# A GPU-Architecture Optimized Hierarchical Decomposition Algorithm for Support Vector Machine Training

Jan Vaněk, Josef Michálek, and Josef Psutka

**Abstract**—In the last decade, several GPU implementations of Support Vector Machine (SVM) training with nonlinear kernels were published. Some of them even with source codes. The most effective ones are based on Sequential Minimal Optimization (SMO). They decompose the restricted quadratic problem into a series of smallest possible subproblems, which are then solved analytically. For large datasets, the majority of elapsed time is spent by a large amount of matrix-vector multiplications that cannot be computed efficiently on current GPUs because of limited memory bandwidth. In this paper, we introduce a novel GPU approach to the SVM training that we call Optimized Hierarchical Decomposition SVM (OHD-SVM). It uses a hierarchical decomposition iterative algorithm that fits better to actual GPU architecture. The low decomposition level uses a single GPU multiprocessor to efficiently solve a local subproblem. Nowadays a single GPU multiprocessor can run thousand or more threads that are able to synchronize quickly. It is an ideal platform for a single kernel SMO-based local solver with fast local iterations. The high decomposition level updates gradients of entire training set and selects a new local working set. The gradient update requires many kernel values that are costly to compute. However, solving a large local subproblem offers an efficient kernel values computation via a matrix-matrix multiplication that is much more efficient than the matrix-vector multiplication used in already published implementations. Along with a description of our implementation, the paper includes an exact comparison of five publicly available C++ SVM training GPU implementations. In this paper, the binary classification task and RBF kernel function are taken into account as it is usual in most of the recent papers. According to the measured results on a wide set of publicly available datasets, our proposed approach excelled significantly over the other methods in all datasets. The biggest difference was on the largest dataset where we achieved speed-up up to 12 times in comparison with the fastest already published GPU implementation. Moreover, our OHD-SVM is the only one that can handle dense as well as sparse datasets. Along with this paper, we published the source-codes at `https://github.com/OrcusCZ/OHD-SVM`.

**Index Terms**—Support Vector Machines, SVM Training, GPU, CUDA, Optimization

## 1 Introduction

Support Vector Machines are popular general purpose learning methods. They offer a good generalization ability through maximizing of the margin controlled by a manual setting regularization constant. SVMs can also deal with a high variability of problems because of a user-defined kernel function. SVMs were originally developed for binary classification, but the multi-class variant is also possible.

Training an SVM amounts to solving a quadratic programming problem. A good overview of optimization techniques can be found in [1]. Very efficient solutions were developed especially for linear or linearized SVMs [2], [3], [4], [5]. Nonlinear SVMs solvers are mostly based on a decomposition technique in the dual formulation of the SVM criterion. The most frequent approach is SMO with the subset of two components which has a simple analytical solution introduced by Platt in [6]. The two-components SMO was generalized to a three-components SMO by Lin in [7]. In contrast, Joachims solves small subproblems by Cholesky factorization [8]. A decomposition technique with a gradient projection of subproblems was proposed by Zani in [9]. Platt's SMO was further improved by Keerthi in [10]. Fan implemented a LibSVM which is based on Keerthi improved SMO [11] and it is still used as a reference due to a robust working set heuristic, a kernel caching, and a shrinking technique.

However, large SVM problems require high-performance implementations to train a model in reasonable time. One option for large dense datasets is to compute a kernel function via CPU optimized Intel or AMD libraries which have also multi-core support. More advanced multi-core and multi-node CPU implementations were described by Elad, Cao, Goncalves, and You in [12], [13], [14], and [15], respectively. Dong in [16] proposed an approach based on a reduced block-diagonal Gram-matrix and Graf in [17] proposed an SVM cascade that has similar behavior: faster elimination of non-support vectors.

In the last decade, GPU computation power have been utilized by machine learning applications widely. Because this paper is about the GPU implementation, we analyzed already published GPU implementations in more detail. Most SVM training GPU implementations were focused on dense data where a higher algorithmic complexity and the regular data structure match better with the GPU architecture. However, there are some sparse implementations, also. Recently, we published a review of the open-source CUDA C implementations [18], where a performance of most of the bellow mentioned implementations were compared.

• *The authors are with the University of West Bohemia, New Technologies for the Information Society, Pilsen, Czech Republic E-mail: {vanekyj, orcus, psutka}@ntis.zcu.cz*

### 1.1 GPUSVM

GPUSVM from Brian Catanzaro et al. [19] is an open-source CUDA implementation of SMO. It supports dense data and two-class SVM classification (C-SVM) with linear, polynomial, radial basis function (RBF), and sigmoid kernel functions. The CUBLAS library for kernel function values was not used. Custom CUDA kernels are used instead. Both, the first and the second order heuristic, were implemented to select the SMO working set. Using the second order heuristic decreases the number of the training iterations needed to converge but the individual iterations require significantly more work than the first order heuristic due to extra kernel function evaluations. Therefore, an adaptive heuristic is used in GPUSVM. It chooses between the two selection heuristics dynamically, with no input or tuning from the user. The adaptive heuristic reduces the total training time on most of the datasets. Better computation effectiveness is also obtained in the case of the first order heuristic with the cache-miss of both working-set vectors. Then, the kernel function is evaluated for both of the vectors at once by a special, more effective, CUDA kernel. GPUSVM achieves a speed-up of a magnitude on dense data sets with the comparison to LibSVM, and it is one of the best performing already published GPU SVM training implementations.

### 1.2 cuSVM

cuSVM from Austin Carpenter [20] is practically just a CUDA reimplementation of the LibSVM. It is open-source and it supports the RBF kernel function only and two-class SVM classification and regression for dense data. It has a Matlab interface for train and prediction functions. The CUBLAS library is used for the matrix-vector multiplication in the RBF kernel function calculation. Although, cuSVM is much faster than LibSVM, in most cases it is slower than GPUSVM.

### 1.3 MultiSVM

Herrero-Lopez in [21] introduced a GPU multi-class SVM training. It has been the first GPU SVM implementation that allowed multi-class classification in a one-vs-all manner besides the two-class problem. It supports dense data and linear, polynomial, RBF, and sigmoid kernel functions. The multi-class implementation does the training of partial SVMs in parallel. So, there is another parallel layer that helps to fully occupy current high-performance GPUs with thousands of cores. A cross-task kernel caching technique is used to significantly reduce the total amount of computations needed to calculate kernel function values where the CUBLAS library is used for the matrix-vector multiplication. It is open-source.

### 1.4 Li-GPUSVM

Qi Li and others from Vojislav Kecman group presented in [22], [23] an SVM package under the same name like Catanzaro: GPUSVM. It offers multi-class and cross-validation abilities. However, it supports only dense data and we were not able to find source codes, therefore we cannot add it into the comparison.

### 1.5 GPUMLib

GPUMLib is a larger open-source machine learning project with a GPU implemented SVM module from Joao Goncalves et al. [24]. Our first preliminary tests showed that it was not stable on some of the tested datasets and first performance results were significantly worse than GPUSVM.

### 1.6 WUSVM

The most recent dense GPU implementation is WUSVM published by Tyree [25]. A sparse primal SVM variant of the training algorithm is implemented in the open-source WUSVM. Linear algebra operations are accelerated via Intel MKL and OpenMP in the CPU variant case, and via the NVIDIA CUBLAS library in the GPU case. Two-class dense problems are supported with all common kernel functions. The algorithm contains random shuffling of the training data by default that brings a stochastic component that produces models with variable performance in variable training times. In most cases, the training times were significantly higher than e.g. GPUSVM.

