University of Glasgow | School of Computing Science

A Parallel Implementation of the Self Organising Map using OpenCL

Gavin Davidson

School of Computing Science
Sir Alwyn Williams Building
University of Glasgow
G12 8QQ

Level 4 Project — 27th March 2015

Abstract

The self organising map is a machine learning algorithm used to produce low dimensional representations of high dimensional data. While the process is becoming more and more useful with the rise of big data, it is hindered by the sheer amount of time the algorithm takes to run serially.

This project produces a parallel version of the classical algorithm using OpenCL that is suited for use on parallel processor architectures. The use of the Manhattan distance metric is also explored as a replacement for the traditional Euclidean distance in an attempt to decrease run times. The output from the parallel program is compared to that of a widely used package, SOM_PAK, to ensure validity. The parallel implementation is tested on a variety of architectures, including graphics hardware, to determine which parameters run times. A 10x speed up is demonstrated when compared with SOM_PAK.

Education Use Consent

I hereby give my permission for this project to be shown to other University of Glasgow students and to be distributed in an electronic format. **Please note that you are under no obligation to sign this declaration, but doing so would help future students.**

Name: _____ Signature: _____

Acknowledgements

I would firstly like to thank my supervisor Dr Vanderbauwhede for proposing such an interesting project and also for providing sound advice whenever it was needed.

I'd also like to thank Rachel for her constant patience while I continually talked about a subject she has no interest in and for her help in proof reading this report.

Contents

Chapter 1

Introduction

The idea of an intelligent machine is one that has long fascinated computer scientists and engineers that strive to mimic the skills and behaviors of humans. The field of artificial intelligence aims to achieve this through a combination of robotics and machine learning. While robotics provides the physical, motor skills necessary, machine learning is tasked with providing perception and decision making abilities to these machines.

There are many machine learning algorithms being pursued by those interested in the area [19, 35, 9]. Of these algorithms, the one that this project will explore is the Self Organising Map, originally proposed by Tuevo Kohonen [19].

1.1 Motivation

Since their inception, self organising maps have been used in genetics [6], climatology [27] data mining [16] and many other fields. The self organising map is able to reduce highly dimensional data to much lower dimensions while still preserving topology. It is this ability that gives the algorithm its value. With the recent rise in big data, and with scientific data in general, it is becoming more and more necessary to analyse massive, often highly dimensional data sets.

The drawback of the classical self organisng map algorithm is that when run serially on even the most powerful machines it can take a substantial amount of time to complete. This is true with even moderately sized maps. However, the algorithm itself is inherently parallel and is therefore well suited to an implementation designed for multi-core and many-core architectures.

Graphics hardware provides the architecture necessary to take advantage of the highly parallel nature of the self organising map. The rise of general purpose graphics hardware has been facilitated by frameworks like nVidia's CUDA [24] and Khronos's OpenCL [13]. These frameworks have made graphics hardware programmable and have opened them up to non-graphics related applications. This new programmability coupled with the fact that GPUs are inexpensive when compared to highly parallel CPUs has made graphics hardware increasingly popular when processing large data sets.

1.2 Aims

It is the aim of this project to study the classical self organising map algorithm and devise an efficient implementation that takes advantage of the highly parallel nature of the algorithm. The implementation will be suited

for use on many-core architectures, such as general purpose graphics processors. A further aim is the demonstration of a measurable speedup when compared to a widely used self organising map package, SOM_PAK [17]. This parallel implementation of the self organising map will be written using the OpenCL framework for heterogeneous architecture

1.3 Outline

Chapter 2 will explain the original self organising map algorithm in great detail and provide a more formal algorithmic specification. Background including applications, software packages and research work will be discussed in chapter 3 and chapter 4 explains the basic concepts of OpenCL and how to write OpenCL compatible code. Chapter 5 provides a detailed explanation of all software that was produced during the project and how they are all related to one another. Evaluations of the main program, *parallel_SOM*, are presented in chapter 6 and chapter 7 provides concluding remarks and ideas for future work.

Chapter 2

The Self Organising Map

A self organising map is a form of clustering or classification algorithm that involves projecting highly dimensional data into much lower dimensions and is an example of an artificial neural network [19]. The intention of these networks is that they somehow imitate the properties of biological neural networks, like the central nervous system. The network is made up of a set of neurons that are tuned to respond to certain input signals. In the case of the classical self organising map, the neurons are arranged in a rigid grid.

As a machine learning algorithm, the self organising map is classified as an unsupervised learning process. This categorisation is a result of the fact that it does not require an explicit training data set that has already been processed before being able to understand new data. The map is trained continually during the organising process with every input that it processes.

The main use of the self organising map is its ability to reduce highly dimensional data to a much smaller dimensionality, so that it may be represented on a two dimensional map. As such, the input to the map is a set of n-dimensional vectors.

2.1 Algorithm

The organising process can be split into 4 stages. An overview of the process is explained and then each of the four stages is clarified along with associated definitions.

It is important to note that while the algorithm is explained in this section, optimal parameters for each stage are not known. This is a problem within the field as optimal parameters for the organising process are specific to the data set being processed. In order to find the best parameters for a data set, trial and error is necessary.

2.1.1 The Process

The process starts with *map initialisation*. From there, the first input vector is considered and a *winner* from the map is found. Next, all of the neurons in the *neighbourhood* are updated according to the *neighbourhood function* and *learning rate*. This is then done for the each of the input vectors. Input vectors are considered many times during the process so exhausting the input vectors does not result in the termination of the algorithm.

After a set time and then periodically afterward, the *neighbourhood size* and *learning rate* are both reduced. These reductions allow for the map to become more finely tuned as the algorithm runs.

Termination of the algorithm happens when the map *converges*. Convergence can be defined by a number of mathematical criterion but in actuality it means that the neurons in the map have reached a stable state that accurately resembles the input data set. The criterion for convergence will not be explored in this work.

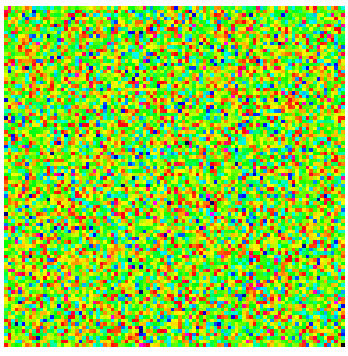
Algorithm 1 Self Organising Map Algorithm

```
1:  function MAIN
2:      map ← INITIALISE()                          ▷ Map is initialised with random vectors
3:      while CONVERGENT(map)=False do              ▷ Iterate until map convergences
4:          currentInputVector ← GETINPUT()
5:          winner ← FINDWINNER(map, currentInputVector)
6:          UPDATEMAP(map, winner)
7:          if currentIteration mod 100 = 0 then    ▷ After a chosen time (100 is arbitrary)
8:              REDUCENEIGHBOURHOOD                 ▷ Reduce neighbourhood size
9:              REDUCELEARNINGRATE                  ▷ Reduce learning rate
        return map

10: function FINDWINNER(map, inputVector)
11:     minDistance ← ∞
12:     for all neurons in map do
13:         distance ← EUCLIDEANDISTANCE(neuron, inputVector)
14:         if distance < minDistance then
15:             winner ← neuron
16:             minDistance ← distance
        return winner

17: function UPDATEMAP(map, winner)
18:     for all neurons in map do
19:         neighbourhoodCoeff ← NEIGHBOURHOODFUNCTION(winner, neuron)
20:         UPDATENEURON(neuron, neighbourhoodCoeff, winner)
```

2.1.2 Map Initialisation

The first stage in the algorithm is to initialise the map. A map of predefined size is created where the size is the number of neurons in the map. Each neuron is assigned a vector that has the same number of dimensions as all of the vectors in the input data set. The vector is sometimes called the *weight* of the neuron. The vector at every neuron is set to hold random values. Figure 2.1 shows an example of a map after initialisation, where each neuron is represented by a pixel.

Figure 2.1: Initial map example



2.1.3 Find a Winning Neuron

The algorithm works on a single input vector at a time. When each vector is considered, a *winning* neuron in the existing map must be found. The winning neuron in the map is the neuron whose vector is most closely related to the current input's vector. In the classical self organising map algorithm, the relationship between two n dimensional vectors is defined by the Euclidean distance between them.

$$Euclideandistance(\mathbf{p}, \mathbf{q}) = \sqrt{(p_1 - q_1)^2 + (p_2 - q_2)^2 + ... + (p_n - q_n)^2}$$

The neuron whose vector is the shortest Euclidean distance away from the current input vector is the winner.

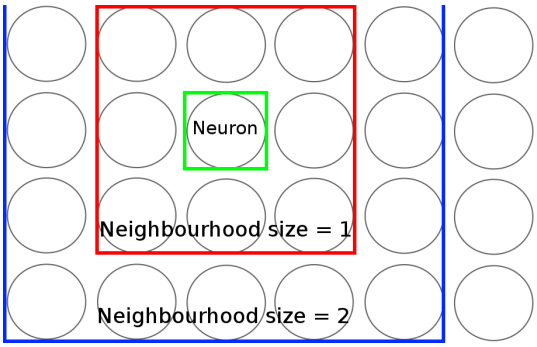
2.1.4 Update Neurons in Neighbourhood

In order to describe the update phase, first a *neighbourhood* and a *neighbourhood function* must be defined.

The neighbourhood of a neuron is simply an area on the map surrounding that neuron. A neuron is in the neighbourhood if it is within a defined distance away from the original neuron. This distance that defines neighbourhood size decreases to zero during the runtime of the algorithm. Neurons within the neighbourhood of the winning neuron are changed according to the neighbourhood function.

The neighbourhood function is often a Gaussian function. This function defines how much a neuron's vector is changed during an update according to how far away the neuron is from the winning neuron on the map. The winning neuron and its close neighbours will be changed the most, while the neurons on the outer edges of the neighbourhood are changed the least. Figure 2.2 shows an example neighbourhood. The neurons in the neighbourhood of size 1 will be changed more than the neurons in the neighbourhood of size 2.

Figure 2.2: Neighbourhood example



Once the winning neuron has been found, all of the neurons within the current neighbourhood must have their vectors updated. The vectors are updated so that they are closer to the input vector. How much each vector changes is determined by a combination of the learning rate and the neighbourhood function at this position in the neighbourhood.

2.1.5 Reduce Neighbourhood and Learning Rate

When the algorithm begins, the neighbourhood size and the learning rate at every neuron in the neighbourhood are maximal. Usually this means that the neighbourhood of every neuron encompasses every other neuron in the map. The effect of this is that, in the early stages of the organisation process, all of the neurons are affected by every update and form a vague initial organisation. From this initial organisation, it is necessary to more finely tune the neurons in the map to produce an accurate representation.

In order to produce this finely tuned map, the size of the neighbourhood and the learning rate for each neuron is reduced as the algorithm continues. At a set time after the start of the algorithm, the neighbourhood size for each neuron is decreased and so is the learning rate for the map. Reducing the neighbourhood size means that less neurons are affected during an update. Reducing the learning rate means that the neurons that are affected by an update are changed less than with a larger learning rate. Both the learning rate and neighbourhood size are reduced periodically as the algorithm runs.

Chapter 3

Background

3.1 Applications

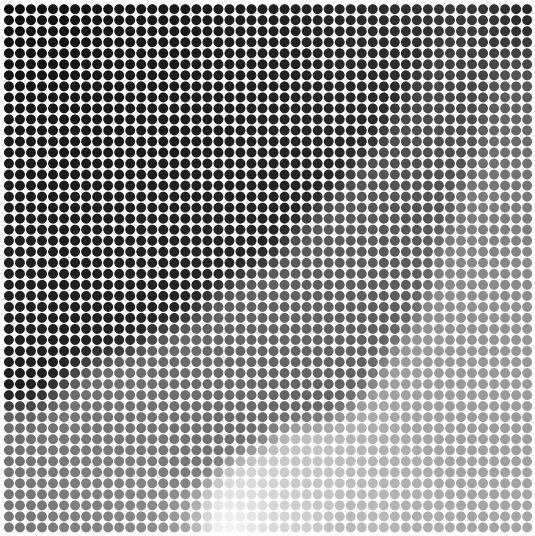
Since its original proposal, the self organising map has been widely explored for use in many areas. Work was done by Kohonen et al. [20] to implement a document classification system using a self organising map. The resulting system created maps using histograms of the vocabularies of a set of documents. The maps were then presented as HTML pages that allowed a user to explore related documents. In the largest experiment of the work, a map of 1 002 240 neurons was created to represent 6 840 568 documents and each input document was represented by a 500 dimensional vector. At the time (2000) these computations took 6 weeks to complete. Even with modern CPUs however, the computations involved to create a map of that size using inputs of that specification could take days.

Work carried out by Caridakis et al. [4] uses self organising maps to recognise hand gestures in real time. The system that is described in the paper processes images to determine position and trajectory values for a particular hand. These values are then used as the input to a self organising map which classifies the data. The organised map is used in the decision process to determine which gesture was performed. In order to ensure that responses from the system are real time, the self organising map is implemented using a parallel computing model, namely nVidia's CUDA [24].

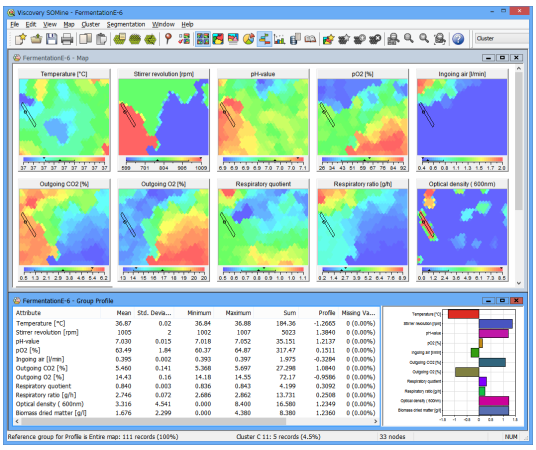
Self organising maps are used heavily in meteorology and climatology. According to Liu and Weisberg [21], the algorithm is used as a pattern recognition tool in the analysis of air temperature, precipitation, snow, humidity and sea ice among many others. The book goes on to explain that the self organising map has been used all over the world in numerous meteorological experiments and studies.

