An Embedded Support Vector Machine

Rasmus Pedersen[1], and Martin Schoeberl[2]

[1]Department of Informatics,
CBS, Copenhagen, Denmark
rup.inf@cbs.dk

[2]Institute of Computer Engineering,
Vienna University of Technology, Vienna, Austria
mschoebe@mail.tuwien.ac.at

Abstract — *In this paper we work on the balance between hardware and software implementation of a machine learning algorithm, which belongs to the area of statistical learning theory. We use system-on-chip technology to demonstrate the potential usefulness of moving the critical sections of an algorithm into HW: the so-called hardware/software balance. Our experiments show that the approach can achieve speedups using a complex machine learning algorithm called a support vector machine. The experiments are conducted on a real-time Java Virtual Machine named Java Optimized Processor.*

1 Introduction

In this paper we exploit the fact that FPGA technology enables us to focus more on the balance between programming the machine learning algorithm entirely in a high level language such as Java and programming it entirely in a hardware (HW) language such as VHDL. We have chosen to focus on statistical learning theory because it represents a field characterized by both mathematical rigor and applicability to practical problems like embedded machine learning [1]. The SVM is an algorithm out of this field and we focus on one such algorithm in this paper. It is the non-linear binary classifier SVM. Starting with this fundamental binary classifier will later make it feasible to analyze the subsequent research into other areas such as novelty detection, regression, ranking, clustering, and corresponding on-line variants. We take the same position as [2] regarding the potential of moving embedded systems from being ”dumb” reactive systems to ”intelligent” proactive systems. The open source Distributed Support Vector Machine (DSVM) [3] is used in this paper to discuss many of the motivational reasons laid out in [2]. The experimental platform is the Java Optimized Processor [4] (JOP). JOP allows for a fine-grained analysis of resource consumption (mainly speed and HW area) for different setups of the algorithm. We provide the key terms that are used in this paper in the appendix.

2 Background

The SVM is the first algorithm produced by Vladimir Vapnik's statistical learning theory frameworks [5]. Later, the SVM is extended by Cortes and Vapnik to cover binary classification problems with misclassifications [6]. The most significant discovery in terms of enabling the using SVMs in embedded systems [1] is probably attributed to John Platt [7]. Using his sequential minimal optimization method we are able to train SVMs using an insignificant memory footprint. Later, the SVM is extended to most of the other classic machine learning frameworks such as regression, novelty detection, ranking, clustering etc. [8] [9] [10].

2.1 Support Vector Machine

The support vector machine has been chosen because it represents a framework both interesting from a machine learning perspective and from an embedded systems perspective. A SVM is a linear or non-linear classifier, which is a mathematical function that can distinguish two different kinds of objects. These objects fall into *classes*, which is not to be mistaken for a Java class.

Training a SVM can be illustrated with the following pseudo code:

Algorithm 1 Training an SVM

Require: $\boldsymbol{X}$ and $\boldsymbol{y}$ loaded with training labeled data, $\boldsymbol{\alpha} \Leftarrow 0$ or $\boldsymbol{\alpha} \Leftarrow$ partially trained SVM

1: $C \Leftarrow$ some value (10 for example)
2: **repeat**
3: **for all** $\{\boldsymbol{x_i}, y_i\}, \{\boldsymbol{x_j}, y_j\}$ **do**
4: Optimize α_i and α_j
5: **end for**
6: **until** no changes in $\boldsymbol{\alpha}$ or other resource constraint criteria met

Ensure: Retain only the support vectors ($\boldsymbol{\alpha_i} > 0$)

We use the sequential minimal optimization (SMO) method to train the SVM. The algorithm is — as the name indicates — a sequential optimization algorithm. Line 5 of Algorithm 1 indicates an optimization step. The details has been nicely explained in important papers such as Platt's SMO paper from 1999 [7]. The essence of SMO is just a pair of two *Lagrange multipliers* α_i and α_j is being optimized at a time. A key term is the *kernel* k, which is a mathematical function to compare two data points at a time. The engineering part of using SVMs is to choose the right kernel for the task at hand. The optimization takes place according to the imposed constraints, which are based on the *KKT* (Karush-Kuhn-Tucker) conditions as well the soft margin parameter C. Perhaps most importantly, the SMO algorithm takes away the need for a dedicated matrix library. Such a library is generally needed when solving quadratic optimization tasks. An important side effect of using SMO is that it can scale from requiring virtually no memory cache up to the point where the whole kernel matrix is cached. The kernel matrix is generated by taking the kernel $k(\boldsymbol{x}_i, \boldsymbol{x}_j)$ of all pairs of inputs. Note that the kernel matrix is symmetric and semi-definite. One very straightforward optimization technique is to

cache the matrix in its full $n \times n$. size.

