \cdots \end{array}$$

(c)

Figure 1. Examples of counted fingerprints (a) in dense format, (b) in sparse format, and (c) in ALLPACK format with column-major order.

```
Algorithm: GPU algorithm for calculating common counts of two fingerprints.
Data:
    a) Matrix FP for fingerprints pool with l columns and k rows, on global memory
    b) Vector Q for query fingerprint, with length k, on shared memory
    c) variable i indicates the fingerprint column that assigned to current thread
Result:
    Vector C for sums of counts, on global memory
begin
  pos = 0
  count = 0
  (1) for (j=0; j<k; j=j+1) do
        (1.1)   bit = FP[j,i]
        (1.2)   while ( Q[pos].index < bit.index ) do
                     pos++
        (1.3)   if ( Q[pos].index = bit.index )
                 then count += min ( Q[pos].value, bit.value )
  (2) C[i] = count
end
```

Figure 2. GPU algorithm for calculating common counts of two fingerprints.

operations on shared memory and registers and do not overly limit the overall performance.

Moving the query fingerprints to shared memory is crucial for performance. There are still some small tricks for improvement. For example, by sorting the fingerprints in the pool by the number of nonzero bits in each fingerprint, adjacent threads in the same warp will have similar workloads and lead to significant speedups. Furthermore, with the sorted fingerprints, early stopping from the for-loop when j reaches the sentinel row can improve performance.

GPU Parallelization of SVM Training. Our GPU code for SVM training is optimized from SVM-light by repeatedly finding the bottleneck and parallelizing it. Some of the steps include:

1. Start with a Clean CPU Version of SVM-Light. SVM-light is a general purpose package, including many functions that we seldom use. We set up a clean CPU version that only handles classification and regression problem and only uses the Tanimoto kernel. We used the caching technique but omitted the shrinking strategy to reduce algorithm complexity. For classification problems, our CPU version is often slower than SVM-light because we disabled the shrinking. For regression problems, SVM-light doubles the size of the training set for +1 label and −1 label, which doubles the time to calculate every kernel row. We slightly modified the caching strategy for regression problems so that both the +1 label and the −1 label of every training sample share the same cache position because they have the same kernel values. In this way, regression in our CPU version is about 2 times faster than in SVM-light. The diagram of our CPU version is as Figure 3.

At each iteration, most time is spent on CALCULATE_KERNEL_ROWS. This step has a time complexity of O(qlk), where q is working set size, l is size of whole training set, and k is the maximum number of nonzero features in any of the training examples. Using the kernel rows stored in cache, UPDATE_S_VARIABLE is done in time O(ql). The other three steps (SELECT_WORKING_SET, OPTIMIZE_SUBPROBLEM, and CHECK_OPTIMALITY) also require time O(ql) as well.

2. CALCULATE_KERNEL_ROWS on GPU. Because kernel evaluation is the major bottleneck, it makes sense to transfer this part of work to the GPU. In the previous section, we have discussed how to calculate one row of the Tanimoto kernel values on the GPU. Here, we simply launch q calls to the GPU, each calculating one kernel row, and then copy the results back to the cache on CPU RAM.

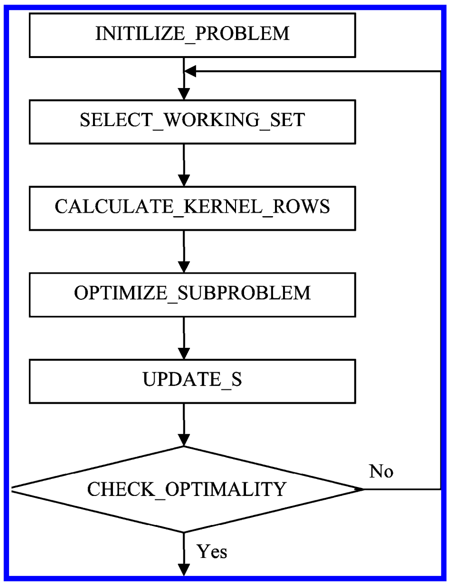


Figure 3. Diagram for simplified CPU version of SVM-light algorithm.