3.2 Self Organising Map Software

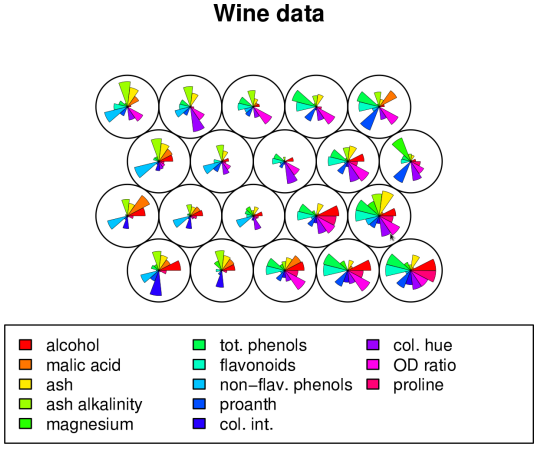
Possibly the most widely used self organising map package is the freely available SOM_PAK. The program was developed by Teuvo Kohonen, the original proposer of the self organising map, along with his colleagues at Helsinki University of Technology in the early 90's [17]. The package itself is written using C and seems to have compatibility issues with newer versions of the C compiler. For example, when SOM_PAK was installed during the research part of this project, it was found that the original authors had implemented functions that now exist in standard libraries. This caused clashes during compilation and the original code had to be changed slightly to avoid this. Once the program is installed, it works on the basis of running a number of trial maps and then measuring the *quantisation error* on each of the maps that are produced. The trial map that produces the lowest



(a) SOM_PAK



(b) Viscovery [30]



(c) Kohonen Package [36]

Figure 3.1: Visualisations of Maps from Various Software

quantisation error is output as the best map for that input data set. Along with the programs that perform the organisation and calculate quantisation error, the package also provides programs to create visual representations of maps. While the production of visualisations is a necessity for a good self organising map package, SOM_PAK also provides a program that takes a number of input vectors and shows where on a particular map they would end up. It is this extra program that provides some form of classification in SOM_PAK.

A self organising map package that seems to have been kept more up to date is *Viscovery* [29]. The development of Viscovery, like SOM_PAK, started in the early 90's. However, Viscovery's development took a more commercial approach and was funded by venture capitalists. While this commercialisation means that Viscovery is not free, it has allowed the package to continue to develop and grow into a suite of data mining programs. According to Viscovery's marketing literature [30], the package provides a cacophony of features for data mining and exploratory data analysis. These features include visual cluster analysis, correlation analysis, principal component analysis and a range of graph production abilities. The package claims to be intuitive even for users with little knowledge of statistical analysis while also being extensible and flexible enough for experts in the field. Viscovery has been used in research into document classification [7] and medicine [34] among other areas.

For those familiar with the statistical language R [10] there is a self organising map package called *Kohonen* [36]. The Kohonen package puts emphasis on visualisations and aims to provide functions for the production of self organising maps that are easy to use. Though free to use like SOM_PAK, the Kohonen package seems to

have many more features than its C based counterpart. Along with functions for the classical self organising map, the package also provides functions for variants of the original algorithm. Where visualisations are concerned, the Kohonen package produces a wide array of graph and map representations.

3.3 Related Work

Previous to this work, it has been widely appreciated that the self organising map is extremely useful in the analysis of large, highly dimensional data sets. It is also universally accepted that the time required to run the algorithm on such data sets is a hindrance. Due to this, much work has been done to create a self organising map program that organises data in a greatly reduced time.

A graphics hardware version of the self organising map is presented by Gajdo and Plato [11]. The work focuses on a version of the algorithm implemented using CUDA. The algorithm updates map vectors in parallel and also demonstrates a parallel reduction. The algorithm is evaluated in four experiments running on both a CPU and a GPU. Two experiments varied the size of the map, the others changed the dimensionality of the vectors. The results showed that changing vector dimensionality did not separate the CPU from the GPU until the number of dimensions was as high as 100. However, the GPU outperformed the CPU when the size of the map was made larger. The evaluation strategy used by Gajdo and Plato [11] is similar to the strategy that will be employed in this work, as is the use of a parallel reduction.

The system detailed by Prabhu [26] implements a self organising map in a context more traditionally associated with a GPU. The system is designed to recognise patterns in an input image. The input image is reduced somewhat through mathematical transformations before being processed by the self organising map in order to reduce complexity. The strategy employed by Prabhu[26] is designed for GPU use and involves processing more than one input vector at a time. After a winner for each input vector is found, a matrix representing winning neurons in the map is constructed and acts as a mask for the whole map during the update phase. In this case, the GPU is used in both finding winning neurons and updating vectors on the map. This strategy shows a marked speedup on a GPU when compared to a CPU in all tests performed. The paper mentions the overhead of transferring data between main memory and GPU memory as a concern in some of the tests. While the strategy Prabhu [26] employs is a successful one for the organising process, it is not the strategy that will be used in this work. However, the consideration of the overhead of data transfer between devices is one that this work will take into account.

Work by McConnell et al. [22] compares a number of implementations of the self organising map for multi-core and many-core hardware. The map is implemented in OpenCL [13] and CUDA [24] along with a version written using message passing. The strategy described by McConnell et al. [22] is very similar to the one described by Prabhu [26] in that many input vectors are processed at a time before the map is updated in batches. The value of McConnell's work [22] comes from the comparisons between parallelism frameworks that are presented. The work shows that all three implementations produced a speedup when compared to a serial version but also that the CUDA implementation outperformed the OpenCL implementation when run on the same platform. While all the tests presented by McConnell et al. [22] show CUDA performing twice as fast as OpenCL, CUDA will not be used in this work. It is the aim of this work to produce a self organising map program that can be run on varying architecture. If the program were to be implemented using CUDA then it would only be able to run on nVidia graphics hardware. Though more in favour of a CUDA implementation, the results of McConnell's work [22] still show that vast speedups are possible when using OpenCL to implement a self organising map.

A graphics hardware implementation of a varient of the self organising map is presented by Campbell et al. [3]. This variant of the original map is called a "parameterless" self organising map and eliminates the need to decide on the necessary parameters of the original algorithm (neighbourhood size, learning rate etc). While Campbell et al. [3] does not use recent parallel computing frameworks like OpenCL [13] or CUDA [24], the

strategy for parallelising is very similar to the one that will be employed in this work. The work shows that when the algorithm is run on a GPU it outperforms a desktop computer with ease. With larger map sizes, the GPU even out performs a super computer of the time (2005).

The self organising map algorithm is used by Strong and Gong [31] to group visually similar images. The desired use of the system is a way for users to navigate through a library of images. The hope is that users can easily find the image they are looking for by locating similar images and from there moving to the desired image. The organising process is parallelised using a GPU and OpenCL and shows a speed up of up to 37 times when compared to a sequential version running on a CPU. The work carried out supports the use of OpenCL and also shows examples of the use of larger maps and input dimensionality.

In Kohonen's book on self organising maps [18] a hardware implementation for the self organising map is suggested. This suggested implementation makes use of SIMD [28], a hardware component present in graphics hardware, to accelerate the self organising process. The book outlines a system where each neuron in a map is dealt with using a part of the SIMD and each input vector is broadcast across the SIMD. The distance from the input vector to each neuron is therefore calculated simultaneously. When the winner is found from these neurons, the updates to every neuron's vector also happen simultaneously. It is a form of this suggested model that will be implemented in this work.

A short discussion of the use of more efficient distance metrics is present in Kohonen's book [18]. The book explains that it would be possible to greatly reduce the computations necessary by using what is calls *city block distance* rather than the traditional Euclidean distance. City block distance is more commonly called *Manhattan distance* and is a simple operation involving only addition rather than the square and square root operations used by Euclidean distance. When this distance metric is applied to the *find a winning neuron* phase of the organising process the number of necessary computations is reduced. However, the book does not recommend this approach. It is stated in the book that the use of Euclidean distance leads to a more *isotropic* display of the map. This means that the layout of the neurons on the map are more uniform when comparing horizontal and vertical orientations. The book's negative view of Manhattan distance is supported by Marqués et al. [23]. In this case, the main opposition to Manhattan distance comes from the fact that it is shown to produce maps with slightly more *distortion*. Distortion in this context means that some of the topological information of the map is lost. However, this distortion is only slightly increased by the use of the simpler metric and Marqués et al. [23] accept that it may be more appropriate to use Manhattan distance for some implementations. The use of Manhattan distance as a way to accelerate the organising process will be explored in this work.

3.4 Discussion

It is clear that self organising maps have been widely explored since their original proposal. Their applications are numerous across many fields and disciplines where it is necessary to reduce huge, highly dimensional data to more understandable representations. The applications mentioned in 3.1 are just a small subset of the situations that the self organising map has been applied and, with the use of data mining on the increase, the number of applications of the algorithm are set to rise. However, in order to see the usefulness of the algorithm rise, it is necessary to make it much faster than the classical, serial version. A measured speedup allows for self organising map applications to become more real time and possibly even interactive. Faster applications may lead to the use of the self organising map in situations where the time taken to process data with the original algorithm made the output useless due to volatility of data.

Of the packages discussed in 3.2, none have support for many-core architectures like graphics hardware that would allow any sort of speedup when compared to a serial version. SOM_PAK has absolutely no support for parallel multi-core and many-core processors. This is simply due to the fact that it outdates the widespread use of these parallel architectures. Without further analysis, it seems that Viscovery does not make use of parallel

architectures like GPUs as this feature does not appear in marketing literature. It is a reasonable assumption to make however, that this more professional package will take advantage of the parallelism made available by the widespread use of multi-core CPUs. The third package mentioned, the kohnonen package for R, does not have direct support for the use of multi-core and many-core architecture. As a language, R does have support for these parallel devices but a version of the Kohonen package designed for these architectures is yet to be made available.

The work referenced in 3.3 is only a small portion of the research that has been performed in an attempt to create a fast self organising map program. The sheer amount of research shows that the currently available applications that perform the algorithm are inadequate for many situations. Most of the work into self organising map programs for a GPU have been done using nVidia's CUDA. Work by McConnell et al. [22] shows that using CUDA rather than OpenCL is the best approach when speed is the only concern. However, the use of CUDA limits the program to only being compatible with nVidia graphics hardware. Using OpenCL for a parallel applications means that the program is compatible with any architecture whose vendor has released OpenCL support. This list of hardware vendors includes Intel, nVidia and AMD [12].

All of the work mentioned in 3.3 suggests that a speedup compared to the sequential self organising map algorithm is highly possible using many-core and multi-core hardware, regardless of whether the program is written using CUDA, OpenCL or message passing. The suggestion of using parallel architecture is even present in Kohonen's 2001 book [18] which predates the widespread use of multi-core and many-core architecture.

Many different strategies for parallelising the original self organising map algorithm are presented by research into the area. The majority of strategies focus on processing data for all of, if not some of, the neurons in the map at the same time. Others process more than input at a time. As the original algorithm causes the neurons in the map to change with every input vector, it seems apparent that any parallel algorithm that truly duplicates the outcome of the original algorithm should be constrained to process inputs one at a time. This logical constraint forces the strategy to be followed in this work to be one of evaluating many neurons at once rather than many inputs at once.

Chapter 4

OpenCL

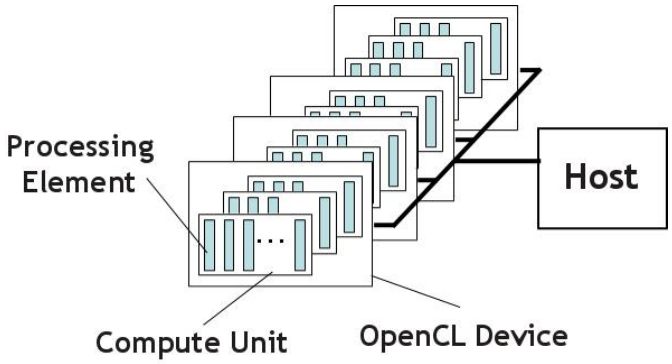
OpenCL is a framework for writing software that runs in parallel across *heterogeneous platforms*. In this context, *heterogeneous platforms* are different kinds of processors and this means that OpenCL allows a programmer to write software that is compatible with more than one type of processor or graphics processor. The OpenCL code requires little to no modification to be run on different platforms.

4.1 Basic Structure

OpenCL works on the basis of running the same instructions in parallel on large data sets. These instructions are called *kernels* and are explained in 4.2. In order to understand how these kernels are executed it is necessary to understand the device hierarchy that is present within OpenCL.

The base object in this hierarchy is the *host*. The host is a CPU based computer that handles the initialisation phase of the program as well as harvesting the final result of all of its operations. Contained in the host are *devices*. Each device contains a number of *compute units* which in turn contain a number of *processor elements*.

Figure 4.1: OpenCL Host Structure (from [33])

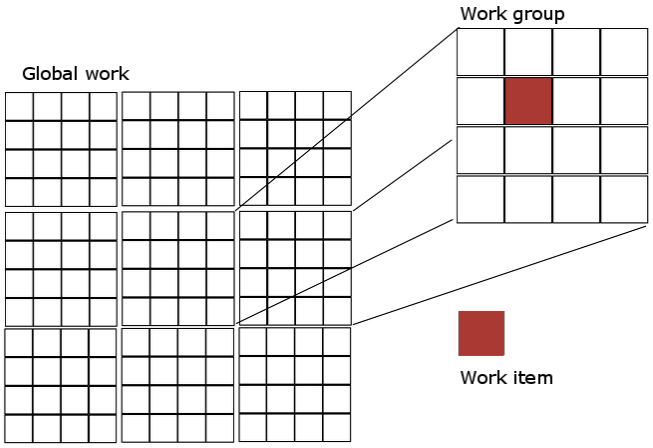


The host is responsible for allocating memory (explained further in 4.3), loading kernel code and assigning work to devices. Work is submitted to compute units as kernels (explained in 4.2) which are executed on processing elements.

Each individual element of execution is called a *work item* in OpenCL. Work items can be thought of as the single, smallest parts of a problem. For example, a single value to be processed. Work items are grouped into independent *work-groups* where a work group is an up to three dimensional array of work items that is executed in

parallel by the device. The size of the work group is limited to the size of CL_KERNEL_WORK_GROUP_SIZE which is the maximum number of concurrent threads that are possible for this device's compute units. That is to say, the work group can only have as many work items as there are threads to process them on a compute unit. Figure 4.2 shows the structure of global work in OpenCL. In the figure, each work group is a 4 x 4 matrix (a 2 dimensional array) and there are 9 work-groups. In order for this to be possible, the compute unit used in this situation must have at least 16 concurrent threads available.

Figure 4.2: OpenCL work structure



When writing programs that use OpenCL, there are two categories of code that must be considered. The first of these is the host code. The host code is the program that specifically runs on the host that deals with initialisation, distributing work and harvesting results, along with other things. The host code can be written in any language for which there is an OpenCL implementation available. For example, the host code in this work is written using C++.

The second code base that must be considered is the kernel code and will be discussed in 4.2.

4.2 Kernels

As mentioned before, OpenCL kernels are sets of instructions that are executed in parallel. These instructions are written in a variant of C99 and have mostly the same syntax. OpenCL C has some extra built in abilities, like extra mathematical functionality, but suffers from being more restricted in some cases, such as the fact that recursion is not possible and random number generation is not available [8].

Each kernel program is stored in a separate file, often with the extension *.cl*. During the initialisation phase of the host program, each kernel program is compiled using the OpenCL compiler and is stored as a *kernel* object in the host code. Once the kernel has been built, the arguments that the kernel needs to run are assigned to it.

