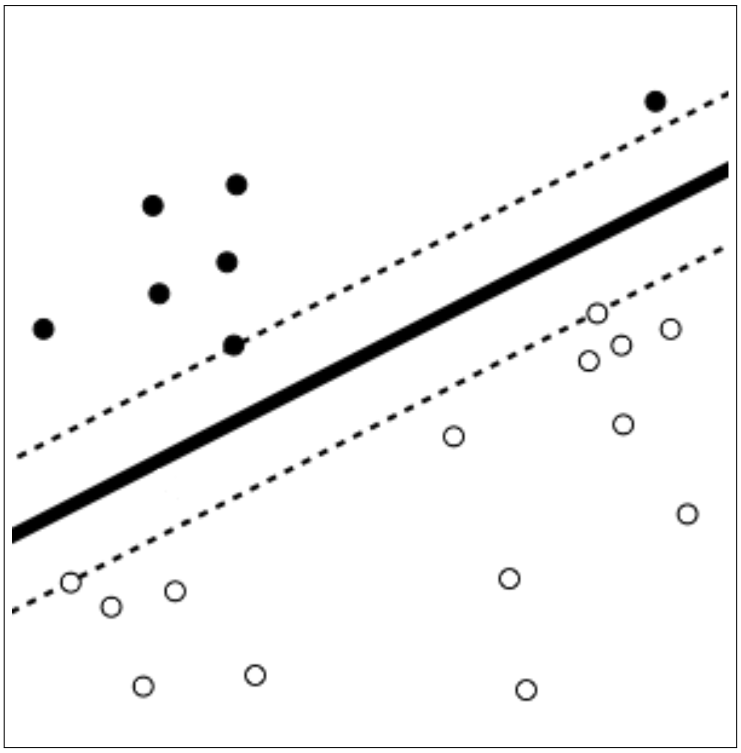
UNIVERSITY OF COPENHAGEN
FACULTY OF SCIENCE



Master's thesis

Emil U. Weihe — `emil@weihe.dk`

Support Vector Machines in Futhark

Supervisor: Troels Henriksen

Handed in: August 31, 2020

Abstract

The support vector machine is a supervised learning model. It is sometimes referred to as the best "off-the-shelf" learning model since it is relatively simple to use and can achieve very accurate results. The model is expensive to train, which motivates the use of graphics processing units (GPUs) to accelerate it. Futhark is a high-level programming language designed to be compiled into efficient GPU code. In this thesis, a new support vector machine library written in Futhark is presented. It introduces kernel function modules, which allow users of the library to define and use their own choice of kernel function. The library was benchmarked against the popular GPU-accelerated support vector machine library, ThunderSVM, which showed that their performance is relatively similar when training. On one dataset, the library was 2.2 times faster than ThunderSVM, but it performed a bit worse on others. For prediction, it was consistently between 2.4 to 9.6 times faster. Since the current trend shows that Futhark is becoming faster over time, these results are very promising.

Keywords: SVM, SVC, GPU, Futhark, pretty fast

Contents

Chapter 1

Introduction

In order to solve today's data science problems, a drastic increase in computational power is needed. Problems such as detecting cancer in X-ray images, automated face recognition, and even getting cars to drive autonomously, all require massive amounts of data to be processed. To keep up with the demand, it has been necessary to utilize the compute power of graphics processing units (GPUs) to accelerate the systems. The extensive amount of cores and high memory bandwidth of GPUs enables them to solve highly parallel problems very efficiently. The support vector machine (SVM) is an excellent supervised learning model with parallel characteristics. It is used to solve problems such as classification, regression, and outlier detection. It is very computationally expensive to apply SVMs to large and complex problems. This motivates the use of GPUs to accelerate them.

In this thesis, I present a novel support vector machine implementation in the data-parallel language, Futhark. Programs written in Futhark can compile into heavily optimized code that runs on GPUs.

1.1 Related works

LIBSVM is a very popular support vector machine toolkit for CPUs [CL11]. It was released in 2001 and has been maintained ever since. It is used in the popular Python library scikit-learn [Ped+11]. It implements classification (SVC), regression (SVR), and the one-class SVM for outlier detection. ThunderSVM[1] is a more modern implementation for multi-core CPUs and GPUs. It was released in 2018. On most datasets, it is more than 100 times faster than LIBSVM when using GPUs [Wen+18].

1.2 Thesis objective

The main goal of this thesis is to use Futhark to develop a GPU-accelerated support vector machine library. Futhark is a high-level language that hides many of the rather complicated low-level details of writing GPU code, such as thread blocks, synchronization, and memory management. As such, I want to investigate how the efficiency of a support vector machine library written in Futhark compares with other established solutions. Since many different types of SVMs exist, the objective is limited to the implementation of a C type multiclass SVM classifier (as introduced in Chapter 2).

1.3 Introduction to Futhark

Futhark[2] is a programming language designed to be compiled into efficient parallel code [Hen+17]. Its syntax is derived from ML. It is purely functional, has a static type system, and it comes with a set of constraints which allows the compiler to produce highly performant GPU code. Currently, it can produce GPU code via OpenCL and CUDA. It can also compile to C.

Futhark uses *array combinators* to perform operations on array elements in parallel. These are the building blocks of Futhark. It uses them to produce GPU code and as such, they are the key to achieving fast programs. Futhark defines second-order array combinators (SOACs) such as `map`, `reduce`, and `scan`, that takes a function as an argument, which indicates the operation to perform. As a small example, we can write the dot product $\sum_{i=1}^{n} u_i v_i$ of two vectors $\boldsymbol{u}$ and $\boldsymbol{v}$ using `map` and `reduce` by:

```
let dot [n] (u: [n]f32) (v: [n]f32): f32 =
  reduce (+) 0 (map2 (*) u v)
```

[1] `https://github.com/Xtra-Computing/thundersvm`
[2] `https://futhark-lang.org/`

For parametric polymorphism Futhark uses *type parameters*, which allows functions and types to be polymorphic. Type parameters are written as a name preceded by an apostrophe. Futhark also has *size parameters*, for imposing constraints on the sizes of arrays. A type for a pair of arrays of the same type `t` and size `n` can be expressed as:

```
type array_pair 't [n] = ([n]t, [n]t)
```

While Futhark is a purely functional language, it allows in-place array updates. The ith element of an array `A` can be set to 0 with `A with [i] = 0`. Mulitple elements can be updated in parallel with the `scatter` function. Futhark employs a special type system feature called *uniqueness types* to ensure the safety of in-place updates. Functions can take *unique* parameters, prefixed by `*`, which indicates that the argument will be consumed by the function, e.g., if the function updates the argument value in-place.

Futhark has an ML-style higher-order module system. It has module types that lets us specify the contents of a module. They contain specifications of what a module should implement. We can define a module type `mt` by:

```
module type mt = {
  type t
  val add1: t -> t
}
```

It specifies a type `t` and a function `add1` that takes a parameter of type `t` and returns `t`. Futhark also has parametric modules, which can take other modules as arguments. For example, this allows us to define a module `m` of type `mt` that takes a module of type `real` as a parameter:

```
module m (R: real): mt = {
  type t = R.t
  let add1 (n: t): t = R.(n + i32 1)
}
```

Here `R.(n + i32 1)` is the shorthand syntax for `(n R.+ R.i32 1)`. Dot-notation is used to access the functions and types of modules. `real` is the module for real numbers in Futhark. It implements the operator `+` and a function `i32` that casts to `R.t` from the primitive integer type `i32`.

Futhark imposes a set of rules that allows it to produce efficient code and ensure correctness. For example, irregular arrays are not allowed. Another important example is that the function argument given to `reduce` and `scan` must be *associative* and have a *neutral element* in order to work correctly on GPUs. While it can prevent irregular arrays at compile-time, it cannot detect if a function is associative, so that has to be ensured by the developer.

1.4 Source code repository

The finished implementation is a small SVM library written in Futhark. The source code can be found at the GitHub repository: github.com/fzzle/futhark-svm. The repository includes instructions on how to build and use it as a library in Python, or as a Futhark package. It also includes instructions in the `bench/` folder on how to run the benchmarks presented in this thesis.