3. Parallel Execution of CALCULATE_KERNEL_ROWS and OPTIMIZE_SUBPROBLEM. The CPU is idle when GPU is executing the kernel calculation. It will be helpful to execute the next step (OPTIMIZE_SUBPROBLEM) on the CPU while waiting for the GPU. The only constraint is that OPTIMIZE_SUBPROBLEM uses a small $q*q$ matrix of kernel values from CALCULATE_KERNEL_ROWS. By recalculating these kernel values on the CPU on-the-fly, instead of reading from cache, the CPU running time for OPTIMIZE_SUBPROBLEM is fully hidden in the shadow of GPU running time.

4. Transfer UPDATE_S to GPU. With the cache on CPU RAM, q rows of kernel values must be copied from the GPU back to the CPU at each iteration. The memory copy between GPU and CPU is very slow and largely limits the performance. Because the cache is only used by UPDATE_S step, we transferred the UPDATE_S step to the GPU, and thus the cache can be allocated on the GPU's global memory. The UPDATE_S calls q iterations of for-loops (eq 6), which is easy to parallelize by passing the q values of $(\alpha_j^{t+1} - \alpha_j^t)$ to the GPU function. The advantage of transferring UPDATE_S to the GPU is dual; for one thing, UPDATE_S itself achieves a large speedup, but also only one instance of GPU−CPU memory copying (the vector **s**) is needed at each iteration, instead of q rows of kernel values.

5. Maximize the Parallelism between GPU and CPU. There is often still some CPU idle time when the GPU is running CALCULATE_KERNEL_ROWS. To make full use of this idle time, we postpone the CHECK_OPTIMALITY until the next iteration so that it is done in parallel with GPU execution. Furthermore, the working set is selected by several steps in SELECT_WORKING_SET step. We launch the GPU call immediately after every working set sample is selected; thus much of the SELECT_WORKING_SET running time is also parallel with the GPU execution.

The diagram of our final parallel version is shown in Figure 4. Two levels of parallelism are included: the first is parallelism between CPU and GPU via a master−slave mode; the second is parallelism of two large for-loops (CALCULATE_KERNEL_ROWS and UPDATE_S) purely on many-core GPU.

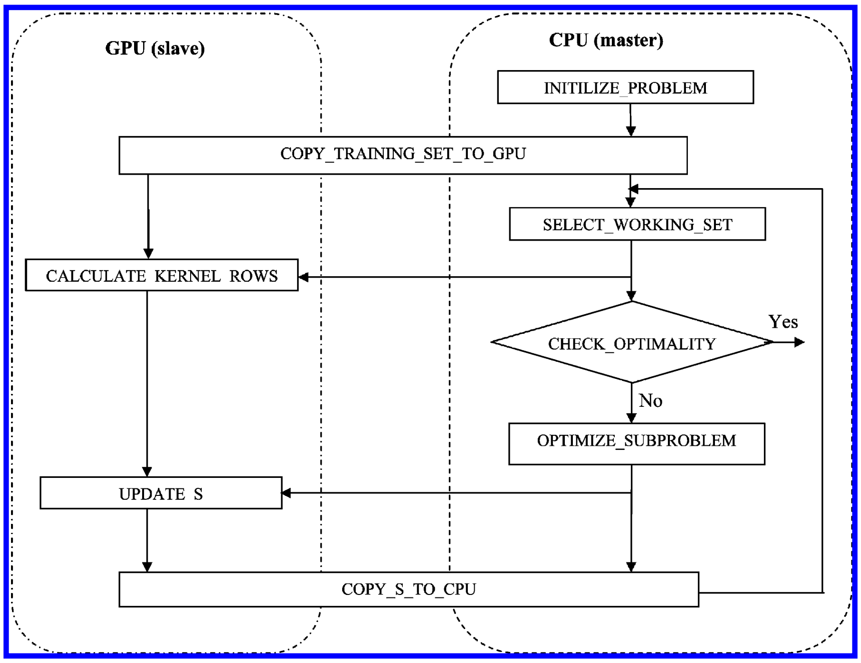


Figure 4. Diagram for parallel version of SVM training algorithm.