The SVM follows a three stage approach like many other machine learning algorithms: Train, test, and predict. We focus on the kernel function which is common for all three states. Our focus is to test whether we can implement a faster kernel function in HW (VHDL) than in SW (Java).

2.2 A Java Processor

The Java Optimized Processor (JOP, [4]) is an FPGA based hardware implementation of the Java Virtual Machine. One of the motivations of JOP is that is has been created with worst-case execution time (WCET) in mind. This design principle is consistent throughout the JOP processor. In particular, this makes it possible to analyze certain data mining algorithms with respect to their WCET properties. As the processor is a soft core and all sources are available it is easy to add hardware accelerators to this processor.

3 Hardware Acceleration

As we are using FPGA technology we can experiment with moving certain parts of the algorithm between hardware and software. The critical –ie. time consuming– sections of the SVM can be implemented in hardware. That means that the kernel functions can be moved to hardware for faster execution. The kernel functions are the dot products, and the Gaussian kernel functions. We use 32-bit integers (16.16 fixed point representation) to represent the numbers in the system. The SVM can be implemented to use fixed point (FP) math in the kernel operations, and the error calculations. The disadvantage is that we would need to call a specialized library each time the variable was used, which would result in many method calls. It is possible by post-processing the code such that all such calls are replaced with the actual source code just like inline code in *C*. This would avoid the expensive method invocations. A property of the fixed point math class could be to move the radix point. In some algorithms it might be desirable to use an 8.24 representation and in others it can be necessary to use a 24.8 setup. To achieve our goals of this paper we only need the mathematical functions of `add`, `sub`, `mul`, `div` and `sqrt`. We did choose to implement this operations in Java and the time critical vector dot product in VHDL.

3.1 Hardware Implementation

In our setup with a soft core processor in an FPGA we have a large design space to investigate. A software only implementation and an almost full hardware implementations are two extreme points in this design space. However, we can also configure the processor with respect to caches size and hardware/software implementation of Java bytecodes. The software only implementation with an real-time method cache [11] of 4KB and 16 blocks is our baseline for the measurements in Section 4.

The most time consuming operation in the SVM (during training and during classification) is the dot product between two vectors. This is similar to the well known FIR filter in digital signal processing. However, for the dot product in the FIR filter one vector contains constants. In our case both vectors contain values that have to be loaded into the hardware accelerator.

We use fixed point calculations and therefore the hardware unit is a simple integer Multiply-and-Accumulate (MAC) unit. One design trade-off is the internal length of the multiplier and the accumulator. The input values are in 16.16 fixed point and the output is scaled to 16.16. In one extreme we can implement a 32x32 multiplier and an accumulator of $64 + n$ bits (with n as the vector size). The other extreme is to scale the input values to less than 32 bits to reduce the hardware resources for the multiplier.

Another design decision is how we implement the multiplier. A single-cycle hardware implementation of a 32x32 multiplier with a 64 bit result is not a useful approach as it leads to a low system frequency. In Table 1 the resource consumption and maximum operation frequencies for different multiplier implementations in a Cyclone EP1C6 FPGA [12] are given. The resource consumption is given in logic cells (LC) that are basic building blocks in an FPGA. As a reference a basic configuration of JOP with 1KB method cache consumes about 2000 LCs and can be clocked at 100MHz [13].

Pipeline	Implementation	Size [LC]	fmax [MHz]
1	single cycle	1620	66
4	pipelined	1660	96
16	pipelined	1698	103
-	serial Booth multiplier	144	106

Table 1: Different multiplier implementations in a Cyclone FPGA

A serial Booth multiplier as a viable option as it consumes few resources and has no impact on the whole systems operating frequency (100MHz). However, this is the same implementation as used for the bytecode `imul`, which implements a 32x32 signed multiplication with 32 bits result, in JOP. The Booth multiplier is not pipelined and we have to wait for the result 32 cycles. The speed gain is only due to the hardware accumulator and we get a higher accuracy when scaling the result back to 16.16 after accumulation (we use the full 64 bits result from the multiplication during accumulation).