Chapter 2

Support Vector Machines

The support vector machine (SVM) is a supervised learning model. In its purest form, an SVM takes data points (m-dimensional vectors) of two classes as input and attempts to find an $(m - 1)$-dimensional hyperplane, which separates them — a *decision surface*. The class of a new data point can be determined based on which side of the hyperplane it lies. This is a binary linear SVM classifier. The inherent problem is to find the best hyperplane.

In 1992, Boser et al. [BGV92] proposed the *hard margin* hyperplane. If the classes in a set of data are *completely separable*, it is the hyperplane with the largest margin between the classes. Two classes of data points are completely separable if you can put a hyperplane between them such that there is only data points of one class on each side. The limitation of the hard margin hyperplane is that it only works on completely separable data. Besides, it's been shown that it doesn't generalize well if there exist outliers. It was extended in 1995 by Cortes & Vapnik [CV95], who proposed a *soft margin* hyperplane, which also works on data that can't be separated without error. It introduces slack variables $\boldsymbol{\xi}$, which lets it tolerate training data that falls on the wrong side of the hyperplane. It also introduces a regularization parameter C, which controls how much the model should be penalized for misclassifying data when training. It is commonly referred to as a C-SVM and it is a type of SVM that most of the popular toolkits implement.

There exist simple modifications to the C-SVM, which makes it usable for other problems, namely regression, and outlier detection (one-class SVM). However, in this chapter, I will primarily focus on the principles and ideas that have been used to develop efficient C-SVM implementations, as they can be used as a foundation for the modified versions.

2.1 Training a C-SVM

Given a training set $\boldsymbol{X}$ of m-dimensional data points $\boldsymbol{x}_i$ and their corresponding labels $y_i \in \{+1, -1\}$ for all $i \in \{1, \ldots, n\}$, the goal of a C-SVM is to find the hyperplane which separates the positive and negative data points with a maximal margin $m = 1/\|\boldsymbol{w}\|$ and, at the same time, a minimal number of training errors. Training amounts to solving an optimization problem:

$$\begin{aligned}
&\underset{\boldsymbol{w},\boldsymbol{\xi},b}{\operatorname{argmin}} && \frac{1}{2}\|\boldsymbol{w}\|^2 + C\sum_{i=1}^{n}\xi_i && && (2.1\text{a})\\
&\text{subject to} && y_i(\boldsymbol{w}\cdot\boldsymbol{x}_i + b) \geq 1 - \xi_i, && \forall i \in \{1,\ldots,n\}, && (2.1\text{b})\\
& && \xi_i \geq 0, && \forall i \in \{1,\ldots,n\}. && (2.1\text{c})
\end{aligned}$$

Here $\boldsymbol{w}$ and b are the normal and intercept of the hyperplane, $\boldsymbol{\xi}$ are the slack variables, and C is a regularization parameter. The optimization problem can be rewritten into a dual form quadratic program [CV95; BB00]:

$$\begin{aligned}
&\underset{\boldsymbol{\alpha}}{\operatorname{argmin}} && W(\boldsymbol{\alpha}) = \frac{1}{2}\boldsymbol{\alpha}^{\mathrm{T}}\boldsymbol{Q}\boldsymbol{\alpha} - \sum_{i=1}^{n}\alpha_i && (2.2\text{a})\\
&\text{subject to} && 0 \leq \alpha_i \leq C, \quad \forall i \in \{1,\ldots,n\}, && (2.2\text{b})\\
& && \boldsymbol{y}\cdot\boldsymbol{\alpha} = 0. && (2.2\text{c})
\end{aligned}$$

Here $\boldsymbol{\alpha}$ is an n-dimensional set of Lagrange multipliers, and $\boldsymbol{Q}$ is an $n \times n$-dimensional symmetric matrix with $Q_{i,j} = y_i y_j K_{i,j}$ and $K_{i,j} = \boldsymbol{x}_i \cdot \boldsymbol{x}_j$. $W(\boldsymbol{\alpha})$ is the objective function and the goal is to find Lagrange multipliers $\boldsymbol{\alpha}$ that minimizes it. Once an optimal $\boldsymbol{\alpha}$ has been found, it is possible to derive the normal $\boldsymbol{w}$ and the intercept b of the hyperplane by:

$$\boldsymbol{w} = \sum_{i=1}^{n} y_i\alpha_i\boldsymbol{x}_i, \tag{2.3}$$

$$b = \boldsymbol{w}\cdot\boldsymbol{x}_k - y_k, \qquad \text{for any } k \text{ where } \alpha_k > 0. \tag{2.4}$$

Thereby, the label y^{test} of a test data point $\boldsymbol{x}^{\text{test}}$ can be predicted by:

$$\begin{aligned}
y^{\text{test}} &= \operatorname{sgn}\left(\boldsymbol{w}\cdot\boldsymbol{x}^{\text{test}} - b\right) && (2.5\text{a})\\
&= \operatorname{sgn}\left(\sum_{i=1}^{n} y_i\alpha_i\boldsymbol{x}_i\cdot\boldsymbol{x}^{\text{test}} - b\right) && (2.5\text{b})
\end{aligned}$$

The data points $\boldsymbol{x}_i$ that have an associated Lagrange multiplier α_i that is greater than 0 are referred to as *support vectors*. These are the data points that are used for the label prediction of unseen data.

For the set of $\boldsymbol{\alpha}$ to be an optimal solution of (2.2), it must satisfy the Karush-Kuhn-Tucker (KKT) conditions, which are as follows:

$$\alpha_i = 0 \quad \Rightarrow \quad y_i\left(\boldsymbol{w}\cdot\boldsymbol{x}_i + b\right) \geq 1 \tag{2.6}$$

$$\alpha_i = C \quad \Rightarrow \quad y_i\left(\boldsymbol{w}\cdot\boldsymbol{x}_i + b\right) \leq 1 \tag{2.7}$$

$$0 < \alpha_i < C \quad \Rightarrow \quad y_i\left(\boldsymbol{w}\cdot\boldsymbol{x}_i + b\right) = 1 \tag{2.8}$$

2.2 Sequential Minimal Optimization

In 1998, John C. Platt proposed Sequential Minimal Optimization (SMO) [Pla98]. It's an efficient algorithm used to solve the quadratic programming (QP) optimization problem that arises when training the C-SVM (2.2). SMO works by iteratively improving the α's until the KKT conditions are satisfied within a threshold ε. At each iteration, a pair of Lagrange multipliers $\{\alpha_u, \alpha_l\}$ are selected and the objective function $W(\boldsymbol{\alpha})$ is reoptimized with respect to this pair while the rest of the Lagrange multipliers are kept fixed.

SMO can be considered a decomposition method, by which the QP problem is broken into QP subproblems. In 1997, Osuna et al. proved a theorem which shows that solving QP subproblems will decrease the objective value of the full QP problem, as long as at least one Lagrange multiplier of the subproblem violates the KKT conditions [OFG97]. Decomposing the problem is a recurring theme in C-SVM training algorithms since the matrix $\boldsymbol{Q}$ with its n^2 elements is often too big to be kept in memory. However, SMO differs because it breaks the problem into the *smallest* possible subproblems that can be solved analytically and thus, very efficiently. Other algorithms mostly use numerical QP optimization which tends to be slow in practice.

2.2.1 Optimization of a pair

Let's consider the problem of minimizing the objective function $W(\boldsymbol{\alpha})$ with respect to a pair of Lagrange multipliers α_u and α_l while keeping the rest of the $n-2$ α's fixed. When updating α_u and α_l, they have to satisfy the constraints of the full problem (2.2b-c) so it's a requirement that:

$$0 \leq \alpha_k \leq C, \quad \forall k \in \{u, l\}, \tag{2.9a}$$

$$\boldsymbol{y}\cdot\boldsymbol{\alpha} = 0. \tag{2.9b}$$

The equality constraint (2.9b) can be rewritten as:

$$y_u\alpha_u + y_l\alpha_l = -\sum_{i\notin\{u,l\}} y_i\alpha_i \tag{2.10}$$

By multiplying with y_u on both sides it becomes:

$$\alpha_u + y_u y_l \alpha_l = -y_u \sum_{i \notin \{u,l\}} y_i \alpha_i \tag{2.11}$$

Since the α_i's for $i \notin \{u, l\}$ are fixed, the right-hand side will be a constant. To simplify, let's denote this constant by δ. The constraint becomes:

$$\alpha_u + y_u y_l \alpha_l = \delta \tag{2.12}$$

The constraints on α_u and α_l are visualized in Figure 2.1.

