School of Computer Science and Engineering

Faculty of Engineering

The University of New South Wales

Implementing neural networks in Accelerate for GPU execution

by

Ji Yong Jeong

Thesis submitted as a requirement for the degree of

Bachelor of Engineering in Software Engineering

Submitted: 24 May 2016 Student ID: z2250189

Supervisor: A/Prof. Gabriele Keller Topic ID: 3733

Abstract

GPU-accelerated programming is used to speed up neural network computations. However, the main languages for GPU programming are CUDA and OpenCL, which are very low-level languages. This thesis proposes implementing a simple feed-forward back-propagation (FFBP) neural network, namely a convolutional neural network (CNN) for image recognition using Accelerate. Accelerate is an Embedded Domain-Specific Language (EDSL) in Haskell for GPU programming and has several advantages over CUDA, such as accessibility. Once the implementation is successful, I plan to assess its performance, benefits and disadvantages against the traditional approaches with CUDA.

"accessibility advantages over CUDA, while still offering competitive performance."

my comments are in blue, sometimes i crossed stuff out with the blue highlighter.

Acknowledgements

First, I would like to express my deepest thanks to my supervisor, Gabriele Keller for her gentle encouragements, patience and guidance. I would also like to thank all the members of UNSW Programming Languages and Systems (PLS) group for additional support and assistance, and for being an invaluable wealth of knowledge. In particular, I would like to thank Liam O'Connor for his generous aid.

I am very grateful to have learned so much during Thesis A, and look forward increasing my knowledge further in Thesis B!

Abbreviations

"Glossary" is a better title, but this should be incorporated into the text. E.g, on first use, say "Batch gradient descent learning algorithm (BGD)" and thereafter just use BGD.

Looks like you've already done that though.

BGD Batch gradient descent learning algorithm

CNN Convolutional neural network

CUDA Computer Unified Device Architecture

DNN Deep neural network

EDSL Embedded domain-specific language

FFBP Feed-forward back-propagation algorithm

GPU Graphics Processing Unit

OpenCL Open Computing Language

PLS UNSW Programming Languages and Systems group

ReLU Rectified Linear Unit activation function

SGD Stochastic gradient descent learning algorithm

Contents

List of Figures

Chapter 1

Introduction

Neural networks are widely used for computer vision and pattern recognition [illegible] problems. Deep neural networks (DNN) are more complicated neural networks and can already perform at human level on tasks, such as handwritten character recognition (including Chinese), on various automotive problems (traffic sign detection, lane detection, pedestrian tracking, automatic steering), speech recognition and natural language processing [NVI14]. The issue with neural networks is that there may be thousands of features, but testing for the validity of each hypothesis (a certain set of features) can oft times be most time efficient and economically beneficial when assessed by a quick trial and error type of test [Ng16].

no comma

no "on"

comma

"features" is a ML term not used in PL. Define it before use

"often be most efficiently and cheaply assessed by rapid trial and error."

oft times or oftentimes are colloquial. Just use "often"

Neural networks implemented using GPU-accelerated computing are generally faster than with just CPU as neural networks typically involve large, repeated computations. The language of choice for GPU programming is Compute Unified Device Architecture (CUDA), which is a C-like language. Accelerate on the other hand, is a embedded domain-specific language in Haskell created for GPU programming, with comparatively much simpler syntax and type-checking, etc.

why is that suitable for GPUs?

the main simpler part is the semantics, as you don't need to worry about memory. So I'd replace "type-checking" with "semantics" and drop the "etc."

Thus, the motivations for this thesis is to explore the feasibility of implementing a neural network in a more on-the-fly, user friendly way using Accelerate, and consequently, enabling us to more readily test for its hypotheses. The performance of the implemen-

"… test a variety of hypotheses."

tation will indicate if there exists a loss of performance, and if so, whether it is within an acceptable range for hypothesised trade-off in functionality.

the

The following section, Chapter 2 outlines the background relating to this topic, such as an overview of neural networks, previous approaches to neural network implementations using the GPU, and the current neural network implementation using Accelerate. Chapter 3 explains my proposed implementation and some projected issues. Finally, Chapter 4 summarises the contents of this report.

where's your plan here?

Chapter 2

Background and Related Works

2.1 Neural network architecture

Broadly speaking, a neural network can be described as a certain layering of nodes, or units, connected to each other by directed links, where each link has a certain numeric weight that signifies the strength of connection between the connected nodes.

In latex, you should use backticks (``) to get the double quotes going the right way on the LHS of a quote.

