Increasing Portable Machine Learning Performance by Application of Rewrite Rules on Google Tensorflow Data Flow Graphs

Rudi Horn

Master of Science

School of Informatics

University of Edinburgh

2016

Abstract

Machine Learning is an important field that is usually limited by execution performance. The approach commonly used to solve this problem is to make use of parallelism provided by hardware such as Graphics Processing Units. This allows for much higher execution performance but introduces the new challenge of needing to write efficient code for them. Languages such as OpenCL and CUDA provide methods to write performant code that can run on such hardware, but these still require the user to manually map the parallelism and know how to produce efficient code. Yet even with extensive knowledge of the underlying platform, performing these optimizations to produce efficient code can be difficult.

The solution to this has been to use domain specific languages such as Google Tensorflow, made specifically for machine learning applications. Tensorflow allows the user to define the machine learning problem without knowing the technical details. Tensorflow makes use of the basic linear algebra package from NVIDIA called cuBLAS, which offers binaries of efficient kernels for common mathematical operations. However these binaries are only usable on specific devices. Instead of using preimplemented libraries an attempt is made to convert Google Tensorflow segments into a functional representation of the Lift programming language. This functional representation can then be converted into the OpenCL language and executed on a number of devices including CPUs and GPUs. It also allows the functional representation to be optimized using rewrite rules so that it can be run efficiently on any device that is supported by the Lift programming language. This approach also allows multiple operations to be combined in order to reduce unnecessary overhead required for calling multiple kernels.

It is demonstrated that it is possible to convert Google Tensorflow graphs into this functional programming language and that they can be compiled. Furthermore it is shown that an automatically generated and then hand tuned functional expression can be optimized so that it has comparable and up to 2 times the performance of the cuBLAS implementation. Additionally the work presented here is demonstrated to work efficiently on a platform not supported by the cuBLAS library.

Acknowledgements

I would like to thank the team responsible for the Lift programming language, as the work presented here is all based on and was only possible through their project. I would like to thank Christophe Dubach for supervising this project and giving me feedback and ideas during its progress. My thanks also goes to Toomas Remmelg, who helped out with his experience on how to efficiently run and execute Lift kernels and also to Michel Steuwer who helped setting things up and supported me with any questions that arose

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Rudi Horn*)

Table of Contents

Chapter 1

Introduction

Machine learning has been proven to be a very successful approach for many different applications that are not easily solvable using traditional methods. Recent advances in machine learning have allowed previously difficult tasks such as image recognition to become usable. An example of this is the use of deep neural networks in the field of image recognition, where neural network based algorithms have not only been able to outperform other approaches, but have also been able to offer a usable approach to the problem [1, 2]. However machine learning is a computationally expensive task requiring complicated calculations such as softmaxes on large amounts of data. Furthermore the modern trend has been to introduce networks of great depth, which require more and more of these calculations to be performed more often [3].

The solution to this problem has been to make use of extensive parallelization. The most common approach has been to make use of Graphics Processing Units (GPU), which contain thousands of computation cores [4]. Using GPUs for computations is called General-purpose computing on GPU (GPGPU) and has become readily available on consumer hardware, while also having an increase in enterprise GPGPU hardware popularity. This allows the problem to be divided into many subtasks and have the GPU process them in parallel.

The problem with having such massively parallel hardware, is that it is necessary to write efficient code that makes good use of the parallel hardware. This is especially important, since the same code is usually executed so many times, that a small impact in performance can make a very big difference in the total runtime of the application. There have been many attempts to write frameworks and offer solutions for executing

code on GPUs. One approach has been to develop Domain Specific Languages (DSL), which allow someone who has knowledge of the problem to be able to solve it without knowing much about the technical details. This is the approach taken by Google Tensorflow, which allows the user to define a neural network out of tensors (also called nodes) which define specific mathematical operations such as sums and softmaxes [5]. These tensors then take inputs in the form of other tensors, in order to be able to execute steps of computations. Tensorflow internally makes use of highly performant libraries for specific operations, such as the CUDA Basic Linear Algebra Subroutines (cuBLAS) [6].

It is also possible to solve the problem by using OpenCL [7], which relies on the user programming C-type code, which is then compiled and executed on the hardware. The problem with the compiler based approaches is that it is up to the user to determine how best to map parallelism to the specific underlying hardware. This makes it quite difficult to achieve ideal performance, especially since there are many underlying hardware factors that can affect this. Furthermore it makes it much more difficult to port code to different hardware or even hardware platforms, which is described as the portable performance. The approach taken by Steuwer et al. was to define the problem as a functional approach, on which rewrite rules can be applied that affect the paralellel performance, but don't affect the outcome [8]. This functional approach is then converted into highly performant OpenCL code, that can then be executed on the underlying hardware.

The approach taken here is to try and take a data flow graph produced by Google Tensorflow, and to convert it into a functional expression. This approach could allow a single kernel to be produced for a subsection of a data flow graph. Then the rewrite rules from Steuwer et al. could be applied in order to increase its performance for the specific underlying platform. That way it is possible to generate computationally efficient code that can run on all sorts of different platforms. It is also possible that by combining different kernels together, it may be possible to reduce overhead and thus also compete with the NVIDIA cuBLAS implementation. The exact determination of which rewrite rules to use could be done using further machine learning approaches or similar.