### 1.7 ELLPACK-R SVM

Tsung-Kai Lin in [26] used the regularized ELLPACK sparse matrix format. It was the first published sparse GPU implementation. However, source-codes are not available. The ELLPACK format is regularized to the CUDA warp size. Than sparse-matrix dense-vector multiplication is very effective on GPU hardware. The SVM training implementation supports the multi-class tasks. In the multi-class training process, all the classifiers are being trained at the same time. All training tasks share the same cache. However, the operation in each task is computed serially. Every different task evokes a kernel at each iteration, instead of evoking only a single kernel like Herrero-Lopezs MultiSVM mentioned above. The serial approach is generally less efficient.

### 1.8 gtSVM

Andrew Cotter in [27] used sparse data format and supported the two-class and even the multi-class SVM classification with the RBF kernel only. The open-source gtSVM offers a Matlab interface. Computation is divided between the CPU and the GPU, with large parallel computations being performed on the graphics hardware and lightweight serial tasks on the host processor. The working set values are passed to the CPU at each iteration. It does not use SMO but it uses a larger working set of size 16. The CPU optimizes the subproblems and the GPU updates all elements and selects the new working set for the next iteration. A clustering algorithm is used to regularize sparsity patterns in data and permits better memory access. The size of the clusters can be selected from two options: large or small which means 256 or 16 samples, respectively. We marked those two variants in the results section of the paper as *gtSVM LC* and *gtSVM SC*.

### 1.9 KMLib

Krzysztof Sopyla in [28] used the standard CSR sparse format that has lower memory requirements than the regularized ones. KMLib is open-source and it was written

in a now outdated version of .net with CUDA platform. Therefore, it is not a standard C++/CUDA project and we have failed to compile and run it successfully.

### 1.10 Non-GPU Implementations

All the above-mentioned GPU implementations were developed in nVidia CUDA. However, other platforms were also in focus: e.g. Cadambi in [29] used FPGA, You in [30] used Intel Xeon Phi coprocessor.

## 2 Support Vector Machine Training

We will start with the simplest case, linear machines trained on separable data. Nonlinear machines trained on non-separable data result in a very similar quadratic programming problem. Suppose we have some hyperplane which separates positive from negative examples. The training data are labeled as $\{\boldsymbol{x}_i, y_i\}, \quad i = 1, \ldots, l, \quad y_i \in \{-1, 1\}, \quad \boldsymbol{x}_i \in \boldsymbol{R}^d$. The points lying on the hyperplane satisfy $\boldsymbol{w} \cdot \boldsymbol{x} + b = 0$, where $\boldsymbol{w}$ is normal to the hyperplane. Define the margin of a separating hyperplane to be $d_+ + d_-$, where $d_+$ ($d_-$) is the shortest distance from the separating hyperplane to the closest positive (negative) example. For linearly separable case, the support vector algorithm simply looks for the separating hyperplane with the largest margin. This can be formulated as follows:

$$y_i(\boldsymbol{x}_i \cdot \boldsymbol{w} + b) - 1 \geq 0 \quad \forall i \tag{1}$$

We can find the solution by minimizing $||\boldsymbol{w}||^2/2$ subject to constraints (1).

The Lagrangian formulation of the problem is as follows:

$$L_P = \frac{1}{2}||\boldsymbol{w}||^2 - \sum_{i=1}^{l} \alpha_i y_i (\boldsymbol{x}_i \cdot \boldsymbol{w} + b) + \sum_{i=1}^{l} \alpha_i \tag{2}$$

We must now minimize $L_P$ with respect to $\boldsymbol{w}$, $b$ and simultaneously require that the derivatives of $L_P$ with respect to all $\alpha_i$ vanish, all subject to $\alpha_i \geq 0$. We can equivalently solve the following dual problem: maximize $L_P$, subject to the constraints that the gradient of $L_P$ with respect to $\boldsymbol{w}$ and $b$ vanish and subject to $\alpha_i \geq 0$. Requiring that the gradient of $L_P$ with respect to $\boldsymbol{w}$ and $b$ vanish gives the following conditions:

$$\boldsymbol{w} = \sum_i \alpha_i y_i \boldsymbol{x}_i \tag{3}$$

$$\sum_i \alpha_i y_i = 0 \tag{4}$$

We can substitute these equality constraints into (2) to get:

$$L_D = \sum_i \alpha_i - \frac{1}{2} \sum_{i,j} \alpha_i \alpha_j y_i y_j \boldsymbol{x}_i \cdot \boldsymbol{x}_j \tag{5}$$

Support vector training is maximizing $L_D$ with respect to $\alpha_i$ subject to constraints (4) and positive $\alpha_i$. Solution is then given by (3). Points with $\alpha_i > 0$ lie on separating hyperplanes and are called support vectors. If the training was performed with all other training points removed, the same separating hyperplane would be found.

This algorithm will not find a feasible solution when used on non-separable data. The dual Lagrangian $L_D$ will grow arbitrarily large. We can overcome this problem by relaxing condition (1) when necessary. This is done by introducing positive slack variables $\xi, \quad i = 1, \ldots, l$ in constraints (1). New constraints are:

$$y_i(\boldsymbol{x}_i \cdot \boldsymbol{w} + b) - 1 + \xi_i \geq 0, \quad \xi_i \geq 0 \quad \forall i \tag{6}$$

For an error to occur, $\xi_i$ must exceed 1, so $\sum_i \xi_i$ is an upper bound on the number of training errors. The objective function is changed from $||\boldsymbol{w}||^2/2$ to $||\boldsymbol{w}||^2/2 + C(\sum_i \xi_i)^k$, where $C$ is a parameter set by user. This is a convex programming problem for any positive integer $k$, for $k = 2$ or $k = 1$ it is also quadratic programming problem and if we choose $k = 1$, then neither $\xi_i$ or their Lagrange multiplier appear in the dual problem. The dual problem is still (5) with the same constraints except that $\alpha_i \geq 0$ becomes $0 \leq \alpha_i \leq C$. The solution is again given by (3).

In this algorithm, the decision function is a linear function of data, but it can be extended for a nonlinear case. Training data in the training problem appear only in the form of dot products $\boldsymbol{x}_i \cdot \boldsymbol{x}_j$. They can be first mapped to some other Euclidean space $H$ using a function $\Phi$:

$$\Phi : \boldsymbol{R}^d \rightarrow H \tag{7}$$

Now the training algorithm depends only on the dot products of the training data in $H$, that is $\Phi(\boldsymbol{x}_i) \cdot \Phi(\boldsymbol{x}_j)$. Let us define a kernel function $K$ as:

$$K(\boldsymbol{x}_i, \boldsymbol{x}_j) = \Phi(\boldsymbol{x}_i) \cdot \Phi(\boldsymbol{x}_j) \tag{8}$$

In the training algorithm we can now use just the values of the kernel function $K$ without the need to explicitly know $\Phi$.

The most frequent kernel functions are the following:

- **linear** $\Phi(x_i, x_j) = x_i \cdot x_j$
- **polynomial** $\Phi(x_i, x_j) = (ax_i \cdot x_j + b)^d$
- **radial basis function (RBF)** $\Phi(x_i, x_j) = \exp(-\gamma||x_i - x_j||^2)$
- **sigmoid** $\Phi(x_i, x_j) = \tanh(ax_i \cdot x_j + r)$

An optimal solution of equation (5) fulfills the Karush-Kuhn-Tucker (KKT) conditions. The KKT conditions are:

$$\begin{aligned} \alpha_i = 0 &\Rightarrow y_i f(x_i) \geq 1, \\ 0 < \alpha_i < C &\Rightarrow y_i f(x_i) = 1, \\ \alpha_i = C &\Rightarrow y_i f(x_i) \leq 1. \end{aligned} \tag{9}$$

Most of the parallel implementations use Platt's SMO algorithm improved by Keerthi and Fan. A simple algorithm is as follows:

1) Initialize $\alpha_i = 0, f_i = -y_i, \forall i \in 1, \ldots, l$
2) Select working set $\rightarrow j$ and $k$
3) Optimization step $\rightarrow \Delta\alpha_j$ and $\Delta\alpha_k$
4) Update $f_i, \forall i \in 1, \ldots, l$
5) Go back to step 2 if KKT conditions not fulfilled

The working set selection may use the first or the second order heuristic. Usually, the first index $j$ is selected by the first order and, with known $j$, the index $k$ is selected by the second order heuristic. Alternatively, both the indexes may

be selected by the first order heuristic. Using the second order heuristic improves convergence and decreases the number of iterations [11]. On the other hand, the second order heuristic requires a $j$-th row of the Gram matrix $Q$. If the row is not in the cache, its computation affects the total performance. Catanzaro in [19] introduced a dynamic switching between the first and the second order heuristic. However, the update step requires $j$-th and $k$-th row of the matrix $Q$ anyway. All the steps are parallelizable, except the optimization step that is computationally undemanding.