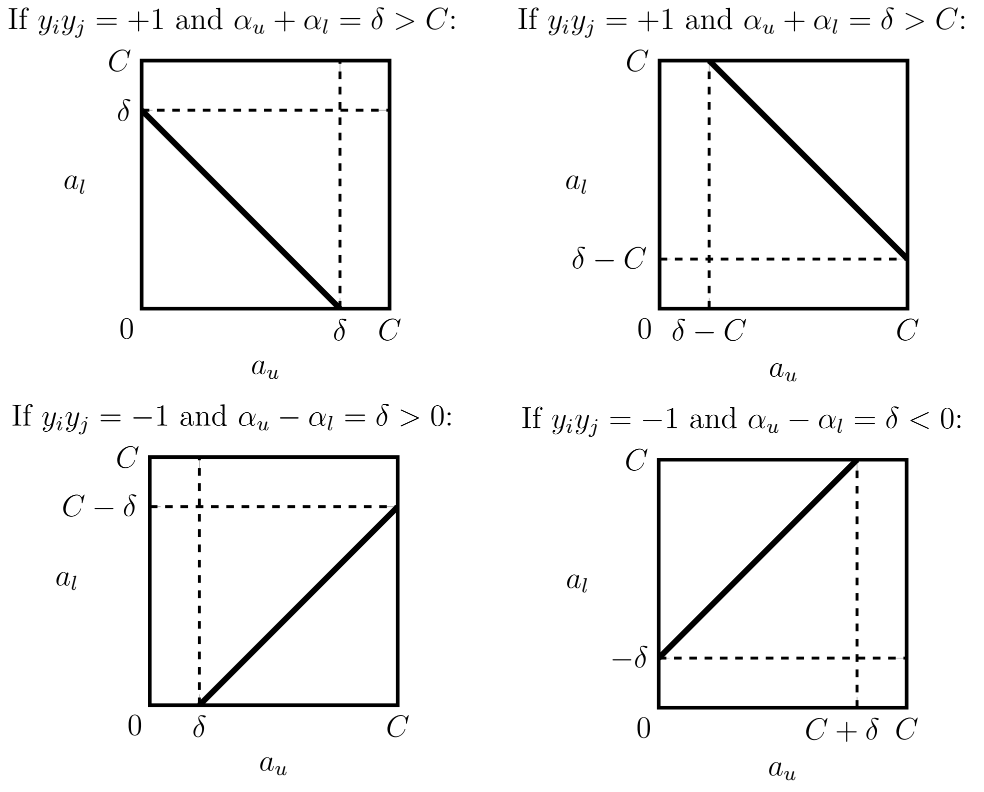

If $y_i y_j = +1$ and $\alpha_u + \alpha_l = \delta > C$:
C
δ
a_l
0
δ
C
a_u
If $y_i y_j = +1$ and $\alpha_u + \alpha_l = \delta > C$:
C
a_l
$\delta - C$
0
$\delta - C$
C
a_u
If $y_i y_j = -1$ and $\alpha_u - \alpha_l = \delta > 0$:
C
$C - \delta$
a_l
0
δ
C
a_u
If $y_i y_j = -1$ and $\alpha_u - \alpha_l = \delta < 0$:
C
a_l
$-\delta$
0
$C + \delta$
C
a_u

Figure 2.1: The inequality constraint (2.9a) causes α_u and α_l to lie within a box $[0, C] \times [0, C]$ and the equality constraint (2.12) causes it to lie on a diagonal line. The two topmost frames show the constraints if $y_u y_l = +1$ (which means $y_u = y_l$), and the two bottom ones show the constraints if $y_u y_l = -1$ (and $y_u \neq y_l$).

The equality constraint (2.12) can be rewritten to a function of α_l:

$$\alpha_u = \delta - y_u y_l \alpha_l \tag{2.13}$$

In the SMO paper by Platt, α_l is updated first, bound to the constraints, and then α_u is found by (2.13). As seen in Figure 2.1, a_l has to lie within the bounding box of the diagonal line. For example, in the top-left frame we have $L \leq \alpha_l \leq H$ for $L = 0$ and $H = \delta$. To summarize all the frames, if y_u equals y_l, then the lower bound L and upper bound H that apply to a_l are:

$$L = \max\{0, \delta - C\} = \max\{0, \alpha_u + \alpha_l - C\}, \tag{2.14}$$
$$H = \min\{C, \delta\} \qquad = \min\{C, \alpha_u + \alpha_l\}. \tag{2.15}$$

If y_u does not equal y_l, then:

$$L = \max\{0, -\delta\} \qquad = \max\{0, \alpha_l - \alpha_u\}, \tag{2.16}$$
$$H = \min\{C, C - \delta\} = \min\{C, C + \alpha_l + \alpha_u\}. \tag{2.17}$$

To update α_l, we have to find a value α_l' which minimizes the objective function $W(\boldsymbol{\alpha})$ of (2.2). If the α_i's for $i \notin \{u, l\}$ are kept fixed, and α_u is substituted with $\delta - y_u y_l \alpha_l$ (by (2.13)), then the objective function $W(\boldsymbol{\alpha})$ becomes a quadratic function of α_l. If the constraints in (2.9) are ignored, then it's easy to find the the value α_l' that minimizes $W(\boldsymbol{\alpha})$. It amounts to taking the derivative of $W(\boldsymbol{\alpha})$, setting it equal to 0, and solving it. The full proof is given in [Pla98]. It yields the following equation:

$$\alpha_l' = \alpha_l + y_l(f_u - f_l)/\eta_{u,l} \tag{2.18}$$

Here $\eta_{u,l} = K_{u,u} + K_{l,l} - 2K_{u,l}$, and f_k is the error on the kth training point. f_k is defined as the difference between the current output of the model and the desired output y_k. They are sometimes referred to as the *optimality indicators*. They can be obtained by the following:

$$f_k = \left(\sum_{i=1}^{n} \alpha_i y_i K_{i,k}\right) - y_k \tag{2.19}$$

The unconstrained α_l' has to be bound by L and H:

$$\alpha_l^{\text{new}} = \max\{L, \min\{\alpha_l', H\}\} \tag{2.20}$$

Finally, the new α_u value can be found by:

$$\alpha_u^{\text{new}} = \delta - y_u y_l \alpha_l^{\text{new}} \qquad \text{due to (2.13)} \tag{2.21}$$
$$= \alpha_u + y_u y_l \alpha_l - y_u y_l \alpha_l^{\text{new}} \qquad \text{due to (2.12)} \tag{2.22}$$
$$= \alpha_u + y_u y_l(\alpha_l - \alpha_l^{\text{new}}) \tag{2.23}$$

Rather than computing f_u and f_l at every update of α_u and α_l, it's possible to update all f_i for $i \in \{1, \ldots, n\}$ by the following formula:

$$f_i^{\text{new}} = f_i + (\alpha_u^{\text{new}} - \alpha_u)y_u K_{u,i} + (\alpha_l^{\text{new}} - \alpha_l)y_l K_{l,i} \tag{2.24}$$

This requires the f_i's to be initialized as $-y_i$.

2.2.2 Selection of a pair

SMO will work towards a minimum as long as at least one of the Lagrange multipliers selected violates the KKT conditions. However, since only a pair of multipliers are updated every step, it becomes an important task to choose and optimize those that will improve the overall problem the most. To speed up convergence, heuristics can be used to select a good pair of multipliers.