1.1 Contributions

The goal of this project is to be able to execute as much of an existing Google Tensorflow machine learning application within the data parallel programming language Lift. This approach allows the Lift programming language to find a method in order to find an optimized kernel for the specific target platform, by applying different rewrite rules in order to produce functionally equivalent code. Google Tensorflow produces a data flow graph, consisting of different nodes that can perform operations on data. These nodes define a graph which can be exported into the Google Protocol Buffers[1] format. Google Tensorflow then offers a runtime environment, which can handle variables and allow specific nodes to be executed under the input of certain parameters. Some of these nodes are control nodes, and represent variable inputs, variable assignments, placeholders and constants. Other nodes perform operations on arrays such as reshaping them or determining their shape. Finally the most important nodes are those used to perform mathematical operations, such as matrix multiplications, additions and softmaxes.

Using the Tensorflow data flow graph it is possible to try and group as many of the computationally expensive mathematical operations and to convert them into a functional representation that can be executed by the Lift runtime. This functional representation of the combined mathematical operations can then be optimized using the Lift framework. This is done by producing different yet equivalent kernels and trying to find the most optimal one. The idea behind this is that even though it is not possible to use highly optimized kernels for specific operations such as matrix multiplications, it may still be possible to produce a kernel which makes use of most of the performance capabilities while avoiding the overhead of having to apply multiple different kernels. Furthermore it is not necessary to have many different kernels for different specific target platforms, but instead one can produce near optimal code for the specific target platform.

As such the goal of this project is to write a Tensorflow runtime, that can read and execute nodes from Tensorflow data flow graphs. This is then applied to the Google Tensorflow example, which makes use of gradient descent and a softmax in order to perform handwriting recognition on the MNIST dataset[2]. As such it is not only nec-

[1] `https://developers.google.com/protocol-buffers/`

[2] `https://www.tensorflow.org/versions/r0.9/tutorials/mnist/beginners/index.html`

essary for the runtime to be capable of handling nodes which can be converted into a functional representation that can be compiled by the Lift framework, but also for it to be able to manage variables, placeholder inputs, constants, array operations and any operation that could only be executed in software. This requires it to have different components. The first is a method of loading and passing the graph from the Google Protocol Buffers format it is stored in. Secondly a method of group graph nodes into units that are either executable in software or convertable into a functional approach is required. Next, a type inference system is needed, since the functional representation is often dependent on the input type and certain operations such as *rank* and *shape*. Finally a session manager is necessary in order to handle variables and to handle node execution.

The runtime is then evaluated on the simple MNIST handwriting example consisting of a single layer neural network. The execution performance was measured on a representative subgraph in order to set an initial benchmark. It is also shown that by hand tuning the selected subgraph's kernel it is possible to outperform the Google Tensorflow cuBLAS implementation. Finally the subgraph is also tested on hardware that is not supported by the cuBLAS library in order to demonstrate that it is platform portable.

1.2 Thesis Structure

Section 2 contains some background information required to understand the work presented here. It contains some information on Google Tensorflow, the Lift programming language and the OpenCL programming language. Section 3 lists similar work and explains how this project differentiates itself from them. Section 4 describes how the runtime is implemented in Section 4.1 and how the nodes are converted into Lift in Section 4.2. In Section 5 it is shown how the work presented here compares to the original Google Tensorflow implementation. Finally, Section 6 contains a summary of the results, future work and a critical analysis of the work presented here.

Chapter 2

Background

This section explains some relevant background information that is necessary in order to understand this thesis. This project involves taking Google Tensorflow data flow graphs and executing them in a self written execution engine. This execution engine is written in Scala and tries to calculate portions of the data flow graph in the Lift execution environment, by converting them into their functional representation first. Section 2.1 explains what the Lift framework is and how functional expressions can be combined. Section 2.2 gives a brief introduction into what OpenCL is and how it functions. Finally Section 2.3 explains the Google Tensorflow framework and how it can be used to export graphs.

2.1 Lift

The Lift programming language is a data parallel functional programming language. It was specifically designed for producing parallel code, without the user having to worry about how exactly the code should be parallelized. The user defines a functional expression, consisting of *Map*, *Reduce*, *Zip* and other similar operations. This function is then rewritten in a way that more closely defines the technical parallelization that will occur. The rewritten function can then be compiled into OpenCL code, which can be compiled to a kernel that can run on the underlying platform.

The most important elements in the Lift language are expressions and lambdas. An expression is essentially code elements and in the Lift language currently consists of parameters (**Param**) and function calls (**FunCall**). A lambda expression consists out

of an argument list and an expression making use of those parameters. Listing 1 shows how to define lambda functions.

```
// define a simple lambda
val l1 = fun(a => a)

// define a lambda with and specify the parameter type
val l2 = fun(ArrayType(Float, 10), a => a)

// define a lambda with the alternative syntax
val l3 = \(a => a)
```

Listing 1: Defining lambda expressions in Lift.