Some implementations speed up convergence by using a technique called shrinking. Shrinking reduces the problem size by temporarily eliminating variables that are unlikely to be selected by SMO algorithm because they have reached their lower or upper bound. SMO iterations continue on this reduced working set. Shrinking reduces the number of kernel values needed to update the gradient vector.

## 3 Our Implementation

Most GPU implementations are based on SMO. They decompose the quadratic problem into a series of the smallest possible sub-problems, which are then solved analytically. For large datasets, the majority of elapsed time is spent by a large amount of matrix-vector multiplications that cannot be computed efficiently on current GPUs because of limited memory bandwidth. Also, a huge number of iterations that requires CPU-GPU communication decreases the total efficiency.

In contrast, our algorithm (OHD-SVM) is composed of a hierarchy of 2 levels, we call them global and local. Global level is described in Algorithm 1. It selects a working set of predefined size $N_{WS}$, usually equal to the number of threads a CUDA kernel can execute in one block. Kernel matrix for this reduced working set has size $N_{WS} \times N_{WS}$, small enough to be computed all at once. The local level of our algorithm is a solver which optimizes this reduced problem using working set selected by global level. We use SMO as a local solver.

This local solver is implemented as a one-block CUDA kernel that does many iterations without a need of costly global synchronizations or CPU-GPU communication. Each thread optimizes one point from the working set. The local kernel matrix values are already computed when the local solver is executed and the matrix is small enough to efficiently use the GPU global memory cache. Compared to the naive SMO, this approach needs only one CUDA kernel launch for each global iteration and uses high parallelism to compute all local kernel matrix rows at once. After the local solver optimizes the reduced problem, the global gradient vector is updated and a new working set is selected. The gradient vector update needs full rows of the kernel matrix belonging to vectors from the current working set. Computing $N_{WS}$ rows of the kernel matrix in each global iteration is too costly and the entire kernel matrix is too large to fit into the GPU memory. Therefore we had to implement our own cache mechanism. It is described in Section 3.1.

This algorithm needs a greater number of total iterations to converge than the SMO algorithm alone. However, most of these iterations are done in SMO in our local solver, which is a CUDA kernel launched only once per global iteration and does not compute any kernel values during its local iterations.

Alg. 1. Global-Level of our Algorithm

1: Precalculate $x^2$
2: Precalculate a diagonal of $K$
3: $iter \leftarrow 0$
4: **loop**
5: **if** $iter = 0$ **then**
6: Select random working set
7: **else**
8: Select new working set ▷ See algorithms 3 and 4
9: **end if**
10: Compute $KTile$
11: $\alpha' \leftarrow \alpha$
12: Execute local solver
13: **if** local solver iterations $= 0$ **then**
14: **break**
15: **end if**
16: $\Delta\alpha \leftarrow \alpha - \alpha'$
17: Compute all rows $i$ of $K$ and save them to cache $\forall\Delta\alpha_i \neq 0$
18: Update $g$
19: $iter \leftarrow iter + 1$
20: **end loop**

### 3.1 Kernel Matrix Cache Management

In each iteration, the SMO algorithm requires the kernel function values for a pair of training vectors. Such values can be used once or many times and take up a vast majority of total computation time of the whole SVM training algorithm. There is a total of $N^2$ kernel values for training set of size $N$. It is desirable to calculate kernel values only once and use their values during SVM training, but kernel matrix is larger than the available computer or GPU memory for larger data sets. Therefore it is very important to implement an efficient cache to remember often used kernel values.

In our implementation, the entire rows of kernel matrix are saved into the cache. We use GPU for all cache management while other available implementations use CPU. This approach avoids CPU–GPU synchronization and even the selection of cache rows to compute is faster than if it was done on CPU because we compute several rows of kernel cache at once and use several parallel algorithms when determining which rows to compute. This kernel is described in detail in Algorithm 2.

Several variables are needed for cache management:

- *K* – Pointer to buffer with cached kernel values. The buffer size is $N \times N_{Cache}$ where $N$ is the size of the training set and $N_{Cache}$ is the maximum number of rows in the cache.
- *KTile* – Submatrix of *K*. Contains kernel values for all vectors in the current working set.
- *KCacheRemapIdx* – Array of pointers to the matrix *K*. It contains offsets of rows from the buffer beginning for each training vector. A value of $-1$ means the corresponding row is not cached. Array size is $N$.
- *KCacheRowIdx* – Array containing an index of a training vector to which a particular row of the cache

belongs. A value of $-1$ means that the row does not belong to any training vector. Array size is $N_{Cache}$.
- *KCacheRowPriority* – Array of kernel cache row priorities. It is used to determine which rows to discard if the cache is full.
- *KCacheRowsToCompute* – Array of indices of cache rows which should be computed this iteration. *KCacheRowIdx* can be used to retrieve the index of the training vector to whom the particular cache row belongs.

The local solver uses only variable *KTile*. It contains all the kernel values for all the vectors from the current working set and it is all that the local solver needs to know. We compute *KTile* in one CUDA kernel.

The kernel function evaluation is the most expensive part of the SVM training algorithm, therefore it is important to focus on the optimization of the kernel matrix cache to achieve a high SVM training speed. Vector norm in the expression for the RBF kernel function can be modified to

$$||x_i - x_j||^2 = x_i^2 + x_j^2 - 2x_i \cdot x_j$$

All values of $x_i^2$ for all $x_i$ from the training set are calculated in the beginning of our training algorithm. $x_i \cdot x_j$ is a dot product of two points from the training set. If we want to compute the value of RBF kernel for many points, we can use matrix multiplication to compute all these dot products and this operation is very effective on graphic processors.

#### 3.1.1 Dense Data

All training vectors $x_i$ are saved in the matrix $X$

$$X = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}$$

After the selection of the new working set *WS*, we must compute all the rows $R$ of the kernel matrix where $R \subset WS$ and rows $R$ are not already cached. If we wanted to compute the entire kernel matrix, we would have to compute $x_i \cdot x_j$ for all $i$, $j$. This can be done on many different GPUs efficiently by evaluating $X \cdot X^T$ using CUBLAS. To compute the dot products only for rows $R$, we must evaluate $X' \cdot X^T$, where $X'$ is a matrix constructed from $X$ by using only rows $R$.

CUBLAS GEMM gives the best performance when one matrix is transposed, therefore we save the training data $X$ twice, once in normal order and once transposed.

We were not able to implement a faster function to calculate these dots products than already highly optimized CUBLAS GEMM function. Our algorithm uses these steps to compute rows $R$ of the kernel matrix cache:

1) Copy $R$ rows from $X$ to $X'$ in a CUDA kernel
2) Use CUBLAS GEMM to compute all the dot products $x_r \cdot x_j$, where $r \in R$.
3) Launch a CUDA kernel to compute the RBF values from the dot products using the expression

$$k_{r,j} = \exp -\gamma \cdot (x_r^2 + x_j^2 - 2x_r \cdot x_j)$$

and save them to appropriate rows in the kernel matrix cache. Column index $j$ is an index of the point in the training set. The kernel matrix row $r$ is saved in the kernel matrix cache row $KCacheRemapIdx[r]$, where *KCacheRemapIdx* is defined above.