Table 1. Summary of Selected PubChem Bioassay Data Sets

data set	PubChem AID	assay description	number of molecules	fingerprint max-bits
PB1044	1044	assay for agonists of Sphingosine 1-Phosphate receptor 1 (S1P1)	15 999	137
PB83	83	MCF-7 human breast tumor cell line growth inhibition assay	27 674	459
PB179	179	AIDS antiviral assay	41 173	407
PB893	893	HSD17B4 hydroxysteroid (17-beta) dehydrogenase 4 inhibition assay	68 476	397
PB778	778	CYP2C19 assay	95 856	197
PB1458	1458	assay for enhancers of SMN2 splice variant expression	196 108	417

RESULTS AND DISCUSSION

Data Sets. We tested the performance of our GPU implementation versus SVM-light on six HTS data sets contained in PubChem bioassay (Table 1), with AID 1044, 83, 179, 893, 778, and 1458. The data sets are randomly selected by evenly covering a wide range of data size, from 15 999 compounds to 196 108 compounds. Because the purpose of this study is not to really discover the structure−activity relationships, we do not select the data sets by their biological targets. In normalized PubChem bioassay data, the activity can be either labeled as 1 (inactive), 2 (active), 3 (inconclusive), or 4 (missing) in the "PUBCHEM_ACTIVITY_OUTCOME" field, or as integers between 0 and 100 in the "PUBCHEM_ACTIVITY_SCORE" field. We cleaned up the data sets by only considering the compounds labeled as inactive or active. Each data set is used to build either a classification model or a regression model, when the response is activity outcomes (active or inactive) or activity scores (0−100), respectively. ECFP descriptors with 4 layers of neighbors are generated for each data set and output to the SVM-light file format. The experiments are all running on a Dell Precision 490 workstation equipped with 4 Intel Xeon 5120 1.86 GHz processors, 2 GB RAM, and an NVIDIA GTX280 card with 1 GB VRAM.

Speedup of Tanimoto Similarity Calculation. To compare the GPU and CPU speed for Tanimoto similarity calculation, we performed an all-to-all computation, which calculates an $l*l$ Tanimoto kernel matrix for data set with size l. The results are listed in Table 2. When only considering the calculation time, the GPU is 66−124× times faster than the CPU; while adding the time for copying the kernel row back to CPU after every GPU call, the speedup decreases to 55−94×.

Table 2. Comparison of All-to-All Tanimoto Calculation on CPU and GPU

data set	CPU (s)	GPU[a] (s)	speedup[a] (×)	GPU[b] (s)	speedup[b] (×)
PB1044	288.8	3.4	84.9	4.4	65.6
PB83	942.1	14.2	66.3	17.1	55.1
PB179	1934.4	24.2	79.9	30.4	63.6
PB893	6139.1	59.7	102.8	76.4	80.4
PB778	12 292.0	99.2	123.9	131.6	93.4
PB1458	52 081.5	420.3	123.9	553.6	94.1

[a] Time is for calculation only. [b] Time is for both calculation and copying the results from GPU back to CPU.