Historically, the concept of "net of neurons" whose interrelationship could be expressed in propositional logic was first proposed by [MP43] in 1943, inspired by a "all-or-none" behaviour of the biological nervous system. The first basic unit, also called the *perceptron*, was invented by [Ros62]. A perceptron will be activated if the sum of all the input values from its input links, say $x_1, ..., x_m$, multiplied by the links' corresponding weights, say $\theta_1, ..., \theta_m$, is above that unit's certain threshold value, or *bias* b, such that

$$\text{output} = \begin{cases} 0 & \text{if } \sum_{i=1}^{m} x_i \theta_i \leq b \\ 1 & \text{otherwise} \end{cases}$$

The above is equivalent to vectorizing the inputs to $\boldsymbol{x} = [x_1, ..., x_m]$, weights to $\boldsymbol{\theta} =$

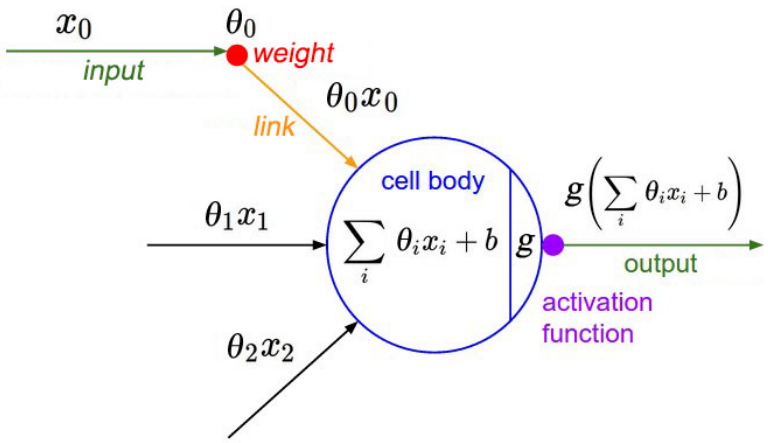

Figure 2.1: Structure of a modern unit by [Kar16].

$[\theta_1, ..., \theta_m]$ and inverting the sign on b, so that,

$$\text{output} = \begin{cases} 0 & \text{if } \boldsymbol{x} \cdot \boldsymbol{\theta} + b \leq 0 \\ 1 & \text{otherwise} \end{cases}$$

Consider italicising terms that are new, like “activation function”.

The perceptron eventually evolved into the modern unit, which computes the output value as a *range* of values, obtained by applying an activation function to the sum of its inputs and bias. With this modification, a small change in the inputs only resulted a small change in the output, allowing a more convenient way to gradually modify the weights and consequently, improve the learning algorithm [Nie15].

There are various activation functions; historically, the most commonly used is the sigmoid function, $\sigma(x) = 1/(1+e^{-x})$. Its advantages and disadvantages are outlined in 2.6. [Kar16] recommends using less expensive functions with better performance, such as,

1. Tanh function, $tanh(x) = 2\sigma(2x) - 1$.
2. Rectified Linear Unit (ReLU) function, $f(x) = max(0, x)$.
3. Leaky ReLU function, $f(x) = 1(x < 0)(\alpha x) + 1(x >= 0)(x)$
4. Maxout function, $max(w_1^T x + b_1, w_2^T x + b_2)$.

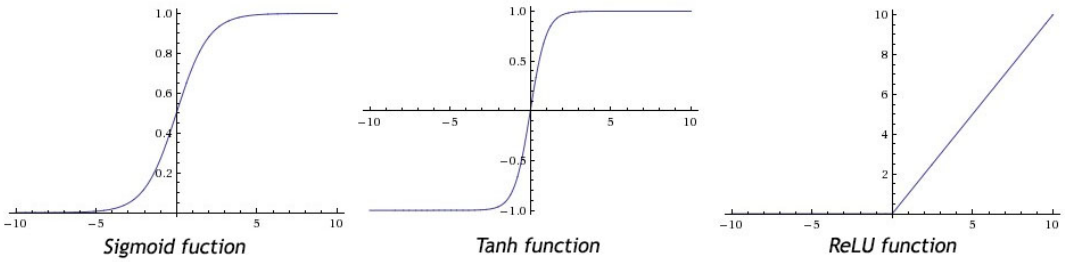

Figure 2.2: Graphs of some common activation functions [Kar16].

Remove "thus to summarise".

Thus to summarise, a unit's output can be expressed as $g(\boldsymbol{x} \cdot \boldsymbol{\theta} + b)$, where $g(x)$ is the chosen activation function.

As previously mentioned, the general architecture of a neural network can be described as distinct layers of these units, which are connected to units in its adjacent layers. The most common layer type is *fully-connected*, which means that all units in the adjacent layers are pairwise connected [Nor14].

pairwise connected /= fully connected. I think you mean "each unit in a layer is connected to every unit in an adjacent layer".

