Daniel Palmer 10265229

Parallel Computing and Distributed Systems
Support Vector Machine

Introduction

In this report I have created a Support Vector Machine (SVM) for classification of points. The reason for making this application parallel is the nature of the calculation of used to classify the points.

$$I(z) = sign\left(\sum_{SupportVectors} \alpha_i K(z, c_i) + b_0\right)$$

α = Support vector weight
c = Support vector
z = Test vector
b = bias
K = kernel function

Figure 1

The kernel of each vector can be done in parallel I have chosen to uses the linear kernel for this example due to it's simplicity but if you where to use the B-Spline or Histogram Intersection Kernel that contain further sigma functions to sum kernel input would allow for greater parallelism and so give a greater speed up verses a cpu bound implementation.

$$\text{Linear: } K_{Linear}(a, b) = a \cdot b^T$$

Figure 2

The linear kernel from figure 2 replaces K(z,c) in figure 1 (both figures came from *GPU Acceleration of Object Classification Algorithms Using NVIDIA CUDA[2])*. The linear kernel uses the dot product of the input matrices as it's classification method.
During training the alpha and bias variables are varied to find the best fit to the support vectors this process is similar to training neural networks where the neuron weights are varied.
The differences between Support Vector Machines and Neural Networks. Neural networks do not the data that they are trained with present when they are used however they do have a the wight of all the neurons for multiple levels, depending on the number of neurons in the network this can be quite large. Support vector machine only have one set of calculations to classify the the data, a neural network has to calculate the activation function for each neuron and since multi level networks require the output from previous neurons before they are calculate this gives them a greater serial part to the calculation leading to a small speed up when pluralised.
The sum of the support vectors can also be done in parallel. I have used the example from Navidia (tutorials week 6) as this provides the best speed performance compared with other ways of doing this type of sum.

Implementation

I have written an SVM that uses the linear kernel on 2D matrices split into 2 arrays I did try to use one 2D array to hold the support vectors but I could not get the padding or flattening to work correctly within Cuda.

The serial implementation involved a using a for loop to go through the array of support vectors and apply the necessary dot matrix calculations, then add all of the results together.

In the parallel version of the application each thread performed the necessary calculations and was then synchronised before producing the sum. The sum was then returned to the host
I performed the timing of both implementations from the time they where called to the time they returned the value. This meant including the time taken for the device to copy the result back to the host. I believe this is the best way to time the implementations as the result of the parallel implementation is not very use full if it is still on the device. This way of timing the implementations would increase the time for the parallel version to complete the task. However by not including the time taken to copying the support vectors, test vector and the alpha to the device I have reduced the timed section of the total GPU function.

Comparison Discussion
In this report I used a SVM with twenty five support vectors, and the twenty five associated alpha values. Both implementations where tested with the same support vectors multiple time as we are concentrating on the speed up that the GPU can provide to the problem, and not the accuracy of the SVM.
As the application shows I achieved a speed up of 20% over the serial version. I Used a relativity small number of support vectors for testing but believe that with a greater number of support vectors the over heads of copying the result back to the host would become a small percentage of the total time and so lead to a greater speed up verses the serial version.

Emerging GPU applications
GPUs where originally used for gaming and digital design, but over the last few years have started to be used in science and research due there ability to do complicated mathematics relatively quickly and cheaply verses any CPU. Now GPUs have start to be included with in many consumer electronics, this was intended to give the users better graphics and gaming on the devices, however they can also be used to speed up other operations. Nividia recently launched CARMA witch is a version of Cuda for the arm platform. This could be used in the same way as Cuda on any desktop in sorting and math problems, likely around security, I.e. encryption and decryption of messages, calls or hard drives.
The Linux Kernel has recently start to use GPUs reduce the response time on some tasks similar to the use of the GPU within the consumer electronics market. With the launch of windows 8 Microsoft also lunched new libraries and updates to the .net run time environment that among other things maid lync able to use GPUs with out changing the code this means .net developers can take advantage of the performance GPUs can give without having to learn new languages or methods of programming. Microsoft also released a new C++ library called AMP (Accelerated Massive Parallelism) that can be included in any C++ application to provide GPU interaction and mathematical function done on the GPU.
With a wider use within all areas of computing GPUs are providing a cheaper alternative increasing any devices speed. This bleed of GPUs into every device is good not only for manufacturers and developers who can make there programs faster and more responsive but also to the consumer as GPUs are cheaper that CPUs and use less power in mobile devices. These three examples show that the new application for GPUs is, common computing.

Daniel Palmer 10265229

Refinances

Pricing of Cross-Currency Interest Rate Derivatives on Graphics Processing Units.
Author: Duy Minh Dang
Institution: Department of Computer Science University of Toronto IEEE
Year: 2010

GPU Acceleration of Object Classification Algorithms Using NVIDIA CUDA
Author: Jesse Patrick Harvey
Year: 2009