High Performance Implementation of Support Vector Machines Using OpenCL

Ethan Peters

Follow this and additional works at: http://scholarworks.rit.edu/theses
High Performance Implementation of Support Vector Machines Using OpenCL

by

Ethan Peters

A Thesis Submitted in Partial Fulfillment of the Requirements for the Degree of Master of Science in Computer Engineering

Supervised by

Professor Dr. Andreas Savakis
Department of Computer Engineering
Kate Gleason College of Engineering
Rochester Institute of Technology
Rochester, New York
May 2014

Approved by:

Dr. Andreas Savakis, Professor
Thesis Advisor, Department of Computer Engineering

Dr. Muhammad Shaaban, Associate Professor
Committee Member, Department of Computer Engineering

Dr. Sonia Lopez Alarcon, Assistant Professor
Committee Member, Department of Computer Science
Thesis Release Permission Form

Rochester Institute of Technology
Kate Gleason College of Engineering

Title:

High Performance Implementations of Support Vector Machines Using OpenCL on Heterogeneous Systems

I, Ethan Peters, hereby grant permission to the Wallace Memorial Library to reproduce my thesis in whole or part.

______________________________________________________________
Ethan Peters

______________________________________________________________
Date
Dedication

This thesis is dedicated to my wife Cassandra Madison-Peters, without whose encouragement this work would not have been possible. I would also like to dedicate this work to my family, who have supported me throughout my academic career. Finally, I dedicate this work to my colleagues and professors, who have been instrumental in my education as well as my research.
I would like to thank Dr. Savakis for all of his guidance and help on this research. I would also like to thank the members of my committee, Dr. Shaaban and Dr. Lopez Alarcon, for their feedback and assistance to my work. Finally, I would like to thank my colleagues Riti Sharma and Breton Minnehan for sharing their knowledge with me, and Jack Stokes for his help in the writing process.
Abstract

High Performance Implementation of Support Vector Machines Using OpenCL

Ethan Peters

Supervising Professor: Dr. Andreas Savakis

Support Vector Machines are a machine learning approach that is well studied, thoroughly vetted and effective in a large number of applications. The objective of this thesis is to accelerate an implementation of Support Vector Machines (SVM) using a heterogeneous computing system programmed using OpenCL in C/C++. LIBSVM, a widely-available, popular and open source implementation of SVM is chosen, allowing the presented work to be integrated seamlessly into existing systems. The proposed framework is evaluated in terms of speed and accuracy when performing training and classification on a number of standard data sets. Testing was based on two work station GPUs, the NVIDIA GTX 480 and Tesla K20, and a modern, work station CPU (Intel i5 Quad Core, 3 GHz).

We find that, for large data sets, training is accelerated by a factor ranging from 9 to 22. In general, speedup increases with the total number of training samples in the data set until the GPU device is fully utilized. While these gains in speedup are significant, they do not match the ideal parallel speedup, that is the total number of cores in the parallel system. Our findings indicate that performance is hampered by the portions of the SVM training algorithm that are sequential. In addition, we find that the classification phase of the SVM system is accelerated by a factor of up to 12. During classification only a relatively small number of samples are classified compared to the typical number of training samples, and
the computational complexity of classification grows only linearly with the number of samples processed, as opposed to the training phase where it grows quadratically. The contributions of this thesis include the use of OpenCL for accelerating SVM training and testing on heterogeneous systems, and the performance analysis of the acceleration of SVM.
## Contents

Dedication ........................................................................................................ iii

Acknowledgments ............................................................................................... iv

Abstract ............................................................................................................... v

1 Background ......................................................................................................... 1
   1.1 Computer Vision ......................................................................................... 1
   1.2 Heterogeneous System Architecture ......................................................... 2
   1.3 OpenCL ........................................................................................................ 5
   1.4 Support Vector Machines ............................................................................ 5
   1.5 Thesis Overview .......................................................................................... 6

2 Support Vector Machines ................................................................................... 7
   2.1 SVM Applications in Computer Vision ....................................................... 16
   2.2 LIBSVM ....................................................................................................... 18
   2.3 GPU Assisted LIBSVM ............................................................................... 20

3 OpenCL .............................................................................................................. 22

4 SVM Acceleration .............................................................................................. 27
   4.1 Acceleration ................................................................................................. 27
      4.1.1 Overview .............................................................................................. 27
      4.1.2 Training ................................................................................................. 33
      4.1.3 Classification ......................................................................................... 36
   4.2 Comparisons ................................................................................................. 39
      4.2.1 Overview .............................................................................................. 39
      4.2.2 Timing .................................................................................................... 40
      4.2.3 Error ..................................................................................................... 40
5 Results ................................................................. 42
  5.1 Performance ...................................................... 42
    5.1.1 Training .................................................... 45
    5.1.2 Classification ............................................. 70
  5.2 Error ............................................................ 71
    5.2.1 Training .................................................... 71
    5.2.2 Classification ............................................. 76
  5.3 Summary of Results ............................................ 77
  5.4 Adapting to the Tesla K20 ..................................... 77

6 Conclusion ............................................................ 80
  6.1 Summary ........................................................ 80
  6.2 Future Work ..................................................... 83

Bibliography ............................................................. 86
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>SVMs seek to find a hyperplane which maximizes margin between two classes</td>
<td>8</td>
</tr>
<tr>
<td>2.2</td>
<td>SVM Formulations Offered by LIBSVM</td>
<td>18</td>
</tr>
<tr>
<td>2.3</td>
<td>SVM Kernels Offered by LIBSVM</td>
<td>18</td>
</tr>
<tr>
<td>2.4</td>
<td>LIBSVM Algorithm Flowchart</td>
<td>20</td>
</tr>
<tr>
<td>3.1</td>
<td>OpenCL Platform Model</td>
<td>23</td>
</tr>
<tr>
<td>4.1</td>
<td>Illustration of Matrix-Vector Multiplication</td>
<td>31</td>
</tr>
<tr>
<td>4.2</td>
<td>Parallelization of Matrix-Vector Multiplication. Colors indicate division of</td>
<td>31</td>
</tr>
<tr>
<td></td>
<td>labor amongst work groups</td>
<td></td>
</tr>
<tr>
<td>4.3</td>
<td>Illustration of Dot Product Parallelalization. Colors indicate division of</td>
<td>31</td>
</tr>
<tr>
<td></td>
<td>labor amongst threads</td>
<td></td>
</tr>
<tr>
<td>4.4</td>
<td>LIBSVM Algorithm Flowchart</td>
<td>36</td>
</tr>
<tr>
<td>4.5</td>
<td>Kernel Matrix Row Calculation Flowchart</td>
<td>37</td>
</tr>
<tr>
<td>4.6</td>
<td>Kernel Matrix Row Calculation Flowchart</td>
<td>37</td>
</tr>
<tr>
<td>4.7</td>
<td>Kernel Matrix Calculation Flowchart</td>
<td>38</td>
</tr>
<tr>
<td>5.1</td>
<td>Working Set Selection Comparison</td>
<td>47</td>
</tr>
<tr>
<td>5.2</td>
<td>BIO ID Raw Image Training Performance (GTX 480)</td>
<td>49</td>
</tr>
<tr>
<td>5.3</td>
<td>BIO ID HOG Training Performance (GTX 480)</td>
<td>50</td>
</tr>
<tr>
<td>5.4</td>
<td>BIO ID HOG Training Performance Across GPUs</td>
<td>51</td>
</tr>
<tr>
<td>5.5</td>
<td>PASCAL VOC Training Performance (GTX 480)</td>
<td>53</td>
</tr>
<tr>
<td>5.6</td>
<td>PASCAL VOC Training Perform Across GPUs</td>
<td>54</td>
</tr>
<tr>
<td>5.7</td>
<td>BIO ID Raw Image Training Performance (Non-Linear Kernels) (GTX 480)</td>
<td>56</td>
</tr>
<tr>
<td>5.8</td>
<td>BIO ID Raw Image Training Performance (Non-Linear Kernels) (GTX 480)</td>
<td>57</td>
</tr>
<tr>
<td>5.9</td>
<td>BIO ID HOG Training Performance (Non-Linear Kernels) (GTX 480)</td>
<td>60</td>
</tr>
<tr>
<td>5.10</td>
<td>BIO ID HOG Training Performance (Non-Linear Kernels) (GTX 480)</td>
<td>61</td>
</tr>
<tr>
<td>5.11</td>
<td>BIO ID HOG Training Performance (Non-Linear Kernels) On All GPUs</td>
<td>62</td>
</tr>
<tr>
<td>5.12</td>
<td>BIO ID HOG Training Performance (Non-Linear Kernels) Across GPUs</td>
<td>63</td>
</tr>
<tr>
<td>5.13</td>
<td>Speedups for All Training Kernels On All Available GPU Devices</td>
<td>64</td>
</tr>
<tr>
<td>5.14</td>
<td>Objective Function Calculation Performance Breakdown</td>
<td>65</td>
</tr>
<tr>
<td>5.15</td>
<td>Comparison of methods for updating objective function</td>
<td>66</td>
</tr>
</tbody>
</table>
Chapter 1

Background

1.1 Computer Vision

Computer vision deals with interpreting image data and obtaining an understanding of the depicted scene or object. It is an important pursuit for fields including artificial intelligence, robotics and human-computer interaction.

Parallel processing systems present major performance improvements for algorithms which are well suited to them. They are of particular interest for image processing applications, as most such algorithms have a high degree of inherent parallelism; that is to say they are well suited to being performed on parallel processing systems because they involve a large number of independent (typically light-weight) operations. For instance, in pixel and neighborhood processing algorithms, each output pixel value can be computed simultaneously by performing a relatively simple operation on a small amount of input data. Such systems can be made to scale well with the size of input data, because the computation time of such a parallel algorithm decreases linearly with the number of computing elements in the system, often without the need for individual computing elements to become significantly more sophisticated (and consequently more costly).

A Graphics Processing Unit (GPU) is a computational device that typically serves as the processing engine of a larger graphics processing system. These systems were originally designed to accelerate image rendering in desktop systems, and are of major importance in
systems designed for processing computer generated images (such as video game systems and 3D animation work stations). To benefit this specialized task, GPUs are designed to be low-cost, highly parallel processing systems, employing a large (and growing) collection of relatively weak cores. These characteristics make the GPU an attractive platform for applications with inherent parallelism, such as image processing, and so General Purpose computing on a GPU (GPGPU) has drawn a large amount of interest in recent years.

In addition to providing a low-cost parallel system, many image processing algorithms don’t suffer from the inherent weakness of each individual core, as the tasks to be performed are light weight and involve small amounts of data, therefore GPUs provide a platform which very closely matches many of the work loads encountered. This close match between the requirements of these algorithms and the benefits offered by GPUs means that in many cases they provide the optimal tradeoff between cost and performance. Even in situations where a GPU is not sufficient for the final intended application of a system, it can still prove useful as a testbed for a new parallel algorithm or data access model.

GPUs are not without their drawbacks, however. Standard processing cores are not typically used in graphics systems (for cost and chip resource reasons), thus GPUs present a specialized programming environment which requires a greater effort on the part of the programmer. Furthermore, manufacturers of GPUs adhere to different standards, not only for individual processing units but for the interconnect and command structure of the chip as well, meaning that each GPU is a different, specialized programming environment.

1.2 Heterogeneous System Architecture

Heterogeneous Computing (HC) refers to computing systems that not only use multiple computational elements, but elements of different types. Computing elements can vary widely in their internal structure and intended task, and consequently a single algorithm can have different performance implications depending on the platform on which it is run and the platform for which it is optimized. For instance, as previously mentioned GPUs tend to
have hundreds of processing cores, making them a good target platform for tasks requiring small, independent computations; however, these cores tend to be relatively weak, so a task that is inherently serial (requiring a long chain of operations to be performed in sequence) and generally intensive (such as floating point arithmetic) would be a poor fit for a GPU and a much better fit for a traditional CPU.

Additionally, NVIDIA GPUs make use of a VLIW type of instruction dispatch, where some number of instructions are dispatched in parallel to a single group, or warp, of cores. For Kepler architectures, two independent instructions can be dispatched to 32 cores at once[43]; this is a highly restrictive form of parallelism, thus an algorithm must be highly conducive to GPU acceleration for its benefits to overcome its drawbacks.

We can take advantage of this kind of system by segmenting the algorithm being implemented into separate algorithms based on the level and type of parallelism present in each part; then these separate algorithms are mapped onto and optimized for the available hardware that best meets their needs. Desktop work stations can be a good gateway into larger, more complete heterogeneous systems because modern work stations typically contain a multicore CPU, a discrete graphics card and an integrated GPU living on the same die as the CPU.

A multicore CPU, which is itself a traditional parallel processing system containing a limited number of very powerful cores, is extremely well suited to serial operations; individual CPU cores benefit from decades of research and development into optimizing their performance on traditional, serial operations, and include physical constructs that implement optimizations such as high accuracy branch prediction, deep pipelining, dependency resolution, multi-level caching and prefetching, and hardware conducive to instruction level parallelism such as reservation stations (to avoid having to share registers between unrelated operations) and redundant operative elements such as ALUs and FPUs (to allow unrelated operations to take place simultaneously).