Some operations such as *Map* and *Reduce* require a lambda expression in order to determine what exactly they do. In the case of the *Map* operation, it takes a lambda expression which defines what to perform on each element in an array. In order to use it, a **Map** instance is constructed with a given lambda expression and it can then be called with a given argument. An example of this is shown in Listing 2. The user functions *mult* and *add* are both functions which take two floating point scalar values and respectively multiply and add them. The example shows how to perform a *Map* by multiplying each element of an array by two and a *Reduce* by summing up all elements of an array. The **Reduce** class requires a lambda function which takes the accumulator and the next array element and performs an operation on them. Furthermore it requires an expression for an initial accumulator value.

```
// multiply all elements of an array by 2
val m1 = \(ArrayType(Float,10), a => Map(\(b => mult(b,2f)))(a))

// sum up an array of elements
val m2 = \(ArrayType(Float,10), a => Reduce(add, 0.0f)(a))
```

Listing 2: Using the *Map* and *Reduce* operations.

Function calls have different syntaxes depending on the type of call. In general it is possible to call lambda expressions using the syntax shown in Listing 2. However using a functional composition operator may in some situations be a nicer syntax. A functional composition is equivalent to calculating the result of the function on the

right and passing it as the sole argument for the function on the left. Lift uses the *o* operator for this. There is an exception for the first functional composition which is performed, as this requires the use of a *$* operator instead. How to perform these function calls is shown in Listing 3.

```
1  // a typical function call
2  val fun = \(ArrayType(Float,10),ArrayType(Float,10), (A,B) =>
3    Map(\(Acell =>
4      Map(\(Bcell => add(Acell, Bcell)))(A)
5    ))(B)
6  )
7
8  // using functional composition, Transpose has to be called
9  // using a $ operator, while Map can be called using an o
10 // operator
11 val p1 = \(ArrayType(ArrayType(Float,14),10),ArrayType(Float,10),
12   (A,B) =>
13   Map(\(Arow => fun(Arow,B))) o Transpose() $ A
14 )
```

Listing 3: Calling functions and functional composition.

This code can the be converted into OpenCL code, which runs on any OpenCL target platform. The next section contains a short introduction to what OpenCL is and how it works.

2.2 OpenCL

The Open Computation Language (OpenCL) is a programming language for writing software that scales across massively parallel hardware such as CPUs and GPUs [7, 9]. In order for an expression written in Lift to be ready to run on a GPU, it is converted into OpenCL code, which is then compiled for the target platform. It works by defining a kernel which is run as different work-items on the individual cores. During execution each work-item is assigned an index in either a one, two or three dimensional index space. It is then possible to determine which index the work-item has and to use this to determine which part of the data the individual computation core has to handle. This

is done by using the *get_global_id(dimension)* function. These work-items can also be grouped into work-groups that allow further synchronization between them.

```
1 kernel void dp_square
2     (global const float *a, global float *result)
3 {
4   int id = get_global_id(0);
5   result[id] = a[id]*a[id];
6 }
```

Listing 4: A simple OpenCL kernel example [9].

Explaining OpenCL in depth is beyond the scope of this thesis, but it is important to have some understing of the memory model. In OpenCL there are different memory spaces which can be seen by different items. Private memory can only be seen by the specific work item. Local memory can only be seen by work-items within the same work-group and global memory can be accessed from all work-items. Computed results have to eventually be written to global memory for them to be accessible. For this the Lift framework offers the *toGlobal* operation. Furthermore Lift also has an operation to write to local memory called *toLocal*.

2.3 Google Tensorflow Data Graphs

Google Tensorflow is a domain specific language primarily designed for machine learning applications. It allows the generation and execution of data flow graphs [5].

An example of a simple graph that takes two variable placeholders is shown in Listing 5. In this example two variables called a and b are defined as placeholders for matrices of dimensions 64×32 and 32×64 (lines 5-8). Using these two placeholders a tensor node called c is defined as the multiplication of a and b (line 10), and then a tensor node called d is defined as the multiplication of c and d (line 12).

Google Tensorflow supports the exportation of dataflow graphs. When exporting the graph generated in Listing 5 to the Google Protocol Buffers representation, the placeholders are converted into tensor nodes containing the shape and data types of the placeholder, as shown in Listing 6. Using these placeholder tensors, the Protocol

```
import tensorflow as tf
import numpy as np

with tf.Session() as sess:
        a = tf.placeholder(tf.float32, shape=[64,32],
                name='a')
        b = tf.placeholder(tf.float32, shape=[32,64],
                name='b')

        c = tf.matmul(a,b, name = 'c')

        d = tf.matmul(c,a, name = 'd')

        tf.train.write_graph(sess.graph_def, 'models/',
                'mult_graph.pb', as_text=True)
```

Listing 5: Code in order to produce a simple matrix multiplication graph.

Buffers file then contains a tensor node for the matrix multiplication that refers to the two placeholder tensors as shown in Listing 7.

Google Tensorflow also supports variables. Variables are tensors that are stored in memory and can be written to or read from. The advantage of variables is that they are stored in the target device memory, meaning that in the case of a GPU it is stored in GPU memory and can be quickly accessed by operations. In contrast, whenever a placeholder is used, it is read from main memory and written to device memory, which can be quite a costly operation.