The entire rows of kernel matrix are important for the gradient update, but we also have to compute $KTile$ for the local solver. $KTile$ is $N_{WS} \times N_{WS}$ matrix, where $N_{WS}$ is the working set size. Unlike the entire kernel matrix, $KTile$ cannot be arbitrarily large and we can optimize its computation for several specific values of $N_{WS}$.

Our kernel for $KTile$ computation is based on matrix multiplication kernel with using shared memory. The biggest difference is that we do not compute dot products for rows of the output matrix if the appropriate values are in kernel matrix cache. In this case, we just copy these values from the cache and save computation time.

#### 3.1.2 Sparse Data

To efficiently work with sparse data, we must choose the appropriate data representation. We use both Compressed Sparse Row (CSR) and Jagged Diagonal Storage (JDS) sparse matrix formats.

A dense matrix can be transformed to CSR representation by omitting zero values and saving rows of all non-zero values into an array. Such sparse matrix is then represented by 3 arrays:

- Array of non-zero values
- Array of column indices
- Array of row offsets, pointing into the both arrays above

The process of converting dense matrix to JDS is as follows. All zero values are removed and the remaining non-zero values are shifted to the left. Rows are sorted in a decreasing order by the number of non-zeros. After that, the columns are stored consecutively. This representation uses 5 arrays:

- Array of non-zero values
- Array of column indices
- Array of column offsets, pointing into the both arrays above
- Array of row lengths, contains the length of each row after sorting
- Row permutation array, contains indices of original rows

The training set is stored twice in the memory, once for each format. This is needed because different parts of the training algorithm work faster with different sparse matrix formats and short training time was our goal.

JDS is used when computing full rows of the kernel matrix. In each iteration, a new working set of size $N_{WS}$ is selected, of which the kernel function values for $N'_{WS}$ points are not in cache, where $0 \leq N'_{WS} \leq N_{WS}$. All training vectors belonging to $N'_{WS}$ non-cached rows are then expanded to the dense representation. This allows us to write a simpler and more efficient multiplication kernel, but more GPU memory is needed for this step of the algorithm. Each thread in a thread block computes one output value, a kernel function

value of training points $blockIdx.y$ and $perm[tidx]$, where *perm* is the row permutation array and *tidx* is the thread global index. Sorting the rows in JDS by length allows us to minimize divergent threads in a warp, there can be only one warp with less than *warpSize* threads running.

Only full rows of *KTile* are copied from the cache, therefore $N'_{WS} \times N_{WS}$ new values must be computed each iteration. If JDS was used, there could not be enough thread blocks executed to keep the GPU busy. Our CUDA kernel using JDS launches with the total number of threads in $x$ dimension equal to $N_{WS}$ and it is a low number. The maximum possible size of $N_{WS}$ is 2048 due to the CUDA shared memory size limit. The working set of size 2048 is not usually faster. It introduces overhead for each thread, which has to calculate values of 2 training points. Output values are also not written in coalesced manner, because of the JDS row sorting. Therefore we use CSR format to compute *KTile*. Each warp calculates one output value, a kernel function value of training points $blockIdx.y$ and $wi$, where $wi$ is the global warp index. When calculating a matrix multiplication, each thread loops over training data dimension, accumulating partial sums. CUDA shuffle instructions are then used to reduce these partial sums efficiently.

### 3.2 Working Set Selection

Working set selection is a very important part of the training algorithm. In SMO, a working pair is selected using either first order or second order heuristic. Our local solver uses first order heuristic to find out if the solution is optimal and if not, second order heuristic is used to select new working pair $i, j$.

These selection methods are defined only for 2 points, but we need to select $N_{WS}$ points for the new working set in each global iteration. The simplest approach would be to use a similar algorithm to the first order selection, but to take first $N_{WS}/2$ values instead of one point for each $i$ and $j$.

However, we use the working set selection method proposed by Serafini [31]. The method is as follows:

1) Define *NC*, usually equal to $N_{WS}/4$.
2) Use first order heuristic to select *NC* points for both $i$ and $j$ (Algorithm 3).
3) Put these points to the new working set.
4) Sort points in the old working set by a number of iterations they were in *WS*. Old ones go to the back.
5) Fill the new working set with free, lower bound and upper bound vectors from the old working set in order until it is full. The sorting from the previous step and the working set filling is in detail in Algorithm 4.

This method allows us to select up to $2NC$ new points each iteration and reuse at least half of the old working set. The assumption is that if the working set is optimized and then some points in it are changed, already optimized points are not optimized in the new working set but are close to their final values. Also, the longer the point is in the working set, the higher chance it has to be removed in the next iteration. This avoids zigzagging of points during iterations [31]. Our sorting algorithm is a bitonic sort. It is easily implemented in parallel and performs well if the number of sorted items is $2^n$. We use only powers of 2 as our working set size.

Alg. 2. Algorithm Determining which Cache Rows to Compute

$num \leftarrow 0$ ▷ total number of rows to compute
Declare $o_i$ for all $i$ in working set
Allocate $NIdx[N_{WS}]$
**for** $i = 0$ to $N_{WS} - 1$ **do** ▷ in parallel, each thread processes one $i$
  **if** $\Delta\alpha_i \neq 0$ **then** ▷ zero $\Delta\alpha$ is not used in gradient update, skip it
    $w \leftarrow ws[i]$ ▷ $w$ contains global index of training vector
    **if** $KCacheRemapIdx[w] < 0$ **then** ▷ if vector $w$ is not in cache
      $o_i \leftarrow num$ ▷ atomic operations needed due to parallelism
      $num \leftarrow num + 1$
    **else**
      $r \leftarrow KCacheRemapIdx[w]$ ▷ get cache row belonging to vector $w$
      $KCacheRowPriority[r] \leftarrow cacheUpdateCnt$
    **end if**
  **end if**
**end for**
**for** $n = 0$ to $num - 1$ **do**
  Find cache row $i$ with the lowest priority ▷ parallel reduction
  $NIdx[n] \leftarrow i$
  $KCacheRowPriority[i] = \infty$
**end for** ▷ *NIdx* now contains indices of cache rows which will be computed
**for** $i = 0$ to $num - 1$ **do** ▷ in parallel, each thread processes one $n$
  $r \leftarrow NIdx[o_i]$
  $KCacheRowsToCompute[o_i] \leftarrow cache_row$
  $v \leftarrow KCacheRowIdx[r]$ ▷ vector which is already cached in row $r$, if any
  **if** $v \geq 0$ **then** ▷ if cache row $r$ contains valid data
    $KCacheRemapIdx[v] \leftarrow -1$ ▷ remove vector $v$ from cache
  **end if**
  $KCacheRowIdx[r] \leftarrow ws[i]$
  $KCacheRemapIdx[ws[i]] \leftarrow r$
  $KCacheRowPriority[r] \leftarrow cacheUpdateCnt$
**end for**

### 3.3 Local Solver

The local solver is used to optimize the sub-problem consisting of points in the current working set. The rest of the training set is ignored in this step. We use the SMO algorithm as our local solver.