Unfortunately, as a result of these advancements, modern CPU cores tend to occupy a
large die area and thus there are limitations on the number that can be fit onto a single chip. While these limitations can be accommodated to some degree by connecting large numbers of chips together in larger systems, cooperation between computational cores becomes difficult and costly across chip boundaries, while increasing the area of chip tends to decrease the yield of the process (or increase the likelihood that any single chip produced will be faulty and unusable); that is not to say that these systems can not be effectively applied to parallel problems (indeed, there is definitive proof that they can), but rather that the types and scale of parallel problems for which these kinds of system are optimal is limited by the cost of coordination and communication across chip boundaries, in addition to the cost in power of many-core chips and the monetary cost of fabricating more complex chips.

Discrete GPU devices, which contain a large number of weak cores, offer a different approach; individual GPU cores forego complex hardware constructs typical of standard CPU cores in favor of being small enough to fit tens or hundreds of cores onto a single die. Consequently, discrete graphics cards allow more tasks to be more closely coordinated at a reduced communication cost, at the expense of instruction level parallelism, since individual cores have only a limited capacity for computation.

Desktop workstations also often contain, as a bonus, GPUs integrated onto CPU core dies; such a GPU has fewer cores than a discrete GPU but has access to system memory and requires less overhead for coordination with the rest of the system. Integrated GPUs can communicate using the system bus, compared to discrete GPU devices which connects to a computer’s motherboard using PCI, or something similar, which presents significant overhead for communicating between computational devices.

The term Heterogeneous System Architecture is also applied to the design of integrated circuits consisting of tightly coupled heterogeneous computing units, the terms "system" and "architecture" being relatively flexible. For clarity, it is emphasized that this thesis does not present a hardware implementation, specification or design and the terms "HSA" and "Heterogeneous System Architecture" are used here only to refer to software systems
that are design to operate in a heterogeneous environment.

### 1.3 OpenCL

OpenCL (Open Computing Language) is a software platform designed to provide a consistent interface for programming computational devices for specialized hardware such as GPUs, Field Programmable Gate Array (FPGA) platforms, certain CPUs, etc. It is an open specification maintained by the Khronos Group, and it allows software written in a standard, familiar programming language (C99) to run on any of the devices supported, with an emphasis on parallel computation. It is of particular interest to developers working on GPGPU applications, because OpenCL works across all of the major GPU hardware vendors and doesn’t require learning a new programming language for each target device.

More importantly, however, is that OpenCL allows one’s application to work across a generalized, heterogeneous computing system. That is, OpenCL applications can make use of systems with many different types of computational devices. It allows this by first its ability to run on almost any major computational device, and second by providing utilities for an application to understand what hardware is available and to adjust task allocation across devices to best make use of the available hardware.

Having discussed the general means by which HC is applied to accelerate computation, in the next section (and the chapter following), we consider a particular application which is well suited to, and in serious need of the advantages described. Namely, the algorithms used in constructing and using Support Vector Machines, and their applications, are presented.

### 1.4 Support Vector Machines

Support Vector Machines (SVMs) are a particular style of machine learning algorithm commonly used in computer vision problems. SVMs operate in two phases: training and classification. During the training phase, the SVM is given data that is labeled by the class
to which it belongs; SVMs treat all data as vectors or points in some (potentially) high dimensional space. Traditionally, an SVM is used to distinguish between two classes of objects, so two labels are accepted. The algorithm then attempts to find a hyperplane that separates the two classes by maximizing the margin between the classes. During classification, the trained SVM identifies the class to which a point belongs by determining on which side of the plane the point lies.

1.5 Thesis Overview

In this thesis we contribute the following items: first, an accelerated implementation of LIBSVM, a popular and powerful library for machine learning, which is reasonably easily to use optimally on a variety of hardware and reasonably easy to use in place of the original; secondly, a case study in HC acceleration using the OpenCL platform. The work is laid out as follows: in the following chapter, we discuss the particular set of algorithms (namely those used in SVM) we are attempting to optimize; the next chapter presents our plans for optimization of the algorithms discussed; in the chapter which follows we present the experiments we have conducted as well as the results we have obtained and we draw general conclusions from them; finally we present a summary of our work, its meaning and logical next steps.
Chapter 2

Support Vector Machines

The Support Vector Machine (SVM) is a type of machine learning algorithm for binary (two class) classification problems, discussed at length in [2] and [17]. SVMs operate in two phases: a training phase, in which labeled data is used to build an SVM model, and a classification phase, during which the trained model is used to determine the class where some new set of data belongs. SVMs work by viewing input data as vectors, and consequently points in some high dimensional space. During training, the SVM finds a hyperplane which separates the labeled training data into distinct classes based on a maximum margin criteria. During classification, the SVM calculates where new data points lie with respect to this hyperplane using a decision function outlined below.

\[ D(\vec{x}) = \vec{w} \cdot \vec{x} + b \]  

(2.1)

where, \( \vec{x} \) is the new data vector being classified, and \( \vec{w} \) and \( b \) are a vector and a scalar, respectively, which are learned during training and describe the separating hyperplane. This function relates to the distance between \( \vec{x} \) and the hyperplane, and consequently the sign of \( D(\vec{x}) \) will determine to which class the point belongs.

It is not always possible to separate the training data using a linear hyperplane; for instance, positive samples may cluster around a single point, with negative samples scattered further out in which case it would be impossible to separate the data with a straight line.
Figure 2.1: SVMs seek to find a hyperplane which maximizes margin between two classes

However, for many data sets it may be possible to find some other shape to separate the data; that is, a hyperplane in some non-linear space may be ideal for these data sets. To do this, a kernel is introduced that maps input data into some space that is more conducive to separating the data, which manifests in the decision function as follows.

\[ D(\vec{x}) = \vec{w} \cdot \phi(\vec{x}) + b \]  

(2.2)

The actual Euclidian distance between the given data point, \( \vec{x} \), and the SVM’s characteristic hyperplane can be expressed as

\[ d(\vec{x}) = \frac{D(\vec{x})}{||\vec{w}||} \]  

(2.3)

An SVM model is in part characterized by its margin; that is, the distance from the hyperplane to the nearest points in the training set.

\[ M = \min_k \frac{y_k D(\vec{x}_k)}{||\vec{w}||} \]  

(2.4)

Here, \( y_k \) are data point labels, indicating negative or positive samples. A good SVM classifier seeks to create as much separation between the two classes as possible to reduce the possibility of confusing points that are near the class boundary. The approach presented by Boser et. al. [2] is to maximize this margin for the vectors closest to the boundary, called
the supporting vectors. This goal is expressed in the optimization problem below.

$$\max_{\vec{w}} \left( \min_k \left( \frac{\vec{y}_k D (\vec{x}_k)}{||\vec{w}||} \right) \right)$$  \hspace{1cm} (2.5)

Here, $\vec{y}_k = \pm 1$ is the class corresponding to the vector $\vec{x}_k$. The constraint is added that $M||\vec{w}|| = 1$ to restrict the problem space from an infinite number of possible solutions to a single feasible solution, and consequently the problem becomes

$$\min_{\vec{w}} ||\vec{w}||^2$$  \hspace{1cm} (2.6)

subject to

$$\vec{y}_k D (\vec{x}_k) \geq 1 \forall k$$  \hspace{1cm} (2.7)

This problem is theoretically solvable, but it is computationally impractical when the dimensionality of $\vec{w}$ becomes very high. For this reason, the Lagrangian of the problem is taken and the dual space optimization problem is formed, as per work presented in [19][20][21].

The Lagrangian of the problem is

$$L (\vec{w}, b, \vec{\alpha}) = \frac{1}{2} ||\vec{w}||^2 - \sum_k \vec{\alpha}_k [\vec{y}_k D (\vec{x}_k) - 1]$$  \hspace{1cm} (2.8)

where $\vec{\alpha}_k$ are the Lagrange multipliers and together form a vector $\vec{\alpha}$. At the solution, the following condition holds

$$\frac{dL}{d\vec{w}} = \vec{w} - \sum_k \vec{\alpha}_k \vec{y}_k \phi (\vec{x}_k) = 0$$  \hspace{1cm} (2.9)

$$\vec{w} = \sum_k \vec{\alpha}_k \vec{y}_k \phi (\vec{x}_k)$$  \hspace{1cm} (2.10)
As per [2], the solution to this problem should be a maximum of \( L(\alpha, b, w) \) with respect to \( \alpha \). We reformulate the term in the above equation that depends on \( \alpha \).

\[
\sum_k \bar{\alpha}_k [\bar{y}_k D(x_k) - 1] = \sum_k \bar{\alpha}_k \bar{y}_k D(x_k) - \sum_k \bar{\alpha}_k
\]

(2.11)

\[
= \sum_k \bar{\alpha}_k \bar{y}_k (w \cdot \phi(x_k) + b) - \sum_k \bar{\alpha}_k
\]

(2.12)

We can substitute into the above equation to get an equation in terms of \( \alpha \). Additionally, we set the offset \( b \) to zero and solve for it later.

\[
\sum_k \bar{\alpha}_k \bar{y}_k (w \cdot \phi(x_k) + b) - \sum_k \bar{\alpha}_k = \sum_k \bar{\alpha}_k \left( \left( \sum_z \bar{\alpha}_z \bar{y}_z \phi(x_z) \right) \phi(x_k) \right) - \sum_k \bar{\alpha}_k
\]

(2.13)

By expanding vector operations we can rewrite the above

\[
\sum_k \bar{\alpha}_k \bar{y}_k \left( \sum_i \sum_z \bar{\alpha}_z \bar{y}_z \phi_i(x_z) \phi_i(x_k) \right) - \sum_k \bar{\alpha}_k
\]

(2.14)

\[
= \sum_k \bar{\alpha}_k \sum_i \sum_z \left[ \bar{y}_z \bar{y}_k \phi_i(x_z) \phi_i(x_k) \phi(x_k) \phi(x_k) \right] \bar{\alpha}_z - \sum_k \bar{\alpha}_k
\]

(2.15)

where \( \phi_i(x) \) is the \( i^{th} \) element in the vector \( \phi(x) \). Finally, we reformulate this into its final representation.

\[
\min_{\alpha} \left( f(\alpha) \right)
\]

(2.16)

\[
f(\alpha) = \frac{1}{2} \alpha Q \alpha - \sum_k \bar{\alpha}_k
\]

(2.17)

\[
\bar{w}_i = \sum_k \bar{\alpha}_k \phi_i(x_k)
\]

(2.18)

\( Q \) is the kernel matrix:

\[
Q_{ij} = \bar{y}_i \bar{y}_j K(x_i, x_j)
\]

(2.19)
The label factors are often omitted in this work because their product calculation is trivial and can be circumvented using flow control.

In this problem domain, values of a kernel function $K(x, y)$ are chosen such that

$$K(\vec{x}, \vec{y}) = \sum_i \phi_i(\vec{x}) \phi_i(\vec{y}) \quad (2.20)$$

With these definitions, the decision function can be rewritten in terms of $\vec{\alpha}$.

$$D(\vec{x}) = \sum_k \vec{\alpha}_k K(\vec{x}_k, \vec{x}) + b \quad (2.21)$$

Here, $\vec{x}_k$ is the $k^{th}$ support vector and $\vec{x}$ is the vector being classified.

**Sequential Minimal Optimization**

Platt et. al. [4] present a simple, efficient and effective technique for finding the optimal $\vec{\alpha}$ vector for Support Vector Machines that has become the basis for popular SVM implementations, called Sequential Minimal Optimization (SMO). The basic tenet of this approach is to break down the large optimization problem presented into the smallest possible optimization problem, that is solving for two elements of $\vec{\alpha}$, and repeating this optimization until the entire vector is optimized. In this way, the inner loop of the SMO algorithm is a single calculation, rather than an optimization problem itself. There are three phases in the SMO algorithm as described by Platt:

1. Check for Terminating Conditions

2. Select Working Set

3. Optimize Working Set

Also worth noting is that SMO actually attempts to solve a slightly modified version of $f(\vec{\alpha})$, introduced in [5]; it introduces a tolerance term that allows some adjustable number of training vectors to spill over their class boundary and into the opposing class space.
in cases when the training samples are not linearly separable.

\[
\min_{\vec{w}, b, \zeta} \frac{1}{2}||\vec{w}||^2 + C \sum_i \zeta_i 
\]  

(2.22)

This extra term leads to another constraint in the dual space.

\[
0 \leq \vec{\alpha}_k \leq C \forall k 
\]  

(2.23)

**Terminating Conditions**

The Karush-Kuhn-Tucker (KKT) [35] conditions for optimality dictate that \( \vec{\alpha} \) is an optimal solution of \( f(\vec{\alpha}) \) if and only if there exist values which satisfy

\[
\nabla f(\vec{\alpha}) + b\vec{y} = \vec{\lambda} - \vec{\zeta} 
\]  

(2.24)

\[
\vec{\lambda}_i \vec{\alpha}_i = 0, \vec{\zeta}_i (C - \vec{\alpha}_i) = 0, \vec{\lambda}_i \geq 0, \vec{\zeta}_i \geq 0, \forall i 
\]  

(2.25)

We can rewrite this equation.

\[
\nabla_i f(\vec{\alpha}) + b\vec{y}_i = \begin{cases} 
\geq 0 : & \vec{\alpha}_i < C, \\
\leq 0 : & \vec{\alpha}_i > 0 
\end{cases} 
\]  

(2.26)

Since \( y = \pm 1 \), we know that \( b \) is constrained within a range.

\[
\max_{i \in I_{up}} - \vec{y}_i \nabla_i f(\vec{\alpha}) \leq b \leq \min_{i \in I_{down}} - \vec{y}_i \nabla_i f(\vec{\alpha}) 
\]  

(2.27)

\[
I_{up}(\vec{\alpha}) = \{ t | \vec{\alpha}_t < C, \vec{y}_t = 1 \text{ or } \vec{\alpha}_t > 0, \vec{y}_t = -1 \} 
\]  

(2.28)

\[
I_{down}(\vec{\alpha}) = \{ t | \vec{\alpha}_t < C, \vec{y}_t = -1 \text{ or } \vec{\alpha}_t > 0, \vec{y}_t = 1 \} 
\]  

(2.29)

Once the two values are sufficiently close, \( b \) is calculated as their average, the KKT conditions are met and the algorithm stops. To track this condition, \( \nabla f(\vec{\alpha}) \) is initialized to
a vector of all ones and updated at each iteration according to

$$\nabla f (\tilde{\alpha})^{r+1} = \nabla f (\tilde{\alpha})^{r+1} + \Delta \tilde{\alpha}_i Q_i + \Delta \tilde{\alpha}_j Q_j$$ (2.30)

Here, $\Delta \tilde{\alpha}_i$ is the amount by which the selected $\alpha$ component has changed and $\tilde{Q}_i$ is a column of the kernel matrix. In this way, $\nabla f (\tilde{\alpha}) = \tilde{\alpha} Q - \tilde{e}$ is maintained. Here $\tilde{e}$ is a vector of all 1. It is worth noting here that this formula is a vector addition and can potentially be of very high degree; this makes this step in the operation a good candidate for parallelization, since vector additions contain a large number of simple, completely independent operations which all rely on data that exhibits spatial locality and is thus cache friendly (meaning that only the first set of accesses will suffer full communication penalty).