Google Tensorflow is designed for fast execution. It can make use of different devices, such as CPUs, GPUs and dedicated hardware computation units such as the Tensorflow Processing Unit [10]. For executing code on NVIDA GPUs it makes use of NVIDIA cuBLAS [6] and the CUDA Neural Network Library [11].

```
node {
  name: "a"
  op: "Placeholder"
  attr {
    key: "dtype"
    value {
      type: DT_FLOAT
    }
  }
  attr {
    key: "shape"
    value {
      shape {
        dim {
          size: 64
        }
        dim {
          size: 32
        }
      }
    }
  }
}
```

Listing 6: Example protobuf code for a placeholder.

```
node {
  name: "c"
  op: "MatMul"
  input: "a"
  input: "b"
  attr {
    key: "transpose_b"
    value {
      b: false
    }
  }
  attr {
    key: "transpose_a"
    value {
      b: false
    }
  }
  attr {
    key: "T"
    value {
      type: DT_FLOAT
    }
  }
}
```

Listing 7: Example protobuf code for a matrix multiplication.

Chapter 3

Related Work

There are many different frameworks and approaches for making use of parallelization. The most common method is to define C-type programming language that can be compiled into a kernel that can be run on the computation device. Examples of this include CUDA [5], OpenCL [7] and OpenMP [12]. These approaches all offer only a low level method to program parallelism for the target device. As such they still require the user to explicitely implement parallelism into his code. The approach taken here is not to define a low level language such an OpenCL. Instead it is to make use of functional expressions which are generated from the data flow graph. These are optimized for the available hardware and then converted into OpenCL code.

This is different to Domain Specific Languages (DSL) where a language is defined for a specific problem domain, so that the user doesn't have to have knowledge of the technical details. An example of such a Domain Specific Language is Google Tensorflow, where the user can define a data flow graph, which consists of mathematical operations that should be performed [5]. Google Tensorflow maps operations to fast implementations that are part of the NVIDIA cuBLAS library. This allows it to achieve fast performance, without having to hand tune the parallelization. This however means that the code is restricted to NVIDIA hardware, and thus cannot efficiently run on other platforms easily. The approach taken here is not to define another DSL, but rather to take the existing Tensorflow DSL, and allow it to compile data flow graphs into Lift, a functional programming language which can make use of rewrite rules. Further examples of similar DSL / machine learning library approaches are GPUMLIB [13] and Torch7 [14].

The approach of using rewrite rules in order to improve performance is different to the approach of performing tuning of parallel applications such as in [15, 16]. Tuning only tries to change certain parameters, while rewrite rules can reconstruct and change the code according to equivalences of functionial list expressions. The approach of using rewrite rules allows for more flexibility in adapting the kernel in order to improve performance [17].

Another approach taken is to not use GPUs, but to rather use specialized hardware. This could be reprogrammable hardware such as Field Programmable Gate Arrays (FPGA), which is the approach taken by Cadambi et al. [18]. Similarly specialized hardware can be developed in the form of an Application-Specific Integrated Circuit (ASIC). This was done by Google in order to produce the Tensor Processing Unit [10]. The approach presented here is somewhat compatible to the use of FPGAs, since the Lift programming language could potentially be extended in order to make use of FPGAs for processing the data.

Chapter 4

Technical Implementation

This section contains a description of the technical implementation of the Tensorflow data flow graph execution environment, which makes use of the Lift programming language in order to optimize portable execution performance. Figure 4.1 shows an overview of the required components for this project. This project involves the implementation of the runtime, which includes components for loading Tensorflow graphs, grouping nodes and then converting them into the Lift language. Section 4.1 describes the general structure of a runtime, showing how the user can execute a node in a graph and how such an execution call is handled by the runtime. Section 4.1.1 describes how a graph is loaded. Next, Section 4.1.2 explains how the fusion of nodes into execution units works. Section 4.1.3 describes how, given Lift implementations of different nodes, they can be combined into a single node. Section 4.1.4 is about the functionality of the type inference system. Finally Section 4.2 contains the implementations of different individual Tensorflow operations in Lift.

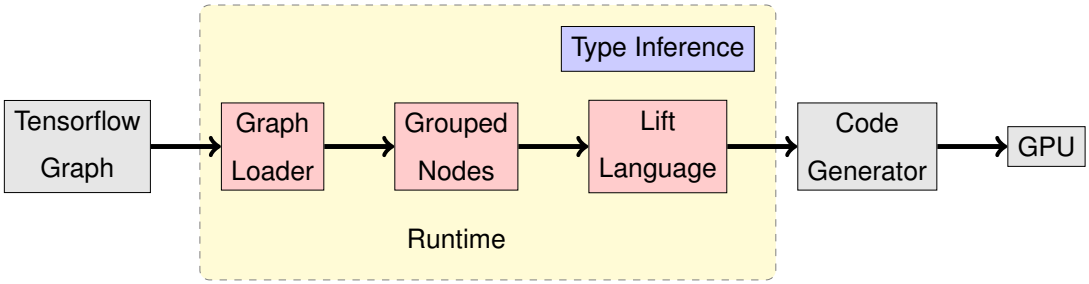


Figure 4.1: An overview of the components required in order to execute Tensorflow graphs in Lift.