Figure 4.3: OpenCL kernel to calculate Manhattan distance

```
__kernel void Manhattan_distance(
  __global float * input,              // __global keyword
  __global float * map,
  __global float * distance_map,
  int vector_length,
  int input_start_index)
{
  size_t tid = get_global_id(0);         // Global work item number
  int base_map_position = tid*vector_length;

  float sum = 0;
  for (int component = 0; component < vector_length; component++)
  {
    sum += fabs(input[input_start_index + component] -
      map[base_map_position+component]);     // 'fabs()' function
  }
  distance_map[tid] = sum;
}
```

In order to assign work to a device, a kernel must be queued. Queuing involves setting up work-groups (see 4.1) of appropriate sizes that represent pieces of the total work to be processed. It can aid understanding to think that when a kernel is queued many copies of that kernel program are made and assigned to each work item. In that case, every kernel is executed sequentially to process a single work item.

Figure 4.3 shows a kernel that calculates Manhattan distances. The first point to be made here is the way in which arguments are passed to the kernel. Passing arguments to kernels is done by the host code before the kernels are enqueued. In the case of array arguments, buffers are assigned (see 4.3 for buffers). From there, a number of other things are important to observe in this figure. Firstly, the use of the keyword *__global* in the arguments to the kernel. This keyword will be explained in 4.3.

Another observation is the use of the function *get_global_id(0)*. If the global work amount is thought of as an n dimensional array, *get_global_id(0)* returns the size of the first dimension. This means that *get_global_id(1)* returns the second dimension and *get_global_id(2)* the third.

In the particular case of the kernel in Figure 4.3, the kernel is run using a one dimensional global work size, meaning that the global work can be thought of as a normal, linear array. In turn, this means *get_global_id(0)* returns a value that is unique to every work item. This value is used to determine which elements in the *map* and *distance_map* arrays are accessed. There are similar functions that allow a programmer to access the dimensions of the current local work group.

A third observation is the use of the function *fabs()* that returns the absolute value of a floating point number. This is an example of the functions that are included as part of standard OpenCL. In C99, the function *abs()* performs the same action but is only available as part of the *math.h* header.

4.3 Memory Model

4.3.1 Memory Hierarchy

The memory model used in OpenCL reflects the hierarchical nature of the device structure that is employed. Figure 4.4 demonstrates this hierarchy.

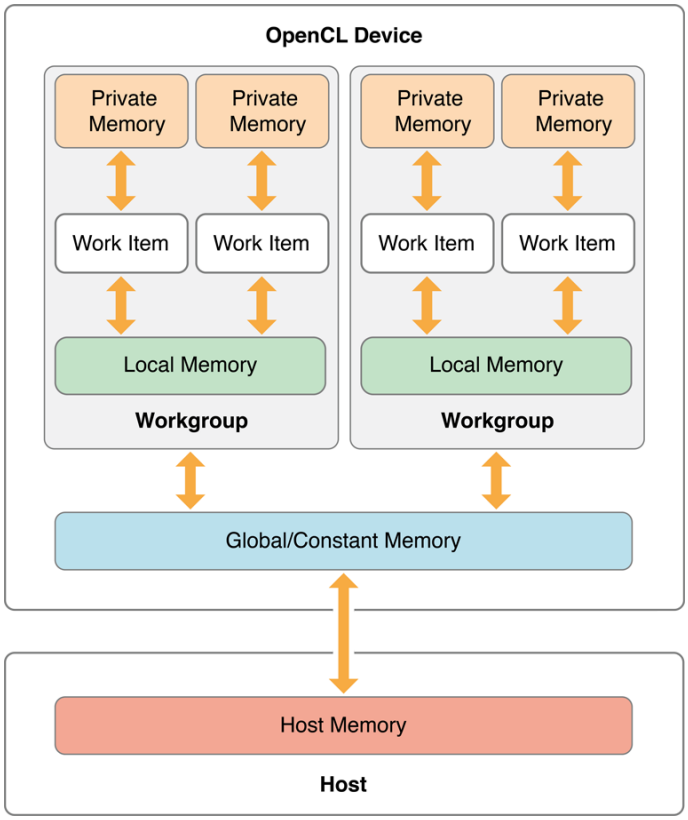
The smallest, most exclusive element of the memory model is the *private memory* of a work item. This is the memory that a single kernel uses for variables during its execution. In Figure 4.3 for example, the variable *base_map_position* is stored in private memory. The variable is only accessible by this kernel and only exists during the processing of the current work item.

Local memory is accessible to all kernels that execute as part of a single work group. In more hardware-centric terms, local memory is only accessible to kernels that are running on the same compute unit. The contents of this local memory only exists for as long as the work group is being executed. In order to designate local memory, the keyword *__local* is used. As an example from Figure 4.3, if the keyword *__global* were to be replaced by the keyword *__local* when defining the *distance_map* argument then the *distance_map* array would only be accessible to kernels in the same work group.

The peak of the device memory hierarchy is the *global memory*. Intuitively, global memory is accessible from all kernels that are executed while processing all work groups. The contents of global memory exist until they are manually dereferenced by the host.

The *host memory* is not present in this hierarchy as it is not directly accessible by any kernel during execution, it is only accessible by the host.

Figure 4.4: OpenCL Memory Hierarchy



4.3.2 Buffers

All memory transactions in OpenCL are explicit and must be done manually by the programmer. In order to pass data to a device, the host must create *buffers* of appropriate sizes to contain all of the values that the device will need. For example, if the device requires a set of 10 *floats* in order to execute, the buffer size is the size of a single float in *bytes* multiplied by 10.

Buffers for both local and global memory must be made and assigned appropriately as kernel arguments to avoid memory inconsistencies. When global memory buffers are made, the host can transfer values from host memory to the buffers. The host can also transfer values from buffers back into host memory. It is this ability to transfer data between host memory and global device memory that allows the original data to be passed to the device and the final result of all computations to be harvested by the host.

4.3.3 Barrier Synchronisation

In general, OpenCL attempts to remove the need for synchronisation between threads but sometimes it is necessary to coordinate for the sake of keeping data valid. *Barriers* provide the desired synchronisation in OpenCL. There are two types of barrier that can be used in kernel code, local and global.

When a local barrier is placed in a kernel, the effect is that no kernel within a work group can continue past that barrier until all other kernels in the group have reached that barrier. That is to say that until all the kernels have executed the program on their work unit up to the point where the barrier is placed, the program cannot continue. The use of local barriers means that any changes to local memory before the barrier can be guaranteed to have happened when code after the barrier is executed.

A global barrier is very similar, other than the fact that when a global barrier is placed in kernel code, all kernels in the global work must reach the barrier before any of them can continue. Global barriers should be used over local barriers when changes to global buffers are made.

4.4 Example OpenCL System

This section will attempt to give an outline of how a simple OpenCL program works.

Firstly the host must determine which devices in the system it can access and choose one. After choosing a device to run the kernels on, a context is set up so that it can be used when initialising other structures that OpenCL programs need. When the device has been set up properly, a command queue for that device is made that allows the host to pass work to the device.

The next thing the host does is initialise buffers to correct sizes for kernels to use. The data that kernels will need to execute is loaded onto appropriate buffers while those buffers that will have data loaded into them by the kernels are left just initialised.

Next, files containing kernel code are read by the host and compiled. At this point, if the code in the kernel files does not compile for any reason the host program ends and informs the user of its failure. In the case that the kernel code does compile, arguments are assigned to the compiled object. When the host code is assigning an array argument it actually assigns a reference to the buffer that was built to hold that array. For constants, the argument for the kernel is simply set to the value rather than a buffer.

Once arguments have been set, it is time to enqueue work to the device. To do this, the command queue must be passed information on the kernel that is to be run as well as a set oof values that specifiy global and local work sizes. The queue is also passed an *Event* object that it uses to signal when the work has been finished. For those familiar with more traditional concurrency techniques this *Event* object is essentially a condition variable that is signalled by the command queue when all of the current batch of work has been finished. When the host has given work to the command queue, it waits on the *Event* object before continuing.

When the command queue signals that the work is done, it is the time for the host to harvest the results of the computations. This is done by queueing a read request from a particular buffer and supplying information about where to put the read values and how many bytes should be read from the buffer. The results are now stored in the location specified to the read request and can be accessed normally by the host code to do whatever it needs to with it.

Chapter 5

Implementation

In the course of this work a number of programs were produced. Along with the main program, the parallel self organising map, there were some smaller, utility programs produced. All of the applications that were produced by this work are explained in this section, along with explanations of design choices for the main program.

5.1 Single Threaded SOM

The main result of this work is a program called *parallel_SOM*. Parallel_SOM is written using C++ and uses OpenCL to provide parallelism on multi-core and many-core processor architecture. However, before OpenCL and GPU programming could be even considered it was necessary to develop a single threaded version of the program.

For consistency, this section will follow the same structure as section 2.1 - where the original organising algorithm is explained.

5.1.1 The Process

As explained before, the algorithm begins with *map initialisation*, followed by continually loading an input vector, *finding a winning neuron* in the map for that input vector and then *updating neurons in the neighbourhood* of the winning neuron. This process continues until the program's termination condition is fulfilled and periodically there is a *reduction of the neighbourhood and learning rate*.

There are two differences in the process implemented here when compared to the original specification of the algorithm in 2.1, one rather major and the other less so. There is also an extra constraint applied to the whole process.

The extra constraint in the process is that the map that is output by the process must be square. That is, the x dimension and y dimension of the map must be equal. This is simply due to the way that some features have been implemented and this rule would likely be removed in later versions of the program.

The major difference in this implementation is the termination condition for the algorithm. In the original definition of the algorithm, the process terminates when the map *converges*. In simple terms, the process ends when the neurons in the map change very little between processing input vectors or stop changing all together. This implementation does not support any notion of convergence but rather terminates when a certain number of

iterations is reached. The justification for this decision is taken from the implementation of the self organising map package SOM_PAK as the same concept of maximum iterations for termination is used there. The value for this maximum number of iterations is calculated as follows:

$$max\_iterations = (map\_x\_dimension) * (number\_of\_input\_vectors) * 20$$

The second, almost trivial difference between this implementation and the original algorithm is the inclusion of the idea of *trials*. That is to say, the implementation allows a user to specify how many times the program should run the whole algorithm to make a map before deciding which of the maps produced is the best and outputting it. The *best map* is defined as being the map that has the lowest *quantisation error* when compared to the original dataset. After each trial, if the map that has just been produced is deemed to be better than the previous best map then the best map is over written with the new map. At the end of all of the trials it is this *best map* that is output.

Quantisation error will be explained in 6.3.1. This is another feature that is present in SOM_PAK and was seen as a useful feature.

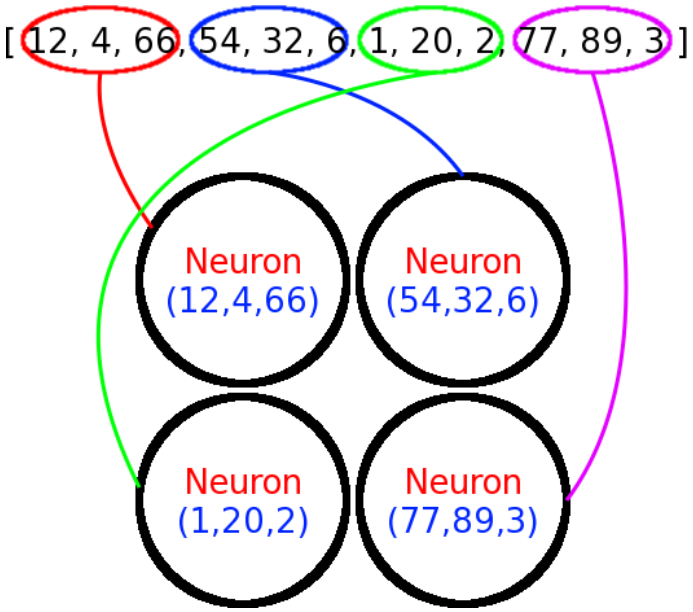
5.1.2 Map Initialisation

In the classical organising algorithm, the map initialisation phase simply involves assigning random vectors to every neuron in the map. This simple initialisation takes place in this implementation with one small constraint. The constraint here is that only values from a certain range can be used as the components of the vectors. This range is calculated before the map is created by finding the maximum and minimum of all of the vector components in the input data set and using these values as the maximum and minimum of the range.

In the single threaded version, the map is stored in a one dimensional array of *floats* that has length equal to the number of input vectors multiplied by the dimensionality of the input vectors. This simplistic data structure was chosen because the access time for a static array cannot be beaten and time is of the utmost concern. This array simply stores all of the values from the vectors associated with the neurons. For example, if the dimensionality of the vectors is *n* then the first *n* elements of the map array represent the first neuron, the second *n* elements of the map array represent the second neuron and so on.

The topology of the map is defined in a line by line basis. That is, if the width of the map is *w* then the first *w* elements of the map array represent the top row of the map. The second *w* elements represent the second row and the last *w* elements represent the bottom row of the map. The first element of the map array is the top left corner of the map and the last element is the bottom right corner of the map. Figure 5.1 attempts to further explain this idea.

Figure 5.1: Map Topology from Map Array



5.1.3 Find a Winning Neuron

To find a winning neuron for a particular input in the original organising process, the input vector is compared to every neuron in the map and the distance between the two vectors found. This distance is found using the Euclidean metric shown below.

$$Euclideandistance(\mathbf{p}, \mathbf{q}) = \sqrt{(p_1 - q_1)^2 + (p_2 - q_2)^2 + ... + (p_n - q_n)^2}$$

In this implementation, it was decided that the use of Euclidean distance was an unnecessary drain on processor resources as it involves a number of squaring functions and a square root function. The alternative is *Manhattan distance*. Manhattan distance is a simplification where the distance between two points is calculated by finding the differences between all of the components of the two points and then summing those components, rather than using complex square and square root operations.

$$Manhattandistance(\mathbf{p}, \mathbf{q}) = (p_1 - q_1) + (p_2 - q_2) + ... + (p_n - q_n)$$

As before, the neuron whose vector is the minimum distance away from the input vector is the winning neuron. When this winner is found its position in the map array is returned as an integer. This integer is the index in the map array at which the first component of the neuron's vector is contained. It follows that the consecutive elements of this index are the other vector components.

5.1.4 Update Neurons in Neighbourhood

The neighbourhood of a particular neuron is the area around that neuron that contains a number of other neurons from the map. In the beginning this neighbourhood covers most of the map and reduces to be much smaller around the original neuron. In the original algorithm, the neighbourhood function is used to determine how much a neuron should change during an update, according to how far away it is from the winning neuron. That is, neurons that are further away from the winner are changed less than those closer to the winner. The neighbourhood function's result for a particular neuron is combined with the learning rate at that time to form a multiplier that is applied when updating that neuron.