The input layer receives input values corresponding to the number of features[1] in the neural network. The last, or output layer may be composed of different classifications in a multi-classification problem or ouput a certain value in a prediction problem (FIX THIS). The layers in between input and output layers are called hidden layers; a neural network is classified as DNN if it contains more than one hidden layer. Increasing the size and numbers of hidden layers also increases the *capacity* of the neural network [Kar16]; that is, the space of its representable functions. However, this may undesirably result in *overfitting*[2].

italicise

There are numerous neural network classifications depending on their architecture; the design relevant to this thesis is supervised[3] FFBP neural network. Its training process

[1]For instance, in a 200×200 pixel image recognition problem, there may be $40,000$ features corresponding to each individual pixel's RGB values.

[2]Overfitting refers to modeling the learning algorithm to excessively fit to the training samples. It thereby increases the risk of including unnecessary noise in the data, resulting in more inaccurate model.

[3]As in "supervised learning", a concept in machine learning where a set of training examples is paired up with a set of corresponding desired output values. A supervised

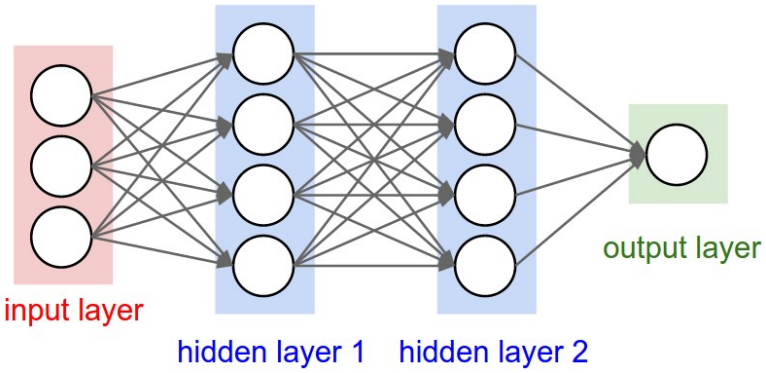

Figure 2.3: An example of a 2-layer neural network [Kar16].

and mathematical representation are briefly outlined in 2.2.

Finally, [Nor14] summarises the attractiveness of neural networks as follows:

Might be worth briefly comparing to other ML techniques here in a proper section. This paragraph currently sticks out a bit awkwardly.

1. Its capacity to support parallel computation.

2. Its fault tolerant nature against "novel inputs".

3. Graceful degradation[4].

   Don't put this in a footnote, explain it properly.

4. The usage of inductive learning algorithms to train the networks.

2.2 Feed-forward back-propagation learning algorithm

This section explains the mechanics and the mathematical representation involved in training a fully-connected, supervised FFBP neural network, based on the works

in, not by. [Ng16] is a citation not a person

by [Ng16]. Given a set of features and training samples, this learning algorithm aims to find the correct weight distribution in the neural network in two stages: feed-forward propagation and back-propagation.

---

neural network thus adjusts its links' weights to get the correct output values during its training phase.

[4]A gradual performance drop-offs in worsening conditions.

First, the weights are randomly initialised within the permitted range of the chosen activation function. For instance, this range is $[0, 1]$ for a sigmoid function; for a tanh function, it is $[-1, 1]$.

Now, in a k-layer neural network, let the size or the number of units in layer j be denoted as $|j|$. Let a particular training sample, s, be denoted as $(\boldsymbol{x}^{(s)}, \boldsymbol{y}^{(s)})$, such that $\boldsymbol{x}^{(s)} = [x_1^{(s)}, ..., x_m^{(s)}]$ is the sample input and $\boldsymbol{y}^{(s)} = [y_1^{(s)}, ..., y_n^{(s)}]$ is the matching desired output. Let $a_i^{(j)}$ be the activation value of unit i in layer j, where $1 \leq i \leq |j|$, $1 \leq j \leq k$. Let $g(x)$ be the activation function; $\Theta_{qp}^{(j)}$ be the weight of the link from unit p in layer j to unit q in layer $j+1$; and, let $\Theta^{(j)}$ be the matrix of weights controlling function mapping from layer j to $j+1$. Aren't these last two "lets" defining the same thing?