4.1 Runtime

The goal of this project is to be able to compile and execute parts of a Tensorflow data flow graph using the Lift programming language. As such, it is necessary to either find a way to hook into the existing Tensorflow runtime or to write a runtime which can load and execute data flow graphs. Since the Lift programming language is tightly integrated with the Scala programming language and execution from within Scala is easier because the integration of different programming languages is not required, the approach of writing a new Tensorflow runtime is preferred. Listing 8 shows how to instantiate a session for an example graph consisting of a simple graph reduction and demonstrates execution with a randomly initialized three dimensional array.

```
1  // Generate a 3-dimensional input array
2  val A = Array.fill(32, 32, 32)
3          (util.Random.nextInt(5).toFloat)
4
5  // create a session and load the graph from the
6  // specified path
7  val session = TensorFlowSession(
8          TestGraphLoading.TestPath +
9          TestGraphLoading.SimpleSumReduction)
10
11 // execute the node called ''res'' and assign the array A
12 // to the placeholder called ''A'', finally cast the
13 // result to an Array[Array[Array[Float]]]
14 val res = session.executeNode("res", "A" -> A)
15         .asInstanceOf[Array[Array[Float]]]
```

Listing 8: Code that creates a session from the specificed graph and executes it on a randomly generated matrix.

When the `TensorFlowSession(/* ... */)` line is executed, the first step that happens internally is that a **GraphLoader** instance is created, which opens and parses the Google Protocol Buffers file, which is described in Section 4.1.1. After this a session object is created, which takes the **GraphLoader** instance.

4.1.1 Graph Loading and Parsing

Before any compilation or execution of code can be performed, it is necessary to be able to read in the data flow graph. The data flow graph is stored in a Google Protocol Buffers format. Protocol Buffers is a binary format which also offers a textual representation of data that follows the format which is defined in a format file. This format file can be compiled into numerous programming languages and is compiled into a Java library for this project. An example of the textual representation of a data flow graph is shown in Listing 9. This section explains how the nodes are represented internally as **LiftNode** classes. It explains how Protocol Buffer data flow graphs are then converted into these **LiftNode** instances. Finally it also explains the different input types, including control dependencies.

The Protocol Buffers input file is parsed into a Scala case class depending on the node type, which for the case of matrix multiplication is shown in Listing 10. The case class extends a general **LiftNode** class and in addition is marked with a **LiftExecutableNode** trait, in order to signify that a functional Lift implementation exists. Each **LiftNode** has a set of inputs and a name. Alternatively a class can be marked with the **SoftwareExecutableNode** trait, if a software implementation exists. The implementation of the **LiftNode** class and **SoftwareExecutableNode** and **LiftExecutableNode** traits are shown in Listing 11.

Once loaded, the list of nodes of type **NodeDef** are initially converted into a map called $tfNodeMap$, which maps the node name to the **NodeDef** instance. Once a **LiftNode** instance is requested using the *getNode* function, the software first checks if it is stored in a cache map called *nodes* and if not it is parsed from the list of nodes. This approach allows only those nodes to be parsed, which are actually going to be executed later. Listing 12 shows how a simplified extract of how the MatMul class instance is loaded. The $tfNode$ variable is the **NodeDef** instance loaded from the Protocol Buffer format. The code `allInputsCheck(2):_*` code snippet finds all the input nodes for this node, and converts them into **LiftNodeInputBase** instances. It also ensures that there are only two actual (non control dependency) inputs. The exact meanings and differences of input types are explained below.

Most Tensorflow operations have a fixed number of inputs, that define where the numerical data comes from, on which the specific operation is executed. In this example the matrix multiplication operation takes two matrices as an input. Inputs are always

```
node {
  name: "gradients/MatMul_grad/MatMul"
  op: "MatMul"
  input: "gradients/add_grad/tuple/control_dependency"
  input: "Variable/read"
  attr {
    key: "T"
    value {
      type: DT_FLOAT
    }
  }
  attr {
    key: "transpose_a"
    value {
      b: false
    }
  }
  attr {
    key: "transpose_b"
    value {
      b: true
    }
  }
}
```

Listing 9: Example for a matrix multiplication stored in the Google protcol buffers file.

```
case class MatMul(name: String, dataType: DataType,
        transposeA: Boolean, transposeB: Boolean,
        inputs: LiftNodeInputBase*)
  extends LiftNode with BinaryNode with LiftExecutableNode
```

Listing 10: The matrix multiplication case class.

other nodes, which could be constants, user inputs or the result of a previous calculation. There are also some operations such as the *DynamicStitch* operation[1], which

[1] https://www.tensorflow.org/versions/r0.9/api_docs/python/array_ops.html#

```scala
abstract class LiftNode {
  val name: String

  val inputs: Seq[LiftNodeInputBase]

  def actualInputs = inputs.filter(input =>
        !input.isInstanceOf[LiftNodeInputDependency])

  def dependencyInputs = inputs.filter(input =>
        input.isInstanceOf[LiftNodeInputDependency])
}

trait LiftExecutableNode extends LiftNode
trait SoftwareExecutableNode extends LiftNode
```