Speedup of SVM Training. We trained the SVM model on all six data sets using the SVM-light code and the GPU code. Both are using the Tanimoto kernel, with a 500 MB cache size. Other parameters remain set to the default values. For the largest PB1458 data set, a 100 MB cache size is used because the large fingerprint data consumes most of the GPU memory. As described in the previous section, five optimization steps are utilized to parallelize the major bottlenecks in the training time. Tablse 3 and 4 give the step-by-step comparison of the training time for classification and regression problems. Because of the tens of times speedup in the GPU accelerated Tanimoto kernel calculation, step 2 gives the most significant contribution among the five steps. Step 4 is another significant step, which moves another time-

Table 3. Comparison of SVM Training Time (s) for Each Optimization Step (Classification)

optimization steps	PB1044	PB83	PB179	PB893	PB778	PB1458
0. original SVM-light version	83.2	603.7	1038.9	5492.0	24 555.6	17 802.3
1. cleaned CPU version	73.2	970.3	1519.1	7449.6	26 964.0	28 733.1
2. kernel calculation on GPU	3.9	29.6	40.8	159.2	538.0	583.2
3. parallel kernel calculation and subproblem optimization	3.6	27.7	39.0	153.7	518.3	576.3
4. update S vector on GPU	1.9	17.4	23.1	89.4	299.7	343.1
5. max parallelism of GPU and CPU	1.4	14.2	19.4	75.1	237.3	265.4

Table 4. Comparison of SVM Training Time (s) for Each Optimization Step (Regression)

optimization steps	PB1044	PB83	PB179	PB893	PB778	PB1458
0. original SVM-light version	861.6	2648.4	7922.0	15 889.8	42 797.5	65 124.0
1. cleaned CPU version	339.6	1075.7	4472.9	11 060.6	18 936.6	49 966.6
2. kernel calculation on GPU	16.2	41.1	125.8	274.2	463.3	1243.5
3. parallel kernel calculation and subproblem optimization	14.8	37.4	115.9	262.9	438.6	1214.6
4. update S vector on GPU	9.7	26.1	79.4	175.8	308.2	824.7
5. max parallelism of GPU and CPU	7.7	20.5	57.9	124.5	202.2	513.5

Table 5. Comparison of SVM Training Time on GPU with CPU (SVM-Light)

	classification			regression		
data set	GPU (s)	CPU (s)	speedup (×)	GPU (s)	CPU (s)	speedup (×)
PB1044	1.4	83.2	59.4	7.7	861.6	111.9
PB83	14.2	603.7	42.5	20.5	2648.4	129.2
PB179	19.4	1038.9	53.6	57.9	7922.0	136.8
PB893	75.1	5492.0	73.1	124.5	15 889.8	127.6
PB778	237.3	24 555.6	103.5	202.2	42 797.5	211.7
PB1458	265.4	17 802.3	67.1	513.5	65 124.0	126.8

Table 6. Comparison of the SVM Models Derived from GPU and CPU (SVM-Light)

	classification				regression			
	support vectors		threshold *b*		support vectors		threshold *b*	
data set	GPU	CPU	GPU	CPU	GPU	CPU	GPU	CPU
PB1044	3494	3498	1.0313	1.0313	15 215	15 221	−0.5814	−0.5815
PB83	10 216	10 214	1.0235	1.0233	26 934	26 933	−42.0118	−42.0119
PB179	10 351	10 364	0.9741	0.9740	37 581	37 598	−37.0116	−37.0116
PB893	19 823	19 803	0.9189	0.9190	30 611	30 660	−0.2017	−0.2018
PB778	54 557	54 555	1.3115	1.3114	94 759	94 763	−1.7549	−1.7548
PB1458	24 923	24 961	1.2466	1.2466	42 710	42 762	−0.0084	−0.0083

consuming function onto the GPU and thus minimizes the slow memory copy between GPU and CPU. The parallel execution of the GPU and the CPU (step 3 and step 5) only results in a low level of parallelism, but can effectively hide the time for nonparallelizable parts of the program and therefore further increase the overall speedup.

When comparing the original SVM-light version and the final optimized GPU version (Table 5), the GPU version is 43−104× faster than SVM-light for classification problems, and 112−212× faster for regression problems.