In 2001, Keerthi et al. [Kee+01] proposed a selection method which finds the *maximal violating pair*, the pair which violates the KKT conditions the most. It selects u and l by the following equations:

$$u = \text{argmin}_i\{f_i \mid i \in I^{\text{upper}}\}, \tag{2.25}$$
$$l = \text{argmax}_i\{f_i \mid i \in I^{\text{lower}}\}. \tag{2.26}$$

Where:

$$I^{\text{upper}} = I_1 \cup I_2 \cup I_3, \tag{2.27}$$
$$I^{\text{lower}} = I_1 \cup I_4 \cup I_5, \tag{2.28}$$

$$I_1 = \{i \mid 0 < \alpha_i < C\}, \tag{2.29}$$
$$I_2 = \{i \mid y_i = +1, \alpha_i = 0\}, \tag{2.30}$$
$$I_3 = \{i \mid y_i = -1, \alpha_i = C\}, \tag{2.31}$$
$$I_4 = \{i \mid y_i = +1, \alpha_i = C\}, \tag{2.32}$$
$$I_5 = \{i \mid y_i = -1, \alpha_i = 0\}. \tag{2.33}$$

A property of this pair is that if $f_u \geq f_l$ then the KKT conditions are satisfied. Since it's often difficult, if not impossible, to achieve a perfectly optimal solution, it can be rewritten to incorperate a small threshold ε:

$$f_l - f_u < \varepsilon \tag{2.34}$$

In 2005, Fan et al. [FCL05] proposed an improvement to this heuristic. It uses the same u, but it incorporates information from the derivative of the objective function when selecting l. It greatly decreases the amount of iterations needed to find an optimal solution. The pair is selected by:

$$l = \text{argmax}_i \left\{ \frac{(f_u - f_i)^2}{\eta_{u,i}} \middle| f_u < f_i, i \in I^{\text{lower}} \right\}. \tag{2.35}$$

2.2.3 Revised updates

Due to the assumption that $f_u < f_l$ of the selection heuristic (2.35), and that $\eta_{u,l}$ is positive under normal conditions as described in [Pla98], the update step described in 2.2.1 can be simplified slightly. Let's set:

$$q = (f_l - f_u)/\eta_{u,l} \tag{2.36}$$

Since $f_u < f_l$ and $\eta_{u,l} > 0$ we know that $q > 0$. This makes it possible to bound the change q with respect to the inequality constraints on α_l and α_u. Let's denote the bounded q by q'. Consider if we want to update α_l (2.20) and α_u (2.23) using q'. Since the change q' is bounded such that α_l and α_u doesn't exceed their inequality constraint (2.9a), we can write:

$$\alpha_u^{\text{new}} = \alpha_u + y_u q', \qquad \alpha_l^{\text{new}} = \alpha_l - y_l q'. \tag{2.37}$$

These updates also satisfy the equality constraint (2.12) since the same change is applied in each direction. Finding the constraints on q' requires solving $0 \leq \alpha_u + y_u q' \leq C$ and $0 \leq \alpha_l - y_l q' \leq C$. We have:

$$0 \leq \alpha_u + y_u q' \leq C \implies -\alpha_u \leq y_u q' \leq C - \alpha_u \tag{2.38}$$

Since $q' > 0$ and $\alpha_u > 0$, this can be simplified to:

$$\begin{aligned} q' &\leq C - \alpha_u, \quad && \text{if } y_u = +1, && (2.39)\\ q' &\leq \alpha_u, && \text{if } y_u = -1. && (2.40) \end{aligned}$$

Conversely, for α_l these bounds are:

$$\begin{aligned} q' &\leq \alpha_l, && \text{if } y_l = +1, && (2.41)\\ q' &\leq C - \alpha_l, \quad && \text{if } y_l = -1. && (2.42) \end{aligned}$$

Thereby, it's possible to find q' by binding q to these constraints.

Additionally, the update of f_i's can be simplified by:

$$\begin{aligned} f_i^{\text{new}} &= f_i + (\alpha_u^{\text{new}} - \alpha_u) y_u K_{u,i} + (\alpha_l^{\text{new}} - \alpha_l) y_l K_{l,i} && (2.43)\\ &= f_i + ((\alpha_u + y_u q') - \alpha_u) y_u K_{u,i} + ((\alpha_l - y_l q') - \alpha_l) y_l K_{l,i} && (2.44)\\ &= f_i + y_u y_u q' K_{u,i} - y_l y_l q' K_{l,i} && (2.45)\\ &= f_i + q'(K_{u,i} - K_{l,i}) && (2.46) \end{aligned}$$

2.2.4 The full algorithm

Algorithm 1 summarizes the entire training process as carried out by SMO using the components that have been defined. Note that $f^{\max}$ and f_u are the optimality indicators of the maximal violating pair, and thus, if $f^{\max} - f_u < \varepsilon$ the KKT conditions are satisfied and the training is done.

Algorithm 1: Sequential Minimal Optimization

```
Input: A set of training points $\boldsymbol{X}$ and their labels $\boldsymbol{y}$
Output: A set of Lagrange multipliers $\boldsymbol{\alpha}$
for $i \leftarrow 1$ to $n$ do // initialization
    $f_i \leftarrow -y_i$;
    $\alpha_i \leftarrow 0$;
end
loop
    find $u$ using (2.25);
    find $f^{\max}$ using (2.26);
    if $f^{max} - f_u < \varepsilon$ then stop;
    find kernel row $\boldsymbol{K}_u$;
    find $l$ using (2.35);
    find kernel row $\boldsymbol{K}_l$;
    update $\alpha_u$ and $\alpha_l$ using (2.37);
    update $\boldsymbol{f}$ using (2.46);
end
```

Actual implementations should use additional termination checks to ensure that the training is not stuck. Floating-point underflow can potentially cause q to be 0. If that happens, no change is applied to the α's. Most SMO solvers also set a limit on the number of iterations they allow.

2.3 Decomposition methods

A strategy that has greatly improved GPU implementations of SMO is adding another level of decomposition. Since SMO will work towards a global minimum as long as it can find a pair α_u and α_l that violate the KKT conditions, it is possible to select a subset of the problem that has violating pairs and solve it with SMO. Let's call this subset the *working set*. As mentioned before, the matrix $\boldsymbol{Q}$ is often too big to store in memory, yet this allows us to compute a part of it, keep it in memory, and solve the subproblems using it. While it won't be able to select the best pair u and l according to the SMO selection heuristics, it won't have to search the entire problem for u

and l either, a process which is performance-critical since it is done every iteration. Additionally, all the α's that already satisfy the KKT conditions can be avoided, such that they do not have to be considered by the selection heuristics. Thereby, it achieves the same effect as *shrinking*, a heuristic employed by LIBSVM, that filters out α's that are not believed to be updated any further [CL11]. If we want to perform SMO on a subset of n^{ws} data points, a natural strategy would be to select the $\frac{n^{\text{ws}}}{2}$ most violating pairs. This amounts to selecting the $\frac{n^{\text{ws}}}{2}$ data points with the smallest optimality indicators from I^{upper}, and conversely, the $\frac{n^{\text{ws}}}{2}$ largest indicators from I^{lower}.

2.3.1 First-in first-out

To decrease the amount of data points that oscillate in and out of the working set, ThunderSVM reuses a part of the previous working set. It uses a first-in first-out (FIFO) retention policy to choose which data points to keep. It can thereby also reuse a part of the previous cached kernel matrix, instead of having to compute a full new one. It keeps half of the working set by default.

2.4 Nonlinear classification

Not all data is linearly separable. For such data, the linear classifier won't be able to find a good decision surface. To be able to handle such data, SVMs can use *kernel functions* to map the data into a very high-dimension feature space where it may be linearly separable. E.g., we could have a kernel function $k(\boldsymbol{x}_i, \boldsymbol{x}_j) = \varphi(\boldsymbol{x}_i) \cdot \varphi(\boldsymbol{x}_j)$ where $\varphi : \mathbb{R}^m \to \mathbb{R}^t$ is a mapping from m to t dimensions. It is difficult and expensive to map into high-dimensional space with φ. As such, a "kernel trick" is employed, which allows us to calculate the inner product implicitly without using a mapping φ. In the definition of the training problem (2.2), we replace $K_{i,j} = \boldsymbol{x}_i \cdot \boldsymbol{x}_j$ by $K_{i,j} = k(\boldsymbol{x}_i, \boldsymbol{x}_j)$. Additionally, the output is modified to incorporate the kernel function:

$$y^{\text{test}} = \operatorname{sgn}\left(\sum_{i=1}^{n} y_i \alpha_i k(\boldsymbol{x}_i, \boldsymbol{x}^{\text{test}}) - b\right) \tag{2.47}$$

In SMO, it is assumed that $\boldsymbol{K}$ is a positive definite matrix. Thus, the kernel function $k(\cdot, \cdot)$ has to be positive definite to be valid. If not, there are cases where $\eta_{i,j}$ is not positive as otherwise assumed. However, bounding $\eta_{i,j}$ to a small positive constant (e.g. $\tau = 10^{-12}$ as used by LIBSVM) has enabled it to train with kernel functions that aren't fully positive definite [CL11]. Even if the kernel function is positive definite, there are rare cases where $\eta_{i,j} = 0$, and thus, it is a good idea to bound it [Pla98].