Neighbourhood Function and Learning Rate

The neighbourhood function and learning rate are contained within one concept in this implementation, another one dimensional, static array of floats. The array, the *neighbourhood multiplier array*, is of a length equal to the x dimension of the map (or the y dimension, since they are constrained to be equal) and is populated when the program starts with values from a Gaussian function.

$$f(n'hood) = \frac{1}{gaussValue\sqrt{2\pi}} * e^{(-\frac{(n'hood/5)^2}{2*gaussValue^2})}$$

The *n'hood* value is a value representing how many steps the current neuron is away from the winning neuron (The neighbourhood size value in figure 2.2). The *gaussValue* here is a constant that has been determined to be optimal using trial and error. This value helps adjust the Gaussian curve to have desirable properties for its use here. Each position in the static array mentioned above is calculated by using the index value for that position as the *n'hood* value in the above function. Each element in the array therefore holds a multiplier for a neuron that is in the neighbourhood of size equal to the index of that element. That is to say that the multipler sorted at index *n* in the array is the multiplier that is applied when updating a neuron that is in a neighbourhood of size *n* of a winning neuron.

The Update

When the update to a neuron actually happens, each component of the neuron's vector is considered individually with the corresponding component of the input's vector. The idea behind the update here is that the current neuron's component is updated to be a little bit closer to the value of the input's component. The difference between the neuron's component and the input is found and this value is multiplied by the value supplied by the neighbourhood multiplier array at the position equal to the neuron's current position in the neighbourhood. Every component in a neuron's vector is updated in turn. The calculation is shown below.

$$neuron_{update} = neuron_{original} + (input - neuron_{original}) * neighbourhoodMultiplier$$

In this implementation, all neurons are considered in every update, regardless of whether they fall inside the neighbourhood. This is where the size of the neighbourhood multiplier array comes into the fore. The array is the size of the x dimension (and the y dimension, since they are equal) so that every possible neighbourhood is accounted for in the array. When a particular neuron falls outside of the neighbourhood, the value at the appropriate position in the neighbourhood multiplier is zero. This is explained in 5.1.5.

5.1.5 Reduce Neighbourhood and Learning Rate

It is necessary to reduce the neighbourhood size and learning rate throughout the runtime of the algorithm to ensure that more accurate training of the map occurs as time progresses. Since the neighbourhood, neighbourhood function and learning rate are implemented using the same concept, a static *neighbourhood multiplier* array, the reduction in neighbourhood and learning rate is performed in one simple step - all of the values are shifted into the element before and an element with value 0 is added on the end. The first element in this case is removed. This produces the desired affects of reducing the neighbourhood size and lowering the learning rate.

The neighbourhood size is effectively reduced because an element that was previously non zero is now zero and neurons that attempt to update using that value as the *neighbourMultiplier* will not change anything. The size of the array does not change either.

Due to the fact that the array is populated using a Gaussian function, every element is smaller than the one that appeared before it, with the first element being the largest. This means that the effect of shifting all the elements in the array is that every element gets a lower value than it had previously. This is how the learning rate is reduced.

5.2 Adaption to OpenCL

Once the single threaded version was complete, it was time to adapt the code to work with OpenCL. The first step here is to consider which parts of the algorithm are easily parallelised.

5.2.1 What to Parallelise?

As was discussed in 3.3, the usual strategy for parallelising a self organising map algorithm is to have the neurons of the map considered in parallel. That is to say, every neuron processes the same input at the same time. This is the strategy that was be adopted here. Three kernels were specified for the three elements of the program that are run in parallel.

The first kernel calculates the Manhattan distance between every neuron and the current input vector and stores these distances in an array. The second kernel uses this distance array to determine the minimum distance that was found and determines which neuron is the winner. This winning neuron value is then used by the third kernel to update the map appropriately.

Manhattan_distance.cl

The Manhattan_distance.cl kernel, as the name suggests, calculates Manhattan distances. As input, the kernel receives the whole *input* data set as a static one dimensional array structured in the same way as the map array. In this case, the whole input data set contains every input vector to the program. As well as the input data set, the *map array*, the number of *dimensions in each of the input vectors* and an integer representing where the *current input vector starts* in the input array are passed to the kernel. The final argument in the kernel code is another static, one dimensional array called *distance_map*. During the runtime of the kernel this array will be populated with distance values.

When this kernel is queued by the host, the number of work items is equal to the number of neurons in the map. As such, each instance of the kernel is responsible for finding the distance between a single neuron and the current input vector. The calculation is very simple. The first step is using the *global work item id* and the number of dimensions in the input vector to determine the starting index for this particular kernel's work item. That is, since the map array is one dimensional, the neuron's vector that this kernel is meant to consider starts at the element stored at the index equal to *work item id* multiplied by *vector dimensionality*.

Once the base has been determined, the sum of the differences between each of the components of the current input vector and the current neuron's vector is calculated as described previously. The result of this calculation is stored in the *distance_map* array at the index equal to the global work item number.

The distance map array is used by another kernel, later in the cycle of the program.

min_distance.cl

This kernel takes arguments that are the *distance_map* calculated by the Manhattan distance kernel, a integer constant *chunk_size* and two global, one dimensional, static arrays of floats, *winner_index_array* and *winner_distance_array*. The last two arguments are local, one dimensional, static arrays of floats *local_winner_index_array* and *local _winner_distance_array*. This kernel is an example of a reduction and the calculation of minimum distance is performed in stages.

The integer *chunk_size* is calculated as the number of work units that each compute unit on a device should be given so that each unit has roughly the same amount of work to do. The first stage in the reduction uses the processing elements of the device and starts by determining a *local_chunk_size* by dividing the *chunk_size* by the size of the local work group. This value is the amount of work units each processing element must deal with. The kernel determines a starting point for the *distance_map* array using it's global id and iterates from that start point until a point that is *local_chunk_size* later. The minimum value found by the kernel in *distance_map* is stored in one of the local arrays and the index of this minimum distance is stored in the other. This first stage ends with a local barrier synchronisation (see 4.3.3) forcing the work group to come to the same point. At this point, the local arrays contain a set of candidate minimum distances.

The second stage is carried out by each compute unit. After the local barrier, the minimum value from the local distance array is found and stored it in one of the global arrays at a position specified by the work group id. The index of this minimum distance is stored in the other global array.

The final calculations are carried out by the host. Firstly, it reads the values from two global arrays that were populated by the kernel. These arrays are both of length equal to the number of compute units on the device. Next, it must find the minimum distance value from the appropriate array. Upon finding that value, the related index value from the other array is stored as the value of the winning neuron.

update_weight.cl

The third kernel, update_weight.cl, handles changing the vectors associated to each neuron. These vectors are sometimes called *weights*, hence the name of the kernel. As input, this kernel takes the whole *input data set*, the *map array*, the number of *dimensions in each of the input vectors* and an integer representing where the *current input vector starts* in the input array - the same inputs that the Manhattan distance kernel takes. As well as these inputs, the update weight kernel is passed an integer representing the *winning neuron* in the map, an integer representing the *dimensions of the map* and the *neighbourhood multiplier array* mentioned in 5.1.4.

Like the manhatan distance kernel, each instance of this kernel is to process one neuron of the map. As such, when the kernel is enqued the number of work items is equal to the number of neurons in the map.

The first thing that needs be done by the kernel during execution is determining which work item to work on. A call to *get_global_id(0)* does this since the global work is only singlely dimensional (this function is explained in 4.2). The next stage is using this global work id and the integer representing the winning neuron to determine what neighbourhood the current neuron is in. A code snippet showing this calculation is shown in figure 5.2. The figure shows that this small function determines the x and y coordinates of the current neuron and the winning neuron in the map using modulo arithmatic and integer division. From there, the function calculates the diffence between each of the components of the coordinates and returns the maximum of the two. This has the affect of creating the square neighbourhood shown in Figure 2.2

Figure 5.2: Function contained within update_weight.cl to determine a neuron's neighbourhood

```
int squareNeighbourhood(
  int winner_index,
  int map_side_size,
  int current_id)
{
  int neuron_x, neuron_y, winner_x, winner_y;
  neuron_x = current_id % map_side_size;
  neuron_y = current_id / map_side_size;
  winner_x = winner_index % map_side_size;
  winner_y = winner_index / map_side_size;

  return max(abs(neuron_x-winner_x), abs(neuron_y-winner_y));
}
```

Once the neighbourhood has been determined, the calculation shown in 5.1.4 can be carried out on each of the components in the current neuron's vector. This is done with a for loop that updates each position in the map array as the new value is calculated.

5.2.2 Host Program

In 4.1, it was explained that there are two code bases that must be considered when writing for OpenCL: the kernel code and the host code. The host code is the code that runs on the normal CPU while the kernel code runs on the parallel processors and does the bulk of the work. The host code deals with the initialisation phase of the program as well as enqueueing work to devices and then harvesting the results of computations. The host code in this work is written with C++.

Initialisation

Since the basis of OpenCL is passing work to devices for them to process, the first stage in the host program is to set up the device to be used. To do this, the system is queried for a list of OpenCL compatible devices and the host then chooses one of the devices from the list. With the device in mind, a context must be set up. Contexts differ slightly between types of device (CPU, GPU, Accelerator etc.) and applying the wrong context to a device will cause the program to crash. The context is used to set up all manner of constructs necessary to run an OpenCL program, such as buffers, command queues and kernels.

In the initialisation phase all buffers are made and the buffers that hold the map array and the whole input dataset are populated. All three kernel files are then loaded and compiled in preparation for being assigned arguments by the host. In the runtime of the program, while the contents of arrays and buffers may change, the buffers themselves are the same buffers. This means that in all cases of using buffers as arguments, the arguments to each kernel need only be assigned once. This is due to the fact that buffers can essentially be thought of as memory locations in that the contents of the buffer may change but the location of the data is always the same. In the case of assigning values as arguments rather than buffers, this is not true. There are some cases, such as the map dimension value handed to the update weight kernel, where these values are constant for the whole of

the runtime of the program and so do not need to be changed but there are others where a kernel argument must be reassigned every time a batch of work is queued to the device. Examples of values that need to be reassigned continuously are the integer representing the winning neuron and the integer representing where the current input vector is stored in the whole input array.

Enqueueing Kernels

Since the self organising map algorithm is a very repetitive, cyclic process it is obvious that there are many instances of enqueuing kernels in the program. Every time a new input vector is considered three batches of work are queued, one for each kernel to process the data.

The first kernel to be enqueued is the Manhattan distance kernel. Before this is done however, the argument representing where the *current input vector starts* is set to suit the current input. The kernel is then enqueued and the host waits for the batch to complete.

When the Manhattan distance kernel batch is complete, the minimum distance kernel is enqueued. This kernel does not need any new arguments and its main source of input is a buffer that will have recently been populated by the Manhattan distance kernel. The host program waits for this kernel to complete but, before enqueueing the third kernel, must now do some work of its own.

The next phase in the runtime involves the the host program reading the two buffers that are populated during the runtime of the minimum distance kernel. One of these buffers contains a set of integers that represent a set of possible winner neurons, the other contains the distance values calculated by the Manhattan distance kernel. The two buffers are paired so that one buffer contains the index of a certain possible winner and the other contains the distance for that winner. Both of these values are stored in the same position in each buffer. It is the job of the host to search the distance array for the minimum value. When this minimum value is found, the corresponding integer from the possible winner buffer is accessed and used as an argument to the update weight kernel.

The final kernel to be enqueued is the update kernel. Using the integer representing the winning neuron, this kernel updates all of the appropriate neurons in the map.

Harvesting Results

Throughout the runtime, the map buffer is updated continually. However, once the host has loaded the map buffer at the beginning of the process it does not access it again until the whole process is complete. At this point, when the process's termination condition has been reached, the host program reads the whole map buffer into an appropriately sized array of floats. This array is then examined to determine whether it is the best map so far (see 5.1.1).

If there are more trails still to be run (see 5.1.1 for trials), the whole process starts again from map initialisation. Otherwise, the program outputs the best map and ends.

5.3 Visualisation

It has been mentioned previously that a self organising map package is made infinitely more useful when some form of visualisation tool is included. It is for that reason, and for the fact that this visualisation tool was extremely useful for debugging during development, that a lightweight visualisation tool was developed in the course of this work.

5.3.1 HTML Visualiser

The first version of the visualisation tool was written to take an array of floats representing a map and output a file containg html code. The HTML produced was essentially a table that had a cell for every neuron in the map. Each cell's background colour was changed to represented the vector stored at that neuron. The original version of the program was written to produce HTML because the program itself could be developed quickly and easily thanks to previous knowledge.

The visualising code was written in C++ and the header file to access it was very simple. The header only contained 2 functions that do the same thing but take different arguments. Figure 5.3 shows a condensed version of the header file.

Figure 5.3: Condensed header file for DrawHTMLMap.cpp

```
#include <vector>
// ... other includes
#include <iostream>

// Class used to draw a html representation of a map

int drawMap(vector< vector<float > >, string);
int drawMap(float *map, int map_size, int vector_length, string filename);
```

Calculate Colours

It is the intention of the program to produce a representation where the colours on the map vary in such a way that the whole range of neurons in the map are appropriately shown. That is to say, the range of values that the colours represent follow exactly the range of values that are contained in the vectors of the neurons in the map. The values that are used to calculate colours are the distances of each of the neuron's vectors from the origin. For example, if the map vectors are three dimensional, the Euclidean distance from *[0,0,0]* to each of the vectors is calculated and that value is used to calculate colours values.

The colour values themselves are encoded as RGB values. There are three values where one of the values represents the red value, one the green value and the third the blue value for that colour. To make these three dimensional colour values, the range mentioned earlier was split into six smaller ranges of equal size. Whether or not the distance value for each of the neurons falls within each of the ranges determines how the colour is made. If a value falls into a particular range then two of the RGB values are set and the third varies according to the value. For example, if the distance value falls into the first range the red and green values are set to 0 and the blue value is set according to the value. If the distance value falls into the fourth group, the green value is 255, the blue value is 0 and the red value varies according to the distance value. In order to use the distance value, it is adapted to be within the range of the colour values, 0 to 255. Figure 5.4 shows how the colours are calculated and also the start and end values for that range of colour.

Figure 5.4: Table showing how colour values are calculated

	Red	**Green**	**Blue**	**Start Colour**	**End Colour**
Range 1	0	0	*dist*	0, 0, 0	0, 0, 255
Range 2	0	*dist*	255	0, 0, 255	0, 255, 255
Range 3	0	255	255 - *dist*	0, 255, 255	0, 255, 0
Range 4	*dist*	255	0	0, 255, 0	255, 255, 0
Range 5	255	255 - *dist*	0	255, 255, 0	255, 0, 0
Range 6	255	0	*dist*	255, 0, 0	255, 0, 255

Producing HTML

Once the colour values had been calculated, the production of HTML was simply a case of building a string that was valid HTML and then outputting that string to a file. The start of the string defined the start of the table and then every neuron in the map was iterated through and a cell produced for each. New table rows were started at appropriate times using a map dimension value that is passed to the program.