With a pipelined multiplier we can feed new data into the MAC unit each cycle. We only have to wait the number of pipeline stages at the end of the full MAC operation. This means we get a better speed-up with larger vectors.

From Table 1 we can see that the pipeline registers are almost for free as each LC that is used for the combinatorial path of the multiplier contains an otherwise unused register. We choose a pipelined multiplier with 16 pipeline stages as it fulfills our target system frequency of 100MHz.

4 Experiments

We do experiments with fixed point math and hardware implementation. For each method call done in Java, the JVM creates a new stack frame. This overhead can potentially be a waste of CPU cycles for the most active sections of the code. Therefore we have inlined the critical sections. The speed tests on JOP using the binary SVM is performed using several types of tests. Another one is to use the FixedPoint math class. The latter one is implemented in Java as well as in VHDL.

4.1 Low Level Test

In this section we compare two software versions against a hardware MAC unit of for the vector dot product. The fixed point format is 16.16 and the vector size is 2[1]

Implementation	Time [cycles]
Java	650
Java simplified	286
HW MAC	244

Table 2: Java MAC and hardware MAC

Table 2 shows the execution time in clock cycles. The first line contains the *correct* implementation of 16.16 multiplication in Java. As data type `long` is very expensive we used four multiplications with the necessary shift operations to not loose any precision during multiplication. The second version is a simplified 16.16 multiplication where the operands are scaled before multiplication:

```
result = (a>>8) * (b>>8)
```

In this case we loose the 8 least significant bits of both operands before multiplication. Both software solution perform the accumulation in 16.16 with the possibility of loosing precision and overflow.

The hardware MAC unit performs 32*32 multiplications with a 64 bit result and the accumulator is 64 bits width. Therefore we keep all significant bits during the whole MAC operation. Only the final result is scaled back to 16.16.

From Table 2 we can see that the hardware implementation is about 2.7 times faster than the correct Java implementation and still 17% faster than the simplified Java version. It has to be evaluated how the simplified Java implementation results in more support vectors to be used in the SVM and the influence on the error.

It has to be noted that the access of the coefficients dominates the execution time for the hardware MAC and the simplified Java version. This costly access to the array elements is inherent in Java as all array accesses are boundary checked. A further optimization of the hardware could be to fetch the operands from memory by the hardware without boundary checking. However, this optimization undermines the safety property of the Java language.

4.2 Test Data

We use artificial test data generated by a test program. The data generated by this class will be be binary classification data according to parameterized distributions. The difference between this test and the one done before is that we do the test with the SVM code ”as-is”. Hence, we leave the calls to the fixed-point (FP) math library in the code without inlining it. The rationale behind it is to compare the speedup between using the FP Java

[1]This vector size is also used in the following section. Longer vectors would result in a higher speed-up of the hardware solution.

library with no code optimization done. Then we exchange the calls to the fixed point library with inline code directly to the MAC HW unit. The reason for this approach is to point out the *potential* of optimizing Java machine learning code written without special care of optimization. However, it should be noted that it *can* make sense to avoid too much manual code optimization. From a point of maintainability, it can be more costly to keep highly optimized code updated. Therefore, we have created some test data to test the pure Java approach vs. a Java/VHDL approach.

The minimal interface needed is set up the data generating class is specified in the following paragraph. The only thing that the data generating class needs to know is how the layout of the data is. We use a simple approach with a center for a data distribution of either Gaussian or uniform distribution. The simplest instantiation of this setup is to create one center for each class of the binary data: one center for the positive class (+1) and one center for the negative (-1) class. The data generator can create different proportions of the data. This is accomplished by specifying the intended proportion of each data center. We still need to specify the distribution of the data in a data center and that is accomplished by specifying the distribution for each dimension of the data in the data center. For example, a data center could have a uniform distribution for the first dimension and a Gaussian distribution for the second dimension. We can summarize the setup of a data generating center:

center The center of the distribution

width Each dimension of a data center is specified as either uniform or Gaussian. The supplied parameter is the standard deviation for the Gaussian distribution and the half of the range for the uniform distribution. Note: that the distributions are not alike even for equal widths

proportion This is the proportion that this data center. The proportions for all the specified data centers must add up to 1.0