Listing 11: The **LiftNode** abstract class that all operation case classes inherit from, along with the two traits **LiftExecutableNode** and **SoftwareExecutableNode** signifying how they are implemented.

```scala
val node = tfNode.getOp switch {
        /* ... */
        case "MatMul" => MatMul(tfNode.getName,
                tfNode.getAttr.get("T").getType,
                tfNode.getAttr.get("transpose_a").getB,
                tfNode.getAttr.get("transpose_b").getB,
        allInputsCheck(2):_*)
        /* ... */
}
```

Listing 12: Parsing the protocol buffers file into **LiftNode** instances.

also allow an arbitrary number of inputs. Inputs can also have an arbitrary number of dependencies, which have to be executed first. These are denoted by a ^ before the input name, such as `^gradients/add_grad/tuple/group_deps`, and are called

`dynamic_stitch`

control dependencies[2]. Some operations such as the *BroadcastGradientArgs*[3] can also have multiple outputs. In order to access the second output, the input string is followed by a colon (**:**) and a number denoting the output index. An example of this is `gradients/mul_grad/BroadcastGradientArgs:1`.

These input strings are parsed into case classes of the **LiftNodeInputBaseClass**, which are shown in Listing 13. Each case class contains a variable called *node*, which points to the **LiftNode** that should act as the input. Regular inputs are converted into a **LiftNodeInput**, while inputs with an index are converted into a **LiftNodeInputIndex**, which additionally contains an integer denoting the index of the output. Finally all control dependencies are converted into a **LiftNodeInputDependency**, which should be executed before the current node is executed. Control dependencies are not used as inputs for the node.

```
1  abstract class LiftNodeInputBase {
2    val node: LiftNode
3  }
4
5  case class LiftNodeInput(node: LiftNode)
6         extends LiftNodeInputBase
7
8  case class LiftNodeInputDependency(node: LiftNode)
9         extends LiftNodeInputBase
10
11 case class LiftNodeInputIndex(node: LiftNode, index: Int)
12        extends LiftNodeInputBase
```

Listing 13: The **LiftNodeInputBase** class and its case classes.

In the next section the possibilities of how to combine and group nodes is explained.

[2] `https://www.tensorflow.org/versions/master/api_docs/python/framework.html#control_dependencies`

[3] `https://github.com/tensorflow/tensorflow/blob/master/tensorflow/core/ops/array_ops.cc#L1679`

4.1.2 Optimization / Fusing

One of the ways in which it is hoped to be able to achieve performance benefits is by the combination of different mathematical operations into one. By doing this, one hopes to be able to minimize overhead or find optimizations which could help to improve performance. An example of different nodes that could be fused, from the simple MNIST handwriting application, is shown in Figure 4.2. It is possible to fuse the *Mul*, *Sum* and *Reshape* operations into a single Lift expression that can be compiled into one OpenCL kernel. In general any combination of **LiftNode**'s that implement the **LiftExecutableNode** and as such have a Lift implementation can be combined.

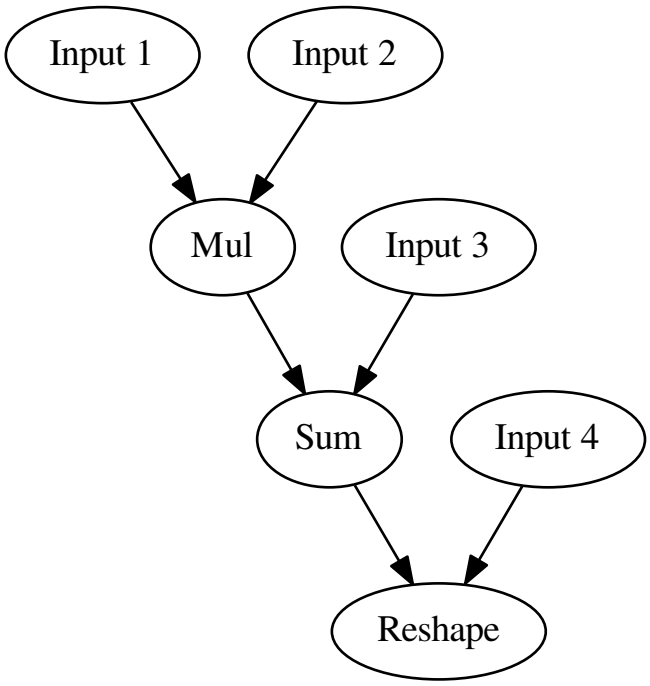


Figure 4.2: An example grouping of a *Mul*, *Sum* and *Reshape* operation automatically generated from the MNIST handwriting example.