5.3.2 PPM Visualiser

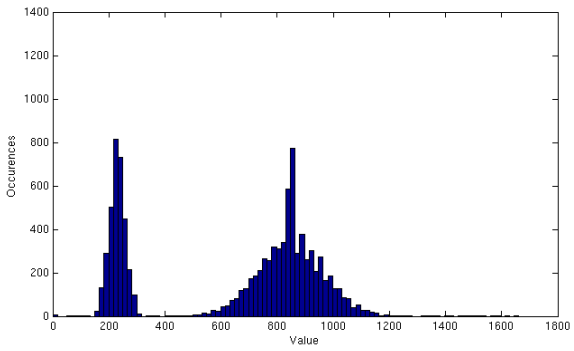
The HTML version of the visualiser tool was put together quickly and served well as a debugging tool in the early stages of development. However, the HTML table is not an efficient way to show a graphical map and so a new version had to be developed that used a more appropriate encoding. The format that was chosen was PPM. The choice here was due mainly to it's simple encoding. It meant that very little changes were necessary to the HTML version of the program to make it produce PPM files.

In order to produce the PPM files it was necessary to produce RGB colour values for each neuron on the map as it was for the HTML version. This meant that the same code was used here. The production the PPM image code is handled by some a program that was supplied by the advisor to this project.

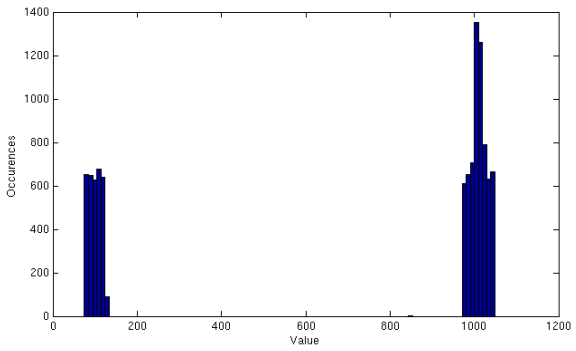
5.4 Data Generator

In the closing stages of development for this work, evaluation become a concern. In order to evaluate the self organising map program well, it was necessary to use data sets with certain properties. These properties were related to how many *clusters* of data were present in the data set and how these clusters were arranged. Tests would be carried out in the evaluation stage to see whether the output from the main program shows the expected properties that these data sets were built to have. Figure 5.5 shows histograms of example data from the program. Each histogram shows the expected properties of the data.

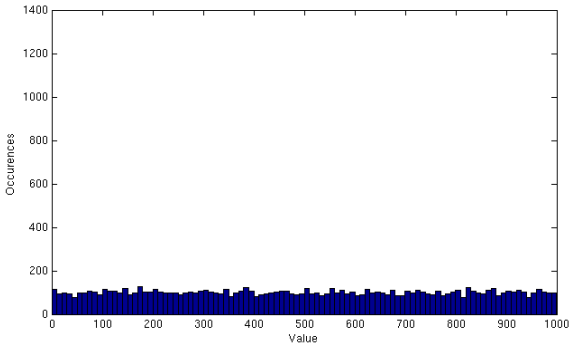
The generator was written using C++ and produces three types of dataset. When the program runs it requests a set of parameters. There are certain parameters that apply to all three data types while others only apply to specific types. The first command defines the data set type, *"uniform"*, *"cluster"* or *"Gaussian"*. From here, the *number of vectors* is requested along with *vector length*, *output file name*, *maximum value* for vector components and *minimum value* for vector components.



(a) Gaussian clustered data



(b) Uniform clustered data



(c) Uniform random data

Figure 5.5: Histograms showing distribution of outputs from data generator

5.4.1 Uniform Random

For uniform random data, only the standard parameters mentioned above are needed. Uniform random, in this case, means that the dataset that is produced is made up of vectors that are randomly generated inside of the range specified by the maximum and minimum value parameters. The vectors are *uniform* because they form no intentional patterns when considered as a large data set.

To produce the uniform random data set, every component of every vector is calculated using the code snippet in Figure 5.6.

Figure 5.6: Code snippet for uniform random data set generation

```
for (int i = 0; i < (num_vectors * vector_length); i++){
    output[i] = (rand()/(float)RAND_MAX) * range + min;
    // 0 < (rand()/(float)RAND_MAX) <= 1
}
```

5.4.2 Uniform Clusters

For clustered data, the standard parameters plus a *number of clusters* are required. Here, uniform clusters means that the dataset has a set number of clusters where the vectors in each cluster vary in a uniform random way. For

example, if a single cluster was to be examined there would be no intentional patterns in the vectors.

Each of the clusters contains an equal amount of vectors. In order to build a cluster, a base value is generated at random using the same technique shown in Figure 5.6. From there, more random values are calculated to be inside a *maximum variance* value and added to the base value to produce vector components.

Figure 5.7: Code snippet for uniform clustered data set generation

```
// Assume variables have been previously initialised
int cluster_size = (array_size * vector_length)/clusters;
for (int outer = 0; outer < clusters; outer++){
    centre = (rand()/(float)RAND_MAX) * range + min;
    for (int inner = 0; inner < cluster_size; inner++){
        component =  (rand()/(float)RAND_MAX) * max_variance + centre;
        output[inner + (outer*cluster_size)] = component
    }
}
```

5.4.3 Gaussian Clusters

The final data type, Gaussian clusters, requires a number of *clusters* and a *spread* value along with the standard parameters. The data that is produced is clustered, but rather than the clusters being uniformly random they are generated using a Gaussian curve. The *spread* value is used essentially as a measure of resolution and will be explained further.

To make Gaussian clustered data, the first step is to define a Gaussian function. The Gaussian function is then used to populate an array of size *spread* where each element in the array is the output from the function when that element's index is passed in. This Gaussian array technique is the same one that is employed in the implementation of the self organising map program. Each value in the array is now normalised so that the sum of the elements is equal to the number of desired values to be produced for that cluster. The number of values to be produced is *number of vectors* multiplied by *vector length* divided by *clusters*. Now that the distribution of values has been set up, it is time to produce the clusters.

As with the uniform cluster generator, a base for the cluster is defined first. The base is a random value within the desired range for the output. The function then iterates over the array containing the distribution values. For each element of the array, a value is produced by adding a fixed offset (defined by the index of the element in the array is being processed) and then a random value to the base value. The same is done again but the offset and random value are subtracted from the base value. The offset value is calculated so that it increases with the index of the element being processed. This is done a number of times equal to the value held at the current element in the distribution array being processed. This process is shown in pseudocode in figure 5.8 The result is that as the offset increases, and values are further from the base, less values are made. The product here is a Gaussian distribution of values.

Figure 5.8: Pseudocode snippet for Gaussian clustered data set generation

```
gauss_array = buildGaussArray(spread);
for (gauss_value in gauss_array){
  for (count in range(0, gauss_array[gauss_value])}{
    newValue = centre + (gaus_list_position*offset) + randomValue;
    output[current_index] = newValue;
    current_index++;

    newValue = centre - (gaus_list_position*offset) - randomValue;
    output[current_index] = newValue;
    current_index++;
    }
  }
}
```

This process is repeated for each cluster. The *spread* input parameter defines how *wide* each cluster is. That is, a spread value of 1 would produce clusters that were completely uniform, where a large spread value would produce clusters that better show the shape of a Gaussian function.

Chapter 6

Evaluation

Evaluation of the product of this work was performed on mostly quantitative basis. Most tests performed aimed to measure how much of a speed up is achieved when comparing to SOM_PAK. Were there more of a user interface to asses, a user study may have been appropriate.

The first evaluation however, was to check whether the product of parallel_SOM was Reliable. Reliability was determined by comparing the results of parallel_SOM with the results from SOM_PAK running with the same parameters.

There were three phases in the speed evaluation. The first phase involved selecting a base test and varying parameters from there. The second phase aimed to test performance with bigger maps. The final set of tests performed were concerned with determining how much of a speed up, if any, was brought about by the use of Manhattan distance over the classical Euclidean metric.

6.1 Architectures

During this evaluation, parallel_SOM was run on three different processors while SOM_PAK was run on a single processor. There were 2 CPUs used and 1 GPU.

6.1.1 AMD CPU

The first of the CPUs used was an AMD Operton 6366HE processor[2]. This processor is a device meant for use in servers and as so has a high number of cores but is also concerned by power consumption and heat production. Due to this, the base frequency is rather low when compared to processors with a lesser number of cores. The device itself has 64 cores. While the base frequency of this processor is only 1.8 GHz, it comes with AMD's *Turbo Core* technology[1]. Use of the turbo mode on this processor means that when a process is running on a single core the frequency can be increased up to 3.1 GHz. This evaluation revealed that this processor does not seem to suit this particular program. It is suspected that this is due to the fact that the processor makes use of shared floating point units and hyperthreading to produce its 64 cores. The application developed in this work is very heavy on floating point operations and performing the same operations at the same time. It is these two factors that lead to a slow runtime on this AMD CPU.

During this evaluation phase, tests were carried out to determine how variable the run times for each architecture were. This test involved running the same test case five times and calculating the standard deviation of

the resulting times. The standard deviated for the tests on this processor was 94.60 seconds or 1.98%.

6.1.2 Intel CPU

The other CPU used in this work was an Intel Core i7-2600[14]. This device has 4 cores and a frequency of 3.4 GHz. It is for this reason that SOM_PAK is run on this processor during this evaluation. parallel_SOM was also tested on this device.

The standard deviation found for tests on this device was 1.232 seconds or 0.18%.

6.1.3 nVidia GPU

The GPU used in the evaluation stage of this work was a nVidia GeForce GTX 590 [25]. This GPU takes advantage of SIMD and claims to have 1024 cores. However, in order to take advantage of all of these cores it is necessary to run highly parallel programs that make use of the GPU's SIMD lanes. parallel_SOM is tested on this device and it is this device that the distance metric tests are run on.

The standard deviation found for tests on this device was 0.3373 seconds or 0.14%.

6.2 Running SOM_PAK

In the process of this evaluation, results from parallel_SOM were compared to results from SOM_PAK [1]. When SOM_PAK runs, it performs the organising process in two phases. For the sake of fair comparison, one of these phases was of length 0 when SOM_PAK was run here.

In order to run SOM_PAK, a great many parameters are required for the map. These parameters include x and y dimensions for the map, file names and learning parameters. One of these parameters is a value that specifies how many input vectors should be processed during the runtime. If this value, *rlen*, is larger than the number of input vectors then the program uses some of or all of the input vectors more than once. It is explained in 5.1.1 that parallel_SOM takes a similar approach to the number of iterations and that this number is calculated using the map size and the size of the input data set. When running SOM_PAK during this evaluation, *rlen* is set to the number of iterations that parallel_SOM would use for that particular scenario.

It is suggested in the documentation for SOM_PAK [17] that users run a program called *vfind* that requests users enter all necessary parameters through a simple text based interface. This program then runs other programs in the package that initialise the map, perform the organisation and then check quantisation errors. However, in order to write automated scripts for testing it was necessary to run these smaller programs directly as they allowed input as arguments to the programs.

When SOM_PAK was compiled the *03* compiler optimisation flag was set in order to produce the most efficient version of the program.

randinit.c

The first of the two programs run was the program that initialises a map with random vectors.

[1] SOM_PAK is available from `http://www.cis.hut.fi/research/som-research/nnrc-programs.shtml`

```
./randinit -xdim 32 -ydim 32 -din test_cases/input_5120_3_gauss.dat -cout
   test_cases/test_base.map -topol rect -neigh Gaussian -rand 123
```

This call to the program initialises a 32 x 32 map (-xdim and -ydim) that is prepared for the input from the file stored at *test_cases/input_5120_3_gauss.dat* (-din). The map has a rectangular topology (-topol) and uses a Gaussian neighbourhood function (-neigh). The random function is seeded with the value 123 (-rand) and the resulting, initialised map is stored at *test_cases/test_base.map* (-cout).

vsom.c

The second program used in test cases is the program that actually performs the organisation process on the map.

```
./vsom -cin test_cases/test_base.map -din test_cases/input_5120_3_gauss.
   dat -cout test_base.map -rlen 3276800 -alpha 0.7 -radius 32
```

This program runs the organisation process on the map stored at *test_cases/test_base.map* (-cin) using the input file stored at *test_cases/input_5120_3_gauss.dat* (-din). The program runs through 3276800 input vectors (-rlen) and the starting neighbourhood radius is 32 (-radius). The alpha value here is the starting learning rate value. This value was chosen to be the same value that parallel_SOM uses by default.

6.3 Validation

The first, perhaps most important, part of the evaluation was to determine whether the maps produced by parallel_SOM are reliable and correct. This was done by comparing results to maps produced by SOM_PAK using the same parameters.

6.3.1 Quantisation Error

It was mentioned previously that when both SOM_PAK and parallel_SOM run they create many maps (according to a trials parameter) and return the map with the lowest *quantisation error*. The quantisation error value is a measure of how well the input data set *fits* with the resultant map. A lower, quantisation error means the data fits well.

The error value itself is an average of the errors found for each of the input vectors. Once the map has been built and the organising process is complete the quantisation error calculation happens. Each input vector in the input data set is considered one more time and the winning neuron for each found. The error for this particular input vector is now found by taking an average of the differences between each dimension in the input vector and the winnning neuron's vector. All of these error values for each input are then considered and averaged to find the overall quantisation error.

6.3.2 Visual Topology Comparison

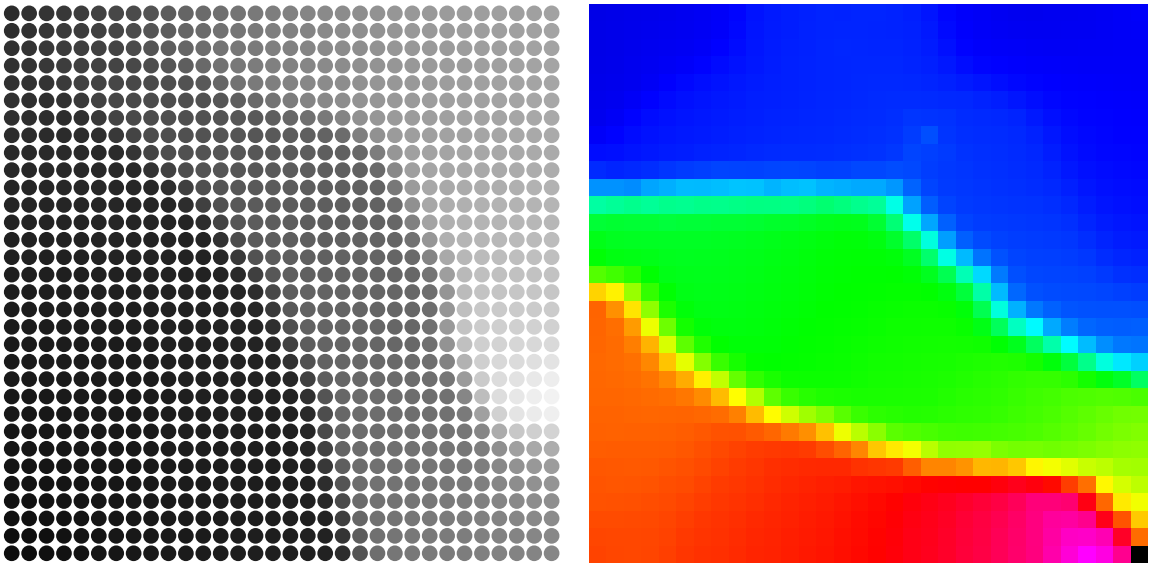
The self organising map algorithm is intended to produce representations of data that preserve the *topology* of the data. It is therefore logical to compare the visual output from SOM_PAK and parallel_SOM to determine

the similarity of their operations by comparing the shapes and groupings that are produced by each for a range of different data sets. It should be noted however that it is unlikely that both programs will produce exactly the same maps from the same data, due to the random nature of the initialisation phase of the process.