The whole local solver is implemented in one CUDA kernel. Launch configuration is one block of $N_{WS}$ threads. Since the number of threads is the same as the working set size, we can use $k^{\text{th}}$ thread to process $k^{\text{th}}$ element in the working set. All training point specific data ($\alpha$, class, ...) are saved in registers. Shared memory is used for parallel reductions when selecting the current working pair in each

Alg. 3. Working Set Selection Phase 1

**Input:**

$N_{WS}$, working set size
$y$, array of training vector classes
$\alpha$, array of Lagrange multipliers $\alpha$
$g$, gradient of the dual objective function

**Output:**

$NewWS$, indices of elements in the new working set, total output size is $2NC$ elements

**Input and Output:**

$WSPriority$, priority of elements used for the working set selection

$NC \leftarrow N_{WS}/4$
$k \leftarrow$ blockDim.x $*$ blockIdx.x
**while** $k < N_{\text{rows}}$ **do**
  **if** $(y_k = 1 \wedge \alpha_k < C) \vee (y_k = -1 \wedge \alpha_k > 0)$ **then**
    $v_k \leftarrow y_k \cdot g_k$
  **else**
    $v_k \leftarrow -\infty$
  **end if**
  Sort all $v_k$ in descending order and keep indices of first $NC$ values in the shared memory buffer *shIdxI*
  **if** $(y_k = 1 \wedge \alpha_k > 0) \vee (y_k = -1 \wedge \alpha_k < C)$ **then**
    $v_k \leftarrow y_k \cdot g_k$
  **else**
    $v_k \leftarrow -\infty$
  **end if**
  Sort all $v_k$ in descending order and keep indices of first $NC$ values in the shared memory buffer for *shIdxJ*
  $k \leftarrow k +$ gridDim.x $*$ blockDim.x
**end while**
Copy all values from shared memory buffers for I and J of all thread blocks to global memory
Terminate all blocks but one
Sort buffers for I and J in global memory and keep first $NC$ values from each buffer
**for** $n = 0$ to $NC - 1$ **do**
  $i \leftarrow shIdxI[n]$
  $j \leftarrow shIdxJ[n]$
  $NewWS[n] \leftarrow i$
  $NewWS[n + NC] \leftarrow j$
  $WSPriority[i] \leftarrow \infty$
  $WSPriority[j] \leftarrow \infty$
**end for**

Alg. 4. Working Set Selection Phase 2

**Input:**

$N_{WS}$, working set size
$\alpha$, array of Lagrange multipliers $\alpha$
$NewWS$, indices of elements in the new working set, contains $2NC$ elements

**Input and Output:**

*WS*, indices of elements in the current working set
$WSPriority$, priority of elements used for the working set selection

$NewWS' \leftarrow NewWS$
Sort *WS* by priority (*WSPriority*) in ascending order
**for all** free vectors $k$ in *WS* **do**
  **if** $WSPriority_k < \infty \wedge NewWS$ is not full **then**
    Add training vector $k$ to *NewWS*
    $WSPriority_k \leftarrow WSPriority_k + 1$
  **end if**
**end for**
**for all** lower bound vectors $k$ in *WS* **do**
  **if** $WSPriority_k < \infty \wedge NewWS$ is not full **then**
    Add training vector $k$ to *NewWS*
    $WSPriority_k \leftarrow WSPriority_k + 1$
  **end if**
**end for**
**for all** upper bound vectors $k$ in *WS* **do**
  **if** $WSPriority_k < \infty \wedge NewWS$ is not full **then**
    Add training vector $k$ to *NewWS*
    $WSPriority_k \leftarrow WSPriority_k + 1$
  **end if**
**end for**
$WS \leftarrow NewWS$
$WSPriority_k \leftarrow 0 \quad \forall k \in NewWS'$ ▷ Reset priority for newly selected elements

iteration of the local solver and to exchange data between threads when updating $\alpha$ coefficients.

Both first order and second order heuristics are used in the kernel. The first order heuristic is used to determine if the solution is optimal and to select the first point in the working pair. The second order heuristic is used to select the second point in the working pair. After the working pair $\{i, j\}$ is selected, both points are optimized against each other. The values $\alpha_i$ and $\alpha_j$ and the whole gradient vector for the working set are updated.

When the solution for the local sub-problem converges, new $\alpha$ values and $\Delta\alpha$ are stored to global memory. It is now necessary to update the gradient vector for all training points, so we can use it for the selection of the new working set. To update the gradient, we need to know the kernel matrix rows for the current working set. However, only gradient elements belonging to non-zero $\Delta\alpha$ must be updated. Therefore we can save time by computing only those kernel matrix rows, which belong to non-zero $\Delta\alpha$. Without this optimization, we would have to calculate the kernel matrix rows for all the points in the working set, so they can be used in the local solver, and then use them to update the gradient vector. Table 6 shows the training time difference between the algorithm with the separate kernel matrix tile calculation and without it.

## 4 Experiments

In this section we compared our OHD-SVM with other available GPU implementations. In addition, we evaluated various variants and setups of our approach.

### 4.1 Datasets Description

We tried to test all the implementations on the same datasets that were frequently used in the referenced publications. However, some regularization was needed. We omitted most of the small datasets that might be challenging for 6 or

7 years old GPUs, but not now. Because some implementations are specialized for dense and some for sparse data, we split datasets and result-tables into dense and sparse groups. Some medium-sparse datasets were also added in the dense form to the dense group for comparison. We performed the two-class SVM training with the RBF kernel function because it is supported by all the tested implementations.

A complete list of used datasets and the training setup is in Table 1. *Epsilon* and *Alpha* datasets come from the Pascal Large Scale Learning Challenge (`http://largescale.ml.tu-berlin.de`). They are very large dense datasets. The entire *Epsilon* set has 400k samples that most of the compared implementations cannot handle. Therefore, we used only 10% of the set. The entire *Alpha* set contains 500k samples. We used first 10k for training and last 50k for testing. We used scaling for *Epsilon* and *Alpha* datasets. The rest of the datasets come from the LibSVM data page (`https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/`). *TIMIT* is a preprocessed speech signal dataset where individual phonemes are classified. It is a multi-class task. We converted it to the binary classification task simply by even-vs-odd manner according to the phoneme index. The same even-vs-odd approach was used for the *MNIST* dataset where hand-written digits are classified. *Adult* and *Web* sets are the most popular SVM datasets and we used the biggest variants of the sets: *Adult a9a* and *Web w8a*. *COV1* is a set of cartographic variables for detection of the cover type. It is a multi-class task but we used it as a detection of cover type 1 that is forest. Therefore in some publications, the dataset is marked as Forest dataset. *News20* (*20 Newsgroups*), *RCV1*, and *Real-Sim* are large sparse sets for the text categorization and the training and testing setup was used according to [28].

### 4.2 Hardware Details

For all the tests we used a desktop PC with Intel Core i7-4790K, 4-core CPU clocked at 4.0GHz with 32 GB RAM at 1600 MHz. We have used Visual Studio 2013 and CUDA 8.0 to compile the project.

Tested GPUs were:

- Pascal-based NVIDIA GTX 1080 with 2560 cores clocked at 1607 MHz and 8 GB GDDR5X with bandwidth 320 GB/s
- Maxwell-based NVIDIA GTX 980 Ti with 2816 cores clocked at 1000 MHz and 6 GB GDDR5 with bandwidth 336.5 GB/s
- Kepler-based NVIDIA GTX TITAN Black with 2880 cores clocked at 890 MHz and 6 GB GDDR5 with bandwidth 336 GB/s

Unless otherwise specified, the tables show the results from GTX 1080, because it has the highest performance of all of our cards. Other GPUs were used only to compare the training speed of our implementation in Table 8.

### 4.3 Results

We have trained models for all datasets using our and several other SVM training implementations. Training times in seconds for dense and sparse datasets are shown in Tables 2 and 4. Trained model accuracy in percent is shown in Tables 3 and 5. gtSVM has two configuration options called ”small clusters” and ”large clusters”. We have trained the model using both options and denote them in the Tables using letters SC and LC.

The model accuracy of the most implementations is comparable. The most popular CPU SVM training tool, LibSVM, achieves the same accuracy. Some implementations trained wrong models, where its accuracy was considerable lower than that of other implementations (marked as *WM* in Tables 2, 4, 3, and 5). multiSVM and gtSVM with small and large clusters were unable to train good model for *COV1* dataset, which is the largest one. gtSVM failed for both dense and sparse representation of *COV1*, therefore our implementation was the only one able to train good model for sparse *COV1*.