**Working Set Selection**

The next step in the algorithm is to select the indices of $\tilde{\alpha}$ to optimize in this iteration. Since the algorithm can finish when the KKT conditions are met, it makes sense to select the two values which most violate these conditions, and in fact this is the initial proposal by Platt. However, [6] presents an optimization which uses second order information to converge in fewer iterations.

As shown in equation 2.27, $b$ is bounded.

$$m (\tilde{\alpha}) \leq b \leq M (\tilde{\alpha})$$ (2.31)

$$m (\tilde{\alpha}) = \max_{i \in I_{up}} - \tilde{g}_i \nabla_i f (\tilde{\alpha})$$ (2.32)

$$M (\tilde{\alpha}) = \min_{i \in I_{down}} - \tilde{g}_i \nabla_i f (\tilde{\alpha})$$ (2.33)

We first select the index $i$ that defines $m (\tilde{\alpha})$.

$$i \in \arg \max_t \left\{ -\tilde{g}_i \nabla f (\tilde{\alpha}^r)_t | t \in I_{up}(\tilde{\alpha}^r) \right\}$$ (2.34)
Then we select a value of \( j \) which defines \( M(\vec{\alpha}) \); should this value exceed \( m(\vec{\alpha}) \), this is a violating pair which should be the next to be optimized.

\[
j \in \arg \min_t \left\{ -\vec{y}_t \nabla f(\vec{\alpha}_t^r) | t \in I_{low}(\vec{\alpha}_t^r) \right\}
\] (2.35)

As [6] points out, this is equivalent to trying to minimize a first order approximation of

\[
f(\vec{\alpha}_t^r + \vec{d}) \approx f(\vec{\alpha}_t^r) + \nabla f(\vec{\alpha}_t^r)^T \vec{d}
\] (2.36)

where \( \vec{d} \) is the change in \( \vec{\alpha} \). The optimization proposed in [6] is to make this minimization of a second order approximation.

\[
f(\vec{\alpha}_t^r + \vec{d}) - f(\vec{\alpha}_t^r) = \nabla f(\vec{\alpha}_t^r)^T \vec{d} + \frac{1}{2} \vec{d}^T \nabla^2 f(\vec{\alpha}_t^r) \vec{d}
\] (2.37)

Therefore, we select \( i \) as above, and then select \( j \) to minimize this function.

\[
j \in \arg \min_t \left\{ \text{Sub}(\{i,t\}) | t \in I_{down}(\vec{\alpha}_t^r), -\vec{y}_t \nabla f(\vec{\alpha}_t^r)_t < -\vec{y}_i \nabla f(\vec{\alpha}_t^r)_i \right\}
\] (2.38)

\[
\text{Sub}(B) = \min_{\vec{d}} \nabla f(\vec{\alpha}_t^r)^T \vec{d} + \frac{1}{2} \vec{d}^T \nabla^2 f(\vec{\alpha}_t^r) \vec{d}
\] (2.39)

Theorem 3 in [6] states that, as long as \( \{i,j\} \) form a violating pair, this function has an optimal value at

\[
-\left( -\vec{y}_i \nabla f(\vec{\alpha}_t^r)_i + \vec{y}_j \nabla f(\vec{\alpha}_t^r)_j \right)^2
\]

\[
2 \left( K_{ii} + K_{jj} - 2K_{ij} \right)
\] (2.40)

Therefore, we select the working set as follows.

\[
j \in \arg \min_t \left\{ -\frac{b_{ij}^2}{a_{ij}} | t \in I_{down}(\vec{\alpha}_t^r), -\vec{y}_t \nabla f(\vec{\alpha}_t^r)_t < -\vec{y}_i \nabla f(\vec{\alpha}_t^r)_i \right\}
\] (2.41)

\[
a_{ij} = K(\vec{x}_i, \vec{x}_j) + K(\vec{x}_j, \vec{x}_j) - 2K(\vec{x}_i, \vec{x}_j)
\] (2.42)
\[ a_{ij}^* = \begin{cases} a_{ij} & \text{if } a_{ij} > 0 \\ \tau & \text{otherwise} \end{cases} \quad (2.43) \]

\[ b_{ij} = -\nabla_i f(\vec{\alpha}^k) + \nabla_j f(\vec{\alpha}^k) \quad (2.44) \]

Note that in this kind of minimization problem, with large computations being optimized over a single value, we can parallelize this by computing the function for all values of \( t \) first, and then simply traversing the resulting data linearly (and quickly) to find the smallest value. This can further be parallelized by doing a tree based reduction of the data to find the minimum; that is, break the data into sets of two elements and find the maximum of each set simultaneously, then repeat on the remaining data set until the maximum is found. This allows all of the computational elements to be applied to the operation and reduces the running time logarithmically.

**Minimal Optimization**

Finally, to optimize the selected working set, the previous optimization problem is approached, but is constrained to only two variables.

\[
\min_{\vec{\alpha}_i, \vec{\alpha}_j} \frac{1}{2} \begin{bmatrix} \vec{\alpha}_i & \vec{\alpha}_j \end{bmatrix} \begin{bmatrix} Q_{ii} & Q_{ij} \\ Q_{ji} & Q_{jj} \end{bmatrix} \begin{bmatrix} \vec{\alpha}_i \\ \vec{\alpha}_j \end{bmatrix} + (Q_i \vec{\alpha} + 1) \vec{\alpha}_i + (Q_j \vec{\alpha} + 1) \vec{\alpha}_j \quad (2.45)
\]

From this equation, the derivative is extracted and set to zero, then the components can be solved. Here, \( N \) represents

\[
\vec{\alpha}_i^{\text{new}} = \vec{\alpha}_i^k + \vec{\gamma}_i \frac{b_{ij}}{a_{ij}^*} \quad (2.46)
\]

\[
\vec{\alpha}_j^{\text{new}} = \vec{\alpha}_j^k - \vec{\gamma}_j \frac{b_{ij}}{a_{ij}^*} \quad (2.47)
\]

Following this solution, the new values of \( \vec{\alpha} \) are then constrained such that

\[
0 \leq \vec{\alpha}_i \leq C_i \quad \forall i \quad (2.48)
\]
and
\[\tilde{\alpha}_i^{r+1} - \tilde{\alpha}_j^{r+1} = \tilde{\alpha}_i^r - \tilde{\alpha}_j^r\] (2.49)

To do this, if one of the new components of \(\tilde{\alpha}\) falls outside of this range, it is clipped (that is, constrained to the bound it violates) and the other component is adjusted to maintain that the difference between the components has not changed. For instance, if \(\tilde{\alpha}_i^r - \tilde{\alpha}_j^r > C_i - C_j\) and \(\tilde{\alpha}_i^{r+1} > C_i\), we set \(\tilde{\alpha}_i^{r+1} = C_i, \tilde{\alpha}_j^{r+1} = C_i - (\tilde{\alpha}_i^r - \tilde{\alpha}_j^r)\).

In this phase, as in other phases, the kernel matrix \(Q\) needs to be calculated by performing a high dimensional matrix vector multiplication operation. These are highly parallel operations, since they require a large number of inner products to be calculated (two element multiplications en masse followed by reductive additions). Furthermore, for reasons discussed later, calculating the kernel matrix requires an extra phase of operation that is entirely element-wise.

### 2.1 SVM Applications in Computer Vision

Support Vector Machines have been applied successfully to a number of machine learning problems. [10] uses both polynomial and radial basis function kernel SVMs to recognize handwritten text in images; here, SVM is chosen in part because the data used is linearly separable and sparse but comes in large quantities of training vectors. An implementation called svmlight is used and the two kernels used achieve 86 and 86.4 percent accuracy respectively. In [11], a soft margin version of SVM is also used in text classification and compared to a number of neural network based approaches. The work done in [39] uses directional element features to apply SVMs to classification of Chinese characters, along with a coarse-to-fine approach. They achieve 100% classification accuracy for certain data sets.
SVMs are commonly used for face detection [26][25] and other human feature detection [24][22][23]. The work done in [12] applies a custom method for performing SVM training to the task of face detection in images. They achieve a correct detection rate of 97 percent for certain test data sets.

Sharma and Savakis, in [13], use LIBSVM, a common implementation of SVM, to classify images using a descriptor based approach. Their system uses a Histogram of Oriented Gradients (HOG) descriptor to extract significant features from eye images, and then performs Principal Component Analysis (PCA) on the resulting HOG descriptors. The resultant data is used to train an SVM model, and then this model is used to classify eye images with accuracies exceeding 98 percent. This illustrates the effectiveness and flexibility of an SVM-based approach.

In [37], SVMs are applied to computer aided diagnosis. Specifically, the work uses a custom formulation of an SVM in order to detect microcalcifications in mammogram images. The custom formulation is called a tangent vector SVM (TV-SVM) meant to introduce rotation invariance, since microcalcification clusters do not vary in their properties with their orientation.

In [38], a multiresolution approach is taken to classify hyperspectral images. They perform wavelet decomposition on the images being tested, then perform classification on the lower resolution images first. This allows them to eliminate images before much effort is invested in their classification, reducing the computational effort of SVM classification.

An interesting application of SVM is [40]; images of olive oil samples are taken in multiple color spaces, used to form histograms and converted to a lower dimensionality using PCA. Then the resultant description of each image sample is run through an SVM to determine its impurity level. This is a multiclass SVM problem, as samples were grouped as “low impurity level”, “acceptable impurity level” or “unacceptable impurity level”; using a polynomial kernel, they were able achieve 87.66% accuracy.
<table>
<thead>
<tr>
<th>SVM Type</th>
<th>Formula</th>
</tr>
</thead>
<tbody>
<tr>
<td>C-Support Vector Classif</td>
<td>( \frac{1}{2} \bar{\alpha} Q \bar{\alpha} - \sum_k \bar{\alpha}_k )</td>
</tr>
<tr>
<td>( \nu )-Support Vector Classif</td>
<td>( \frac{1}{2} \bar{\alpha} Q \bar{\alpha} )</td>
</tr>
<tr>
<td>One-Class SVM</td>
<td>( \frac{1}{2} (\bar{\alpha} - \bar{\alpha}^<em>) Q (\bar{\alpha} - \bar{\alpha}^</em>) - \nu \sum_i (\bar{\alpha}_i + \bar{\alpha}_i^<em>) + \sum_i z_i (\bar{\alpha}_i - \bar{\alpha}_i^</em>) )</td>
</tr>
<tr>
<td>( \epsilon )-Support Vector Regression</td>
<td>( \frac{1}{2} (\bar{\alpha} - \bar{\alpha}^<em>) Q (\bar{\alpha} - \bar{\alpha}^</em>) + \bar{z}^T (\alpha - \alpha^*) )</td>
</tr>
</tbody>
</table>

**Figure 2.2:** SVM Formulations Offered by LIBSVM

<table>
<thead>
<tr>
<th>Kernel Name</th>
<th>( K(\bar{x}, \bar{y}) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linear</td>
<td>( \bar{x} \cdot \bar{y} )</td>
</tr>
<tr>
<td>Polynomial</td>
<td>( (\bar{x} \cdot \bar{y} + c)^d )</td>
</tr>
<tr>
<td>RBF</td>
<td>( e^{-\gamma</td>
</tr>
<tr>
<td>Sigmoid</td>
<td>( \tanh (\bar{x} \cdot \bar{y} + c) )</td>
</tr>
</tbody>
</table>

**Figure 2.3:** SVM Kernels Offered by LIBSVM

### 2.2 LIBSVM

The central SVM engine used in this work is LIBSVM, introduced in [3]. It is a powerful, open source, widely available and thoroughly vetted system that offers very thorough support for SVM. It is popular among research groups because, being open source and reliably hosted, it is easy to access and use LIBSVM and thus gain the benefits of SVM without the need to understand its inner workings. Furthermore, LIBSVM’s feature set is broad enough, and its interface flexible enough, that it can be easily applied to a broad range of applications. Additionally, because LIBSVM is open source, it is a popular platform for experimenting with enhancements to SVM algorithms.