6.3.3 Validation Tests

The maps displayed here were all generated from Gaussian clustered data and were chosen to be shown because they represent a range of parameters for the maps. The data itself was made values ranging from 0 to 10000. The quantisation error value is calculated using the supplied *qerror* program as part of SOM_PAK.

32 x 32 Map trained with 5120 x 3 vectors



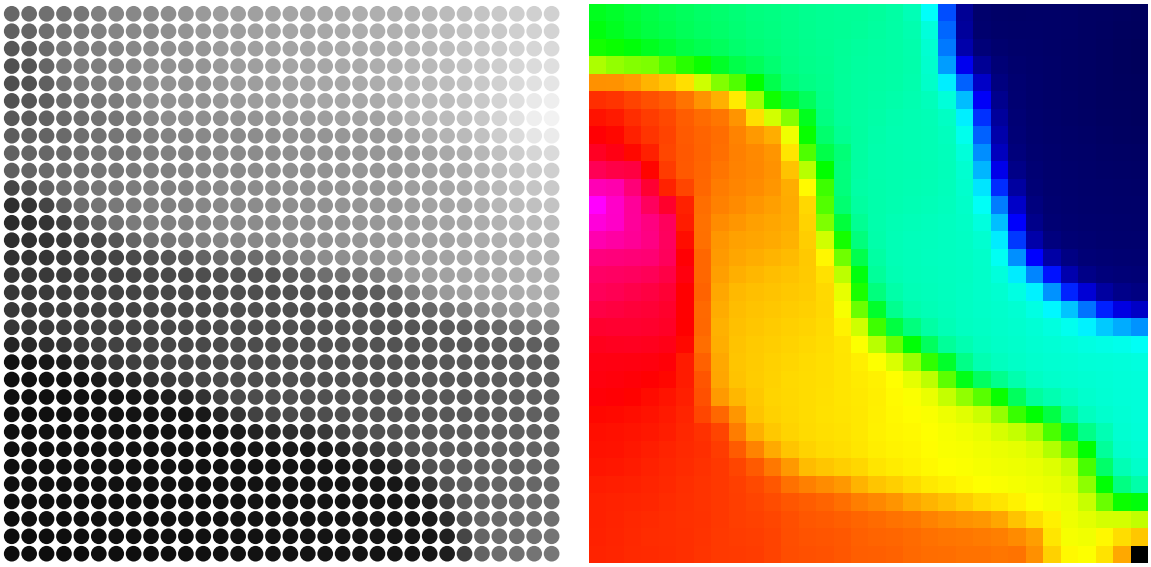
(a) SOM_PAK
Quantisation error = 88.749619

(b) parallel_SOM
Quantisation error = 94.319016

Figure 6.1: Visualisation of 32 x 32 map trained with 5120 x 3 vectors

The maps shown in figure 6.1 do not instantly look the same. However, if examined more closely it can be seen that there are 3 major groups of data in each and that the sizes of the regions are similar between the maps. The quantisation errors of the two maps are both good, and are very similar.

32 x 32 Map trained with 8192 x 3 vectors



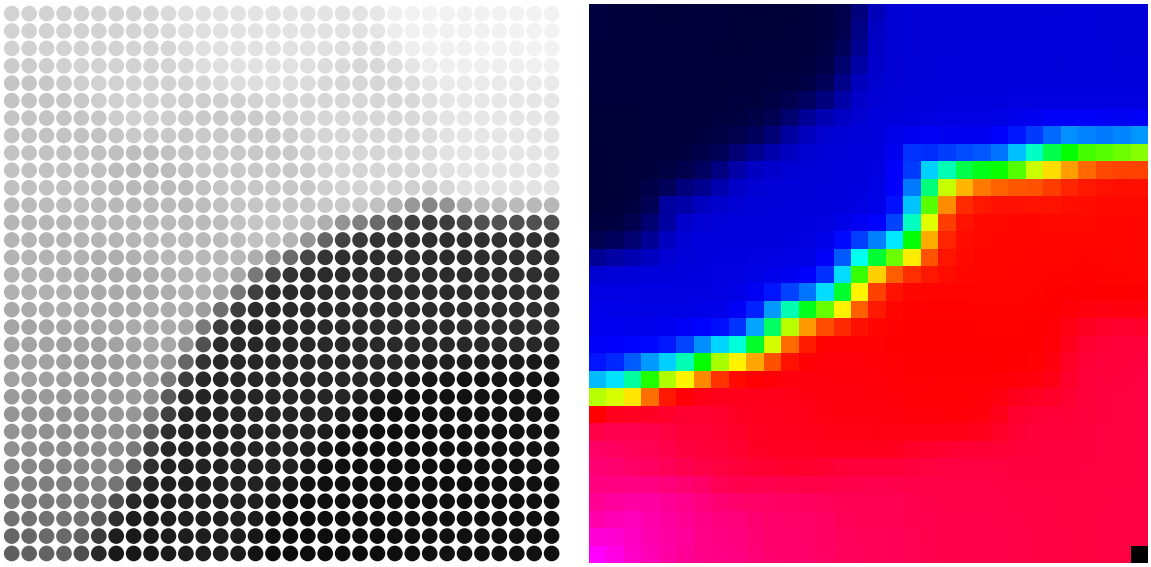
(a) SOM_PAK
Quantisation error = 90.356743

(b) parallel_SOM
Quantisation error = 94.778107

Figure 6.2: Visualisation of 32 x 32 map trained with 8192 x 3 vectors

Figure 6.2 shows that both maps have the same 4 clusters of the same relative sizes. Both maps show that 2 of the clusters are rather similar and seem to meld into one and other. Both maps also show a small, localised spike. In 6.2b this spike is pink and on the left side of the picture and in 6.2a this spike is white and in the top right corner. The quantisation errors here are also similar.

32 x 32 Map trained with 5120 x 8 vectors



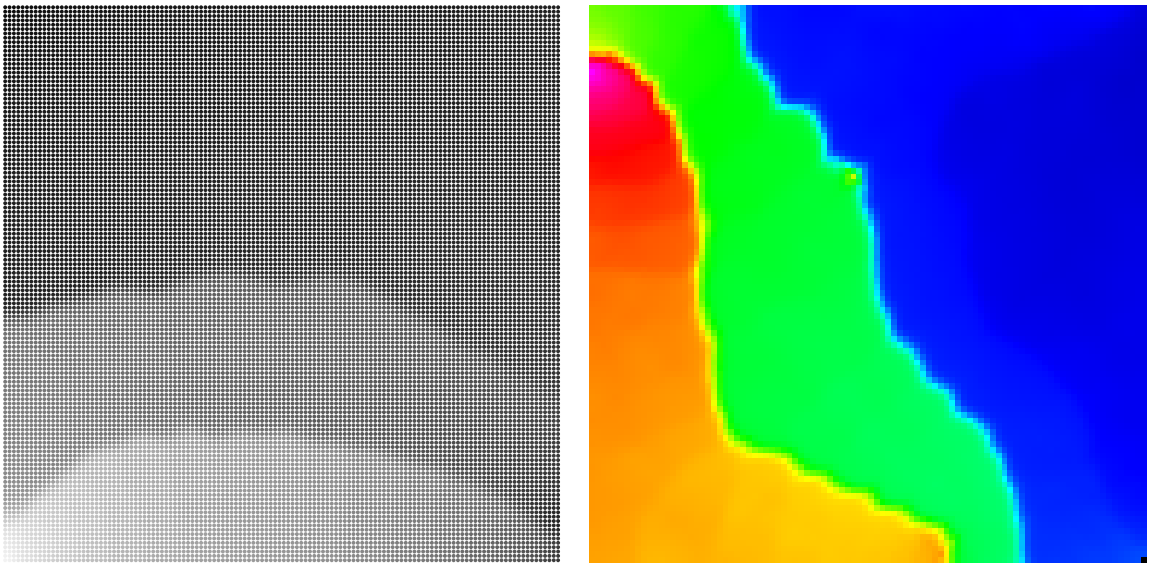
(a) SOM_PAK
Quantisation error = 177.500992

(b) parallel_SOM
Quantisation error = 184.272003

Figure 6.3: Visualisation of 32 x 32 map trained with 5120 x 8 vectors

Both maps in figure 6.3 show at least 2 definite clusters with a possible third that is very similar to another cluster. The quantisation errors here are alike.

96 x 96 Map trained with 5120 x 3 vectors



(a) SOM_PAK
Quantisation error = 40.706688

(b) parallel_SOM
Quantisation error = 38.794975

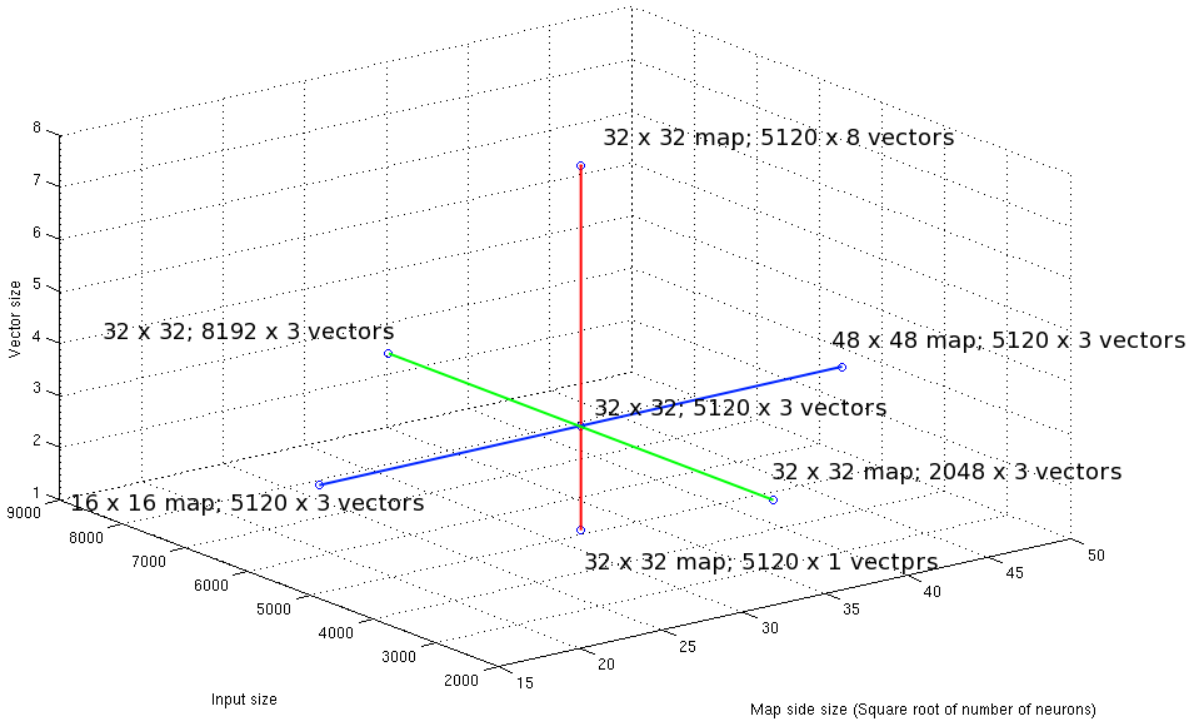
Figure 6.4: Visualisation of 96 x 96 map trained with 5120 x 3 vectors

The maps in figure 6.4 seem to be completely different at first glance. They do show some of the same characteristics however. Both have three clusters where one of the clusters is much larger than the other two. Both exhibit the fact that one of the clusters varies a lot within the cluster. That is, in 6.4b one of the clusters is red at one extreme and yellow at the other. This characteristic is mirrored in 6.4a in the bottom cluster. As with all other maps in this section, the quantisation errors between the maps are similar.

6.4 Base Tests

6.4.1 Strategy

Figure 6.5: Graph demonstrating base evaluation strategy



The strategy undertaken was to vary the three variables that could potentially determine how fast the program runs. Each of the variables chosen contribute to how many calculations are necessary during the runtime of the program. The variables that were changed in the basic tests are the size of the map, the number of input vectors and the number of dimensions in the input vectors.

In order to form comparisons, initial values for the three variables were chosen. From these values a 32 x 32 map formed from 5120, 3 dimensional vectors was chosen as the base case for tests. This base case was chosen as a map that allowed changing all the variables in both directions while still being a map that took a relatively short time to be made. Once the base test had been performed, each of the other variables were tested by increasing the value for that variable for one test and decreasing for another. This strategy resulted in 7 tests, including the base test being performed. Figure 6.5 shows the base evaluation strategy as a 3 dimensional graph.

All tests in this section were performed using data containing Gaussian clusters from the data generator explained in 5.4.