Our algorithm was significantly faster than all other implementations for all datasets. The biggest difference was on the largest datasets. E.g. for *Epsilon* dataset, our implementation was 12 times faster than gpuSVM, which is the fastest already published dense implementation.

We have examined more variants of our algorithm. Our final version uses the working set selection described in Section 3.2 and calculates the kernel matrix tile for the local solver in the separate kernel described in Section 3.1. In Table 6, we compared the training times with the variant with the simple working set selection only. We also compared the variant that does not use the separate kernel for the kernel matrix tile calculation. Simple working set selection produced much worse training times for all datasets except *Web* where the training times were comparable. The algorithm without separate kernel matrix tile calculation was much slower for all dense datasets. For some sparse datasets, it was negligible faster. Note that the last column does not contain the number of iterations, because it was the same as the final version column.

We have compared the training speed for various working set sizes. Table 7 shows that larger working set sizes performed better, especially on the large datasets. Note that dataset *News20* could not be trained with working set of size 1024 because of the limited GPU memory and the need of temporary buffers for dense representation of training points in the working set.

The maximum possible working set size is 2048 due to the shared memory size limit. However, a CUDA kernel can execute only up to 1024 threads in one block on current GPUs. Therefore, the local solver kernel for working set size of 2048 has to process two training points per thread. This very complicates the local solver kernel code. The preliminary results were worse than kernel for working set sizes smaller or equal to 1024. Therefore, we did not add an additional column to Table 7.

We have also analyzed the time spent on all steps of our algorithm. Graphs, showing the relative time for each step for a dense and a sparse dataset, are in Figures 1 and 2. We selected *Epsilon* as a dataset for the dense graph and *News20* for the sparse graph. Graphs show that the majority of the time is spent on kernel matrix cache calculation. This shows why this step is the most important to optimize well. In the case of dense data, the kernel matrix cache calculation uses highly optimized CUBLAS GEMM, therefore there is not much space for improvement. In the case of large and

TABLE 1
List of Used Datasets with the Main Features and the Training Setup

| Dataset | # Training | # Test | # Dim. | Format | C | Gamma |
|---|---|---|---|---|---|---|
| Epsilon | 40,000 | 10,000 | 2,000 | Dense | 32 | 0.0001 |
| Alpha | 10,000 | 50,000 | 500 | Dense | 512 | 0.002 |
| TIMIT | 63,881 | 22,257 | 39 | Dense | 1 | 0.025 |
| Adult a9a | 32,561 | 16,281 | 123 | Sparse | 4 | 0.5 |
| Web w8a | 49,749 | 14,951 | 300 | Sparse | 4 | 0.5 |
| MNIST | 60,000 | 10,000 | 784 | Sparse | 1 | 0.02 |
| COV1 | 522,911 | 58,101 | 54 | Sparse | 3 | 1.0 |
| News20 | 19,996 | 19,996 | 1,335,191 | Sparse | 4 | 0.5 |
| RCV1 | 20,242 | 677,399 | 47,236 | Sparse | 4 | 0.5 |
| Real-Sim | 72,309 | 72,309 | 20,958 | Sparse | 4 | 0.5 |

TABLE 7
Training Time for Various Working Set Sizes [s]

| Dataset | 64 | 128 | 256 | 512 | 1024 |
|---|---|---|---|---|---|
| Epsilon dense | 4.03 | 3.01 | 2.55 | 2.39 | 2.31 |
| Alpha dense | 10.21 | 7.38 | 5.99 | 5.72 | 5.77 |
| Adult dense | 2.53 | 2.05 | 1.85 | 1.89 | 1.69 |
| Web dense | 3.50 | 2.77 | 2.42 | 2.50 | 2.43 |
| COV1 dense | 67.17 | 52.51 | 45.97 | 41.06 | 29.53 |
| MNIST dense | 1.98 | 1.61 | 1.45 | 1.46 | 1.35 |
| TIMIT dense | 2.48 | 2.15 | 1.99 | 2.16 | 1.80 |
| Adult sparse | 2.35 | 2.13 | 2.05 | 2.20 | 2.15 |
| Web sparse | 3.04 | 2.97 | 2.78 | 2.97 | 3.08 |
| COV1 sparse | 131.92 | 133.86 | 133.42 | 135.78 | 119.01 |
| MNIST sparse | 5.12 | 5.04 | 5.14 | 5.32 | 5.37 |
| News20 sparse | 38.67 | 38.19 | 38.41 | 38.57 | N/A |
| RCV1 sparse | 1.87 | 1.83 | 1.86 | 2.03 | 2.08 |
| Real-Sim sparse | 7.02 | 6.66 | 6.44 | 6.52 | 6.83 |

very sparse data, almost all the training time consists of the kernel matrix cache calculation. The rest of kernels, except the *KTile* calculation, are identical to the dense variant. Sparse data do not fit well to GPU architecture, and it is the main reason for the lower efficiency of the kernel matrix cache calculation.

Finally, we have taken an interest in the effectiveness of individual GPU architectures. We have trained the models on 3 high-end GPUs with all the three recent architectures: Kepler, Maxwell, and Pascal. Training times for those GPUs are in Table 8. The table shows expected results, the newer GPUs gave shorter training times. However, there was a surprise: The biggest drop in training times was between the Kepler-based Titan Black and the Maxwell-based GTX 980 Ti, where the theoretical calculation performance does not differ much. The GTX 1080 was not much faster than Maxwell-based GTX 980 Ti, however, theoretically, it should be. On the other hand, the GTX 1080 produced much less heat than its predecessors, due to a new 16nm FinFET fabrication process. An additional reason that GTX 1080 did not perform as fast as expected is the memory bandwidth that did not improve over all the generations. We expect a big step forward when a high bandwidth memory (HBM) comes into play.

## 5 Conclusion

The most GPU implementations are based on SMO. The majority of elapsed time is spent by a large amount of matrix-vector multiplications that cannot be computed efficiently on current GPUs because of limited memory bandwidth,

1. WM means wrong model

TABLE 8
Training Time for Various GPUs [s]

| Dataset | GTX Titan Black | GTX 980 Ti | GTX 1080 |
|---|---|---|---|
| Architecture | Kepler | Maxwell | Pascal |
| Epsilon dense | 3.51 | 3.03 | 2.31 |
| Alpha dense | 13.78 | 7.55 | 5.77 |
| Adult dense | 3.31 | 2.22 | 1.69 |
| Web dense | 4.64 | 3.55 | 2.43 |
| COV1 dense | 66.76 | 39.92 | 29.53 |
| MNIST dense | 2.01 | 1.74 | 1.35 |
| TIMIT dense | 3.22 | 2.20 | 1.80 |
| Adult sparse | 4.08 | 2.90 | 2.15 |
| Web sparse | 6.09 | 3.95 | 3.08 |
| COV1 sparse | 156.97 | 120.14 | 119.01 |
| MNIST sparse | 6.33 | 6.64 | 5.37 |
| News20 sparse | 53.43 | 39.48 | 38.57 |
| RCV1 sparse | 3.38 | 2.31 | 2.08 |
| Real-Sim sparse | 11.88 | 7.57 | 6.83 |