LIBSVM supports five basic configurations of SVM, both for training and classification, listed by the table in Figure 2.2. In addition, LIBSVM supports the use of four kernel functions, defined in Figure 2.3. LIBSVM also supports a precomputed kernel function, allowing the user to supply kernel values within the training data.

LIBSVM has three primary external interfaces; there is an application programming interface (API), written in C/C++, that exposes a flexible set of SVM primitives that are easy to integrate into a larger system. There is a command line interface that is made up of a set
of command line tools for training and classification built around the C/C++ API. This is likely the easiest interface to use as it requires no programming, and training and classification tools both produce and consume data files which are easy to interpret and parse using the C/C++ API in a custom application. Finally, there is a MATLAB interface, also built around the C/C++ interface, that makes the basic API available in MATLAB through mex files, ensuring they are as performant as possible while maintaining a consistent interface with the other approaches. This is an important addition as MATLAB is often an optimal programming environment for research groups, especially those where extensive programming experience is not common, or when speed of development is of great importance.

LIBSVM makes use of an SMO type algorithm, as introduced in the previous section, including the discussed optimization to the working set selection step. However, the library adds some optimizations to reduce the computational load of the approach. In particular, the library uses a shrinking technique, introduced in [15] and discussed at length in [14]. The general premise of this approach is that if certain components of the vector $\alpha$ are not changed over a large number of iterations, they are unlikely to be changed at any point in future iterations of the SMO algorithm; consequently, if certain indices can be proved to be stable they can be removed from consideration for the working set selection step, which can reduce the computational effort of the entire algorithm.

Work done in [16] and [14] show formal proofs that, under certain working set selection algorithms, during the final iterations of the SMO algorithm only a small subset of the vector $\alpha$ is updated. Another logical consequence of this is that only a small subset of the kernel matrix $Q$ is used during these iterations; this reduced set is likely small enough to fit into the system’s memory, and so LIBSVM employs a caching technique that saves values of $Q_{ij}$ for future use.

For clarity, 2.4 shows an overview of the SMO algorithm is implemented in the LIBSVM library.
2.3 GPU Assisted LIBSVM

This thesis will be influenced by the work done in [8], which is a GPU accelerated SVM implementation using the CUDA framework (GPU LIBSVM). It is based on LIBSVM and is an extension on the existing code base, slightly stripped down for focus, implementing only the SVM core and the SVM training command line module. This system uses a GPU to accelerate the calculation of the radial basis function kernel during the cross validation phase of training. Cross validation is a technique used to heuristically find the optimal value for $C$, the factor that dictates leniency in the constraint that all training samples be cleanly separated by the SVM’s characteristic hyperplane. The work done in [8] also introduces a new internal data format for the training vectors used by LIBSVM that is more conducive to intersystem communication; the work presented here uses this format.

There are two limitations in the work in [8] that this work attempts to overcome; the first is that its domain is narrow, accelerating only one kernel (the radial basis function kernel) and affecting only training runs that use cross validation. The second is that [8] is limited
to CUDA systems; this is considered a weakness here because it potentially eliminates a wide range of computationally powerful devices from being utilized.

The work presented in [8] is based on cuSVM, introduced in [7], itself a GPU-based, accelerated version of SVM-based loosely on LIBSVM. CuSVM uses the same general algorithms as LIBSVM, but is largely rewritten in C/C++, using CUDA, designed to interface with MATLAB. Because of the nature of the rewrite, cuSVM is only accessible using MATLAB and thus loses a large portion of its utility and accessibility. Furthermore, it is also narrowly focused on the radial basis function kernel.
Chapter 3

OpenCL

This thesis will make use of OpenCL, the Open Compute Language, which is a collection of libraries that together form a platform for Heterogeneous Computing System design. OpenCL allows any C/C++ program to interface with parallel computation devices, including GPUs, FPGAs, parallel CPUs and even specific accelerators such as the IBM CELL Blade [9]. OpenCL is supported by GPU devices that support the CUDA framework, as well as a number of AMD and ATI GPUs and AMD and Intel CPUs, and even some ARM CPUs, such as some of those provided by Qualcomm [41].

OpenCL primarily supports a data parallel programming model, wherein the same computation is applied to several memory elements in parallel. The main element of computation is referred to in OpenCL as the kernel (not to be confused with the SVM kernel function), and its code is compiled using the OpenCL compiler in a subset of C99 which is extended to allow for certain vector operations. The language also provides certain C standard library operations such as exponential calculation and some trigonometry functions. Kernels are executed in local work groups, which provide the perception of being executed in parallel all at once, thus allowing synchronization between threads; a single work group is therefore ideal for parts of the kernel requiring synchronization.

Local work groups are joined together to form a global work group which defines the total number of work items that need to be executed to complete a parallel task. Global work
groups are then synchronized on the host side; this allows the programmer to view the system as a familiar, serial programming environment that makes use of an asynchronous API to dispatch parallel tasks to devices which are better suited to them.

OpenCL views the platform on which it runs as a host device connected to a number of parallel devices, which can be viewed as either Single Instruction Multiple Data (SIMD) or Single Program Multiple Data (SPMD) devices, as depicted by 3.1 (image taken from [9]). The host is typically the system’s primary CPU, and the parallel devices, here called Compute Units (CU), can include GPUs, FPGAs, other CPUs and any device that is OpenCL compliant. All of the devices provide a consistent, abstracted interface to the programmer and the host device. CUs in OpenCL are assigned a command queue by the host device, after which point the host can place commands, such as kernel executions or data transfers, into the queue asynchronously and they get assigned to the appropriate device and will be handled by the OpenCL runtime.

Due to the nature of its platform model, OpenCL is very flexible with regard to what types of devices can be used; even more importantly, OpenCL provides the capacity for the host program to query the available hardware at run time and get lots of detailed information on the available hardware. This means that even a single application can be written in such a way that it is extensible and can take advantage of any system on which it is executed; for example, an OpenCL application can query for available accelerator devices and
in their absence can search for GPU devices, select the one which best meets the program’s needs and run its kernels on that device.

Furthermore, because OpenCL allows for dynamic (run-time) compilation of the program’s kernels, and because hardware vendors are responsible for providing their own support for the OpenCL specification and runtime, OpenCL kernels can be both extremely performant and extremely portable. For instance, OpenCL supports the use of CPU devices, even as far as allowing the host CPU to be viewed as a CU; in this context, the OpenCL compiler can compile the program’s kernels to make use of CPU SIMD extensions and provide the maximum performance from the device, but because the underlying libraries are allowed to change between differing hardware platforms, this optimization does not tie the application to a particular CPU the same way that using Intel’s Math Kernel Library (or even SSE instructions) might.

For this same reason, OpenCL is a good choice of platform even when the designer only intends to use GPU devices. While using CUDA may tie the accelerated application specifically to NVIDIA’s line of GPUs, OpenCL is supported by NVIDIA on CUDA cards as well as competing GPU devices.

Additionally, OpenCL offers the programmer a variable level of control in the optimization of an application. For instance, OpenCL allows the programmer to schedule communication explicitly, by making manual (blocking or non-blocking) writes into device memory from host memory; alternatively, the programmer can specify a section of memory to be home to some necessary data, and allow the runtime to determine when it should be written. Implicit communication can lead to better performance at less effort, as the runtime is likely (in the general case) to have better insight into the use and status of the system’s communication channels. However, the explicit communication can have two advantages; first, it allows the developer tighter control over when it is safe to write or free host memory, which can be important in memory constrained applications. Secondly, by performing explicit, asynchronous communication, the developer can sometimes better
overlap communication and computation efforts to reduce their cost.

Also, OpenCL allows the developer to choose how a group of tasks is broken into local work groups, or leave this decision to the run time. Some operations can only be done efficiently if the programmer can say with absolute certainty how many threads are operating in a specific work group, and so setting this explicitly can allow greater performance. However, only the runtime knows with certainty how many computational cores may be present (or available) at execution time, and how best to schedule resources.

**OpenCL Applications**

In [27], OpenCL is used to apply a multi-GPU system to accelerate backprojection for image reconstruction. OpenCL is chosen for this work because of its independence from hardware, vendor and platform related constraints. They use this system to accelerate the FDK algorithm for backprojection, achieving a speedup of over 400 for a single GPU. [32] performs a similar acceleration, but also duplicates the work in CUDA so that the two frameworks can be compared for performing the same task on the same hardware. They find that CUDA offers a small advantage in performance, however, we contend that this advantage is not significant enough to warrant adherence to a single hardware vendor. In [28], OpenCL is integrated with the Hadoop language to perform large scale calculations on a distributed system using the MapReduce approach. There, OpenCL is chosen because it greatly simplifies the programming of diverse hardware.

In [29], OpenCL is used to accelerate a neural network implementation which is then applied to modeling and simulating the behavior of complex chemical structures. [30] investigates the use of OpenCL in weather forecasting systems; here, OpenCL is chosen because it is open (non-proprietary) and cross platform (available on OS X, Linux, Windows, etc.). This work also integrates OpenCL capabilities into existing FORTRAN applications to make it more useful in scientific applications.

An interesting application of OpenCL is shown in [31], where it is used with an Altera FPGA to accelerate binomial option pricing at minimal power consumption. OpenCL
was chosen for this application because it allowed researchers to reap the full benefits of FPGA acceleration without having experience in hardware design (a field that differs vastly from software development and requires years of experience to achieve expertise). In [33], OpenCL is applied to document similarity analysis, and compared to Java, C, and CUDA C implementations of the same algorithm. Additionally, because of OpenCL’s portability, an AMD GPU is compared to a NVIDIA GPU for this work.
Chapter 4

SVM Acceleration

4.1 Acceleration

4.1.1 Overview

In this work, OpenCL is used to accelerate the algorithms described; this setup focuses on using a GPGPU configuration for acceleration. OpenCL is specifically chosen because of its flexibility and extensibility, and can be used on any major GPU, including integrated GPUs such as Intel HD hardware. In addition, OpenCL can be used for any future platforms to which the library is ported, as the code written for OpenCL is platform agnostic.

Kernel Matrix Calculation

The primary means by which SVM is accelerated is by parallelizing the Kernel Matrix (KM) calculation.

\[
Q_{ij} = \vec{y}_i \vec{y}_j K(\vec{x}_i, \vec{x}_j) \tag{4.1}
\]

For the purposes of this section, we disregard the values \(\vec{y}_i \vec{y}_j\); since \(\vec{y}_i = \pm 1\), calculating \(\vec{y}_i K\) is trivial and often unnecessary.

\[
Q_{ij} = K(\vec{x}_i, \vec{x}_j) \tag{4.2}
\]
Let us define a more general matrix $Q$ such that

$$Q = \begin{bmatrix}
Q_{00} & \cdots & Q_{0p} \\
\vdots & \ddots & \vdots \\
Q_{p0} & \cdots & Q_{pp}
\end{bmatrix} \quad (4.3)$$

And a single row of this matrix

$$Q_i = [Q_{i0}, \ldots, Q_{ip}] \quad (4.4)$$

Let us also define a more general kernel function which generates an entire row of $Q$.

$$K(X, \vec{x}_i) = Q_i^T \quad (4.5)$$

$$X = \begin{bmatrix}
\vec{x}_1 \\
\vec{x}_2 \\
\vdots \\
\vec{x}_p
\end{bmatrix} \quad (4.6)$$

The kernel functions are selected such that

$$K(\vec{x}_i, \vec{x}_j) = h(\vec{x}_i \vec{x}_j^T) \quad (4.7)$$

Consequently

$$K(X, \vec{x}_j) = h(X \vec{x}_j^T) \quad (4.8)$$

Notice that, regardless of the function $h(\cdot)$, the value $X \vec{x}_j^T$ must be calculated when computing the value of $Q_j$, and that the cost of this computation grows both with the number of training vectors and with their dimensionality. For this section, let us define $N$ to be the dimensionality of the training vectors and $M$ to be the number of training vectors. We also define $p$ to be the available number of processing elements.
Due to the objective function updating stage, it is necessary to calculate at least two rows of $Q$ during each iteration. We do this by first calculating $X \vec{a}_i^T$ and then applying $h(\cdot)$ to each element in the result; therefore we first consider the computational stages and cost of $X \vec{a}_i^T$. First, consider that

$$X \vec{a}_i^T = \begin{bmatrix} \vec{x}_1 \\ \vdots \\ \vec{x}_M \end{bmatrix} \vec{a}_i^T = \begin{bmatrix} \vec{x}_1 \cdot \vec{a}_i \\ \vdots \\ \vec{x}_M \cdot \vec{a}_i \end{bmatrix}$$

Also, that

$$\vec{x}_i \cdot \vec{x}_j = \sum_{k=0}^{N-1} \vec{x}_i[k] \vec{x}_j[k]$$

Each dot product requires $N$ multiplications and $N$ additions, and there are $M$ dot products calculated for each row of $Q$. Thus, calculating $Q_i$ has complexity $O(NM)$.