6.4.2 Results

Figure 6.6: Results from Base Evaluation

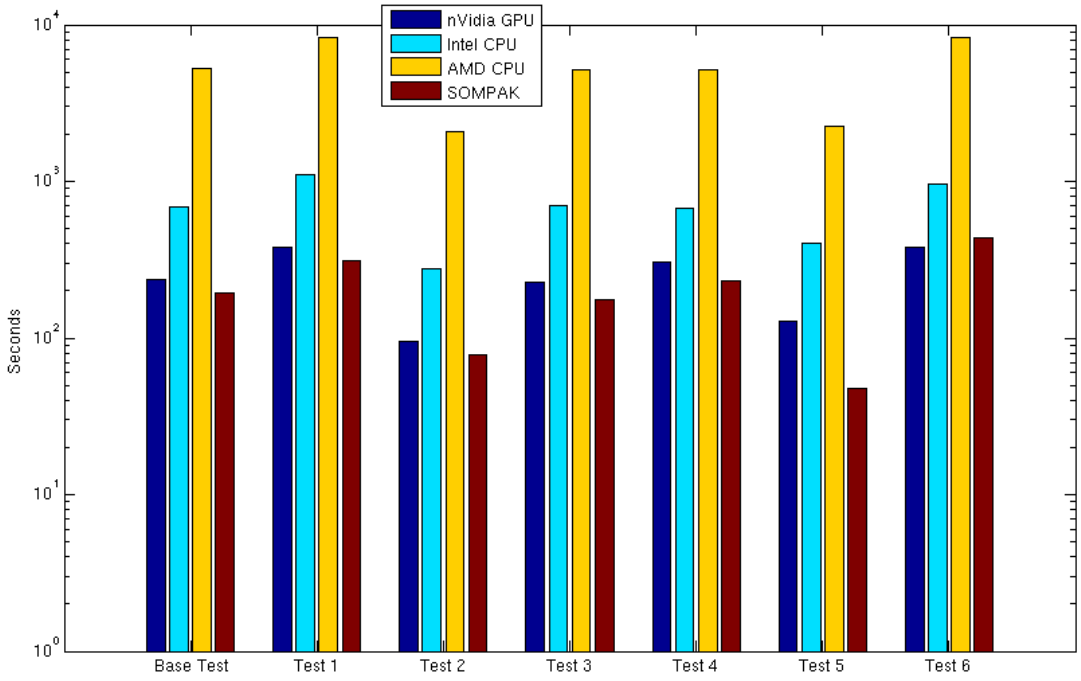
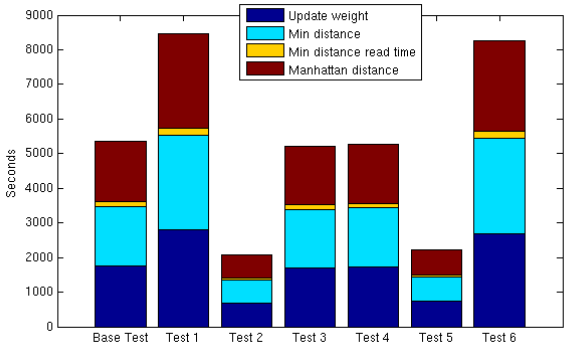


Figure 6.7: Test Data for Base Tests (All times are in seconds)

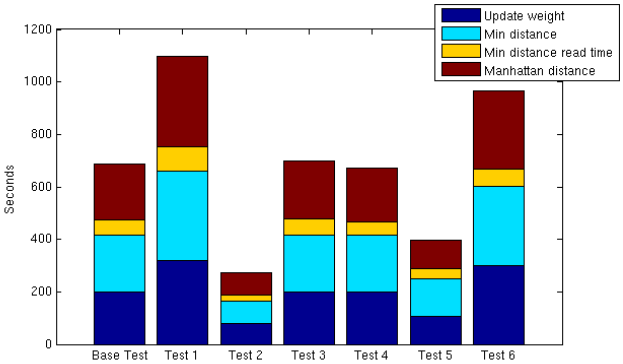
Test	Map Size	Input Size	Vector Size	nVidia GPU	Intel CPU	**AMD CPU**	SOM_PAK
Base	32 x 32	5120	3	237.554	691.885	5181	194
1	32 x 32	**8192**	3	379.744	1106.41	9317	312
2	32 x 32	**2048**	3	95.3717	276.826	2124	78
3	32 x 32	5120	**1**	226.707	703.453	5628	176
4	32 x 32	5120	**8**	302.915	670.234	5903	231
5	**16 x 16**	5120	3	127.156	402.576	2635	48
6	**48 x 48**	5120	3	378.444	955.024	7954	437

6.4.3 Discussion

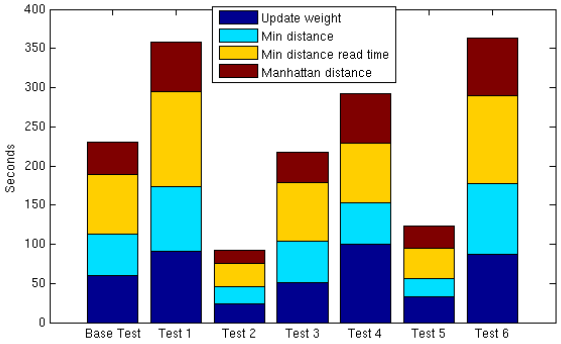
The most obvious result from the graph in figure 6.6 is that the AMD CPU performs very poorly. The AMD CPU seems to take an order of magnitude more time than the other two devices being tested with parallel_SOM. This is believed to be due to the architecutre details of this particular processor, in that it shares floating point units and uses hyperthreading. The use of shared floating point units means that each *core* cannot necessarily execute completely in parallel with other cores as it may be waiting for access to the floating point unit. Since parallel_SOM is heavy on floating point operations, this slows run times. Hyperthreading means that each of the physical processor cores that the AMD CPU has become 2 virtual cores that can execute in parallel when multiple different instructions on the same data are required. parallel_SOM does not require different operations on the same data but rather performs the same operations on a large set of data. Since the AMD processor is



(a) Execution breakdown for AMD CPU



(b) Execution breakdown for Intel CPU



(c) Execution breakdown for nVidia GPU

Figure 6.8: Execution breakdown for nVidia GPU

running every core, the frequency is capped at 1.8 GHz. This, along with the diminished number of cores actually available leads to this processor being much slower than the others for parallel_SOM.

Another point here is that parallel_SOM does not perform any faster than the serially run SOM_PAK for almost all tests. The closest competitor to SOM_PAK in this case is parallel_SOM run on the nVidia GPU. From the data shown in this section it could be postulated that the speed up achieved by parallel_SOM when compared to SOM_PAK increases with the size of the map being produced. This will be further explored in 6.5.

The data also shows that the number of dimensions in the vectors of the test set do not affect run times all that much. It could be logical to assume that multiplying the number of dimensions in the vectors would multiply the runtime but this is not the case. Further tests are needed to determine the relationship between dimensionality and runtime.

According to the data, runtime increases linearly with the size of the input dataset. This was expected as each input vector is considered serially by both parallel_SOM and SOM_PAK.

The execution breakdown graphs in Figure 6.8 show that each element of the program, on each platform, takes proportionally the same amount of time for each test. It also shows that the buffer read that happens after the minimum distance kernel executes is much more expensive on the nVidia GPU than on either of the CPUs.

6.5 Varying Map Size Tests

6.5.1 Strategy

A number of works referenced in 3.3 point to the usefulness of larger maps. The tests discussed is 6.4 show that varying the size of the map affects the time taken to build the map. This section further explores the effect of map size on execution times.

As before, all tests in this section were performed using data containing Gaussian clusters from the data generator explained in 5.4.

6.5.2 Results

Figure 6.9: Line Graph of Results from Varied Map Size Tests

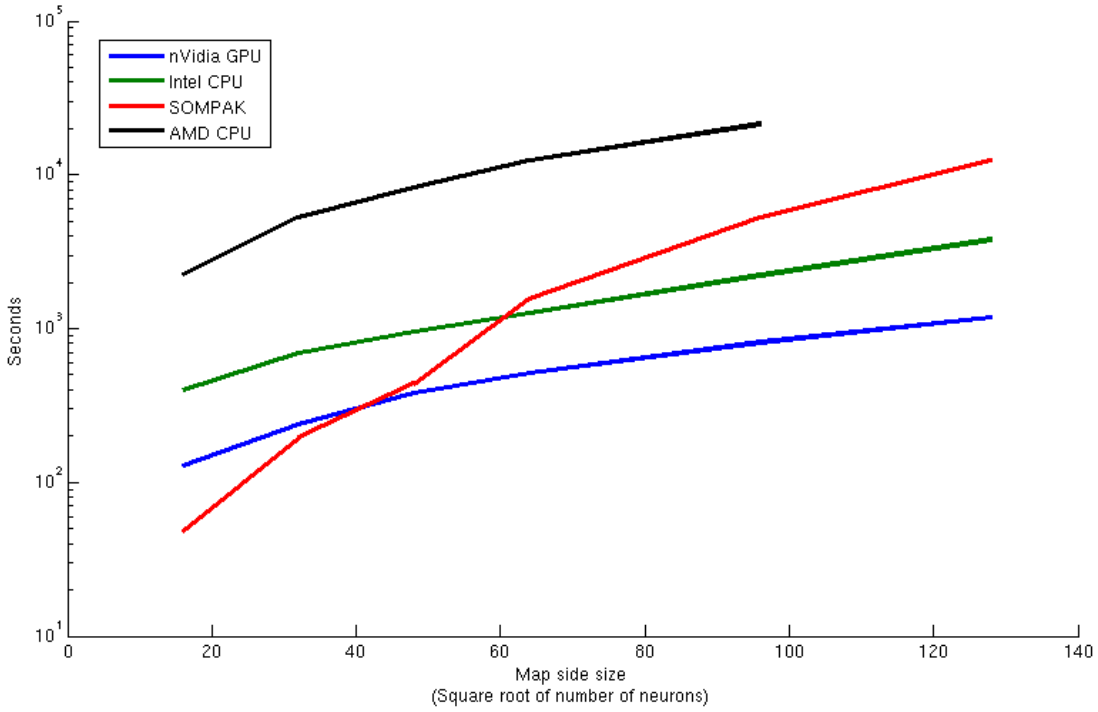


Figure 6.10: Bar Graph of Results from Varied Map Size Tests

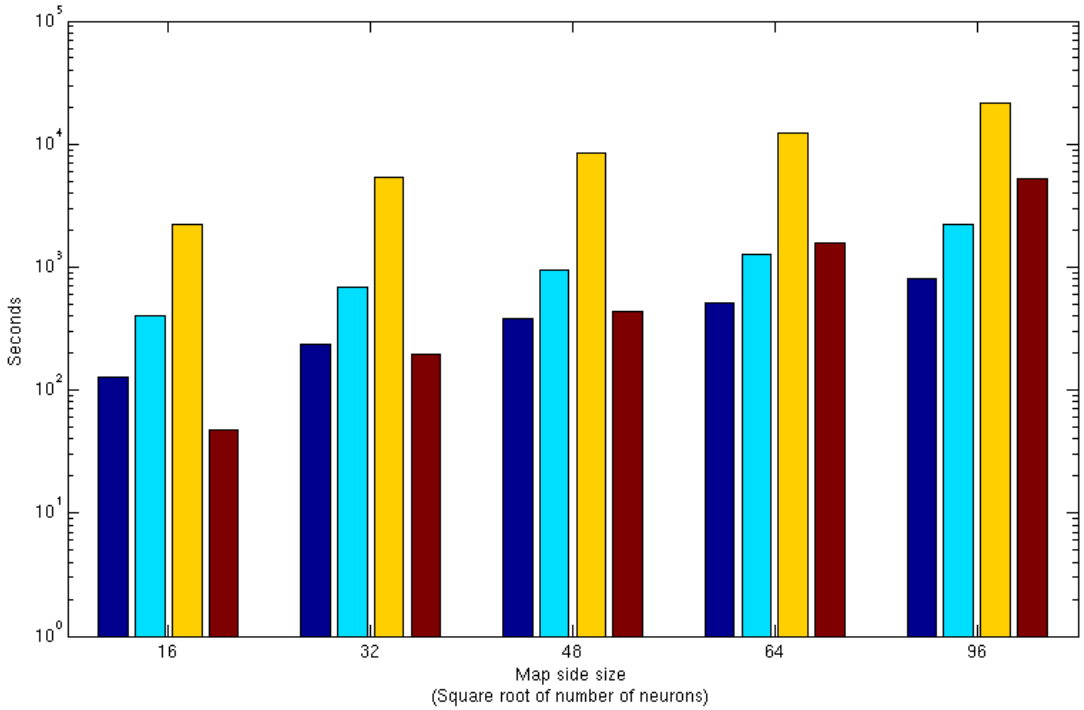


Figure 6.11: Test Data from Varied Map Size Tests (All times are in seconds)

Test	Map Size	Input Size	Vector Size	nVidia GPU	Intel CPU	AMD CPU	SOM_PAK
1	**16 x 16**	5120	3	127.156	402.576	2233.07	48
2	**32 x 32**	5120	3	236.923	692.568	5302.89	194
3	**48 x 48**	5120	3	378.444	955.024	8346.99	437
4	**64 x 64**	5120	3	510.364	1262.16	12 387	1565
5	**96 x 96**	5120	3	814.271	2219.9	21 332.1	5240
6	**128 x 128**	5120	3	1173.72	3795.92	- -	12 360

6.5.3 Discussion

There are two clear results here. The first is that parallel_SOM running on the GPU outperforms everything and that its run time increases almost linearly with the map side size (the square root of the number of neurons in the map). The Intel CPU also performs very well but does not follow this same linear trend.

The second obvious result is that the AMD CPU performs very poorly and is an order of magnitude slower than the nVidia GPU in all tests here. Unfortunately, the graph in figure 6.9 is missing a result for the AMD CPU. This is due to the fact that runtime for this test, the 128 x 128 map, would be huge when considering the previous results and the AMD CPU was required for other research.

This section has shown that map size most definitely effects the run time of the organisation process. It has also shown that the GPU running parallel_SOM produces a speed up of more than 10x when compared to SOM_PAK. This result is comparable to the results found by McConnell et al. [22] who found a maximum speed up of 13.86x when running a parallel version of the self organising map using OpenCL on an nVidia Tesla

graphics card [32]. McConnell et al. compared their results to a single threaded version of the self organising map that they had produced themselves, rather than SOM_PAK.

6.6 Distance Metric Tests

This work aimed to explore the performance benefits from using Manhattan distance rather than Euclidean distance as part of a self organising map. This section details the evaluation of distance metrics.

6.6.1 Strategy

The strategy here is essentially the same as the strategies in 6.4 and 6.5 but *Manhattan_distance.cl* has been changed so that it calculates Euclidean distance rather than Manhattan distance. These tests were carried out on the nVidia GPU and will be compared to the results from the nVidia GPU when the program was run using Manhattan distance.

Data used here was generated using the data generator explained in 5.4 and contained Gaussian clusters.