Training Accuracy. The GPU code uses single-precision float for kernel calculations and uses double-precision for calculation of the **s** vector. This is the same as SVM-light. Because we slightly changed the caching strategy and the working set selection method, as well as the numeric differences between the GPU and the CPU, the model built on a GPU will not be identical to that obtained from SVM-light. Table 6 shows the slight differences in the number of support vectors and the threshold *b* between the GPU version and SVM-light. However, because both the GPU and the SVM-light models are optimized by the same convergence condition, the GPU version provides equivalent accuracy to the SVM-light version. Figure 5 compares the predicted results of 1000 randomly selected molecules by using the GPU and CPU regression models derived from the PB83 data set. The two prediction results are very consistent; the

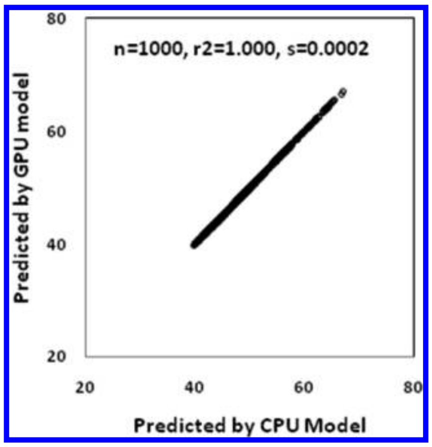


Figure 5. Result predicted by CPU model is consistent with that by GPU model.

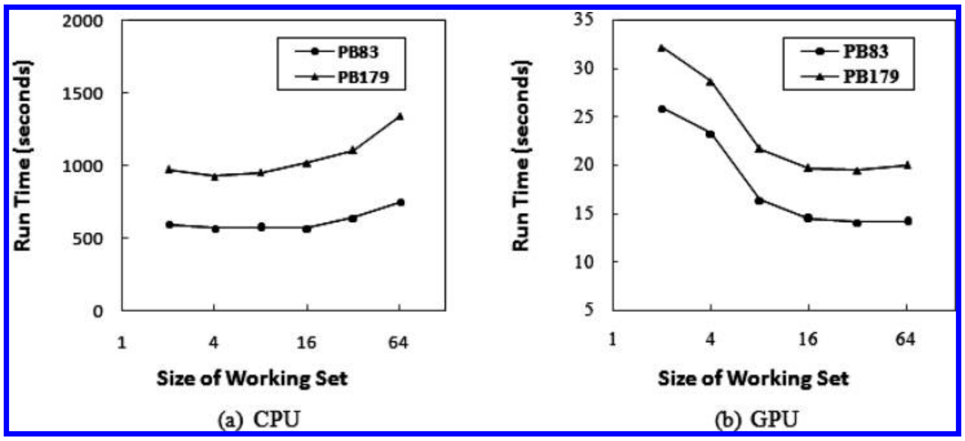


Figure 6. Training time dependent on working set size.

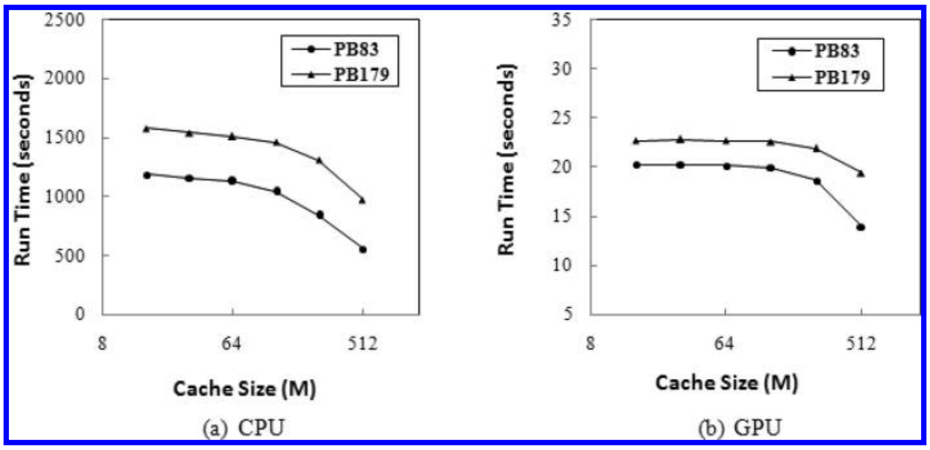


Figure 7. Training time dependent on cache size.

root mean squared deviation between GPU prediction and CPU prediction is 0.0002, and the max absolute deviation is 0.001.