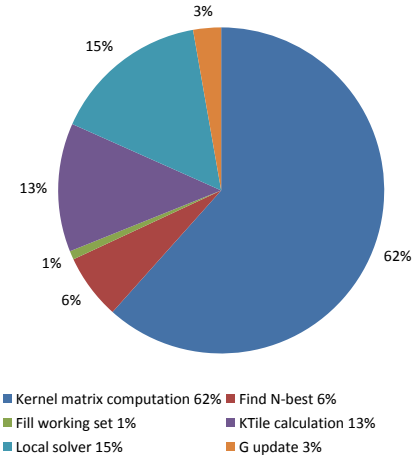


Fig. 1. Relative Time Needed for Each Step of Our Algorithm for Epsilon

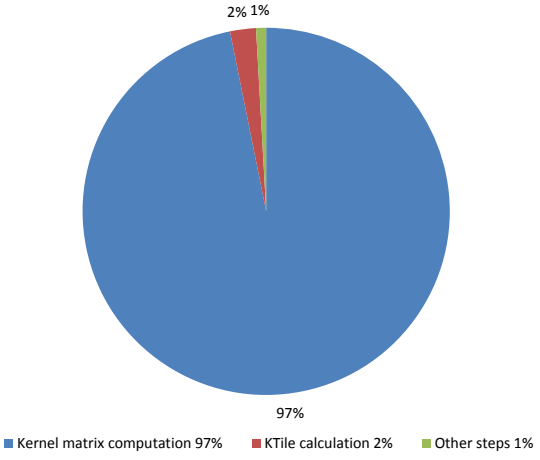


Fig. 2. Relative Time Needed for Each Step of Our Algorithm for News20

TABLE 2
Training Time for Dense Data [s]

| Implementation | Epsilon | Alpha | Adult | Web | COV1 | MNIST | TIMIT |
|---|---|---|---|---|---|---|---|
| cuSVM | 39.25 | 101.38 | 17.45 | 22.02 | 265.27 | 11.92 | 17.04 |
| gpuSVM | 28.04 | 26.26 | 3.36 | 7.58 | 301.04 | 5.89 | 2.86 |
| multiSVM | 51.34 | 90.04 | 18.69 | 22.52 | WM[1] (191.05) | 12.22 | 17.73 |
| wuSVM | 122.08 | 61.96 | 131.56 | 13.61 | 596.86 | 79.21 | 203.86 |
| gtSVM LC | 47.80 | 31.45 | 3.19 | 4.02 | WM[1] (61.25) | 3.72 | 2.43 |
| gtSVM SC | 77.22 | 36.38 | 3.37 | 4.43 | WM[1] (112.12) | 4.55 | 3.17 |
| OHD-SVM | 2.31 | 5.77 | 1.69 | 2.43 | 29.53 | 1.35 | 1.80 |

TABLE 3
Trained Model Accuracy for Dense Data [%]

| Implementation | Epsilon | Alpha | Adult | Web | COV1 | MNIST | TIMIT |
|---|---|---|---|---|---|---|---|
| cuSVM | 88.57 | 78.61 | 82.71 | 99.44 | 84.87 | 98.93 | 87.73 |
| gpuSVM | 88.57 | 78.59 | 82.71 | 99.44 | 84.85 | 98.91 | 87.70 |
| multiSVM | 88.59 | 78.62 | 82.71 | 99.44 | WM[1] (62.75) | 98.93 | 87.73 |
| wuSVM | 88.70 | 78.51 | 83.45 | 97.85 | 82.97 | 98.43 | 87.16 |
| gtSVM LC | 88.46 | 76.27 | 82.71 | 99.44 | WM[1] (50.80) | 98.93 | 87.72 |
| gtSVM SC | 88.53 | 77.14 | 82.71 | 99.44 | WM[1] (62.97) | 98.93 | 87.73 |
| OHD-SVM | 88.52 | 78.61 | 82.71 | 99.44 | 84.87 | 98.93 | 87.73 |

TABLE 4
Training Time for Sparse Data [s]

| Implementation. | Adult | Web | COV1 | MNIST | News20 | RCV1 | Real-Sim |
|---|---|---|---|---|---|---|---|
| gtSVM LC | 3.30 | 4.03 | WM[1] (61.17) | 3.69 | 605.53 | 15.83 | 53.66 |
| gtSVM SC | 3.44 | 4.27 | WM[1] (111.58) | 4.60 | 486.83 | 10.11 | 20.81 |
| OHD-SVM | 2.15 | 3.08 | 119.01 | 5.37 | 38.57 | 2.08 | 6.83 |

TABLE 5
Trained Model Accuracy for Sparse Data [%]

| Implemen. | a9a | w8a | cov1 | mnist | news20 | rcv1 | real-sim |
|---|---|---|---|---|---|---|---|
| gtSVM LC | 82.71 | 99.44 | WM[1] (50.80) | 98.93 | 99.88 | 96.54 | 99.73 |
| gtSVM SC | 82.71 | 99.44 | WM[1] (62.97) | 98.93 | 99.88 | 96.54 | 99.73 |
| ours OHD-SVM | 82.72 | 99.44 | 84.86 | 98.93 | 99.88 | 96.54 | 99.73 |

TABLE 6
Training Time and Number of Iterations of Other Variants of Our Algorithm

| | Final version | | Only first order WS selection | | Without separate *KTile* calculation |
|---|---|---|---|---|---|
| Dataset | Training time [s] | # iterations | Training time [s] | # iterations | Training time [s] |
| Epsilon dense | 2.31 | 43,661 | 82.17 | 116,795 | 4.93 |
| Alpha dense | 5.77 | 852,867 | 505.03 | 3,170,383 | 6.41 |
| Adult dense | 1.69 | 80,261 | 2.38 | 139,533 | 2.04 |
| Web dense | 2.43 | 95,795 | 2.27 | 90,534 | 4.31 |
| COV1 dense | 29.53 | 572,112 | 1,706.65 | 3,644,074 | 91.74 |
| MNIST dense | 1.35 | 22,726 | 4.18 | 48,981 | 3.31 |
| TIMIT dense | 1.80 | 65,321 | 6.33 | 136,624 | 2.36 |
| Adult sparse | 2.15 | 80,440 | 3.20 | 139,533 | 1.96 |
| Web sparse | 3.08 | 91,677 | 3.05 | 90,534 | 2.92 |
| COV1 sparse | 119.01 | 455,878 | 3,051.73 | 2,372,863 | 344.47 |
| MNIST sparse | 5.37 | 22,820 | 8.17 | 49,187 | 6.06 |
| News20 sparse | 38.57 | 38,097 | 50.94 | 56,869 | 37.00 |
| RCV1 sparse | 2.08 | 22,490 | 2.90 | 34,114 | 1.91 |
| Real-Sim sparse | 6.83 | 58,190 | 8.76 | 79,507 | 6.76 |

especially for large datasets. In this paper, we introduced a novel GPU approach of the support vector machine training: Optimized Hierarchical Decomposition SVM (OHD-SVM). It uses a hierarchical decomposition iterative algorithm that allows using matrix-matrix multiplication to calculate the kernel matrix values. This approach is much more effective and results in faster training. The local solver uses a single multiprocessor to efficiently solve a local sub-problem by SMO. The local iterations are very fast because it requires only an intra-block threads synchronization.

We tested our algorithm and other implementations on the most frequently used datasets. We used both dense and sparse datasets, but our implementation is the only one supporting both of them. Our algorithm is significantly faster than all other implementations for all datasets. The biggest difference was on the largest datasets where we achieved speed-up up to 12 times in comparison with the fastest already published GPU implementation. We have also examined our algorithm in more detail. We compared the final variant with the first order WS selection variant and the variant without the separate KTile calculation. We have evaluated the training times for various working set sizes. We have analyzed the time spent on all steps of our algorithm for the dense and sparse cases separately. Finally, we have compared 3 recent GPU architectures: Kepler, Maxwell, and Pascal. The newer GPUs are faster, however, the individual results do not match with our prior expectations.

Our implementation supports both Windows and Linux and is freely available at `https://github.com/OrcusCZ/OHD-SVM`.

## ACKNOWLEDGMENTS

This research was supported by the Grant Agency of the Czech Republic, project No. GAČR GBP103/12/G084.

## REFERENCES

[1] S. J. Wright, "Optimization Algorithms in Support Vector Machines," *Computational Learning Workshop and Summer School, University of Chicago*, 2009.

[2] M. Ferris and T. Munson, "Interior-Point Methods for Massive Support Vector Machines," *SIAM Journal on Optimization*, 2000.