Then we can express $a_i^{(j)}$ as,

$$a_i^{(j)} = g(\Theta_{i1}^{(j-1)} a_1^{(j-1)} + \Theta_{i2}^{(j-1)} a_2^{(j-1)} + ... + \Theta_{i|j-1|}^{(j-1)} a_{|j-1|}^{(j-1)}) \tag{2.1}$$

For instance, the activation of unit i in the first hidden layer can be expressed as,

$$a_i^{(2)} = g(\Theta_{i1}^{(1)} x_1^{(s)} + ... + \Theta_{im}^{(1)} x_m^{(s)})$$

and, in the output layer as,

$$a_i^{(k)} = g(\Theta_{i1}^{(k-1)} a_1^{(k-1)} + ... + \Theta_{i|k-1|}^{(k-1)} a_{|k-1|}^{(k-1)})$$

2.1 can be simplified using vectorised implementation. Let the activated units in layer j be denoted as $a^{(j)} = [a_1^{(j)}, ..., a_{|j|}^{(j)}]$. Then inputs to this layer can be expressed as $z^{(j)} = \Theta^{(j-1)} a^{(j-1)}$ and so $a^{(j)}$ becomes,

$$a^{(j)} = g(z^{(j)}) \tag{2.2}$$

Forward-propagation process ends when the input values are thus propagated to the output layer.

Next, back-propagation involves re-distributing the error value between the expected output, $\boldsymbol{y}^{(s)}$, and actual output, $a^{(k)}$, back from the output layer through the hidden layers [RHW86]. This concept is based on the idea that the previous layer is responsible for some fraction of the error in next layer, proportional to the links' weights.

Let $\delta_i^{(j)}$ denote the error value in unit i in layer j and $\delta^{(j)} = [\delta_1^{(j)}, ..., \delta_{|j|}^{(j)}]$ be the vectorized error values. Then, for $1 < j < k$,

$$\delta^{(j)} = (\Theta^{(j)})^T \delta^{(j+1)} .* g'(z^{(j)}) \tag{2.3}$$

use \! to reduce the space in math mode between . and *

where $.*$ is an element-wise multiplication. Note that the error in the output layer is $\delta^{(k)} = a^{(k)} - \boldsymbol{y}^{(s)}$ and that there is no error in the first layer, because as input values, they cannot contain error.

"Note that" is academic speak for "I forgot to mention earlier that.." Just remove "Note that".

weird space here

Finally, the error in link weight $\Theta_{qp}^{(j)}$ is denoted as $\Delta_{qp}^{(j)}$, such that,

$$\Delta_{qp}^{(j)} = a_p^{(j)} \delta_q^{(j+1)}$$

This, too, can be simplified using vectorized implementation as,

$$\Delta^{(j)} = \delta^{(j+1)} (a^{(j)})^T \tag{2.4}$$

Back-propagation ends for s when the errors from the output layer is propagated to the first hidden layer.

The FFBP learning algorithm is then repeated for all the training samples. $\Delta^{(j)}$ accumulates all the errors in the training set during this process. Once the process is finished, the final values are averaged out by the size of the training set,[a]regularisation parameter[5] added, before finally updating the weights. This entire process is a a form of *batch gradient descent* (BGD) learning algorithm in machine learning [Ng16].

is

and finally the weights are updated.

the

what's a regularisation parameter?

There are other advanced optimization methods that can improve the performance of neural networks, but these has not yet been explored at the time of this report. As with other machine learning algorithms, [Ng16] notes that the performance may also be improved with either altering various parameters, such as regularisation parameter λ and the learning rate parameter α, altering the number of features, gaining more training examples, adding polynomial features, or any combination of these. However, [Ng16]

have yet to be

[5]Features that are vastly different in its range of input values should be regularized to avoid distorting the results [Ng16].

It's not hugely important, but try to avoid "Yin. However, Yang" paragraphs.

This example may be improved by placing the "However" more creatively for emphasis, e.g [Ng16] also states, however, that diagnostic analysis is often…

also states that *diagnostic analysis*[6] is often the fastest and most economic method to [redacted] of features is effective.

economical

determine which set

2.3 GPU-accelerated programming and neural networks

have shown a significant performance improvement.

Neural networks implemented with GPU-accelerated programming has shown to be [redacted]. For instance, [OJ04] had a 20-fold performance enhancement of a multi-layered perceptron (MLP) network to detect text using an ATI RADEON 9700 PRO in 2004. More recently, [GdAF13] showed that performance gains from parallel implementation of neural networks in GPUs are scalable with the processing power of the GPU used. Their results show that performance enhancement over the pure CPU implementation increased with dataset size before reaching a plateau, a limit which they contribute to the saturation of the GPU's processing cores.

[NVI14] explains that GPU computing is particularly well-suited to neural networks due to its massively parallel architecture , consisting of tens of thousands of cores that consists of tens of thousands of cores. Thus, as single GPUs can hold entire networks of neural networks, they state that neural networks benefit from an overall bandwidth increase, reduction in communication latency, and a decrease in size and power consumption compared to a CPU cluster. Furthermore, as neural network units essentially repeat the same computation with just differing input values, it complements the GPU as there is minimal need for conditional instructions that could trigger thread divergences, which can dramatically reduce the GPU throughput.

neurons?

only.

differing

the algorithm

architecture

The main GPU programming language is CUDA, an API model created by NVIDIA in 2007. It has a C/C++ language style and the added benefits that it bypasses the need to learn graphics shading languages, or learn about computer graphics in order to program the GPU. The alternative to CUDA is Open Computing Language (OpenCL) released in 2009; also based on C/C++, it is considered to be the more complicated

no comma

[6]Diagnostic analysis is a form of trial-and-error testing of features sets, or *hypotheses*, with small training examples to assess its performance and thus its suitability.