Effect of Working Set Size. In our implementation, a significant part of the training time is spent on selecting the working set. We re-examined the dependence of training time on the size of the working set (see Figure 6). SVM-light uses 10 as the default working set size, but also works for smaller working set sizes such as 2. Yet in our GPU code, training time seems more sensitive to the smaller sizes. Take PB83 as an example, training time for SVM-light is 568.8 s when the working set size is set to 16. If the working set size is changed from 16 to 2, the training time only increased about 4% to 594.8 s. On the contrary, the training time for GPU version increased 79% (from 14.5 to 25.8 s) when the working set size changed from 16 to 2. According to our experiments, 16−32 working set size works well for most tested training cases, and we use 20 as the default value for our program. The widely used SMO algorithm can be seen as a special case of SVM-light algorithm, allowing only for a working set of size 2. Our results indicate that 2 is not the best choice of working set size for GPU parallelization. This may also partly explain why the GPU accelerated SMO algorithm in Catanzaro's work[21] only achieved a speedup of 9−35× for the classification problem, but our GPU accelerated SVM-light algorithm has a higher speedup (43−104×).

Effect of Cache Size. In SVM-light, caching has a strong impact on training time. We compare the CPU and GPU version's training time dependence on cache size (Figure 7). Interestingly, caching is not as important to our GPU version as to the original SVM-light. When the cache size increased from 16 to 512 M, the training time for SVM-light decreased 52.9% for the PB83 data set and 38.2% for the PB179 data set. On the other hand, the training time for the GPU version only decreased 31.0% and 14.5%. The main reason is that kernel calculation on the GPU is concurrent with other computation on the CPU. Simply increasing the cache size will decrease the GPU time for kernel calculation, but the overall time is in turn bounded by the CPU time.

CONCLUSION

This work has demonstrated the application of GPU for SVM training. The speeds for SVM training on classification problems are 43−104× faster than SVM-light. For regression problems, the speedups are 112−212× over SVM-light. These kinds of performance improvements can largely increase the applicability of SVMs in drug discovery. For example, building an SVM regression model on HTS data with about 100 000 molecules takes less than 5 min on a GPU, as compared to about 10 h on a typical CPU.

Similarity calculations are widely used in drug discovery, such as near neighbor search, clustering analysis, library diversity, and compound selection. Many of them are quite time-consuming for large data sets. The parallelization of the Tanimoto similarity calculation in this work opens the door to accelerating other similarity related calculations using GPU technology. The speedup for the Tanimoto similarity calculation is 55−94× in our experiments, which provides a view for how much improvement might be possible with GPU acceleration of other similarity calculation dominated programs. Our future work will focus on accelerating some of these programs on the GPU.

ACKNOWLEDGMENT

We thank Dr. Zhen Wang for helpful discussions.

REFERENCES AND NOTES

(1) Harper, G.; Pickett, S. D. Methods for Mining HTS Data. *Drug Discovery Today* **2006**, *11*, 694–699.

(2) Winkler, D. A. Neural Networks as Robust Tools in Drug Lead Discovery and Development. *Mol. Biotechnol.* **2004**, *27*, 139–167.

(3) Han, L.; Wang, Y.; Bryant, S. H. Developing and Validating Predictive Decision Tree Models from Mining Chemical Structural Fingerprints and High-Throughput Screening Data in PubChem. *BMC Bioinformatics* **2008**, *9*, 401–408.