Assuming that a single dot product can be calculated on a single processing element in a larger parallel system at no penalty, the cost of the calculation can be reduced linearly with respect to $p$, resulting in complexity $O \left( N \left( \frac{M}{p} \right) \right)$.

An individual dot product can be computed using an element wise multiplication stage followed by a reduction (addition). This changes the time complexity of a single dot product to $O \left( N \log_2(N) \right)$; this is an increase in the sequential case, but by performing the multiplications and the reduction in parallel, the complexity can be reduced to $O \left( \frac{N \log_2(N)}{p} \right)$. Note that even as $p$ tends toward $N$ the time complexity of a single dot product reaches a lower bound of $O \left( \log_2(N) \right)$.

It follows then that the matrix-vector multiplication can be reduced, when parallelized to a complexity of $O \left( \frac{MN \log_2(N)}{p} \right)$. The assumption mentioned previously is, of course, an oversimplification that ignores two basic obstacles in the parallelization of this algorithm; first, there is some amount of overhead, both in coordination and communication that grows with the amount of computation being done. The second limitation to this assumption is
that in many parallel systems used, such as the GPUs used in this thesis, an individual core in the highly parallel computational element does not contain equivalent processing power to an individual core in the serial computational element being replaced. As such, it can not be expected that \( p \) processing cores in a computing environment such as a GPU will do a single task \( p \) times faster than a CPU core.

In an OpenCL environment, libraries exist which perform linear algebra tasks, including matrix-vector multiplication, in an OpenCL environment. A custom algorithm is presented alongside this solution for reasons discussed in a later section. The custom algorithm performs a two-phase reduction for each dot product; first, \( \frac{N}{P} \) multiplications and \( \frac{N}{P} \) additions are performed on each of \( p = P \) processing elements. Following this, additive reduction is performed in \( \log_2(P) \) stages. Figures 4.1, 4.2 and 4.3 illustrate this basic approach.

The value of \( P \) is chosen to satisfy two criteria: \( P = 2^K \) for some integer \( K \), and \( P \) is close to the number of processing elements available. The first criterion ensures that the second phase (binary phase) of the reduction is possible, while the second one ensures that the first phase of the reduction executes as quickly as possible by exploiting the maximum level of parallelism. For many available platforms, both criteria can be satisfied because the total number of processing elements is a power of 2, or the preferred number of simultaneous threads is a power of 2.

Because heterogeneous systems often have a large communication penalty, and the matrix-vector multiplication requires using all of the training data available to the SVM, it is worth considering measures to mitigate this cost. For this reason, a cache is maintained which keeps \( X \) data stored on the device where the calculation is being performed. LIBSVM makes use of a shrinking working set and frequent reordering, so each swap in the main algorithm is mirrored on the OpenCL device used to perform KM calculation in order to keep the cache valid. This trades the communication cost for the training vectors with the cost of executing the swapping kernel on a regular basis, which in practice is found to be a worthwhile exchange despite the overhead associated with small kernels in OpenCL.
Figure 4.1: Illustration of Matrix-Vector Multiplication

Figure 4.2: Parallelization of Matrix-Vector Multiplication. Colors indicate division of labor amongst work groups

Figure 4.3: Illustration of Dot Product Parallelization. Colors indicate division of labor amongst threads.
This kernel calculation is used both in the training and classification phases of the SVM, and thus is central to the overall, accelerated system. However, it makes up a much larger percentage of the training phase than it does the classification phase, and so, given that total acceleration will hinge on this operation, we expect to see that training is accelerated to a greater degree than classification, as per Amdahl’s law.

**Custom Work**

One of the many benefits of OpenCL is its platform independence and flexibility; through the use of its powerful APIs and supporting abstraction layers, OpenCL applications can be made extremely portable while still exploiting the benefits of high performance devices available in a heterogeneous system. Furthermore, third party library support offers a range of functionality that is easy to use and thoroughly debugged and often doesn’t require the user to write any OpenCL kernels. This can drastically reduce development time for large applications.

However, this flexibility could come with a performance penalty. Specifically, third party libraries often have to remain very general, offering a broad range of capabilities; this typically means that the capabilities offered are not highly optimized for a specific problem and a performance gain could be achieved through a fully custom implementation. An opportunity is presented to trade development time for improved performance.

In this work, both approaches are considered. In one approach, a custom set of OpenCL kernels are written and tuned to achieve the maximum performance for the specific problem being approached. This involves a greater effort, but it is expected to provide a worthwhile improvement in performance. In the second approach, the OpenCL AMD BLAS library is used to perform the calculations used in the SMO algorithm and a minimal amount of custom OpenCL code is used. In this way, one can analyze the tradeoff between development time and system performance.
4.1.2 Training

Solving Single Step

The main training algorithm, SMO, is an iterative approach and that the central computation in each iteration is solving for two values of $\alpha$ with respect to each other. This solution requires calculations of $K(\vec{x}_i, \vec{x}_j)$, $K(\vec{x}_j, \vec{x}_j)$ and $K(\vec{x}_i, \vec{x}_j)$, here denoted $Q_{ii}$, $Q_{jj}$ and $Q_{ij}$ in reference to the $Q$ matrix. However, because of the values used in the next step, the work presented calculates two entire columns of the $Q$ matrix instead of just three values, here denoted $Q_i$ and $Q_j$.

Due to this requirement, the basic function of this implementation is a modified kernel function which performs $K(X, \vec{x}_i)$. While it is the case that a single OpenCL kernel could be used to calculate $X\vec{x}_i^T$, and this kernel could be shared by all of the kernel functions followed by a smaller OpenCL kernel which calculates $h(\cdot)$, this approach is not taken. Each kernel function is implemented by a single OpenCL kernel which performs $X\vec{x}_i^T$ and then $h(\cdot)$ to avoid the overhead of executing two kernels.

During calculation of a single column of the kernel matrix, the steps shown in Algorithm 1 are performed on the host device. The corresponding CU device code is shown by Algorithm 2; this is the same operation illustrated in the previous figures.

Once the kernel matrix column is calculated, it is cached on the host side. This is a significant optimization, since each iteration of the training algorithm uses the calculation multiple times. The last step in solving a single pair of values is a calculation which is left to the host, since the host processor is best suited to it, as it involves a small number of expensive operations and some amount of flow control.

Objective Function Updating

The objective function which tracks the gradient of the function being optimized is used for the stopping conditions

$$g(\vec{\alpha}^r) = Q\vec{\alpha}^r - \vec{e}$$

(4.9)
Algorithm 1 Q Matrix Column Calculation

if X is cached on the device then
    store its buffer in X_BUFFER
else
    create OpenCL buffer X_BUFFER
    enqueue command to write X to X_BUFFER
end if

if x_i is cached on the device then
    store its buffer in X_I_BUFFER
else
    create OpenCL buffer X_I_BUFFER
    enqueue command to write x_i to X_I_BUFFER
end if

wait for all write commands to finish
enqueue command to run kernel function kernel on device, operating on X_BUFFER and X_I_BUFFER
wait for kernel function to finish
enqueue command to read Q_i from device to host
wait for read command to finish

Algorithm 2 CU Device Code For Calculating Kernel Matrix Column

1: store row number in R
2: store local work group element number in T
3: store number of columns in COLS
4: store size of local work group in WG
5: sum := 0.0
6: startIndex := T
7: while startIndex < COLS do
8:     sum := sum + A[ R * COLS + startIndex ] * x[ startIndex ]
9:     startIndex := startIndex + WG
10: end while
11: scratch[ T ] := sum
12: local work group barrier
13: offset := WG / 2
14: while offset > 0 do
15:     if T < offset then
17:     end if
18:     local work group barrier
19:     offset := offset / 2
20: end while
21: if T = 0 then
22:     y[ R ] = h( scratch[ 0 ] )
23: end if
This function is updated at each iteration.

\[ \bar{g}^{r+1} = \bar{g} + (\bar{\alpha}_i^{r+1} - \bar{\alpha}_i^r) \bar{y}_i Q_i + (\bar{\alpha}_j^{r+1} - \bar{\alpha}_j^r) \bar{y}_j Q_j \] (4.10)

The value of \( Q_i, Q_j \) is cached prior to this operation, but the update is still a vector addition operation on a vector of length \( M \) with a complexity of \( O(M) \). It is worth noting that each element of \( g^{r+1} \) requires two double-precision floating point multiplications and three double-precision floating point additions (the difference in \( \alpha \) values need only be computed once), so its \( O(\cdot) \) representation belies its true computational cost.

This is another point where both a custom implementation and a solution based on a prefabricated library are presented. The custom implementation is extremely straightforward — it simply performs the five operations in parallel for every index of \( g \). However, this presents an issue: the value of the function is required at certain points in the sequential portion of LIBSVM, so the value needs to stay synchronized between the device where it is computed and the primary CPU. For this reason, two configurations for this algorithm are considered.

First, the calculation is performed on the available GPU, for varying values of \( M \), then the same algorithm is moved to a parallel view of the CPU, using OpenCL to generate parallel extension instructions in a platform agnostic manner, and again observed for varying values of \( M \). Both of these are compared in terms of performance with a closely measured sequential version (found in the unaltered LIBSVM code).

**Working Set Selection**

The working set is selected as follows

\[ i \in \arg \max_t \left\{ -\bar{y}_t \nabla f(\bar{\alpha}^r)_t \mid t \in I_{up}(\bar{\alpha}^r) \right\} \]

\[ j \in \arg \min_t \left\{ -\frac{b^2_i}{a_{ij}} \mid t \in I_{down}(\bar{\alpha}^r), -\bar{y}_t \nabla f(\bar{\alpha}^r)_t < -\bar{y}_i \nabla f(\bar{\alpha}^r)_i \right\} \]
where \( i \) is selected first, which amounts to iterating over the objective value searching for a maximum under a few prerequisite conditions. This is an \( O(M) \) operation, and can be reduced through a parallel reduction to \( O\left(\frac{M\log_2(M)}{p}\right) \). Selection of \( j \) follows sequentially after the selection of \( i \), because it depends on \( i \), and sees the same speed up from \( O(M) \) to \( O\left(\frac{M\log_2(M)}{p}\right) \).

For reference, 4.4 is reproduce here; in addition, the calculation of a row of the Kernel Matrix is expanded into 4.5. This is the phase of the algorithm that has been most significantly altered in order to accelerate LIBSVM; 4.6 shows a high level view of the alteration made. The scalar kernel function calculation is omitted due to its simplicity, however, the parallel kernel function computation algorithm is shown in 4.7.

### 4.1.3 Classification

The classification phase of the SVM system is a much simpler algorithm than the training phase. Specifically, the LIBSVM implementation of SVM reads in a model file
Figure 4.5: Kernel Matrix Row Calculation Flowchart

Figure 4.6: Kernel Matrix Row Calculation Flowchart
Figure 4.7: Kernel Matrix Calculation Flowchart
and the vectors to be classified, and evaluates the decision function.

\[ D(\vec{x}) = \sum_i \vec{\alpha}_i K(\vec{x}_i, \vec{x}) + b \]

Here, \(\vec{x}_i\) are the support vectors and \(\vec{x}\) is the vector being classified. For the sake of efficiency and code reuse, this function's representation is slightly modified.

\[ D(\vec{x}) = \vec{\alpha} K(X, \vec{x}) + b \]

\[ K(X, \vec{x}) = \begin{bmatrix} K(\vec{x}_0, \vec{x}) \\ K(\vec{x}_1, \vec{x}) \\ \vdots \end{bmatrix} \]

In this form, the parallel kernel function code previously described can be reused for this phase. Following this, a new OpenCL kernel is used to perform the inner product of \(\vec{\alpha}\) and \(K(X, \vec{x})\) using parallel reduction as described previously.

### 4.2 Comparisons

#### 4.2.1 Overview

In this work, two comparisons are made between the presented version of LIBSVM and the reference version. First, the performance is compared in terms of execution time. This is important as speedup is the main advantage OpenCL implementation claims to offer over the existing library. Secondly, the difference in output between the two versions is measured, because a performance improvement at the cost of accuracy is unacceptable in many cases. We refer to the error as the amount that the HSA output differs from the serial implementation output.
4.2.2 Timing

High performance and serial implementations are both timed using native, OS supported timing tools such as the time.h library for UNIX-based systems and the performance counter under Windows. Processing time, i.e., the time taken to execute a kernel, is considered separately from communication time and setup time for OpenCL implementations to provide a thorough, fair comparison; this lends insight into which situations are well suited to the methods presented here.

The contributed SVM algorithm is compared to the existing LIBSVM executable for training and classification. The operation is timed at two levels; first, the entire time taken to execute the standalone binaries are measured using the time module inside Python. This is important, because the end user of the system will be primarily concerned with this time and the resultant speedup. Additionally, performance is measured and observed for each stage of the algorithm individually, including OpenCL kernel execution time, communication time and compilation time. This provides a more thorough understanding of the nature and causes of any resultant speedup and will highlight the benefits and drawbacks of the OpenCL system.

4.2.3 Error

Two comparisons are made between the output of the high-performance SVM implementation and the existing LIBSVM implementation. First, the SVM model file produced by the two executables will be compared; this is the most important comparison, because the two files being identical ensures that the new implementation has made the same decisions as the reference implementation, as the same supporting vectors were selected with the same final objective function.