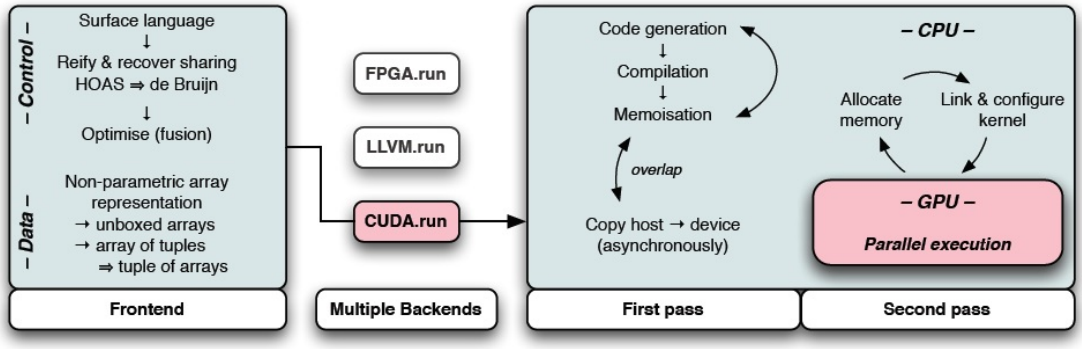

Figure 2.4: Structure overview of `Data.Array.Accelerate`. [CKL+11]

language of the two, but with enhanced portability. As both languages are very low-level [Mar13], this asks for a way to create, manipulate and test neural networks in a more uncomplicated, user-friendly and safe high-level functional language.

there is a need for a way…

less complicated, more user-friendly, safer, higher-level language, such as a functional language.

2.4 Accelerate

How about this definition:
EDSLs are restricted languages that are embedded in a more powerful language, so as to reuse the host language's infrastructure and enable powerful metaprogramming.

embedded domain specific language (EDSL)

Accelerate is an EDSL for GPU programming, released by UNSW PLS in 2011. EDSL is a type of language that integrates into host language and where host language can generate embedded code. In the case of Accelerate, Haskell is the host language and it compiles into CUDA code that runs directly on the GPU.

— accelerate programs take arrays as…

provides

Accelerate is a framework for programming with arrays [Mar13], [illegible] input and outputs one or more arrays. The type of Accelerate arrays is,

output

```
data Array sh e
```

dimensionality

where `sh` is the *shape*, or number of dimensions of the array, and `e` is the *element type*, for instance `Double`, `Int` or tuples. However, `e` cannot be an array type; that is, Accelerate does not support nested arrays. This is because GPUs only have flat arrays and do not support such structures [Mar13].

Types of array shapes and indices are shown in Fig. 2.5 [CKL+11]. It shows that the simplest shape is `Z`, which is the shape of an array with no dimensions and a single

```
data Z            = Z
data tail :. head = tail :. head

type DIM0 = Z
type DIM1 = DIM0 :. Int
type DIM2 = DIM1 :. Int
type DIM3 = DIM2 :. Int

type Array DIM0 e = Scalar e
type Array DIM1 e = Vector e
```

Figure 2.5: Types of array shapes and indices [CKL+11].

element, or a scalar. A vector is represented as `Z:.Int`, which is the shape of an array with a single dimension indexed by `Int`. Likewise, the shape of a two-dimensional array, or matrix, is `Z:.Int:.Int` where the left and right `Int`s denotes the row and column indexes or numbers, respectively.

Common shapes have type synonyms, such as scalars as `DIM0`, vectors as `DIM1` and matrices as `DIM2`. Similarly, common array dimensions of zero and one also have type synonyms as `Scalar e`, `Vector e`, respectively.

Arrays can be built using `fromList`; for instance, a 2×5 matrix with elements numbered from 1 to 10 can be created with,

```
fromList (Z:.2:.5) [1..]::Array DIM2 Int
```

To do an actual Accelerate computation on the GPU, there are two options [Mar13]:

1. Create arrays within the Haskell world. Then, using `use`, inject the array `Array a` into the Accelerate world as `Acc (Array a)`. Then, use `run`.
2. Create arrays within the Accelerate world. There are various methods, such as using `generate` and `fill`. Then, use `run`.