[3] Shai Shalev-Shwartz, "Online Learning: Theory, Algorithms, and Applications," Ph.D. dissertation, Hebrew University, 2007.

[4] C. Hsieh, K. Chang, and C. Lin, "A Dual Coordinate Descent Method for Large-Scale Linear SVM," *Proceedings of the 25th International Conference on Machine Learning*, 2008.

[5] T. Joachims, "Training Linear SVMs in Linear Time," *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp. 217–226, 2006.

[6] J. Platt, "Fast Training of Support Vector Machines Using Sequential Minimal Optimization," *Advances in Kernel Methods – Support Vector Learning*, pp. 185–208, 1999.

[7] Y.-L. Lin, J.-G. Hsieh, H.-K. Wu, and J.-H. Jeng, "Three-Parameter Sequential Minimal Optimization for Support Vector Machines," *Neurocomputing*, vol. 74, no. 17, pp. 3467–3475, Oct. 2011.

[8] T. Joachims, "Making Large-Scale SVM Learning Practical," in *Advances in Kernel Methods – Support Vector Learning*. MIT-Press, 1999, pp. 41–56.

[9] L. Zanni, T. Serafini, and G. Zanghirati, "Parallel Software for Training Large Scale Support Vector Machines on Multiprocessor Systems," *The Journal of Machine Learning Research*, vol. 7, pp. 1467–1492, 2006.

[10] S. Keerthi and S. Shevade, "Improvements to Platt's SMO Algorithm for SVM Classifier Design," *Neural Computation*, vol. 13, no. 3, pp. 637–649, 2001.

[11] R. Fan, P. Chen, and C. Lin, "Working Set Selection Using Second Order Information for Training Support Vector Machines," *The Journal of Machine Learning Research*, vol. 6, pp. 1889–1918, 2005.

[12] E. Yom-Tov, "A Parallel Training Algorithm for Large Scale Support Vector Machines," *Neural Information Processing Systems Workshop on Large Scale Kernel Machines*, pp. 1–9, 2005.

[13] L. Cao, S. Keerthi, and C. Ong, "Parallel Sequential Minimal Optimization for the Training of Support Vector Machines," *IEEE Transactions on Neural Networks*, vol. 17, no. 4, pp. 1039–1049, 2006.

[14] J. Gonçalves, N. Lopes, and B. Ribeiro, "Multi-Threaded Support Vector Machines for Pattern Recognition," *Neural Information Processing*, pp. 616–623, 2012.

[15] Y. You, J. Demmel, K. Czechowski, L. Song, and R. Vuduc, "CA-SVM: Communication-Avoiding Parallel Support Vector Machines on Distributed Systems," *International Parallel and Ditributed Processing Symposium*, 2015.

[16] J. Dong, A. Krzyak, and C. Suen, "Fast SVM Training Algorithm with Decomposition on Very Large Data Sets," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 27, no. 4, pp. 603–618, 2005.

[17] H. Graf and E. Cosatto, "Parallel Support Vector Machines: The Cascade SVM," *Advances in Neural Information Processing Systems*, vol. 17, pp. 521–528, 2004.

[18] J. Vanek, J. Michalek, and J. Psutka, "A Review of GPU Open Source Support Vector Machines Training Implementations," *Submitted to Parallel Processing Journal*, vol. TBD, p. TBD, 2017.

[19] B. Catanzaro, N. Sundaram, and K. Keutzer, "Fast Support Vector Machine Training and Classification on Graphics Processors," *Proceedings of the 25th International Conference on Machine Learning*, pp. 104–111, 2008.

[20] A. Carpenter, "cuSVM: A CUDA Implementation of Support Vector Classification and Regression," *patternsonscreen.net/cuSVMDesc.pdf*, pp. 1–9, 2009.

[21] S. Herrero-Lopez, J. Williams, and A. Sanchez, "Parallel Multiclass Classification Using SVMs on GPUs," *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, pp. 2–11, 2010.

[22] Q. Li, R. Salman, and E. Test, "GPUSVM: A Comprehensive CUDA Based Support Vector Machine Package," *Open Computer Science*, pp. 1–22, 2011.

[23] Q. Li, R. Salman, E. Test, R. Strack, and V. Kecman, "Parallel Multitask Cross Validation for Support Vector Machine Using GPU," *Journal of Parallel and Distributed Computing*, vol. 73, no. 3, pp. 293–302, Mar. 2013.

[24] N. Lopes and B. Ribeiros, "GPUMLib: An Efficient Open-Source GPU Machine Learning Library," *International Journal of Computer Information Systems and Industrial Management Applications*, vol. 3, pp. 355–362, 2011.

[25] S. Tyree, J. R. Gardner, K. Q. Weinberger, K. Agrawal, and J. Tran, "Parallel Support Vector Machines in Practice," *arXiv preprint arXiv:1404.1066*, 2014.

[26] T.-K. Lin and S.-Y. Chien, "Support Vector Machines on GPU with Sparse Matrix Format," in *2010 Ninth International Conference on Machine Learning and Applications*. Ieee, Dec. 2010, pp. 313–318.

[27] A. Cotter, N. Srebro, and J. Keshet, "A GPU-Tailored Approach for Training Kernelized SVMs," *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining*, pp. 805–813, 2011.

[28] K. Sopyla, P. Drozda, and P. Górecki, "SVM with CUDA Accelerated Kernels for Big Sparse Problems," *Artificial Intelligence and Soft Computing*, pp. 439–447, 2012.

[29] S. Cadambi, I. Durdanovic, V. Jakkula, M. Sankaradass, E. Cosatto, S. Chakradhar, and H. P. Graf, "A Massively Parallel FPGA-Based Coprocessor for Support Vector Machines," *2009 17th IEEE Symposium on Field Programmable Custom Computing Machines*, pp. 115–122, 2009.

[30] Y. You, S. L. Song, H. Fu, A. Marquez, M. M. Dehnavi, K. Barker, K. W. Cameron, A. P. Randles, and G. Yang, "MIC-SVM: Designing a Highly Efficient Support Vector Machine for Advanced Modern Multi-core and Many-Core Architectures," *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, pp. 809–818, May 2014.

[31] T. Serafini and L. Zanni, "On the Working Set Selection in Gradient Projection-Based Decomposition Techniques for Support Vector Machines," *Optimization Methods and Software*, vol. 20, no. 4-5, pp. 583–596, 2005.

**Jan Vaněk** Jan Vaněk received the M.Sc. degree equivalent in cybernetics in 2003 and the Ph.D. degree in cybernetics in 2010, both from the University of West Bohemia, Plze, Czech Republic. He is currently a Research Assistant at the New Technologies for the Information Society, University of West Bohemia, since 2014. He was working also at the Institute of Physical Biology in Nov Hrady, since 2006 to 2011. His research interests include GPGPU programming and optimizations, machine learning, automatic speech recognition, acoustic modeling, signal and image processing.

**Josef Michálek** Josef Michálek received the M.Sc. degree equivalent in cybernetics in 2014 from the University of West Bohemia, Plze, Czech Republic. He is currently a Ph.D. candidate at the Department of Cybernetics, University of West Bohemia. His research interests include GPGPU programming and optimizations, machine learning, and support vector machines.

**Josef Psutka** Josef Psutka received the M.Sc. degree equivalent in electrical engineering and the Ph.D. degree in cybernetics from the Czech Technical University, Prague, Czech Republic, in 1974 and 1980, respectively. He worked as an Assistant Professor in the Technical Institute, Plze, Czech Republic, from 1978 to 1991. In 1991, he joined the Department of Cybernetics, University of West Bohemia, Plze, as an Associate Professor, and became a Full Professor in 1997. He is Head of the Department of Cybernetics since 2003. In 2014, he joined the New Technologies for the Information Society. His research interests include speech signal processing, acoustic modeling, large-vocabulary ASR, speech synthesis, and pattern recognition.