Table 2.1 shows a list of popular kernel functions.

Classifier type	**Kernel function**
Linear	$k(\boldsymbol{x}_i, \boldsymbol{x}_j) = \boldsymbol{x}_i \cdot \boldsymbol{x}_j$
Polynomial	$k(\boldsymbol{x}_i, \boldsymbol{x}_j) = (\gamma \boldsymbol{x}_i \cdot \boldsymbol{x}_j + c)^d$
Gaussian (RBF)	$k(\boldsymbol{x}_i, \boldsymbol{x}_j) = \exp(-\gamma \lVert \boldsymbol{x}_i - \boldsymbol{x}_j \rVert^2)$
Sigmoid	$k(\boldsymbol{x}_i, \boldsymbol{x}_j) = \tanh(\gamma \boldsymbol{x}_i \cdot \boldsymbol{x}_j + c)$

Table 2.1: These are the kernels implemented by LIBSVM and ThunderSVM. The kernels use various parameters (γ, c, and d) that can be tuned to achieve better accuracy.

2.5 Multiclass classification

So far, binary classification has been covered. In order to solve a *multiclass* classification problem with three or more classes, it is common to decompose the problem into multiple binary problems and solve them. ThunderSVM uses *one-versus-one*, a technique where a binary classifier is trained for every pair of classes. For k classes there will be a total of $\frac{k(k-1)}{2}$ binary classifiers. After training each classifier, predictions are made by voting; the class that is predicted by most of the binary classifiers is returned as the output of the model. Inevitably, ties can happen when two or more classes receive the same amount of votes. In these cases, it is important to resolve the ties in a fair manner. If a tie involving the same classes happens multiple times, ThunderSVM simply picks the same class every time, and as such, it can be slightly biased towards specific classes. A better solution would be to resolve the ties randomly or use additional information from the binary classifiers.

Chapter 3

Design and Implementation

In this chapter, I will describe the implementation of FutharkSVM (FSVM), a small SVM library in Futhark. In order to exploit the power of the GPU, it is necessary to use the array combinators provided in Futhark to perform operations in parallel. As such, the transformation from the ideas outlined in Chapter 2 to code using Futhark's SOACs will be discussed.

3.1 Library structure

The implementation uses Futhark's higher-order module system. `svm` is a parametric module, and it is the container of the library. It packs all the useful modules into one. It takes a float type module as an argument, making it possible to choose the desired floating-point precision for the entire library. It contains modules `svc` and `kernels`. `svc` is a parametric C-SVM module. In order to enable the user to choose the kernel they want it to use, `svc` takes a kernel module as an argument. `kernels` packs all the predefined kernel modules that can be passed to `svc`: `linear`, `polynomial`, `rbf`, and `sigmoid`. They all implement a module of type `kernel` and can be used to compute their specific kernel matrix. Additionally, `svc` uses the `solver` module which implements an SMO solver. Having the solver in a separate module from `svc` allows other types of SVMs (e.g. `svr` for regression) to be implemented using it in the future. A simple example of how the library is used is shown in Listing 1.

```
import "futhark-svm/svm"

module fsvm = svm f32
module lin_kernel = fsvm.kernels.linear
module lin_svc = fsvm.svc lin_kernel

entry fit_and_predict X y X_test =
  let model = lin_svc.fit X y 10 fsvm.default_fit {}
  in lin_svc.predict X_test model.weights fsvm.default_predict {}
```

Listing 1: Example usage of the library in Futhark. A function `fit_and_predict` is defined, which first fits on `X` and `y`, and then predicts the labels of the data points in `X_test`.

3.2 Full-kernel solver

The `svc` module uses an SMO solver to train. The initial solver that was implemented computes the full kernel matrix and keeps it in memory. Thus, it can implement SMO as summarized in Algorithm 1. It is quite fast but as discussed before; it is rarely possible to store a full kernel matrix in GPU memory due to its n^2 elements.

Listing 2 shows how a single step of SMO is performed in Futhark. It has comments that describe exactly what each part corresponds to. At every iteration of SMO we have to search for u, l, and $f^{\max}$. To implement argmax and argmin as used by the selection heuristics we can use the `reduce` array combinator. $\eta_{i,j} = K_{i,i} + K_{j,j} + 2K_{i,j}$ is used to find l and update the α's. It includes two diagonal elements of the kernel matrix and as such, a separate array `D` is used to cache these for fast accesses. After finding u and l the α's can be updated. Finally, the optimality indicators $\boldsymbol{f}$ can be updated using a simple `map` operation. This step function is the core of the implementation.

3.3 Two-level solver

Since the full-kernel solver uses too much memory it is necessary to use two-level decomposition. We can decompose the problem such that only a few rows of the kernel matrix have to be stored in memory at a time. This amounts to selecting a working set of n^{ws} data points to perform SMO on. As such, there will be an outer loop where a working set is selected and solved with SMO (in an inner loop) every iteration. Note that choosing n^{ws}

```
local let solve_step [n] (K: [n][n]t) (D: [n]t)
    (Y: [n]t) (F: [n]t) (A: [n]t) ((Cp, Cn): C_t)
    (m_p: m_t): (bool, (t, t), [n]t, [n]t) =
  -- Find the extreme sample x_u in I_upper.
  let B_u = map2 (is_upper Cp) Y A
  let F_u_I = map3 (\b f i ->
    if b then (f, i) else (R.inf, -1)) B_u F (iota n)
  let min_by_fst a b = if R.(a.0 < b.0) then a else b
  let (f_u, u) = reduce min_by_fst (R.inf, -1) F_u_I
  -- Find f_max so we can check if we're done.
  let B_l = map2 (is_lower Cn) Y A
  let F_l = map2 (\b f -> if b then f else R.(negate inf)) B_l F
  let f_max = R.maximum F_l
  -- Check if done.
  in if R.(f_max-f_u<m_p.eps) then (false,(f_u,f_max),F,A) else
  -- Find the extreme sample x_l in I_lower.
  let K_u = K[u]
  let V_l_I = map4 (\f d k_u i ->
    let b = R.(f_u - f)
    in if R.(b < i32 0)
      then (R.(b * b / (max tau (D[u] + d - i32 2 * k_u))), i)
      else (R.(negate inf), -1)) F_l D K_u (iota n)
  let max_by_fst a b = if R.(a.0 > b.0) then a else b
  let (_, l) = reduce max_by_fst (R.(negate inf), -1) V_l_I
  -- Find bounds for q.
  let c_u = R.(if Y[u] > i32 0 then Cp - A[u] else A[u])
  let c_l = R.(if Y[l] < i32 0 then Cn - A[l] else A[l])
  let eta = R.(max tau (D[u] + D[l] - i32 2 * K_u[l]))
  let b = R.(F[l] - f_u)
  -- Find new a_u and a_l.
  let q = R.(min (min c_u c_l) (b / eta))
  let a_u = R.(A[u] + q * Y[u])
  let a_l = R.(A[l] - q * Y[l])
  -- Update optimality indicators.
  let F' = map3 (\f k_u k_l -> R.(f + q * (k_u - k_l))) F K_u K[l]
  -- Write back updated alphas.
  let A' = map2 (\a i ->
    if i == u then a_u else if i == l then a_l else a) A (iota n)
  in (R.(q > i32 0), (f_u, f_max), F', A')
```

Listing 2: `solve_step` performs a single iteration of SMO.

to be the number of GPU threads will enable the array combinators used in SMO to utilize the full power of the GPU. FSVM selects the working set by both of the strategies described in 2.3. Note: If the problem has fewer data points than n^{ws}, FSVM simply uses the full-kernel solver.