Didn't mention "acc" before here. How about. "The type "Acc t" denotes an accelerate computation that returns a value of type t. Put it somewhere above this.

copies the final values back into Haskell after execution.

`run` executes the Accelerate computation and retrieves the final values back into the Haskell world. As the first method may require the array data to be copied from

```
-- build an array in Haskell world
fromList :: (Shape sh, Elt e) => sh -> [e] -> Array sh e

-- to execute an Accelerate computation on the GPU
run :: Arrays a => Acc a -> a

-- inject Haskell world array into Accelerate world
use :: Arrays arrays => arrays -> Acc arrays

-- create an array in Accelerate world (array filled with user-specified
    function)
generate :: (Shape ix, Elt a)
         => Exp ix -> (Exp ix -> Exp a) -> Acc (Array ix a)

-- create an array in Accelerate world (array filled with same values)
fill :: (shape sh, Elt e) => Exp sh -> Exp e -> Acc (Array sh e)
```

Figure 2.6: Some Accelerate functions [Mar13].

computer's main memory into the GPU's memory, the second method is generally more efficient [Mar13]. These functions are outlined in Fig. 2.6.

Accelerate is further discussed in 2.6.

2.5 Advantages of Accelerate

There are several advantages of using Accelerate. First, it results in much simpler source programs as programs are written in Haskell syntax; Accelerate code is very similar to an ordinary Haskell code [Mar13]. For instance, Fig. 2.7 shows a dot product function comparison in Haskell and in Accelerate by [McD13]. There are minimal syntactic difference; for example, in the input and output types (Haskell's `dotp` takes and gives Haskell w[illegible] arrays while Accelerate's `dotp` deals with only Accelerate w[illegible] arrays), and in that Haskell's `foldl` is a left-to-right traversal, whereas Accelerate's `fold` is neither left nor right as it occurs in parallel [McD13].

Secondly, as Accelerate is embedded in Haskell, it can benefit from inheriting Haskell's functional language characteristics. For instance, Haskell as a pure language is advantageous for parallel computations as it will prohibit side effects that can disrupt other

```haskell
-- dot product in Haskell
dotp :: [Float] -> [Float] -> Float
dotp xs ys = foldl (+) 0 (zipWith (*) xs ys)

-- dot product in Accelerate
dotp :: Acc (Vector Float) -> Acc (Vector Float) -> Acc (Scalar Float)
dotp xs ys = fold (+) 0 (zipWith (*) xs ys)
```

Figure 2.7: Haskell and Accelerate versions for dot product [McD13]

threads; and, GPUs particularly require an extremely tight control flow due to massive numbers of threads that are generated. Another Haskell characteristic is having a more powerful type system, which could enforce a stronger checking for certain properties - thereby catching more errors - at compile time.

An example of this is the use of two types, Acc and Exp, which separates array from scalar computation and prevents many cases of nested data parallelism, which is unsupported in accelerate.

Lastly, [CKL+11] states that Accelerate is a dynamic code generator and as such, It can

Don't defer to the paper too much. Just say: Lastly, Accelerate is a dynamic code generator [CKL+11], and as such it can

- optimise a program at the time of code generation by obtaining information about the hardware capabilities simultaneously.
- Customise the generated program to the input data at the time of code generation.
- Allow host programs to "generate embedded programs on-the-fly".

On the other hand, the disadvantages of using Accelerate over CUDA are the extra overheads, which may originate at runtime (such as runtime code generation, kernel loading, data transfer, and so on) and/or at execution time (such as from dynamic code generation) [CKL+11].

runtime code generation is the same thing as dynamic code generation.. Kernel loading and data transfer sound like execution time overheads, and code generation sounds like a runtime overhead.

Some of the overheads however, such as dynamic kernel compilation overheads, may not be so problematic in heavily data- and compute-intensive programs, because its proportion to the total time taken by the program may become insignificant [CKL+11]; and, neural networks certainly fit in such a category of programs. Accelerate also uses a number of other techniques to mitigate the overheads, such as fusion, "sharing recovery", caching and memoisation of compiled GPU kernels.

the

no quotes

cite

In terms of performance, Accelerate can be competitive with CUDA depending on the nature of the data input and the program [MCKL13].

2.6 Previous Implementations

A neural network in Accelerate is a fairly new concept and as such, there is only one known work in [Eve16]. In it, Everest (2016) implements a FFBP neural network with a stochastic gradient descent (SGD) learning algorithm. This section briefly covers the details of this implementation.

SGD algorithm is a modification of BGD algorithm, shown in 2.2. Rather than adjusting the neural network parameters after going through the entire training set, SGD updates the parameters immediately after doing the FFBP algorithm for a single training example, which is picked at random [Ng16]. As such, [Ng16] states that SGD is computationally less expensive than the BGD, and allows the training algorithms to scale better to much larger training sets. The disadvantages of SGD are that it is less accurate than BGD, and that it may take longer to reach the local/global minima.

don't defer to the cite, just state it and cite.