This check is important because, while OpenCL provides guarantees about its error for single-precision floating point operations, LIBSVM requires the use of double-precision floating point operations to maintain accuracy. OpenCL provides no guarantees about
the availability, implementation or accuracy of double-precision floating point operations, therefore some calculations may present an error with respect to the reference calculations due to floating point inconsistencies, but this error may be within tolerance for certain datasets and applications. In the next chapter, we describe the specific problems to which our accelerated implementation are applied, and we present the resulting data.
Chapter 5

Results

5.1 Performance

Hardware

All tests for this section are performed using the same desktop computer work station. It contains an Intel i5 quad core processor operating at 3 GHz used to run the reference implementation of LIBSVM; LIBSVM is neither written, nor adapted in this set of tests, to run on multiple cores of a CPU and therefore all serial measurements are making use of a single processor core. However, this device supports OpenCL, meaning that OpenCL kernels can be compiled and run on the CPU. OpenCL views all computational devices as either SIMD or SPMD devices, and its CPU compilers are written to exploit the extensions available on varying CPU devices, such as SSE on Intel and AMD devices.

The work station also contains a NVIDIA GeForce GTX 480 graphics card which is utilized for these tests; there is an additional graphics card which handles the work station’s display, meaning the GTX is entirely dedicated to the tasks presented via OpenCL. This card supports OpenCL by way of a CUDA software layer. The specification for this card is as follows.

- 448 CUDA Cores
- 1.2 GHz Processor Clock
- 1.25 GB DDR3 RAM
• 133.9 GB/s Memory Bandwidth

Additionally, a second workstation is utilized for comparison, containing a NVIDIA Tesla K20c. The CPU for this second workstation is omitted, as it is not used for comparison. The Tesla device’s specifications are as follows.

• 2496 CUDA Cores
• 706 MHz Processor Clock
• 5 GB GDDR5 RAM
• 208 GB/s Memory Bandwidth

**Qualitative Analysis**

One version of the system presented includes the use of preexisting library operations for vector and matrix-vector operations. Specifically, the OpenCL AMD BLAS libraries were to be used to compute the KM and the objective function at each iteration; OpenCL AMD BLAS is an implementation of the existing Basic Linear Algebra Subprograms (BLAS) specification, written by AMD and the open source community specifically for OpenCL, here abbreviated clAmdBLAS. The objective function update step requires an operation of the form

\[ \vec{y} = \alpha_1 \vec{x}_1 + \alpha_2 \vec{x}_2 + \vec{y} \]

which can be accomplished by multiple invocations of

\[ \vec{y} = \alpha \vec{x} + \vec{y} \]

which is supported by clAmdBLAS by way of its Double Precision AX Plus Y (DAXPY) function. However, a custom OpenCL kernel written to perform the first operation (meant to circumvent two invocations of the library) using the trivial solution was measured to be ten times faster than the library, that is, it completed in one tenth the time taken to complete...
the library function.

Similarly, calculating the KM required performing the operation

$$\vec{y} = A\vec{x}$$

which can be accomplished by an invocation of

$$\vec{y} = \alpha A\vec{x} + \beta\vec{y}$$

supported by the GEneral Matrix Vector (GEMV) multiply function of clAmdBLAS. However, the described custom implementation of this operation performed four times faster than the clAmdBLAS equivalent.

This preliminary result is not entirely unreasonable for a number of reasons; first, the clAmdBLAS libraries have to apply to a wide set of cases and are more general. They have to perform redundant dimension and type checks for safety and have a large number of functions implemented as a set of highly flexible primitives; this reduces development time at the cost of performance. Furthermore, the GEMV function in our case has to perform memory synchronization and floating point operations beyond what are necessary because of the general nature of the operation supported, and the double-precision variant is deprecated.

While initial performance results immediately show improvements in the performance of KM calculation, even for small data sets, the same was not initially true for the objective function updating step. Due to overhead issues for this step, and the performant nature of traditional compute units in this realm, it is specifically analyzed and its performance is plotted against data set sizes to show the crossover point and the costs and benefits of implementing this particular step in OpenCL. A similar issue arose for the working set selection portion of the algorithm, but to a much more extreme degree; while the overhead costs are small, and the parallel algorithm is actually very performant, the total computational cost
of this step is just too small to justify its acceleration.

5.1.1 Training

Data Set

The accelerated version of the LIBSVM algorithm was compared to the reference implementation when performing eye detection on the BIO ID database. This database contains annotated images of human faces; image patches of size 25 by 50 pixels are extracted, that either do or not contain a cropped human eye, and are labeled as such. Consequently, the trained SVM attempts to differentiate human eyes from non-eyes using raw grayscale pixel data, reshaped into one-dimensional vectors of length 1,250.

There are approximately 56,000 samples in the resultant data set, split comparatively evenly between positive and negative samples, and the SVM training algorithms are tested on varyingly sized subsets of this data. Subsets are chosen by selecting the first $N$ training vectors from the entire data set, $\frac{N}{2}$ being positive samples and the remaining $\frac{N}{2}$ being negative samples, with the exception of the largest subsets where the distribution between positive and negative samples is slightly uneven.

The data sets are varied in size because the number of iterations of the SMO algorithm is expected to scale with the dimensionality of $\vec{\alpha}$ (number of training vectors), while the size of the Kernel Matrix is known to grow with the number of training vectors, both of which increase the total execution time of the training algorithm.

In addition to the BIO ID data set, a subset of the PASCAL Visual Object Classification (VOC) data set is used, both for training and classification. The data set in its raw form contains images, each of which is annotated with the bounding box of that image’s object of interest and the object class to which it belongs. For the set of experiments performed here, the car class is used.

A two-class SVM is used for these tests. However, the work presented can be extended with relative ease to the other SVM types available in the reference implementation of
LIBSVM. All tests are performed with the LIBSVM default parameters for training: degree equal to 3, gamma equal to one divided by the number of features, coefficient equal to 0, C equal to 1 and epsilon equal to 0.1.

For the sake of emulating a realistic training data scenario, we perform some preprocessing on the images before training on them. Specifically, we perform the Histogram of Oriented Gradients transformation on each image patch, and train on the resulting image description. We also train on raw image data for the BIO ID image data set, for completeness and to ensure that the achieved acceleration is not highly specific to certain types of training data.

**WSS Comparison**

For this test, the execution time dedicated to selecting the working set, here denoted as WSS, is specifically measured. Since this step is computed on every iteration of the larger SMO algorithm, and since it requires iterating over the entire objective function, we expect it to be computationally significant. However, as Figures 5.1(a) and 5.1(b) show, the time spent on the WSS step is insignificant with respect to the entire execution time and shows no significant growth. For this reason, no accelerated version of this step is presented or discussed.

One reason for this discrepancy is hand optimization of LIBSVM. In addition to calculating the objective function at every iteration of the algorithm, the objective function’s gradient is calculated so it is available before the WSS phase; furthermore, since \(y\) is constrained to \(\pm 1\), control flow is used in place of floating-point multiplication. Finally, the Kernel Matrix values are cached once they are calculated, eliminating the greatest expense in calculating the \(j\) index of the working set.

Likely more important is the nature of this WSS compared to modern CPU design, which employs hardware supported loop unrolling, superscalar enhancements and highly accurate branch prediction techniques to optimize this particular operation.
Figure 5.1: Working Set Selection Comparison
**Linear Kernel**

During this phase, it was enforced that the SVM files produced by the two algorithms match exactly. Figure 5.2(a) shows the execution time for both algorithms as a function of the number of samples used for training.

As most HC implementations of an algorithm induce some overhead for communication and hardware setup, it is expected that the new implementation will produce a slow down for small data sets and that its execution time will grow more slowly with data set size than the reference implementation until there is some crossover. Figures 5.2(a) and 5.2(b), therefore, are intended to provide insight as to where this crossover is.

The data suggest that this crossover comes on the order of thousands of samples for this training set; however, since machine learning problems typically rely on very large data sets to achieve high accuracy, we conclude that the typical user will experience the full benefit of the accelerated algorithm.

Furthermore, we measure the speedup achieved by the algorithm as a function of the training set size. Once the data set has crossed over the point where it achieves speedup greater than one, we expect that the speedup will increase with the data set size until the accelerated system is as fully utilized as the original, unaccelerated system is before the crossover, and beyond this point the speedup will converge to a single, maximum value. For this data set, the speedup appears to approach 14 with the full 56,000 training samples for raw image data, and 12 for the HOG image descriptors.

When we perform this same set of experiments on the Tesla device, as shown in Figures 5.4(a) and 5.4(b), we see a similar performance gain. This is unfortunate, considering the increased performance power (and consequently significantly increased cost) of this device, and indeed the benefits of the more powerful device seem to wane as the data size is increased. Discussion of the implications of this performance difference is withheld until the remaining experiments involving the device are presented.
(a) Training Time of Reference and Accelerated Implementations of LIBSVM on BIO ID Raw Image Dataset Graphed Together for Comparison

(b) Speedup of Accelerated Implementation of LIBSVM on BIO ID Raw Image Dataset

Figure 5.2: BIO ID Raw Image Training Performance (GTX 480)
(a) Training Time of Reference and Accelerated Implementations of LIBSVM on BIO ID HOG Dataset

(b) Speedup of Accelerated Implementation of LIBSVM on BIO ID HOG Image Dataset

Figure 5.3: BIO ID HOG Training Performance (GTX 480)
(a) Training Time of Accelerated Implementation of LIBSVM on BIO ID HOG Dataset on Both Available GPU Devices

(b) Speedup of Accelerated Implementation of LIBSVM on BIO ID HOG Image Dataset on Both Available GPU Devices

Figure 5.4: BIO ID HOG Training Performance Across GPUs
Figures 5.5(a) and 5.5(b) show the training time and training speedup, respectively, of the accelerated version of LIBSVM as called from MATLAB. The previously described code, along with a MATLAB interface file were compiled into a mex library that could be called directly from a MATLAB script. Additionally, this was tested on the NVIDIA Tesla device, as shown in Figures 5.6(a) and 5.6(b). Again, we find that the OpenCL version is significantly accelerated with respect to the baseline, we find that the Tesla device runs the same code faster than the GTX version, that the speedup of the Tesla device wanes as both devices approach their maximum speedup, and that the OpenCL version converges to a large speedup over the serial version, but do not approach the ratio of computing cores used in the OpenCL version to those used in the serial version.

**Non-Linear Kernels**

In addition to the standard, linear kernel, the other three kernel types (Polynomial, RBF, Sigmoid) available in LIBSVM are examined using the same test. Kernels are of the form \( K(X, \vec{x}_k) = h(X \vec{x}_k) \), where \( h(\cdot) \) is an element-wise operation. Being element-wise, \( h(\cdot) \) is highly parallel; however, for the linear kernel \( h(x) = x \) presents no computational cost and so the parallelism of \( h(\cdot) \) can not be exploited. For this reason, it is expected that the non-linear kernels will show larger speedups when accelerated using OpenCL.

Figure 5.8(b) shows the speedups of the various kernels tested in this work. Recall that the polynomial kernel has \( h(x) = (x + c)^d \); the speedup for this kernel surpasses the speedup of the linear kernel, as expected, for small numbers of training vectors. However, at full saturation, the linear kernel speedup surpasses the polynomial kernel speedup; this is likely because the element wise function bears a much larger penalty on the highly parallel cores, which are traditionally weaker than standard CPU cores, than it does for the reference implementation.

A similar effect is seen for the sigmoid kernel; recall that the sigmoid kernel has \( h(x) = \tanh(x + c) \). Here, again, the linear kernel’s speedup surpasses the sigmoid kernel’s speedup because the element wise function being performed is much more expensive
(a) Training Time of Reference and Accelerated Implementations of MATLAB Binding of LIBSVM on PASCAL VOC Dataset

(b) Speedup of Accelerated Implementation of MATLAB Binding of LIBSVM on PASCAL VOC Dataset

Figure 5.5: PASCAL VOC Training Performance (GTX 480)
(a) Training Time of Accelerated Implementation of MATLAB Binding of LIBSVM on PASCAL VOC Dataset on Both Available GPU Devices

(b) Speedup of Accelerated Implementation of MATLAB Binding of LIBSVM on PASCAL VOC Dataset on Both Available GPU Devices

Figure 5.6: PASCAL VOC Training Perform Across GPUs
to compute on the highly parallel device than it is on the CPU. This makes sense if one considers the element wise function $h(\cdot)$ to have some per computation cost of $O(k)$ on the CPU and a different cost $O(z)$ on a single GPU core. Assuming the operation can be fully parallelized in its current form, the speedup degrades from the ideal, linear speedup, i.e. parallel cost $O\left(\frac{h(\cdot)}{p}\right)$, to some scaled version of that, i.e. parallel cost $O\left(\left(\frac{z}{k}\right)\left(\frac{h(\cdot)}{p}\right)\right)$, where $p$ is the number of processors employed.

However, Figure 5.8(b) shows exactly the desired result for the speedup of the RBF kernel, which has an exponential calculation as its inner loop. That is to say, the speedup for the RBF kernel grows quickly and surpasses all of the other kernels, including the linear kernel. As to why this operation is favorable on the GPU; this implementation made use of the OpenCL runtime’s version of the C standard libraries (e.g. math.h) to perform all of these element-wise operations. These libraries are allowed to vary between systems and are opaque to the user.

For completeness, Figures 5.7(a), 5.7(b) and 5.8(a) illustrate timing comparisons between the accelerated library and the reference LIBSVM for all of the nonlinear kernels. It is interesting to note in the RBF case that the serial execution time grows at a startling rate compared to the other kernels, which sheds some light on why this kernel saw a greater speedup.