3.3.1 Caching

While the FIFO strategy used by ThunderSVM keeps half of the previous working set, and thus, allows us to reuse a part of the previous kernel matrix rows, it does not act as a cache for kernel matrix rows. It simply copies a fixed part of the previous rows and reuses them. In FSVM, a simple least-recently used (LRU) cache is used to store kernel rows for future requests. It stores all kernel rows for the t previous iterations. When new kernel rows have to be computed for a working set, the kernel rows from $t-1$ previous iterations are searched through first. Note that it is $t-1$ because the kernel rows from the most recently cached iteration are not considered; with the selection heuristics besides FIFO, it is very uncommon that a data point is selected for the working set twice in a row, and thus, the hit rate would be close to zero. It is possible to search the cache simply by comparing all indices of cached rows with all indices of the new rows to compute, which is done quite efficiently with `map` and `reduce`. The rows with indices that were found can then fetched from the cache, and the rest are computed with matrix multiplication. I found that using $t = 2$ (only searching the "previous previous" kernel rows) is quite efficient, and that larger t might not be necessary to achieve high hit rates. For full trainings on the MNIST dataset [LC10] I observed cache hit rates between 75% and 88% with various kernel parameters. In addition, the training became about 3x faster. For the Adult dataset [DG17] the hit rate was 4-71%.

3.4 The `svc` module

`svc` is the module of the C-SVM classifier. It has two functions: `fit` and `predict`. The `fit` function, which trains a model, takes a dataset, some settings, and some kernel parameters. It returns a value of type `output` which has two fields; `weights` and `details` (as shown in Listing 3). The `weights` field is a record that contains all the values needed to predict the labels of unseen data. If the input dataset has k classes, the output contains the weights for $q = k(k-1)/2$ binary models. Each binary model has a different amount of weights and support vectors. As Futhark does not permit irregular arrays, all the weights and support vector indices are stored in flat

1-dimensional arrays. E.g., the field `A` of `weights` is a flat array that contains all $y_i\alpha_i$'s for all the q models. The field `Z` contains the number of weights for each binary model and it indicates where the weights of each model start and end in the flat array. The other field of the output, `details`, contains verbose information about the training of each binary model.

```
type weights 't [m][o][p][q] = {
  -- Flat alphas.
  A: [o]t,
  -- Flat support vector indices.
  I: [o]i32,
  -- Support vectors.
  S: [p][m]t,
  -- Segment sizes of flat alphas/indices.
  Z: [q]i32,
  -- Rhos (-bias/intercept).
  R: [q]t,
  -- Number of classes.
  n_c: i32
}

type details 't [q] = {
  -- Objective values.
  O: [q]t,
  -- Total iterations (inner).
  T: [q]i32,
  -- Outer iterations.
  T_out: [q]i32
}

-- | Trained model type.
type output 't [m][o][p][q] = {
  weights: weights t [m][o][p][q],
  details: details t [q]
}
```

Listing 3: `output` is the return type of the `fit` function and `weights` is used as input to the `predict` function. There are comments in the code to describe each field.

3.4.1 Prediction

Since the training weights come in a flattened format, the prediction function of `svc` is implemented using irregular *segmented* operations. It thereby finds the output of all q binary models in parallel. Initially, it computes the kernel matrix between the test data points and the support vectors. It can then compute $y_i \alpha_i k(\boldsymbol{x}_i, \boldsymbol{x}^{\text{test}})$ using the flattened array `A` of $y_i a_i$. All these values can be summed up by segmented reduction, and we can obtain the value of $y^{\text{test}} = \text{sgn}\left(\sum_{i=1}^{n} y_i \alpha_i k(\boldsymbol{x}_i, \boldsymbol{x}^{\text{test}}) - b\right)$ for each of the binary models. A binary model represents two classes, and $y^{\text{test}} \in \{+1, -1\}$ will be a vote for one of them. The votes can be counted using `reduce_by_index`, a SOAC which can be used to count into bins. Finally, we can do a reduction on the bins to find the index of the bin with the most votes, which also happens to be the predicted class of the full model.

3.5 Modular kernels

A limitation of most SVM implementations is that only a select few popular kernel functions are made available. Studies have shown that other, less frequently used kernels in some cases yield better results than the popular ones. For example, in the article [BTB05], the Laplacian RBF kernel performs better than the Gaussian RBF kernel. In this implementation, kernels are defined with modules. This allows kernels to be defined by users of the library. The definition of the kernel module type can be found in Listing 5. As an example, let the Laplacian RBF kernel be defined by:

$$k(\boldsymbol{x}_i, \boldsymbol{x}_j) = \exp(-\gamma \|a - b\|) \tag{3.1}$$

It can be defined in Futhark as a kernel module as shown in Listing 4:

```
module laplacian_rbf (R: real): kernel
    with t = R.t
    with s = {gamma: R.t} = default_kernel R {
  module util = kernel_util R
  type t = R.t
  type s = {gamma: t}

  let value [n] (k_p: s) (u: [n]t) (v: [n]t): t =
    R.(exp (negate k_p.gamma * sqrt (util.sqdist u v)))
}
```

Listing 4: A simple Laplacian RBF kernel module.

```
module type kernel_function = {
  -- | Kernel parameters.
  type s
  -- | Float type.
  type t
  -- | Compute a single kernel value.
  val value [n]: s -> [n]t -> [n]t -> t
}

-- | Kernel computation module type.
module type kernel = {
  include kernel_function
  -- | Compute the kernel diagonal.
  val diag [n][m]: s -> [n][m]t -> *[n]t
  -- | Extract diagonal from a full kernel matrix.
  val extdiag [n]: s -> [n][n]t -> *[n]t
  -- | Compute a row of the kernel matrix.
  val row [n][m]: s -> [n][m]t -> [m]t -> [n]t -> t -> *[n]t
  -- | Compute the kernel matrix.
  val matrix [n][m][o]:s->[n][m]t->[o][m]t->[n]t->[o]t-> *[n][o]t
}
```

Listing 5: `kernel` is the module type used to define the kernels. It specifies functions that are used to compute kernel values. `s` is the type of the kernel parameters. E.g., the parameter for the Gaussian RBF kernel is γ. `kernel_function` is a shorthand module type for `kernel`, which only specifies the `value` function. Modules with this type can be given as an argument to a parametric module `default_kernel`, which implements all the functions of `kernel` in a standard way using the `value` function.

A new C-SVM module can then defined by:

```
module laplacian_rbf_svc = svc f32 (laplacian_rbf f32)
```

For some `X` and `y`, it can be trained by:

```
laplacian_rbf_svc.fit X y svm.default_training {gamma=0.1}
```

Some kernel functions have special properties that make it possible to compute the kernel matrix, or parts of it, more efficiently. This is another reason why modules have been chosen to implement them. For example, the diagonal of the Laplacian RBF kernel matrix is always all 1's. As such, it is unnecessary to compute or even cache the diagonal, because all accesses to it can be replaced with a constant instead. It would be tedious to replace all accesses to the diagonal with a 1 in the code of the solvers by hand. Futhark's module system should be able to do this for us. While we can define a new kernel using only the `value` function and the `default_kernel` parametric module as done in Listing 4, it is also possible to define each of the functions specified by the `kernel` module type from scratch. For the Laplacian RBF kernel module, we could simply define the `diag` function, which computes the diagonal of its kernel matrix, as:

```
let diag [n][m] _ (_: [n][m]t): *[n]t = replicate n (R.i32 1)
```

3.6 Testing

It is not easy to validate the correctness of the output weights of a support vector machine. The support vectors chosen by this implementation might not be the same as those chosen by other implementations, and thus, the Lagrange multipliers $\boldsymbol{\alpha}$ might not be the same. Also, it will not be possible to use $\|\boldsymbol{w}\|$ to compare the margin achieved since $\boldsymbol{w}$ will be in some obscure feature space induced by the kernel function. The output that can be compared with other implementations to verify the results is, for the most part, the objective value, bias, and accuracy.