(4) Ivanciuc, O. Applications of Support Vector Machines in Chemistry. In *Reviews in Computational Chemistry*; Lipkowitz, K. B., Cundari, T. R., Eds.; Wiley−VCH: Weinheim, Germany, 2007; Vol. 23, pp 291−400.

(5) Geppert, H.; Horvath, T.; Gartner, T.; Wrobel, S.; Bajorath, J. Support-Vector-Machine-Based Ranking Significantly Improves the Effectiveness of Similarity Searching Using 2D Fingerprints and Multiple Reference Compounds. *J. Chem. Inf. Model.* **2008**, *48*, 742–746.

(6) Glick, M.; Jenkins, J. L.; Nettles, J. H.; Hitchings, H.; Davies, J. W. Enrichment of High-Throughput Screening Data with Increasing Levels of Noise Using Support Vector Machines, Recursive Partitioning, and Laplacian-Modified Naive Bayesian Classifiers. *J. Chem. Inf. Model.* **2006**, *46*, 193–200.

(7) Weis, D. C.; Visco, D. P., Jr.; Faulon, J.-L. Data Mining PubChem Using a Support Vector Machine with the Signature Molecular Descriptor: Classification of Factor XIa Inhibitors. *J. Mol. Graphics Modell.* **2008**, *27*, 466–475.

(8) Liao, Q.; Yao, J. H.; Yuan, S. G. SVM Approach for Predicting LogP. *Mol. Diversity* **2006**, *10*, 301–309.

(9) Fang, J. W.; Dong, Y. H.; Lushington, G. H.; Ye, Q. Z.; Georg, G. I. Support Vector Machines in HTS Data Mining: Type I MetAPs Inhibition Study. *J. Biomol. Screening* **2006**, *11*, 138–144.

(10) Liew, C. Y.; Ma, X. H.; Liu, X. H.; Yap, C. W. SVM Model for Virtual Screening of Lck Inhibitors. *J. Chem. Inf. Model.* **2009**, *49*, 877–885.

(11) Li, Q. L.; Bender, A.; Pei, J. F.; Lai, L. H. A Large Descriptor Set and a Probabilistic Kernel-Based Classifier Significantly Improve Druglikeness Classification. *J. Chem. Inf. Model.* **2007**, *47*, 1776–1786.

(12) Kawai, K.; Fujishima, S.; Takahashi, Y. Predictive Activity Profiling of Drugs by Topological-Fragment-Spectra-Based Support Vector Machines. *J. Chem. Inf. Model.* **2008**, *48*, 1152–1160.

(13) Joachims, T. Making Large-Scale Support Vector Machine Learning Practical. In *Advances in Kernel Methods: Support Vector Learning*; Schoelkopf, B., Burges, C. J. C., Smola, A. J., Eds.; MIT Press: Cambridge, MA, 1998; pp 42−56.

(14) Osuna, E.; Freund, R.; Girosi, F. An Improved Training Algorithm for Support Vector Machines. In *Neural Networks for Signal Processing VII*; Principe, J., Gile, L., Morgan, N., Wilson, E., Eds.; Proceedings of IEEE Workshop, Amelia Island, FL, 1997; IEEE Press: New York, NY, 1997; pp 276−285.

(15) Fan, R.-E.; Chen, P.-H.; Lin, C.-J. Working Set Selection Using the Second Order Information for Training SVM. *J. Mach. Learn. Res.* **2005**, *6*, 1889–1918.

(16) Platt, J. C. Fast Training of Support Vector Machines Using Sequential Minimal Optimization. In *Advances in Kernel Methods: Support Vector Learning*; Schoelkopf, B., Burges, C. J. C., Smola, A. J., Eds.; MIT Press: Cambridge, MA, 1998; pp 185−208.

(17) Joachims, T. Training Linear SVMs in Linear Time. In *KDD'06*; Eliassi-Rad, T., Ungar, L. H., Craven, D., Cunopulos, D., Eds.; Proceedings of 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Philadelphia, PA, 2006; ACM Press: New York, NY, 2006; pp 217−226.