The standard data suggestion (as depicted in Figure(1)) shows four data setups. Two is using the uniform data distribution and two are using the Gaussian data distribution. Two have two data centers two have four data centers. The difference between the uniform datacenters ((a1) and (a2)) and the Gaussian datacenters ((b1) and (b2)) is that the latter have overlapping data. We can try to describe the four setups using the terminology introduced in the previous section. With reference to Figure(1):

Testdata	Data	Centers	Type	Param
$a1$	separable	-(3.0, 3.0), +(5.0, 5.0)	uniform	1.0 width
$a2$	non-separable	-(3.0, 3.0), +(5.0, 5.0)	normal	1.0 st. deviation
$b1$	separable	+(3.0, 3.0), +(5.0, 5.0)	uniform	1.0 width
		-(3.0, 5.0), -(5.0, 3.0)		
$b2$	non-separable	+(3.0, 3.0),+(5.0, 5.0)	normal	1.0 st. deviation
		-(3.0, 5.0),-(5.0, 3.0)		

Table 3: Properties of the four test cases

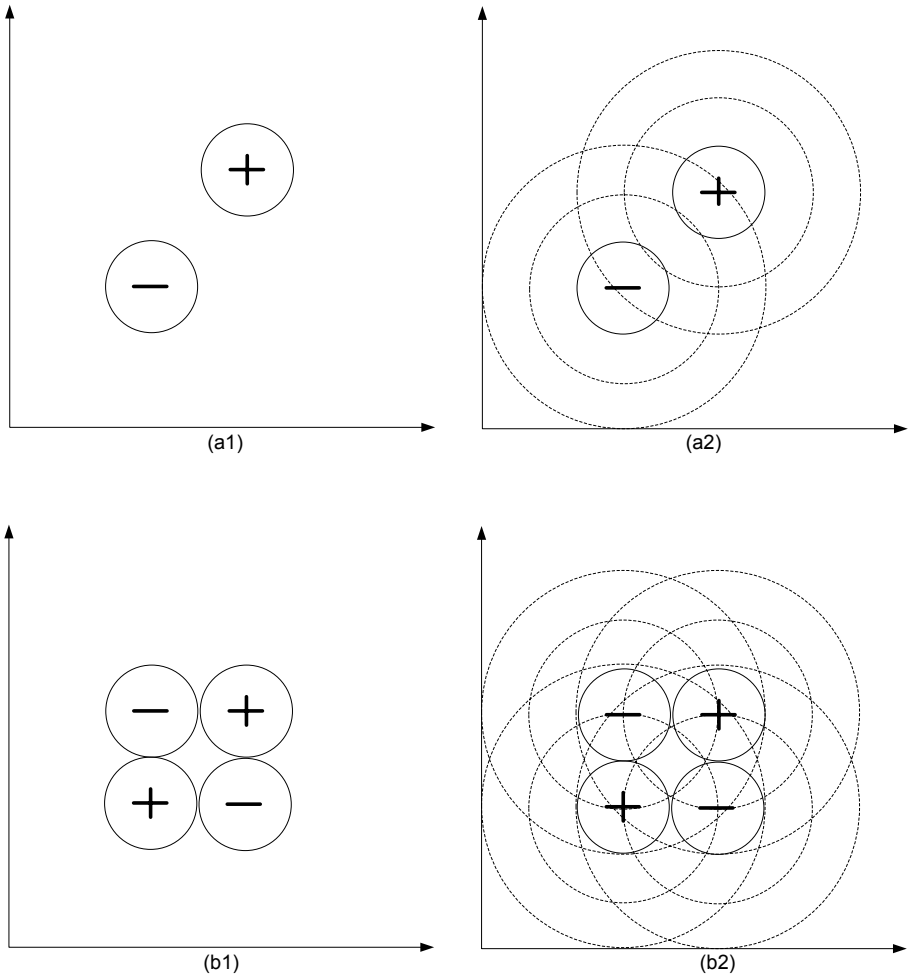

Figure 1: The four suggested standard data setups for the system.

Certain variations of these standard setups are optional. We can increase the number of dimensions while sticking to the same idea of using the same distribution for each dimension. We use 60 training points and 60 test points in the setup.

4.3 Test Evaluation

For each of the four standard tests (see Table 1) $\{$Test 1–4$\}$ we assume that the SVM is not pre-configured to know the center coordinates of the data centers. There are two output parameters of the testing: one is the elapsed time from the first labeled data point of the first test is served to the unlabeled data point of the last test is classified.