6.6.2 Results

Figure 6.12: Graph showing Manhattan distance and Euclidean distance execution time against map side size

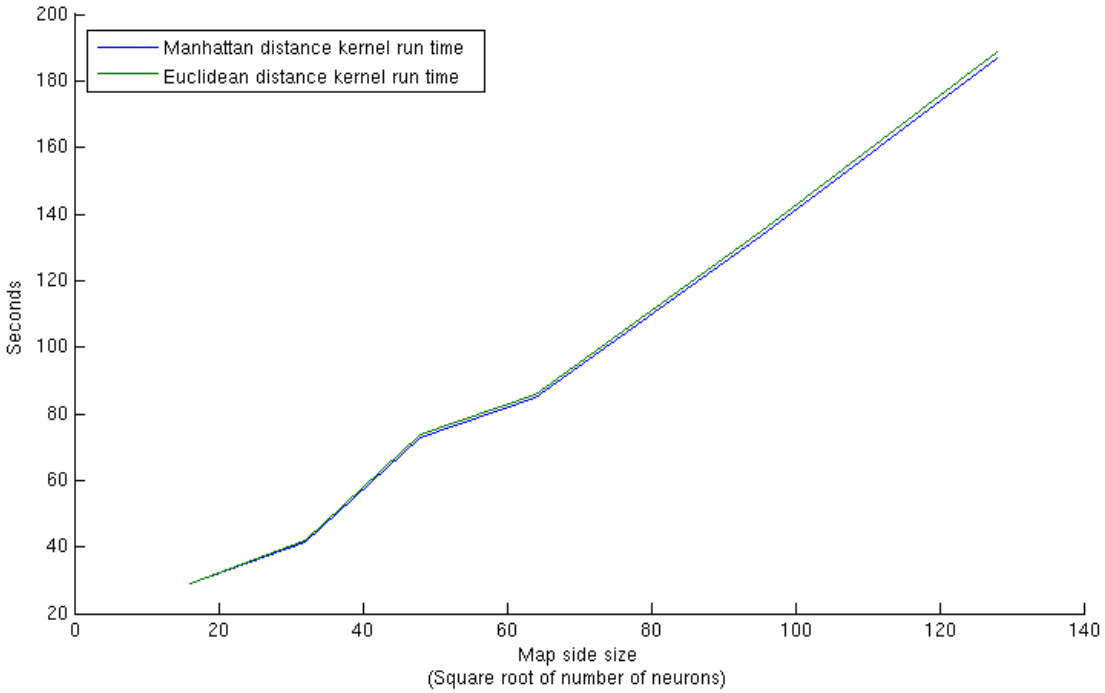


Figure 6.13: Test Data for Distance Metric Evaluation (All times are in seconds)

Test	Map Size	Input Size	Vector Size	Euclidean distance	Manhattan distance
Base	32 x 32	5120	3	42.1586	41.4993
1	32 x 32	**8192**	3	65.66	64.0097
2	32 x 32	**2048**	3	17.3751	16.9048
3	32 x 32	5120	**1**	38.6156	37.9868
4	32 x 32	5120	**8**	63.5136	63.4081
5	**16 x 16**	5120	3	28.9426	28.885
6	**48 x 48**	5120	3	73.9288	73.015
7	**64 x 64**	5120	3	85.7243	84.749
8	**96 x 96**	5120	3	136.108	134.716
9	**128 x 128**	5120	3	188.875	186.794

6.6.3 Discussion

The results here show that there is a negligible difference between the execution times for the Euclidean distance metric and the Manhattan distance metric but that the Manhattan distance metric is always faster. This suggests that the runtime of distance calculation is not dependent on how long the actual computations take and more on other factors. Perhaps these factors are the overheads involved in running OpenCL code in parallel.

6.7 Overall Results

This chapter has demonstrated a number of things. First of all, it has shown that parallel_SOM produces maps that are comparable to those produced by SOM_PAK.

Secondly, it has been shown that for large map sizes and on certain architectures parallel_SOM outperforms SOM_PAK. In these tests, when parallel_SOM was run on a GPU it achieved a speed up factor of more than 10 when compared to the run time of SOM_PAK run serially.

This chapter has shown that the parameters that affect the run time for parallel_SOM are the size of the input dataset and the size of the map to be produced. The dimensionality of the input vectors seems to make little to no difference to run times.

Finally, it has been shown that the use of Manhattan distance over Euclidean distance during the organising process does bring about speed ups. These speed ups are negligible however and seem to show no trend with the size of the map being produced.

Chapter 7

Conclusion

7.1 Future Work

This project has produced a basic self organising map program that achieves speed ups on parallel architecture. There are many improvements that can be made to the program itself and there is much work that can be carried out as a continuation.

7.1.1 Improve Functionality and User Interface of parallel_SOM

Currently parallel_SOM does not provide nearly the amount of customisation of parameters for the organising process that SOM_PAK provides. This customisation is necessary for real world use as there are no globally optimal parameters for the learning process for every possible data set. Optimal parameters for each dataset are unique and can only be found through trial and error currently. It would be desirable to allow a user to specify at least the parameters that SOM_PAK allows through a textual interface of some kind or through loading a file with a specification. This would make parallel_SOM much more usable in the real world.

Currently parallel_SOM runs all kernel code on the first device available in the system. If there were more than one OpenCL compatible device in a system, the program would simply run on the first one and there is no way to force the use of another device without changing program code. It may be desirable in some situations to allow a user to select which device to run kernel code on.

It has been mentioned briefly previously that SOM_PAK allows a user to apply a number of vectors to a map and be informed through a visual representation where those vectors lie on the map. This is a feature that would likely be useful as part of parallel_SOM.

7.1.2 Run parallel_SOM on Intel Xeon Phi

This work has shown that parallel_SOM runs well on multi-core and many-core architecture. The Xeon Phi processor is a many-core processor with up to 61 powerful cores [15]. The device itself is suited to floating point operations and makes use of vectorisation to achieve massive parallelism. parallel_SOM is both heavy on floating point operations and vector calculations and so is suited to running well on a Xeon Phi processor or similar many-core device.

7.1.3 Support Vector Machine in OpenCL

It has been proved in this work, and many previous (see 3.3), that OpenCL is a useful tool for writing parallel code and cutting down run times for computationally heavy programs and operations. The self organising map is just one example of a computationally heavy program. Another example is another machine learning algorithm, the support vector machine [9]. Though work has already been done to create a version of the algorithm that runs on graphics hardware [5] it would perhaps be worthwhile to implement an OpenCL version that could be used on many platforms.

7.2 Final Remarks

From the outset, this project aimed to create a parallel version of the original self organising map algorithm that was suited to run on multi-core and many-core architecture. The program was to produce comparable results to a widely used self organising map package, SOM_PAK, and it would be the ultimate goal to produce these results in a much faster time.

Parallel_SOM achieves both of these goals and makes use of OpenCL. The maps that are produced contain the same features as those produced by SOM_PAK for the same dataset and the program provides a speed up of up to 10x when running on graphics hardware. The fact that the program is written using OpenCL rather than another system like CUDA means that it is not platform specific. The hardware need only support OpenCL for the program to be compatible.

This work has also shown that the use of Manhattan distance rather than Euclidean distance provides further slight speed up. The benefits shown in this work are extremely limited but the maps used for evaluation here were relatively small.

The self organising map has applications in many areas but is limited by the amount of time the organising process takes when run serially. This work has demonstrated the speed ups possible for the organising process when using standard, widely available and affordable graphics hardware. Along with other work in the area, the results of this will lead to widespread use of implementations of the self organising map for parallel architectures and possibly increase the usefulness of the algorithm as a whole.

Appendices

Appendix A

Use of Software

All software produced by this project is available from the github repository at `https://github.com/KombuchaShroomz/ParallelismProject`.

A.1 parallel_SOM

A.1.1 Building parallel_SOM

As well as the visualisation tool provided, parallel_SOM requires C++ 11 and OpenCL to run. A makefile is provided that compiles the program as it should. The original directory structure for parallel_SOM.cpp and drawPPMmap.cpp must be adhered to.

A.1.2 Running parallel_SOM

When parallel_SOM is run it needs 3 parameters, the input file, the size of one side of the map and the number of trials to be run. These parameters can be entered as arguments to the program.

```
>> ./parallel_SOM datasets/input_5120_3_gauss.dat 32 1
```

Where the input file name is *datasets/input_5120_3_gauss.dat*, the map size is 32 and the number of trials is 1.

Otherwise the program will request their entry upon being run.

```
>> ./parallel_SOM
Standard usage: ./parallel_SOM <input file name> <map side size> <trials>
Enter input filename: datasets/input_5120_3_gauss.dat
<Reading input file>
Enter map dimension (x and y dimenions will be equal):  32
Enter number of trials:   1
```

A.1.3 Input File Format

Like SOM_PAK, parallel_SOM takes input vectors from a file where each vector is on a single line and each component is separated by a space. The first line of the file should contain an integer that represents the number of vectors in the file and another that represents the size of the vectors.

```
5120 3
8500.47 8206.1  8601.49
8173.97 8451.26 8318.07
8593.93 8332.43 8540.37
8282.53 8559.09 8310.68
...
```

A.2 dataset_generator

A.2.1 Building dataset_generator

A makefile is provided to compile this program and it has no dependences other than a C++ compiler.

A.2.2 Running dataset_generator

The program does not take any arguments from the command line and provides a simple interface that informs the user of their options and directs them in what to enter.

A.3 drawPPMmap

In order to use this program out with parallel_SOM, the file *PPM.c* is necessary. This file is provided in the same folder as *drawPPMmap.cpp*.

Bibliography

[1] AMD. *AMD Opteron 6300 Series processor Quick Reference Guide*. URL: https://www.amd.com/Documents/Opteron_6300_QRG.pdf (visited on 03/15/2015).

[2] AMD. *AMD Opteron Processor Solutions*. URL: http://products.amd.com/en-us/OpteronCPUDetail.aspx?id=813&f1=AMD+Opteron%E2%84%A2+6300+Series+Processor&f2=6366+HE&f3=Yes&f4=1800&f5=1000&f6=G34&f7=C0&f8=32nm&f9=85+W&f10=6400&f11=16& (visited on 03/15/2015).

[3] Alexander Campbell, Erik Berglund, and Alexander Streit. "Graphics hardware implementation of the parameter-less self-organising map". In: *Intelligent Data Engineering and Automated Learning-IDEAL 2005*. Springer, 2005, pp. 343–350.

[4] George Caridakis et al. "SOMM: Self organizing Markov map for gesture recognition". In: *Pattern Recognition Letters* 31.1 (2010), pp. 52–59.

[5] Bryan Catanzaro, Narayanan Sundaram, and Kurt Keutzer. "Fast support vector machine training and classification on graphics processors". In: *Proceedings of the 25th international conference on Machine learning*. ACM. 2008, pp. 104–111.

[6] Rocio Chavez-Alvarez, Arturo Chavoya, and Andres Mendez-Vazquez. "Discovery of possible gene relationships through the application of self-organizing maps to DNA microarray databases". In: *PloS one* 9.4 (2014), e93233.

[7] William Claster, Subana Shanmuganathan, and Nader Ghotbi. "Text mining of medical records for radiodiagnostic decision-making". In: *Journal of Computers* 3.1 (2008), pp. 1–6.

[8] CMSoft. *Basic aspects of OpenCL C99 language*. URL: http://www.cmsoft.com.br/opencl-tutorial/basic-aspects-opencl-c99-language9/ (visited on 03/07/2015).

[9] Corinna Cortes and Vladimir Vapnik. "Support-vector networks". In: *Machine learning* 20.3 (1995), pp. 273–297.

[10] The R Foundation. *R Homepage*. URL: http://www.r-project.org/about.html (visited on 03/08/2015).

[11] Petr Gajdoš and Jan Platoš. "GPU based parallelism for self-organizing map". In: *Proceedings of the Third International Conference on Intelligent Human Computer Interaction (IHCI 2011), Prague, Czech Republic, August, 2011*. Springer. 2013, pp. 231–242.

[12] Khronos Group. *OpenCL Conformant Products*. URL: https://www.khronos.org/conformance/adopters/conformant-products#opencl (visited on 03/09/2015).

[13] Khronos Group. *OpenCL Homepage*. URL: https://www.khronos.org/opencl/ (visited on 03/01/2015).

[14] Intel. *Intel Core i7-2600 Processor*. URL: http://ark.intel.com/products/52213/Intel-Core-i7-2600-Processor-8M-Cache-up-to-3_80-GHz (visited on 03/15/2015).

[15] *Intel Xeon Phi Product Family*. URL: http://www.intel.co.uk/content/www/uk/en/processors/xeon/xeon-phi-detail.html (visited on 03/19/2015).

[16] Samuel Kaski. "Data exploration using self-organizing maps". In: *ACTA POLYTECHNICA SCANDINAVICA: MATHEMATICS, COMPUTING AND MANAGEMENT IN ENGINEERING SERIES NO. 82*. Citeseer. 1997.

[17] T Hynninen Kohonen. *J., Kangas, J., and Laaksonen, J. 1996. SOM PAK: The Self-Organizing Map Program Package*. Tech. rep. Technical Report.

[18] Teuvo Kohonen. *Self-organizing maps*. Vol. 30. Springer Science & Business Media, 2001.

[19] Teuvo Kohonen. "The self-organizing map". In: *Proceedings of the IEEE* 78.9 (1990), pp. 1464–1480.

[20] Teuvo Kohonen et al. "Self organization of a massive document collection". In: *Neural Networks, IEEE Transactions on* 11.3 (2000), pp. 574–585.

[21] Yonggang Liu and Robert H Weisberg. *A review of self-organizing map applications in meteorology and oceanography*. INTECH Open Access Publisher, 2011.

[22] Sabine McConnell et al. "Scalability of Self-organizing Maps on a GPU cluster using OpenCL and CUDA". In: *Journal of Physics: Conference Series*. Vol. 341. 1. IOP Publishing. 2012, p. 012018.

[23] Nicolás Juris Medrano-Marqués and Bonifacio Martín-del Brío. "Topology preservation in SOFM: an euclidean versus manhattan distance comparison". In: *Foundations and Tools for Neural Modeling*. Springer, 1999, pp. 601–609.

[24] nVidia. *CUDA Homepage*. URL: `http://www.nvidia.com/object/cuda_home_new.html` (visited on 03/01/2015).

[25] nVidia. *GeForce GTX 590*. URL: `http://www.geforce.co.uk/hardware/desktop-gpus/geforce-gtx-590` (visited on 03/15/2015).

[26] Raghavendra D Prabhu. "SOMGPU: an unsupervised pattern classifier on graphical processing unit". In: *Evolutionary Computation, 2008. CEC 2008.(IEEE World Congress on Computational Intelligence). IEEE Congress on*. IEEE. 2008, pp. 1011–1018.

[27] AK Sahai et al. "A new method to compute the principal components from self-organizing maps: an application to monsoon intraseasonal oscillations". In: *International Journal of Climatology* 34.9 (2014), pp. 2925–2939.

[28] *SIMD Explaination*. URL: `https://www.kernel.org/pub/linux/kernel/people/geoff/cell/ps3-linux-docs/CellProgrammingTutorial/BasicsOfSIMDProgramming.html` (visited on 03/03/2015).

[29] Viscovery Software. *Viscovery Homepage*. URL: `http://www.viscovery.net/welcome` (visited on 03/08/2015).

[30] Viscovery Software. *Viscovery SOMine 6 Key Benefits*. URL: `http://www.viscovery.net/somine/` (visited on 03/08/2015).

[31] Grant Strong and Minglun Gong. "Similarity-based image organization and browsing using multi-resolution self-organizing map". In: *Image and Vision Computing* 29.11 (2011), pp. 774–786.

[32] *TESLA GPU ACCELERATORS FOR SERVERS*. URL: `http://www.nvidia.com/object/tesla-servers.html` (visited on 03/24/2015).

[33] Jonathan Tompson and Kristofer Schlachter. "An Introduction to the OpenCL Programming Model". In: *Digital version* (2012).

[34] Lowie EGW Vanfleteren et al. "Clusters of comorbidities based on validated objective measurements and systemic inflammation in patients with chronic obstructive pulmonary disease". In: *American journal of respiratory and critical care medicine* 187.7 (2013), pp. 728–735.

[35] Kiri Wagstaff et al. "Constrained k-means clustering with background knowledge". In: *ICML*. Vol. 1. 2001, pp. 577–584.

[36] Ron Wehrens and Lutgarde MC Buydens. "Self-and super-organizing maps in R: the Kohonen package". In: *J Stat Softw* 21.5 (2007), pp. 1–19.