Since only a few indicators can be compared against those of existing implementations, it becomes an important task to test that the SVM's underlying functionality operates correctly. Futhark's inbuilt testing utility has been used to write unit tests for the modules. The tests can be run with `futhark test tests/`. In addition, these tests are automatically run every day with the GitHub Actions tool, and thus, new modifications are continuously and consistently tested. It is tested with the most recent version of Futhark, and thereby, the tests will indicate if anything needs to be updated.

These tests are primarily made to be quick and concise such that they can be used painlessly under development. Also, they're made to not fail when insignificant numerical errors happen. There are tests of the utilities, kernel implementations, solver, and of the full C-SVM implementation. For testing the full implementation, the tiny Iris dataset [Fis36] (which has two nonlinearly separable classes) is used, and its output is compared with the output of LIBSVM. The objective values achieved are identical (down to the last decimal), refer to Table 3.1.

Kernels	**Objective values: FSVM**		
Linear	−2.7480	−0.2036	−15.7598
Polynomial	−77.3304	−19.8216	−289.5543
Gaussian (RBF)	−3.1437	−3.5925	−74.4895
	Objective values: LIBSVM		
Linear	−2.7480	−0.2036	−15.7598
Polynomial	−77.3304	−19.8216	−289.5543
Gaussian (RBF)	−3.1437	−3.5925	−74.4895

Table 3.1: Objective values found when training with Futhark-SVM and LIBSVM using the shown kernels on the Iris dataset. The table contains three objective values per training since there are three classes, and it solves once for each pair of classes. The values are the same and the test passes.

The implementation has also been evaluated on much larger datasets that are not as suitable for automated tests in a development environment. The results of that will be reviewed in the experimental study, Chapter 4.

3.7 Python binding

Python is one of the most used languages for data science. It has excellent libraries for data loading and preprocessing, and thus, it makes it easier to test and benchmark an SVM library than if it had to be done solely with Futhark. Many other SVM libraries, such as LIBSVM and ThunderSVM, also provide Python bindings. It felt natural to provide a binding such that it is possible to use FSVM in Python. In order to make the compiled Futhark OpenCL program available in Python, I used `futhark_ffi`[1]. The interface was inspired by scikit-learn. Listing 6 shows example usage.

```python
from futhark_svm import SVC
import numpy as np

X_train = np.array([[0, 1], [1, 0]])
y_train = np.array([0, 1])

m = SVC(kernel='rbf', C=10)
m.fit(X_train, y_train)
m.predict(np.array([[0, 1]])) #=> [0]
```

Listing 6: Example usage in Python. The class `SVC` has a similar interface to the class of the same name in scikit-learn.

[1] `https://github.com/pepijndevos/futhark-pycffi/`

Chapter 4

Experimental Study

In this chapter, FutharkSVM (FSVM) is benchmarked versus ThunderSVM (TSVM) in terms of accuracy, objective values, number of support vectors, and time used to train and predict. The steps to run the benchmarks are outlined in the `bench/` folder of the source code repository. Both libraries have Python interfaces with the same input format, and thus, to ensure fairness and make reproduction easier, data loading is done using Python. The three datasets used are detailed in Table 4.1.

Name	**Cardinality**	**Features**	**Classes**
MNIST	60000	784	10
Olivetti	400	4096	40
Adult	48842	113	2

Table 4.1: Datasets used for the benchmarks.

The MNIST dataset comes with 60,000 training samples and 10,000 test samples. Olivetti and Adult has randomly been split into training and test sets, with 10% of the total dataset used for testing (40 and 4,884 test samples, respectively). Adult, that has 14 features originally, comes with some categorical data that has been one-hot encoded. The features of all datasets have been normalized to the $[0, 1]$ interval.

The experiments are conducted with an Intel i3-8100 CPU with 16 GB RAM and an NVIDIA GTX 1060 GPU with 3 GB RAM. Both SVM implementations use 32-bit floating-point numbers for all benchmarks. FSVM is compiled with OpenCL while TSVM uses CUDA-C and C++. The Python interfaces make calls to these compiled libraries.

The Gaussian RBF kernel and the polynomial kernel are used. The parameters C, γ, d, and c used were selected by ad hoc experimentation: Parameters that achieved a decent test accuracy with TSVM were used.

4.1 Training

Table 4.2 shows the times achieved when training. FSVM rivals the performance of TSVM on most of the experiments. For MNIST, FSVM performs a bit worse than TSVM on two out of the six experiments. On the other four, FSVM performs slightly better (1.05 to 1.15 times faster than TSVM). It performs better than TSVM on the Olivetti dataset. When there are binary problems with less than $n^{\text{ws}} = 1024$ data points, FSVM will use an SMO solver that computes the full kernel matrix, rather than the solver using two-level decomposition. This solver is much faster, as it has no overhead from employing a cache, nor does it suffer from attempting to decompose a dataset that already has a suitable size. Datasets with few data points per class, such as Olivetti, can benefit from this. FSVM is significantly slower on the Adult dataset, but it is difficult to rule out if constant factors are at play since the times are low. For the experiments, the features were normalized to the $[0, 1]$ interval because TSVM printed `nan` differences and seemed stuck otherwise. FSVM could handle the non-normalized Adult dataset without any issues. This is perhaps because of the additional safety measures, e.g., ensuring that $q > 0$ and $\eta_{i,j} > 0$ (TSVM employs neither of those checks).

#	**Dataset**	**Kernel**	**Parameters**				**Time (seconds)**		**Speedup**
			C	γ	d	c	**FSVM**	**TSVM**	
0	MNIST	Poly.	10	0.01	2	0	33.13	**23.95**	0.72x
1	MNIST	Poly.	10	0.01	3	0	**23.03**	24.24	1.05x
2	MNIST	Poly.	10	0.01	4	0	**22.94**	23.96	1.05x
3	MNIST	RBF	0.1	0.001	-	-	**28.54**	31.71	1.11x
4	MNIST	RBF	1	0.001	-	-	**21.41**	24.63	1.15x
5	MNIST	RBF	10	0.01	-	-	31.81	**23.60**	0.74x
6	Olivetti	Poly.	10	0.01	3	0	**3.68**	8.08	2.20x
7	Olivetti	RBF	10	0.01	-	-	**4.07**	7.46	1.83x
8	Adult	Poly.	1	0.0001	3	0	2.50	**1.38**	0.55x
9	Adult	RBF	1	0.0001	-	-	2.52	**1.55**	0.61x
10	Adult	RBF	10	0.0001	-	-	3.72	**1.46**	0.39x

Table 4.2: Benchmark results for training. Speedup is the relative performance calculated by $time^{\text{TSVM}}/time^{\text{FSVM}}$. Best if **bold**. FSVM=FutharkSVM, TSVM=ThunderSVM

Table 4.3 shows some important measurements after training; it contains the mean of the objective values, the bias of the last binary model that was trained, and the training error. The objective values and biases of FSVM and TSVM are almost identical, which indicates that the trainings have achieved similar optimizations. The training error of FSVM is slightly higher for some of the experiments. As neither FSVM nor TSVM employs any tie-breaking strategy when predicting, it might be caused by them having sorted the classes differently. E.g., while FSVM will always choose 0 in a tie between classes 0 and 1, TSVM might always choose 1.

#	**Mean obj. value**		**Bias (last)**		**Training error**	
	FSVM	**TSVM**	**FSVM**	**TSVM**	**FSVM**	**TSVM**
0	-860.50	-860.50	0.127	0.128	0.066	0.001
1	-1193.37	-1220.34	0.227	0.227	0.008	0.001
2	-2219.09	-2219.06	0.333	0.332	0.003	0.003
3	-225.39	-225.39	-0.574	-0.564	0.092	0.092
4	-887.25	-887.25	-0.319	-0.319	0.061	0.061
5	-578.46	-578.46	1.346	1.345	0.017	0.000
6	-0.14	-0.14	-2.564	-2.564	0.000	0.000
7	-3.90	-3.90	-3.130	-3.130	0.000	0.000
8	-21113.8	-21113.8	-1.000	-1.000	0.152	0.152
9	-16566.9	-16566.9	-1.022	-1.023	0.164	0.148
10	-152211	-152210	-1.554	-1.556	0.240	0.240

Table 4.3: Mean objective value, last bias, and training error.
#0 corresponds to training setup #0 in Table 4.2, etc.
FSVM=FutharkSVM, TSVM=ThunderSVM

4.2 Prediction

Table 4.4 lists the number of support vectors (SVs) that were outputted by training, the test accuracy, and the time it took to predict the data points in the test set. The number of SVs is listed to show that the tests are fair; the time it takes to predict is related to the number of SVs. It can be seen that FSVM and TSVM have the same amount of SVs, and as such, the prediction times should be "fair" in that aspect. Test accuracies are quite similar except for two deviations where FSVM performs a bit worse than TSVM. Again, the difference might be caused by tie-breaking. Lastly, FSVM is consistently 2.4 to 9.6 times faster than TSVM for prediction.