The SVM undergoes two major states: training and testing. The training is sequenced out in Algorithm 1. Testing or prediction is the time it takes to classify a new data instance using the trained model. For each of the 4 standard test configurations in Table 3, we record the time in μs to complete the training phase and to classify one new point.

The *train* column depicts the total training time (ie. before the system goes real-time. The *test* column depicts the time to classify a new point in real-time mode. We can see a significant difference in the training time (especially with the more demanding test

Test	Java train [μs]	Java test [μs]	MAC train [μs]	MAC test [μs]
1	498,943	216	497,122	213
2	568,579	261	363,977	259
3	1,762,416	950	1,200,544	892
4	1,236,402	1011	773,775	998

Table 4: Java kernel vs. HW kernel

data 2–4) between the simplified Java multiplication (with scaling before multiplication) and the full resolution hardware MAC. The SVM algorithm *tries* to compensate for the computation error during training. In the classification we see only small improvements with the hardware MAC. The speedup of 17% in the vector multiplication is lost in the overall execution time of the SVM algorithm.

The experiments in Table 4 were carried out with a cache size of 4KB and 8 cache blocks. In addition, the effect of reducing the cache to 1KB and 4 cache blocks resulted in 3.2% longer prediction time and a reduction in size from 5,486 to 4,740 LCs. This is an interesting result as it demonstrates a small tradeoff in prediction time versus a larger nice reduction in the processor size of 13.6%.

4.4 Fixed Point Resolution vs. SVM Performance

The SVM algorithm 1 is likely to perform satisfactory over a range of parameters. Some parameters are set explicitly such as the constraint parameter C. Other parameters like the number resolution enter the algorithm in more subtle ways. The FP resolution is an example if this. The SMO SVM [7] has exit criteria built in that determines when convergence toward the *global* maximum of the optimization problem has been reached (see line 6 in the SVM pseudocode in Algorithm 1). It should be clear that the iterative convergence loops in the SMO SVM are sensitive to the FP resolution of the data representations. There are (at least) three different scenarios that can happen when we reduce the resolution of FP numbers:

1. The number of support vectors (#sv) changes
2. The test (and training) error changes
3. The algorithm does not converge

A main goal in machine learning is the ability of the algorithm to *generalize* well. That is the power to classify new data instances correctly, which was not part of the training data. If the number of support vectors generally would decrease as a result of a reduced FP resolution then the generalization error could also be expected to go down. An indication of this could be monitored by the change in the test error. Besides the positive or negative effect of on the test error it can also happen that the algorithm does not converge. This can happen for example when the reduced width leads to the truncation of a significant digit.

The following experiment records the number of support vectors (#sv) and the test (e) error for 4 different symmetric FP resolutions.

	Test 1		Test 2		Test 3		Test 4	
	#sv	e	#sv	e	#sv	e	#sv	e
16:16	8	0	11	8	56	35	60	33
12:12	8	0	NA	NA	56	35	NA	NA
8:8	8	0	56	54	56	35	30	38
4:4	8	0	58	33	49	35	60	33

Table 5: Fixed point resolution vs. support vectors (#sv) and test error (e)

The results in Table 5 show no general reduction in the number of support vectors. Furthermore, the test error does not generally decrease for this particular set of tests. It seems like reducing the FP resolution (which would lead to smaller HW) does not directly lead to smaller test errors. It should, however, be noted that the SVM algorithm [3] has tolerance parameters which could be tuned together with the FP resolution. The algorithm did not converge for the 12:12 resolution in test 2 and 4, which was pointed out at a possibility prior to running the test. There are, however, several methods to reduce the number of support vectors: A direct method to control the number of support vectors is presented by Wu et al. [14], which we predict will have a significant influence on future embedded kernel machines like the SVM.

5 Conclusion

We show that it is possible to optimize the SVM by moving certain sections of the code back and forth between Java and HW on JOP. The results are positive in the regard that the SVM runs faster when the critical sections are implemented in HW. This approach may be applicable for other algorithms that are iterative in nature. It allows one to optimize the program language design space: Java for the main part, with its ease of maintainability and VHDL for the critical sections.