Figures 5.9(a), 5.9(b) and 5.10(a) show similar comparisons for the training performed using non-linear kernels on HOG descriptors of the BIO ID image dataset. Note that the speedups converge to significantly lower values using the HOG descriptors than they did with the raw image data. There are a few possible explanations of this; first, the HOG description transformation changes the shape and relative positions of the vectors being separated by the SVM, and so could change the effort required to form the separating hyperplane. This can drastically affect the computation time required to complete training. Secondly, the HOG description vector is shorter than the raw image data vector for any given training sample; we know that a longer vector requires more computational effort
Figure 5.7: BIO ID Raw Image Training Performance (Non-Linear Kernels) (GTX 480)
(a) Execution Time for Training Radial Basis Function Kernel Using BIO ID Raw Image Data Set

(b) Speedup for Training Non-Linear Kernels Using BIO ID Raw Image Data Set

Figure 5.8: BIO ID Raw Image Training Performance (Non-Linear Kernels) (GTX 480)
to perform the same matrix-vector operation, but it is uncertain if a longer vector requires
more computational effort in a serial computation context in proportion to the effort re-
quired in a parallel computing context.

To explore this question, we perform an experiment wherein we take the same set of
images from the PASCAL dataset and resize them to create vectors of varying length; we
then use those vectors to train an SVM using both the serial and OpenCL implementations,
so that we can see how speedup varies by vector length for our implementation. Figure
5.10(b) shows the results of this experiment; while we can not decisively say which vector
size is optimal for all machine architectures and data sets, it appears that there is a range
between 3,000 and 7,000 data elements where the processing elements in use are fully uti-
lized without being extremely overloaded.

Additionally, Figures 5.11(a), 5.11(b), 5.12(a), 5.12(b) and 5.13 show speedup and ex-
cution time information for nonlinear kernel training time on the Tesla GPU device. What
is interesting to note is that, despite only marginally beating out the less power GTX de-
vice on most of the kernels, the Tesla maintains a near maximum speedup through all of
the kernels. To some degree, this indicates that for certain kernels (such as the polynomial
kernel), the GTX device is fully saturated; that is, it is limited by hardware constraints and
not a weakness in the algorithm. However, that the maximum speedup on the Tesla device
is within a fairly tight tolerance of the maximum speedup on the GTX device indicates that
the algorithm may not fully utilize the available hardware in all cases.

We have calculated previously that the parallel execution time is inversely proportional
to the number of processing elements, or the parallel speedup is directly proportional to
the number of processing elements. Therefore, by increasing the number of cores ap-
proximately five fold, we expect to see an increase in speedup from approximately 12 to
approximately 60, rather than an increase from 12 to approximately 14. However, taking
Amdahl’s law into account

\[ S \approx \frac{1}{B - \frac{1}{n}(1 - B)} \]  

(5.1)
Where $B$ is the percentage of the original algorithm that is completely serial and $n$ is the number of processors dedicated to the parallel version. As $n$ becomes large, the total speedup will converge to some constant value depending on the original algorithm, and we would expect only a marginal increase in speedup for a significant increase in the number of computing cores. Assuming this value is related to the ratio of the computing power of each parallel core to the computing power of the original serial core, we could answer the hypothesis that this is the cause of our result on the Tesla device by running our accelerated algorithm on a system with a large number of powerful cores (such as a number of FPGAs or a traditional multicore system with many CPUs). We would expect to see a speedup curve with the same shape and gradient as the current speedup curve, but converging to a slightly higher value.

**Objective Function Updating**

Upon in-depth profiling, the performance of the code segment which updates the objective function does not present the expected advantage. A comparison of the execution times of the varying implementations is presented. The basic, serial implementation of this step on the CPU using standard x86 instructions shows no crossover with the CPU implementation using OpenCL, which makes use of the CPU’s SIMD extensions.

Figure 5.15 shows the relative execution time for this step; the performance gap between the two approaches appears to be increasing, indicating that this isn’t a matter of initial overhead. Figure 5.14(a) shows how the time is spent in executing the OpenCL CPU implementation; recall that the main computational task in updating the objective function is performing the calculation $\vec{f} = \vec{f} + \Delta \vec{\alpha}_i Q_i + \Delta \vec{\alpha}_j Q_j$, which is denoted here as the DAXPY step (from Double precision AX Plus Y). As one can see, this portion of the calculation is substantially accelerated; however, the gains are overwhelmed by the costs of communication and coordination, both of which grow with sample count. In particular, the Q matrix calculation time, which includes the time taken to communicate the matrix to the device, balloons as excessive memory pressure caused by a growing training data set.
Figure 5.9: BIO ID HOG Training Performance (Non-Linear Kernels) (GTX 480)
Figure 5.10: BIO ID HOG Training Performance (Non-Linear Kernels) (GTX 480)
(a) Execution Time for Training Polynomial Kernel Using BIO ID HOG Data Set On All Available GPU Devices

(b) Speedup for Training Polynomial Kernel Using BIO ID HOG Data Set On All Available GPU Devices

Figure 5.11: BIO ID HOG Training Performance (Non-Linear Kernels) On All GPUs
Figure 5.12: BIO ID HOG Training Performance (Non-Linear Kernels) Across GPUs
Figure 5.13: Speedups for All Training Kernels On All Available GPU Devices

defeats the caching scheme.

Furthermore, Figure 5.15 shows the execution time for this step as implemented on the GPU, and that the GPU implementation is surpassed by both of the other two implementations. While this result is less than ideal, it does illustrate one of the key issues in HSA design over pure GPGPU design, which is that an optimal implementation dispenses work to the computation unit for which it is best suited. Figure 5.14(b) shows the breakdown of the GPU’s execution time, for completeness.

**Kernel Matrix Calculation Breakdown**

In order to understand the performance bottleneck of the new, accelerated system, the portion of time spent calculating the kernel matrix is recorded and compared to the entire training time in Figure 5.16(a). It is clear that, even after accelerating this step of the algorithm, a large portion of the total execution time is still in this phase.

Furthermore, we examine the time spent during the KM step to determine where its
Figure 5.14: Objective Function Calculation Performance Breakdown

(a) Breakdown of time spent updating objective function

(b) Breakdown of time spent updating objective function on GTX 480 GPU
limitations lie. As shown in Figure 5.16(b), we find that the main computation, that is the GEMV operation, consumes the vast majority of this step’s execution time and significantly outweighs its communication time. This indicates the effectiveness of the caching scheme used for this step in the algorithm, as well as hinting at the efficiency of this particular approach.

**Effort Comparison**

We also present a comparison of the full custom implementation to an implementation using prefabricated libraries and implicit communication. Figure 5.17(a) shows how the execution times for all three implementations compare. As is expected, the minimal effort implementation is slower than the full custom version, but its execution time still grows more slowly than the reference implementation and provides an advantage.

However, Figure 5.17(b), which compares the speedup of the various approaches, illustrates that the speedup of the minimal effort implementation reaches a peak relatively early.
Figure 5.16: Kernel Matrix Calculation Performance Breakdown (GTX 480)
(a) Execution Time for Full Custom, Prefabricated and Reference Implementations

(b) Speedup for Full Custom, Prefabricated and Reference Implementations

Figure 5.17: Effort Comparison (GTX 480)
Furthermore, this peak, at slightly over 2, is a modest improvement over the reference implementation when compared to the potential speedup offered by the full custom version, as well as the theoretical speedup made available by the hardware.

**Comparison to Related Work**

Here, we compare the results from this thesis to the similar works on SVM acceleration discussed earlier. Results are summarized in the table below. The raw speedups achieved by the listed works seem to vary wildly; this is most likely due to significant variations in the training sets used, as this thesis has shown the data set to be a key factor in the system’s performance characteristics, as well as variations in the hardware environments used, both in the baseline CPU device and the accelerated GPU device.

<table>
<thead>
<tr>
<th>Work</th>
<th>Environment</th>
<th>Data Set(s)</th>
<th>Speedup (Training/ Classification)</th>
</tr>
</thead>
<tbody>
<tr>
<td>cuSVM</td>
<td>Intel Core 2 Duo @2.66GHz</td>
<td>Adult, Web, MNIST</td>
<td>(12-72)/(22-172)</td>
</tr>
<tr>
<td></td>
<td>NVIDIA GTX260</td>
<td>Forest</td>
<td></td>
</tr>
<tr>
<td>GPU LIBSVM</td>
<td>Intel Core-i7 950 @3.06GHz</td>
<td>TRECVID</td>
<td>(3-9)/NA</td>
</tr>
<tr>
<td></td>
<td>NVIDIA GTS250</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Intel Q6600 @2.4 GHz</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>NVIDIA 8800GTS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CL-SVM</td>
<td>Intel Core-i5 @3GHz</td>
<td>BIO ID, VOC</td>
<td>(9-22)/(10-14)</td>
</tr>
<tr>
<td></td>
<td>NVIDIA GTX480</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>NVIDIA Tesla K20</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 5.1: Comparison of GPU Accelerated SVM Systems
5.1.2 Classification

The SVM model resulting from training with the entire BIO ID data set was used to test the performance and accuracy of the classification phase of the system. Since classification makes use of the same kernel operation being performed on high dimensional vectors, one might expect to see a similar increase in performance as was seen during training, and indeed there is a speedup over the reference implementation. Unfortunately, this speedup is ultimately limited by a number of factors; first, classification is less cache friendly. Since a vector being classified only enters the system once, and is only used once, nothing can be gained by keeping it in memory. Secondly, and much more significantly, one would only expect to classify a small number of vectors at a time.

Classically, one would expect to classify a single vector at a time and this cuts deeply into the potential speedup offered during classification. However, there are applications which call for large bursts of classification at a time; if one were searching for an object in a larger scene a common approach is to break the scene into overlapping windows of the size of the expected object and then classify each of them. The number of samples one would classify in this case can be expressed as

\[
\frac{(W-w)(H-h)}{s^2}
\]  \hspace{1cm} (5.2)

where W and H are the dimensions of the search region, w and h are the dimensions of the expected object and s is the step size. Using the aforementioned 25 by 50 pixel object size, and assuming a video frame of size 640 by 480 with a step size of 3 this would yield approximately 30,000 classifications per frame. For this reason subsets of varying size of the original training data set are classified, up to tens of thousands of samples. Data can be reused like this because the classification code in LIBSVM doesn’t try to detect duplicate data, but rather blindly calculates the decision function.

Linear Kernel
Figure 5.18(a) shows how the resulting execution time varies, and Figure 5.18(b) shows speedup as it varies with the number of samples being classified. The data shows a significant speedup, growing with the number of classifications to be performed at once. This is in line with the expectations gleaned from examining the training data. However, this speedup appears to saturate before reaching the same peak seen during training on the same data.

**Non-Linear Kernels**

Here the classification performance of non-linear kernels is considered. Figure 5.19(a) shows how the speedups of all four available kernels compare as they change with the number of vectors being classified. As is the case with the same comparison for the SVM training phase, the linear kernel speedup is not greatly outmatched by the speedup of the polynomial or sigmoid kernels. However, the RBF kernel shows significant improvement in speedup over the linear kernel, even for smaller classification sets.

In any case, the classification speedup is less than the speedup achieved during training for all kernels; this is, again, due to the cache unfriendly nature of the classification task. For completeness, figures 5.19(b), 5.20(a) and 5.20(b) are included, which show the classification time for reference and OpenCL implementations of the system.

## 5.2 Error

### 5.2.1 Training

For the output models of the training phase of the LIBSVM system, several metrics are presented to characterize the error of the new system compared to the reference. The first discussed here is the relative Mean Squared Error (MSE) between the reference $\bar{\alpha}$ vector and the HSA vector, calculated as

$$\sum_i \left( \frac{\bar{\alpha}_i - \bar{\alpha}'_i}{\bar{\alpha}_i} \right)$$
Figure 5.18: BIO ID Raw Image Classification Performance (GTX 480)
Figure 5.19: BIO ID Raw Image Classification Performance (Non-Linear Kernels) (GTX 480)
Figure 5.20: BIO ID Raw Image Classification Performance (Non-Linear Kernels) (GTX 480)

(a) Classification Time for Sigmoid Kernel

(b) Classification Time for RBF kernel
where $\vec{\alpha}_i$ and $\vec{\alpha}'_i$ are the element at the index $i$ of the reference vector and the vector under test respectively. This is important because the $\vec{\alpha}$ vector characterizes the SVM model extracted, and errors in this parameter may lead to misclassification in the classification phase.

This error is expected to grow primarily with three parameters: the number of training vectors used, the length of the training vectors used and the amount of computation associated with the kernel. Figure 5.21 shows how the MSE grows with the number of training vectors for all four available kernels. It is worth noting that for three of the kernels, the total error is approximately zero for all vector lengths; the RBF kernel on the other hand presents an error level that seems to move randomly between data points. This is not unexpected, as the RBF kernel makes use of the input vectors three times and performs an exponential calculation using a non-standard implementation of the C standard libraries.

Furthermore, this work makes use of double-precision floating-point operations in order to maintain maximum precision. These operations are supported by certain OpenCL devices, but double-precision is not part of the OpenCL standard and OpenCL makes no guarantee as to the accuracy of these operations.