#	Support vectors		Test accuracy		Time (seconds)		Speedup
	FSVM	TSVM	FSVM	TSVM	FSVM	TSVM	
0	8721	8718	0.951	0.980	**0.365**	1.498	4.10x
1	8135	8133	0.976	0.979	**0.352**	1.457	4.14x
2	8424	8423	0.974	0.974	**0.369**	1.495	4.05x
3	37949	37953	0.913	0.913	**1.466**	3.902	2.66x
4	20807	20807	0.941	0.941	**0.837**	2.487	2.97x
5	10868	10868	0.957	0.983	**0.461**	1.758	3.81x
6	353	353	0.975	0.975	**0.003**	0.028	9.33x
7	350	350	0.975	0.975	**0.003**	0.029	9.66x
8	21301	21370	0.769	0.764	**0.075**	0.186	2.48x
9	17694	17689	0.854	0.854	**0.060**	0.161	2.68x
10	15513	15511	0.838	0.854	**0.054**	0.146	2.70x

Table 4.4: Benchmark results for prediction and test accuracies. The number of total unique support vectors found when training are listed, as they impact prediction times. Speedup is the relative performance calculated by $time^{\text{TSVM}}/time^{\text{FSVM}}$. Best if **bold**. #0 corresponds to training setup #0 in Table 4.2, etc. FSVM=FutharkSVM, TSVM=ThunderSVM

4.3 Reflection on results

TSVM boasts a massive 100x performance improvement over LIBSVM when training with GPUs. It is 10-100x faster than LIBSVM for prediction [Wen+18]. The benchmarks show that the performance of FSVM and TSVM is quite similar when training. Especially for MNIST, the largest dataset used in the benchmarks, the difference is very small. For prediction, FSVM is 2.4-9.6x faster than TSVM. This is most likely due to the fact that the *voting* in the prediction function of FSVM is done via the GPU, while it is not in TSVM. Considering that Futhark is a high-level language, with additional run-time safety checks (e.g. bounds checking), and that it is hardware agnostic, and as such, will not be able to provide as many low-level hardware-specific optimizations (TSVM uses CUDA and libraries that run on NVIDIA hardware only), these results are very good. It shows that an SVM implementation in Futhark can compete with TSVM. Furthermore, Futhark is still in development, and it seems to be getting a lot faster over time[1]. If this trend continues, then FSVM will also be getting faster and better.

[1] `https://futhark-lang.org/blog/2020-07-01-is-futhark-getting-faster-or-slower.html`

Chapter 5

Conclusion and Future work

5.1 Conclusion

In this thesis, a new SVM library in Futhark has been introduced. It has modular kernels, such that new kernel functions can be implemented by the users of the library. It compiles into very efficient GPU code, and the benchmarks that were conducted show that it rivals the performance of the popular SVM library ThunderSVM when training. On one dataset it performed 2.2x better, while it performed slightly worse on others. A new method for one-versus-one prediction on GPUs was proposed. It computes predictions for all binary models in parallel and it also counts votes in parallel. The benchmarks showed that it is consistently 2.4 to 9.6 times faster than ThunderSVM at predicting. Since Futhark is hardware agnostic and has multiple backends, it can be compiled to run on more hardware than ThunderSVM which only runs on NVIDIA GPUs. As a high-level language, Futhark also provides additional safety checks to ensure memory safety and the like. Finally, Futhark programs have been shown to become faster over time as the compiler improves. The library introduced has shown that Futhark is an excellent choice for the implementation of SVMs.

5.2 Future work

In order for this library to become a complete SVM toolkit in the future, it needs to implement support vector regression, and one-class SVMs. These would not be very difficult to implement using the SMO solver, and are models that are expected from an SVM toolkit. Besides more models, there are still a lot of strategies and tweaks that can be explored in order optimize the implementation further. For example, as the kernel matrix is symmetric, and n^{ws} kernel rows are computed every iteration, it is possible to reuse n^{ws} columns of the rows every iteration (and thereby compute $n^{\text{ws}} \times n^{\text{ws}}$ fewer dot products). Also, since the matrix is symmetric, it might be possible to achieve better performance by using sparse matrix multiplication.

Another problem that could be fixed in the future is that FutharkSVM does not work on datasets that are too big for GPU memory. As Futhark takes care of all memory management, it is not possible for the programmer to decide if the dataset is stored in CPU or GPU memory. As a result, the entire dataset which we train on is stored in GPU memory. For FutharkSVM to work with very large datasets, or on machines with little GPU memory, a program (written in C or a similar language) could be necessary to call the essential parts of the Futhark program which uses the GPU.

Bibliography

[Fis36] R. A. Fisher. “The Use of Multiple Measurements in Taxonomic Problems”. In: *Annals of Eugenics* 7.7 (1936), pp. 179–188.

[BGV92] Bernhard E. Boser, Isabelle M. Guyon, and Vladimir N. Vapnik. “A Training Algorithm for Optimal Margin Classifiers”. In: *Proceedings of the Fifth Annual Workshop on Computational Learning Theory*. COLT ’92. Pittsburgh, Pennsylvania, USA: Association for Computing Machinery, 1992, pp. 144–152.

[CV95] Corinna Cortes and Vladimir Vapnik. “Support-Vector Networks”. In: *Mach. Learn.* 20.3 (Sept. 1995), pp. 273–297.

[OFG97] Edgar Osuna, Robert Freund, and Federico Girosi. “An Improved Training Algorithm for Support Vector Machines”. In: (1997), pp. 276–285.

[Pla98] John Platt. “Sequential Minimal Optimization: A Fast Algorithm for Training Support Vector Machines”. In: *Advances in Kernel Methods-Support Vector Learning* 208 (July 1998).

[BB00] Kristin Bennett and Erin Bredensteiner. “Duality and Geometry in SVM Classifiers”. In: (Sept. 2000).

[Kee+01] S. S. Keerthi et al. “Improvements to Platt’s SMO Algorithm for SVM Classifier Design”. In: *Neural Comput.* 13.3 (Mar. 2001), pp. 637–649.

[BTB05] S. Boughorbel, J-P. Tarel, and N. Boujemaa. “Conditionally Positive Definite Kernels for SVM Based Image Recognition”. In: (2005), pp. 113–116.

[FCL05] RE Fan, PH Chen, and Chih-Jen Lin. “Working Set Selection Using Second Order Information for Training SVM”. In: *Journal of Machine Learning Research* 6 (Jan. 2005), pp. 1889–1918.

[LC10] Yann LeCun and Corinna Cortes. "MNIST handwritten digit database". In: (2010).

[CL11] Chih-Chung Chang and Chih-Jen Lin. "LIBSVM: A Library for Support Vector Machines". In: *ACM Transactions on Intelligent Systems and Technology* 2 (3 2011). Software available at `http://www.csie.ntu.edu.tw/~cjlin/libsvm`, 27:1–27:27.

[Ped+11] F. Pedregosa et al. "Scikit-learn: Machine Learning in Python". In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[DG17] Dheeru Dua and Casey Graff. *UCI Machine Learning Repository*. 2017. URL: `http://archive.ics.uci.edu/ml`.

[Hen+17] Troels Henriksen et al. "Futhark: Purely Functional GPU-Programming with Nested Parallelism and in-Place Array Updates". In: *SIGPLAN Not.* 52.6 (June 2017), pp. 556–571.

[Wen+18] Zeyi Wen et al. "ThunderSVM: A Fast SVM Library on GPUs and CPUs". In: *Journal of Machine Learning Research* 19 (2018), pp. 797–801.