Perhaps the most important parameter in traditional machine learning and data mining is the test error. We do experiments to monitor the impact of various FP resolutions on the test error. The conclusion from that particular experiment is that blind HW optimization without regard to the SVM was not particular viable.

We demonstrate the idea of striking a balance between hardware implementation and software implementation. The SVM is the unit of analysis in this paper. The processor size can be traded against performance and thus fit well into different deployment scenarios. It is possible to expand this work on several other support vector machine algorithms. Future research is likely to focus on direct control of the number of support vectors [14].

A Key Terms and Definitions

Term	Definition
SVM	Support vector machine that classifies novel points on the nodes and only transmits key information such as support vectors across the network.
SMO	Sequential Minimal Optimization, which is a method for training an SVM efficiently [7].
$\#sv$	Set of support vectors belonging to an SVM model.
$\boldsymbol{\alpha}$	A Lagrange multiplier, which is a key parameter denoted $\boldsymbol{\alpha}$ in the SVM model. Each point with $\alpha_i > 0$ is an SV and its importance can be interpreted as the size of α_i
KKT	The Karush-Kuhn-Tucker conditions, which is a set of constraints that the SVM model must obey within a given error tolerance when trained to optimality.
tol	Tolerance parameter for the SVM that is used as a stopping criterion.
$k(\boldsymbol{x}_i, \boldsymbol{x}_j)$	Kernel evaluation that is a central computation in the SVM model. It is a dot product similarity measure between two data points in a space (usually) of higher dimensionality than the input space associated with the raw data observations.
$\#k$	A count of the number of kernel evaluations the SVM uses while computing the optimal $\boldsymbol{\alpha}$ for the SVM. $\#k$ is used as a proxy for computational time complexity in the machine learning community. it takes to classify a novel point. [15]
μs	Micro seconds, which is the unit we use to measure the execution time of the embedded Java processor with.
LC	Logical cell or logical element in a FPGA.

References

[1] Rasmus Ulslev Pedersen. *Using Support Vector Machines for Distributed Machine Learning*. PhD thesis, University of Copenhagen, 2005.

[2] Wilfried Elmenreich. Intelligent methods for embedded systems. In *Proceedings of the First Workshop on Intelligent Solutions in Embedded Systems (WISES 2003)*, pages 3 – 11, Austria, Vienna, June 2003.

[3] DSVM. Distributed support vector machine server and implementation, www.dsvm.org.

[4] Martin Schoeberl. *JOP: A Java Optimized Processor for Embedded Real-Time Systems*. PhD thesis, Vienna University of Technology, 2005.

[5] Vladimir Naumovich Vapnik. *The Nature of Statistical Learning Theory*. Springer, NY, 1995.

[6] Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Mach. Learn.*, 20(3):273–297, 1995.

[7] John Platt. *Fast training of support vector machines using sequential minimal optimization in Advances in Kernel Methods — Support Vector Learning*. MIT Press, 1999.

[8] B. Scholkopf, R. Williamson, A. Smola, J. Shawe-Taylor, and J. Platt. Support vector method for novelty detection.

[9] B. Schölkopf, P. L. Bartlett, A. Smola, and R. Williamson. Shrinking the tube: a new support vector regression algorithm. In M. S. Kearns, S. A. Solla, and D. A. Cohn, editors, *Advances in Neural Information Processing Systems 11*, pages 330 – 336, Cambridge, MA, 1999. MIT Press.

[10] A. Ben-Hur, D. Horn, H.T. Siegelmann, and V. Vapnik. Support vector clustering. *Journal of Machine Learning Research*, 2:125–137, 2001.

[11] Martin Schoeberl. A time predictable instruction cache for a java processor. In *On the Move to Meaningful Internet Systems 2004: Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES 2004)*, volume 3292 of *LNCS*, pages 371–382, Agia Napa, Cyprus, October 2004. Springer.

[12] Altera. Cyclone FPGA Family Data Sheet, ver. 1.2, April 2003.

[13] Martin Schoeberl. Evaluation of a Java processor. In *Tagungsband Austrochip 2005*, pages 127–134, Vienna, Austria, October 2005.

[14] Mingrui Wu, Bernhard Schölkopf, and Gökhan Bakir. A direct method for building sparse kernel learning algorithms. *Journal of Machine Learning Research*, 7:603–624, 2006.

[15] Bureau International des Poids et Measures. The international system of units (si), http://www.bipm.fr/en/si/.