When trying to execute a node, the first step taken is to generate an execution unit for that node, if one does not already exist. An execution unit defines a node which is executed and a set of dependencies, which have to be executed first and act as inputs. The general procedure for generating execution units is to determine if the **LiftNode**

has a Lift implementation or a software implementation. If it only has a software implementation, then a simple execution unit for that software implementation is created. If the node has a Lift implementation however, it will traverse up the tree and try grouping all nodes that also have a Lift implementation. In the example in Figure 4.2, *Reshape*, *Sum* and *Mul* have a Lift implementation and can thus be combined into a single execution unit. $Input1$ is a *Placeholder*, $Input2$ a *Tile* operation, $Input3$ a *BroadcastGradientArgs* and $Input4$ a *Shape* operation. Since none of the inputs have a Lift implementation, they are simply converted into inputs, which have to be determined in software beforehand. When trying to execute this execution unit, it will first find out which dependencies it has, and it will then proceed to execute these under the **TensorFlowEnvironment** execution environment.

The next section describes how this list of grouped nodes is compiled into a single Lift expression.

4.1.3 Code Generation with Lift

Once a Lift execution unit such as the one shown in Figure 4.2 has been generated as described in the previous section, it is necessary to convert it into a functional expression. For the moment it is assumed that it is possible to generate a functional expression for all the node types. How they are generated is explained in detail in the Section 4.2.

The goal of the code generation is to take a tree of **LiftNode**'s with Lift implementations and to convert them into a single function. In order to do this it is first necessary to understand what a function is comprised of. A function can be described by a body consisting out of expressions, and a list of parameters that can be found within this body. When calling this function, the user passes a set of inputs to the execution environment, and wherever the parameter is found within the body, the parameter is replaced by the passed input. A very trivial example could be a function called $fn1$, which takes two parameters and calls an addition on them:

```
val fn1 = fun((a,b) => add(a,b)).
```

Now let us assume that we have another function called $fn2$, which takes two parameters and calls a subtraction on them:

```
val fn2 = fun((a,b) => sub(a,b)).
```

Assuming the target expression to generate is shown in Figure 4.3, then the first step is to determine what its dependencies are. In this case the *Add* operation is only dependent on *Input 1* and *Input 2*, while *Input 3* is only dependent on itself. As such we can already say that the output expression will be dependent on *Input 1*, *Input 2* and *Input 3* and so we can assume this to be the parameter list. So the generated expression has to call $fn1$ on the inputs *Input 1* and *Input 2*, and then it has to call $fn2$ on the output of the $fn1$ operation and *Input 3*. The final generated expression for the graph shown in Figure 4.3 is then:

```
val fn3 = fun((in1, in2, in3) => fn2(fn1(in1, in2), in3)).
```

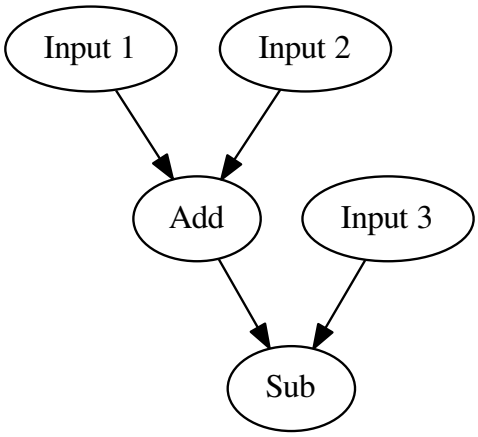


Figure 4.3: The simple example of combining $fn1$ (*Add*) and $fn2$ (*Sub*).

This example can be made more complicated by assuming that an input is used twice within a graph as shown in Figure 4.4. In this case, when generating the input list, one determines the input lists of each of the two subnodes, which in this case is [*Input 1*, *Input 2*] for the node on the right and [*Input 2*] for the node on the left. The correct input list can thus be created by removing all elements which are already in the left input list from the right input list, and to then combine the two. In this case *Input 2* is removed from the right input list and within the expression it is replaced by the *Input 2* param of the left list. Finally the two parameter lists are combined to produce [*Input 1*, *Input 2*]. Combining the two expressions produces the final expression:

```
val fn4 = fun(in1, in2) => fn1(fn1(in1, in2), in2).
```

This can be extended to more complicated graphs such as the one shown in Figure

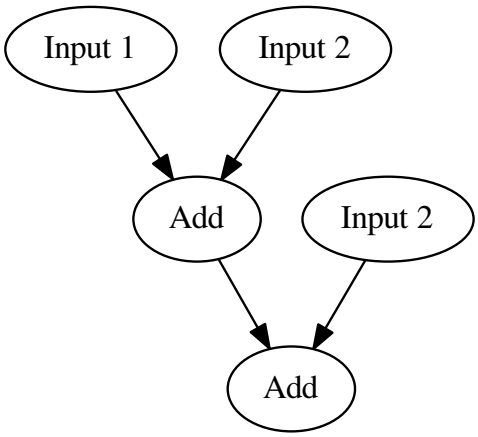


Figure 4.4: An extended example containing duplicate inputs of *Input 2*.

4.5. In this case the resulting expression is calculated recursively, starting from the lowest *Add* node. The expressions for each of its inputs are calculated. In this case the left *Add* node corresponds to $fn4$, while the right *Sub* node corresponds to $fn3$. Then as previously discussed the inputs occuring in the left input are removed from the right input list and replaced in the expression tree, leaving [*Input 3*] as the new inputs occuring in the right node. This leaves a combined input list of [*Input 1*, *Input 2*, *Input 3*]. The final code generated is shown in Listing 14.