The activation function is the sigmoid function. This function is mathematically convenient, because its gradient, $\sigma'(x)$, is easy to calculate:

$$\sigma'(x) = \sigma(x) \cdot (1 - \sigma(x))$$

According to [Kar16], the sigmoid function is now mostly unused due to having these two disadvantages:

1. When the unit's activation *saturates* at either tail of 0 or 1, the gradient of the sigmoid function nears zero (refer to the sigmoid function graph in Fig. 2.2). Zero gradients effectively "kill" any signal flows through the neuron in forward- and back-propagation. Thus saturated neurons results in a "network [that] will barely learn" [Kar16]. Accordingly, extra care must be taken not to initialise the weights with a large value.

```
-- apply supplied function element-wise to corresponding elements of two
    input arrays to produce a third array
zipWith :: (Shape ix, Elt a, Elt b, Elt c)
        => (Exp a -> Exp b -> Exp c)
        -> Acc (Array ix a) -> Acc (Array ix b) -> Acc (Array ix c)

-- map function within the Accelerate world
map :: (Shape ix, Elt a, Elt b)
    => (Exp a -> Exp b) -> Acc (Array ix a) -> Acc (Array ix b)

-- replicate an array across one or more dimensions according to the
    first argument (a generalised array index)
replicate :: (Slice slix, Elt e)
          => Exp slix -> Acc (Array (SliceShape slix) e)
          -> Acc (Array (FullShape slix) e)

-- reduce the innermost dimension of an array
fold :: (Shape ix, Elt a)
     => (Exp a -> Exp a -> Exp a) -> Exp a
     -> Acc (Array (ix:.Int) a) -> Acc (Array ix a)

-- wraps a value in Exp
lift :: Z:.Exp Int:.Exp Int -> Exp (Z:.Int:.Int)

-- deconstruct Exp, get the structured value from within
unlift :: Exp (Z:.Int:.Int) -> Z:.Exp Int:.Exp Int
```

Figure 2.8: Key Accelerate functions used in [Eve16].

2. The range of the sigmoid function is not centered around zero. Hence if the inputs are always positive, it could introduce an undesirable *zig-zagging dynamics* in the gradient updates for weights, as the gradient on the weights will be either all positive or all negative during back-propagation. H[redacted]r [redacted]. This is only a minor inconvenience, however.

Lastly, I briefly analysed what Accelerate functions were used in the implementation. There are six key functions that were used, and their names and operations are listed in Fig. 2.8.

Two key functions were created to support forward- and back-propagation processes, which are `mvm` and `cross` (see Fig. 2.9). In `mvm`, `h` first stores the row number of the matrix input using `unlift`. The vector input is then replicated `h` times across the

```
-- matrix vector multiplication function
mvm :: (Elt a, IsNum a)
    => Acc (Matrix a) -> Acc (Vector a) -> Acc (Vector a)
mvm mat vec
  = let Z:.h:._ = unlift (shape mat) :: Z:.Exp Int:.Exp Int
  in A.fold (+) 0 $ A.zipWith (*) mat (A.replicate (A.lift (Z:.h:.All)) vec)

-- vector multiplication function
cross :: Acc (Vector Float) -> Acc (Vector Float) -> Acc (Matrix Float)
cross v h = A.zipWith (*) (A.replicate (lift (Z:.All:.size h)) v)
                          (A.replicate (lift (Z:.size v:.All)) h)
```

Figure 2.9: Support functions for forward- and back-propagation used in [Eve16].

first dimension using `replicate`. It is then element-wise multiplied with `mat` using `zipWith`. Lastly, the values are summed and folded into a vector using `fold`.

The process is similar in `cross`, which is a vector multiplication function. First, vector `v` is replicated $|h|$ number of times in the second dimension, whereas vector `h` is replicated $|v|$ number of times in the first dimension before they are element-wise multiplied using `zipWith`. This is equivalent to the two vectors multiplying to create a matrix.

Chapter 3

Thesis Proposal

3.1 Choice of neural network

For this project, I have chosen a CNN to be implemented in Accelerate, based on the works of [Eve16]. CNN is currently one of the most reliable and expedient performer in terms of image recognition. One of best known architecture is based on the works of [LeC89] called the LeNet architecture.

CNN uses MLP feed-forward system, but only processes portions of the input image, which are tiled in subsequent layers in such a way that each input regions overlap and is deemed to obtain a better representation of the original image.

The advantages of this network are that it requires relatively little pre-processing, it uses same weight for each pixel the layers while maintaining good performance.

i think you forgot a word?