However, the maximum MSE recorded for this kernel is on the order of $10^{-6}$, and while some applications of SVM call for extreme precision in classification, we expect that this error is within tolerance for most users. Another important metric for error measurement is the difference in the final calculated value of $b$. This parameter also describes the SVM’s characteristic hyperplane and thus has a deep impact on the accuracy of any classification done with the trained model. However, no results for the error in $b$ are presented because within tolerance of the IEEE floating point standards, no error was observed with respect to the reference value.

Additionally, for the purposes of this work, any differences in the vectors that were selected as support vectors is recorded. This error reveals a similar insight as does the MSE.
error in the alpha vector; that is, the separating hyperplane of an SVM is in part characterized by the training vectors that are selected for use in forming it. However, no results for this error are shown because none were observed; that is, the accelerated SVM implementation selected precisely the same set of support vectors as the reference implementation.

5.2.2 Classification

During classification, the only metric observed for error measurement is the number of vectors misclassified by the OpenCL version of the library compared to the serial version. No cases of misclassification were observed. This is not surprising, as classification phase outputs are constricted to a binary set of class labels; a large range of values produced by the decision function can be accepted as precisely correct, as long as they fall on the correct side of the separating hyperplane.
5.3 Summary of Results

In the table below we summarize the performance results of the presented system. We can see that the largest contributor to the speedup of the system is the dataset in use.

<table>
<thead>
<tr>
<th>Training Set</th>
<th>Accelerator</th>
<th>Training Speedup</th>
<th>Classification Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIO ID</td>
<td>NVIDIA GTX 480</td>
<td>12-14</td>
<td>10-14</td>
</tr>
<tr>
<td>BIO ID/HOG</td>
<td>NVIDIA GTX 480</td>
<td>8-12</td>
<td>-</td>
</tr>
<tr>
<td>BIO ID/HOG</td>
<td>NVIDIA TESLA K20</td>
<td>12-14</td>
<td>-</td>
</tr>
<tr>
<td>VOC/HOG</td>
<td>NVIDIA GT 480</td>
<td>22</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 5.2: Performance Results by GPU Device

5.4 Adapting to the Tesla K20

Thus far, when comparing the two NVIDIA devices used in this work, we have only considered how they differ in terms of easily quantifiable specifications such as clock speed, total RAM and core count. However, The GTX 480 and Tesla K20 used here are actually based on substantially different microarchitectures: the (older) Fermi architecture[42] used in the 480 and the (newer) Kepler[43] architecture used in the K20. It is therefore useful not only to consider changes to our algorithm to better use the extra cores available in the K20, but changes that make better use of the improved, underlying architecture. In this section, we consider features that are new to the Kepler architecture, which offer an advantage over the Fermi architecture, which we might exploit in order to accelerate our system.

One of the most heavily advertised Kepler features is called Dynamic Parallelism; in short, kernels operating on the device have the ability to manage and enqueue other kernels, which themselves will run as though they had been enqueued from the primary host. This feature would be useful in the development of the prediction phase of the accelerated
SVM system; specifically, two kernels were employed in sequence which incurs significant overhead. However, software support for this feature in OpenCL was not introduced until the OpenCL 2.0 specification, for which NVIDIA driver support is not forthcoming at the time of this writing. Support exists for this feature on AMD and Intel devices, but NVIDIA devices make up a significant portion of the accelerator devices on the market and thus cutting them out of our user base by requiring features introduced after OpenCL 1.1 is not prudent for this project.

Another novel Kepler feature is called Hyper Q, which allows multiple connections from the host CPU to the GPU device, each with its own hardware work queue. This is designed to allow multiple threads of execution on the host to independently make use of the device and thus, hopefully, keep it fully utilized. During the prediction phase of the accelerated SVM system multiple samples can be classified concurrently, from multiple threads on the host, and increase utilization of the GPU. However, during the training phase of the system, the dispatch of kernel matrix calculations is almost entirely sequential as a result of the organization of the SMO algorithm and little gain is to be had from this feature.

The last of the major, explicitly accessible features of the Kepler architecture is DirectGPU™, which allows devices other than the primary CPU host to perform direct memory access (DMA) on the GPU. These can be other accelerators in the system, or networking devices which allow communication to GPUs on other hosts in a larger, distributed system. This would provide obvious performance benefits for many systems, but the system we present here makes use of only one accelerator at a time and thus could not make use of this feature.

Finally, Kepler introduces many improvements whose use is hard to capture explicitly and must be utilized through subtle code manipulations through the help of a sophisticated compiler. For example, the new architecture allows double precision floating point operations to be dispatched concurrently with dissimilar operations; unfortunately, this does not benefit our application, as it makes heavy use of double precision floating point operations, and their computational complexity is a limiting factor. Kepler also incorporates better
memory bandwidth and more cache per core, which is useful for data intensive applications such as the accelerated SVM system presented here; however, no modifications need to be made to capture it, since we already do as much as we can to cache data on the GPU.

The newer architecture introduces 4 times as many registers per thread as Fermi, which is useful for large kernels which experience significant register pressure; however, our kernels are relatively small, and a manual inspection of the generated assembly for our training kernels indicates that they are not under significant register pressure. There is, in addition, a new supported instruction intended to reduce communication latency within a single warp; the "shuffle" operation, can move data between threads in a warp without a load/store operation. This could reduce communication time for the binary reduction (second phase) portion of our two phase reduction, where data is copied between two threads on every iteration; however, we use a thread block size which crosses warp boundaries and we can not expect a compiler to reliably produce the optimal assembly code in this case.

In the end, we avoid making optimizations for a single, target device at the expense of other accelerators, and we argue that the benefits of portability outweigh the performance cost for specific situations. These optimizations would not only lock the application in to NVIDIA devices, but would actually restrict its usage to one particular micro-architecture and reduce its usefulness.
Chapter 6

Conclusion

6.1 Summary

The presented work is successful in its primary goal, accelerating a widely accessible implementation of SVM training and classification while maintaining its accuracy and usefulness. Furthermore, the resulting product is portable and easy to use, being built on and using open source tools and libraries and maintaining the same programming and command line interfaces as LIBSVM.

The linear and polynomial kernels tested in this system are shown to have no error with respect to the reference implementation, either in support vector selection or hyperplane description parameters. It is important that this error be constrained very closely, as it is expected to grow as more training vectors of higher dimensionality are introduced.

Furthermore, the speedup achieved for training, while falling short of the ideal linear speedup equal to the number of processing elements available, provides a benefit that greatly outweighs the cost of adopting the accelerated system. By accelerating the SMO algorithm by a factor of 10-14, the work presented can reduce days or weeks of training time into hours at minimal cost to the system’s user.

However, portions of the training algorithm used for LIBSVM were not conducive to the approach used. In particular, the KKT condition tracking function update proved to require too much communication and coordination overhead to be sped up, and selection of
the SMO working set bears too little computational load to be worth accelerating. Additionally, non-linear kernels were expected to see a larger speedup than their linear counterpart but fell victim to the complexity of their element-wise operations when compared to the computational elements brought to bear on them.

The new approach is not without its limitations; since the new system requires all of the training data available to be stored on a device’s memory simultaneously (since we calculate the entire kernel matrix with a single invocation of an OpenCL kernel to reduce overhead), there could be a problem adapting this algorithm to use very large data sets for which this isn’t possible. Overcoming this will require some changes to memory management, but most likely not a total overhaul of the primary parallel algorithm. Furthermore, while this system is designed to push different parts of the SMO algorithm to different computational devices, it is not fully capable of splitting an individual operation amongst multiple devices. For instance, multiple GPUs can not currently be used to accelerate the kernel matrix calculation.

Our analysis of the use of OpenCL yields certain insights about its benefits and drawbacks. Among its benefits are its portability across hardware platforms; combined with its ability to dynamically profile the hardware used to run it, this means that programs using accelerated paradigms, like GPGPU and FPGA programming, can be written to be truly portable in much the same way more standard applications are written.

Furthermore, that the OpenCL kernel language implements large portions of the C standard math libraries, along with extensions for vector math operations, greatly eases the development of scientific applications for OpenCL. This is furthered by certain third party libraries, such as the BLAS and FFT implementations provided by AMD.

Among the drawbacks to OpenCL is that these libraries sometimes come with a significant performance penalty, as evidenced by our use of the AMD BLAS library. Additionally, to gain the full performance benefits of an OpenCL compliant platform, one must develop their application in a way that is aware of the platform rather than leaving all decisions
to the runtime environment. However, this is no different, at least, from the performance implications from developing highly generalized applications for a traditional, serial environment.

OpenCL displays many similarities with CUDA, the parallel programming platform developed specifically for NVIDIA GPUs. The two platforms are both designed to operate on a system with a traditional CPU in control of one or more many-core accelerators. To that end, they both make use of a SPMD model with the single program delivered in the form of a kernel developed in C and run asynchronously from a single, sequential, control program. However, the differences are quite stark; primarily, OpenCL supports a much greater variety of accelerator devices and configurations, as has been discussed at length. This offers many advantages over CUDA, which only supports NVIDIA devices. However, in a situation where an application is being developed explicitly and exclusively for NVIDIA accelerator devices, CUDA offers one significant advantage: the CUDA API is much easier to use, which helps to reduce development time for scientific applications. CUDA kernels are compiled statically along with the serial portion of the program, which allows the full error prevention capacity of modern compilers to be brought to bear easily on the parallel portion of the program. Furthermore, the hardware configuration of a system is implicit in a CUDA program, so there is substantially less ”boiler plate” code used to create contexts, kernel objects, program objects, device handles, etc. Calls to CUDA kernels are very easy to write as well as to read and comprehend; rather than a kernel being called using a series of API calls, as it is in OpenCL, a CUDA kernel can be called like any C function with a small amount of metadata encoding the thread arrangement. The overall effect is that CUDA programs are more concise and readable than their equivalent OpenCL counterparts, and are significantly easier to develop.

In speculation of the future of OpenCL, we posit that it will supplant similar solutions for the same application area, owing mostly to the portability of produced applications. The success of OpenCL, in terms of its widespread adoption, will likely hinge on its ability
to supplant NVIDIA’s CUDA platform as the de facto standard for scientific computing. CUDA has two major advantages in gathering market share; the first being that it gained popularity first and thus has some degree of legacy inertia in the scientific computing community. The second advantage that CUDA holds is that it runs on what is currently the only set of accelerators being considered for most professional scientific computing applications, GPU devices created by NVIDIA. This, for many, defeats the argument of OpenCL’s portability. However, emerging research indicates that FPGAs will play a key role in the future of hardware acceleration, and NVIDIA’s current, perceived dominance in scientific computing hardware is unlikely to remain a permanent fixture.

6.2 Future Work

SVM Types

For this work, only one SVM type is considered in order to provide focus, and this type only supports classifying between two classes. While this provides a solid foundational work, it is important that this work be expanded to SVM class types other than binary SVMs in order to be widely useful or accepted. The major limitation to this would be a somewhat fragmented code base, involving work repeated in numerous places.

FPGAs

For this work, the use of GPUs and CPUs as parallel devices through OpenCL is investigated in large part because of their ubiquity in desktop work stations. However, reprogrammable hardware devices such as FPGA devices are not only increasingly common in research settings like universities, but also offer an extremely promising source of hardware acceleration. FPGAs offer two basic routes of acceleration; first, specific hardware could be generated at run time to accelerate exactly the operations involved in the SMO operation, such as matrix multiplication of the exact dimensions the training set requires or the DAXPY calculation required in the objective function update.
This would require a rather sophisticated software stack, including a very intricate compiler, and consequently might introduce a large overhead since OpenCL kernels are typically compiled at run time and would be forced to be compiled at run time in this case. The second option made possible by FPGAs is to build the accelerated system out of accelerated primitives of which FPGAs provide a much wider set than is typically available in traditional CPU-GPU environments. For instance, a reductive adder primitive along with a double-precision multiplier primitive (repeated to form a vector multiplier) together could perform the matrix-vector multiplication operation used in SMO. This would likely be less intensive on the necessary compilers, and would still be able to exploit the inherent parallelism and flexibility of the FPGA hardware environment.

**Structured SVMs**

Structured SVMs (SSVMs) are a generalized form of SVM which seek to find an optimal function mapping some raw data points to a set of discrete output labels. [1] proposes an algorithm for solving general SSVMs that has, as its inner loop, an optimization problem of the type seen in SVM models, as well as additional linear algebra computations of high dimensionality that lend themselves to HSA acceleration. For this reason, SSVMs are a prime target for the type of work presented here, though they are not considered here because LIBSVM does not support SSVMs.

**Further Acceleration**

While the OpenCL language specification benefits from standard library math functions, these functions are implemented serially on a single CU device core. This system could benefit from fully accelerated implementations of these functions. Since the number of computational elements that are involved in calculating a single output of the kernel matrix column decreases as the algorithm progresses, when the calculation of the element-wise operation is performed the majority of the CU has been left dormant; this represents untapped computational power that could be brought to bear on the element wise calculation.
Furthermore, currently each phase of the SMO algorithm is designed to be performed on a single CU device. However, the heart of HSA is dividing single algorithms across multiple devices, so a logical extension of this system is to break down single phases such as kernel matrix calculation so that they can be split across devices.
Bibliography


[38] B. Demir, S. Erturk, “Improving SVM Classification Accuracy Using a Hierarchical Approach for Hyperspectral Images”, Image Processing (ICIP), 2009 16th IEEE International Conference on, 2009