(18) Graf, H. P.; Cosatto, E.; Bottou, L.; Dourdanovic, I.; Vapnik, V. Parallel Support Vector Machines: The Cascade SVM. In *Advances in Neural Information Processing Systems*; Saul, L. K., Weiss, Y., Bottou, L., Eds.; MIT Press: Cambridge, MA, 2005; Vol. 17, pp 521−528.

(19) Zanni, L.; Serafini, T.; Zanghirati, G. Parallel Software for Training Large Scale Support Vector Machines on Multiprocessor Systems. *J. Mach. Learn. Res.* **2006**, *7*, 1467–1492.

(20) Do, T.-N.; Nguyen, V.-H. A Novel Speed-up SVM Algorithm for Massive Classification Tasks. *RIVF'2008; Proceedings of RIVF'2008*; IEEE Press: Ho Chi Minh, Vietnam, 2008; pp 215−220.

(21) Catanzaro, B.; Sundaram, N.; Keutzer, K. Fast Support Vector Machine Training and Classification on Graphics Processors. In *ICML'08*; Cohen, W. W., McCallum, A., Roweis, S., Eds.; Proceedings of 25th International Conference on Machine Learning, Helsinki, Finland, 2008; ACM Press: New York, NY, 2008; pp 104−111.

(22) Harvey, M. J.; Giupponi, G.; De Fabritiis, G. ACEMD: Accelerating Biomolecular Dynamics in the Microsecond Time Scale. *J. Chem. Theory Comput.* **2009**, *5*, 1632–1639.

(23) NVIDIA Corporation NVIDIA CUDA Programming Guide 2.0, 2008. NVIDIA website. http://developer.download.nvidia.com/compute/cuda/2_0/docs/NVIDIA_CUDA_Programming_Guide_2.0.pdf (accessed Nov 26, 2008).

(24) Willett, P. Similarity Based Virtual Screening Using 2D Fingerprints. *Drug Discovery Today* **2006**, *11*, 1046–1053.

(25) Salim, N.; Holliday, J. D.; Willett, P. Combination of Fingerprint-Based Similarity Coefficients Using Data Fusion. *J. Chem. Inf. Comput. Sci.* **2003**, *43*, 435–442.

(26) Holliday, J. D.; Hu, C.-Y.; Willett, P. Grouping of Coefficients for the Calculation of Intermolecular Similarity and Dissimilarity Using 2D Fragment Bit-Strings. *Comb. Chem. High Throughput Screening* **2002**, *5*, 155–166.

(27) Ralaivola, L.; Swamidass, S. J.; Saigo, H.; Baldi, P. Graph Kernels for Chemical Informatics. *Neural Networks* **2005**, *18*, 1093–1110.

(28) Lind, P.; Maltseva, T. Support Vector Machines for the Estimation of Aqueous Solubility. *J. Chem. Inf. Comput. Sci.* **2003**, *43*, 1855–1859.

(29) Wale, N.; Karypis, G. Acyclic Subgraph based Descriptor Spaces for Chemical Compound Retrieval and Classification. *Technical Report TR 06-008*; Department of Computer Science and Engineering, University of Minnesota: Minneapolis, MN, 2006.

(30) Rogers, D.; Brown, R. D.; Hahn, M. Using Extended-Connectivity Fingerprints with Laplacian-Modified Bayesian Analysis in High-Throughput Screening Follow-up. *J. Biomol. Screening* **2005**, *10*, 682–686.

(31) Grimes, R.; Kincaid, D.; Young, D. ITPACK 2.0 User's Guide. *Technical Report CNA-150*; Center for Numerical Analysts, University of Texas: Austin, TX, 1979.

(32) Bell, N.; Garland, M. Efficient Sparse Matrix-Vector Multiplication on CUDA. *Technical Report NVR-2008-004*; NVIDIA Corp.: Santa Clara, CA, 2008.

CI900337F