3.2 Measuring performance

I assume this will be written later?

– the need to find a large enough sample to stress test the implementation

3.3 Areas requiring development

There are a number of problem areas that may require further attention and development in order..

I identify these following points as problem areas requiring further attention and development in order to achieve the thesis goal. Each points accompanied by my strategies to resolve them:

point is

- Need more familiarity with Haskell and Accelerate in general:
  1. Learn how to utilise Accelerate through accelerate-examples package.
  2. Learn Haskell in more depth from the leftover COMP3141 materials.
  3. Learn COMP3161 in semester 2, 2016.
     why? to gain more familiarity with type systems and compilers?
- Gain information about practical implementation of convolutional neural networks:
  1. Learn from [Kar16], which seem cover convolutional neural network in detail. If more information is required, do more research.
  2. Look for implementations in other languages.
  3. Try to make a simple, working implementation as early as possible, and find ways to improve its performance from there.
- Search for ways to improve the performance of the program once implemented:
  1. Learn if program can be improved using parallel algorithms with better performance.
  2. Learn COMP4121 in semester 2, 2016. once again, why?
  3. Further investigation required.
- Identify an appropriate set of large sample materials to test the performance of the implementation. Strategies to target this is outlined in 3.2.
- Obtain a similar convolutional neural network implementation in CUDA for comparison analysis. I believe that this is one of the latter tasks to be completed and plan to discuss with my supervisor regarding the issue at a later point.

3.4 Expected schedule

Current schedule is to continue research and gain necessary knowledge and skills for the rest of 2016. The projected timeline for Thesis B and completion are in Semester 1, 2017. Put the chart in from your talk.

Chapter 4

Conclusion

An implementation of a neural network in Accelerate may offer a convenient alternative to current CUDA implementations to test the validity of hypotheses.

.. need a bit more than that. Let me know if you need help writing here.

Bibliography

[CKL+11] Manuel M. T. Chakravarty, Gabriele Keller, Sean Lee, Trevor McDonell, and Vinod Grover. Accelerating haskell array codes with multicore gpus. *Declarative Aspects of Multicore Programming*, 2011.

[Eve16] Rob Everest. Untitled phd thesis. 2016.

[GdAF13] Sskya T. A. Gurgel and Andrei de A. Formiga. Parallel implementation of feedforward neural networks on gpus. pages 143–149, 2013.

[Kar16] Andrej Karpathy. Neural networks part 1: Setting up the architecture. http://cs231n.github.io/neural-networks-1/, accessed 01/01/2016 to 30/05/2016, 2016. Course notes from CS231n: Convolutional Neural Networks for Visual Recognition at School of Comp. Sci. Stanford University.

[LeC89] Y. LeCun. Generalization and network design strategies. *University of Toronto Connectionist Research Group Technical Report*, 89, 1989.

[Mar13] Simon Marlow. *Parallel and Concurrent Programming in Haskell*. O'Reilly Media, Sebastopol, CA, 1st edition, 2013.

[McD13] Trevor L. McDonell. Gpgpu programming in haskell with accelerate. https://speakerdeck.com/tmcdonell/gpgpu-programming-in-haskell-with-accelerate, accessed 01/04/2016, 2013. YOW! Lambda Jam 2013 Workshop.

[MCKL13] Trevor McDonell, Manuel M. T. Chakravarty, Gabriele Keller, and Ben Lippmeier. Optimising purely functional gpu programs. *ACM SIGPLAN International Conference on Functional Programming*, 2013.

[MP43] Warren S. McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *Bulletin of Mathematical Biology*, 5:115–133, 1943.

[Ng16] Andrew Ng. Machine learning. http://www.coursera.org/learnmachine-learning, accessed 01/01/2016 to 30/05/2016, 2016. Coursera Inc.

[Nie15] Michael Nielson. *Neural Networks and Deep Learning*. Determination Press, 2015.

[Nor14] Russell Norvig. *Artificial Intelligence A Modern Approach*. Pearson Education Limited, New York City, New York, 3rd edition, 2014.

[NVI14] NVIDIA. Cuda spotlight: Gpu-accelerated deep neural networks. https://devblogs.nvidia.com/parallelforall/cuda-spotlight-gpu-accelerated-deep-neural-networks, accessed 14/04/2016, 2014. NVIDIA CUDA Spotlight.

[OJ04] Kyoung-Su Oh and Keechul Jung. Gpu implementation of neural networks. *Pattern Recognition*, 37:1311–1314, 2004.

[RHW86] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning representations by back-propagating errors. *Nature*, 323:533–536, 1986.

[Ros62] Frank Rosenblatt. *Principles of neurodynamics; perceptrons and the theory of brain mechanisms*. Spartan Books, Washington, D.C., 1962.