The Lift compiler defines a class called **LiftNodeExpr**, which takes an expression body, an array of paremeters and an array of **LiftNode** names, that the inputs correspond to. The definition of **LiftNodeExpr** is shown in Listing 15. The function *combineArguments* is defined to produce a single expression making use of all parameters from both **LiftNodeExpr** inputs. This is done by concatenating all names of arguments from the second **LiftNodeExpr** to the names of arguments from the first **LiftNodeExpr** and then throwing away all duplicate values. The corresponding parameter is then attempted to be found in the first expression and if it cannot, it is taken from the second expression. Finally for each parameter the code tries to replace the old parameter in the expression tree with the new parameter.

Given the function *combineArguments* it is then possible to generate a **LiftNodeExpr** using code similar to the one shown in Listing 16. The first step to producing a combined node is to compile the two execution nodes corresponding to the inputs $input1$

```
val fn5 = fun((in1, in2, in3) => fn1(
        fn4(in1, in2),
        fn3(in1, in2, in3)
))

// which when combined produces
val fn5 = fun((in1, in2, in3) => add(
        add(
                add(in1, in2),
                in2
        ),
        sub(
                add(in1, in2),
                in3
        )
)
```

Listing 14: The combined code for the graph shown in Figure 4.5.

```
class LiftNodeExpr(e: Expr, p: Array[Param],
        n: Array[String]) {
  if (p.length != n.length) {
    throw new TensorFlowIncorrectInputCount(
      "LiftNodeExpr requires an identical number of " +
          "parameters as parameter names.")
  }

  val expr = e;
  val names = n;
  val params = p;

  def paramTuple = names.zip(params)
}
```

Listing 15: The **LiftNodeExpr** class which is used to represent a lambda expression.

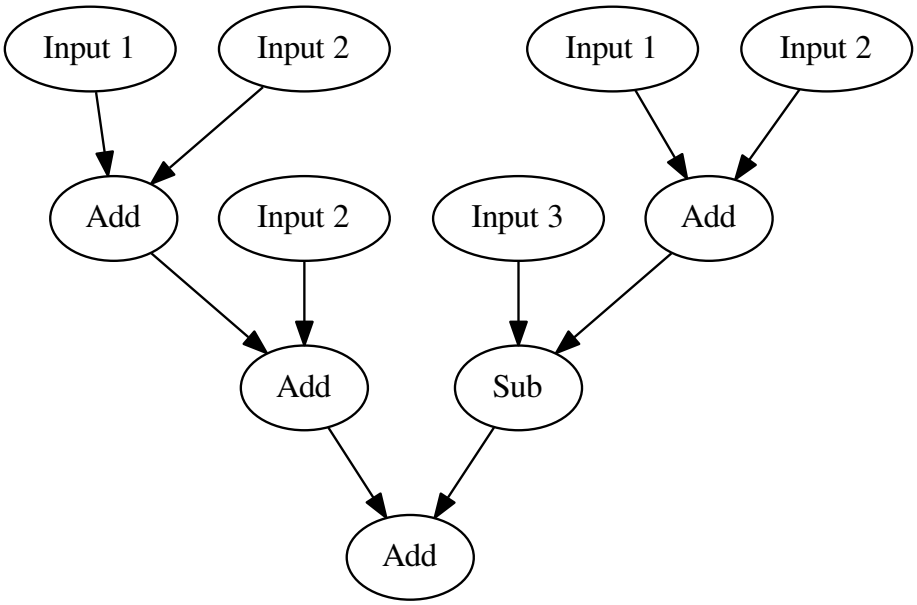


Figure 4.5: A combination of the graphs in Figure 4.3 and 4.4.

and $input2$ using the *compileLiftNode* function. This function calls the corresponding conversion function depending on the node type. Given the two compiled input nodes of type **LiftNodeExpr**, it is then necessary to generate a combined parameter list and to ensure that the second compiled node uses the same parameters as the first. This is done by applying the *combineArguments* function to both of the compiled **LiftNodeExpr** instances. Finally an expression for the specific node can be applied to the expressions of the first compiled input node and the combined second input node. The generated **LiftNodeExpr** uses the same parameter list as $in2appl$.

In the following section a method for inferring the type accross multiple nodes is explained.

4.1.4 Type Inference

The previous section examines how to combine different functional expressions for individual operations into a single functional expression. However, many operations either depend on the input shape or have a Lift implementation that depends on the

```
def convertSomeExpr(env: TensorFlowEnvironment,
      input1: ExecutionNode, input2: ExecutionNode) = {
  val in1 = compileLiftNode(input1, env)
  val in2 = compileLiftNode(input2, env)

  // combine the parameters from input 1 and input 2
  val in2appl = combineArguments(in1, in2)

  // some expression
  val expr = fun((A, B) => /* ... */)

  // generate a new lift node expression
  new LiftNodeExpr(expr(in1.expr, in2appl.expr),
      in2appl.params, in2appl.names)
}
```

Listing 16: Example code which could be used to compile to **ExecutionNode**'s *input*1 and *input*2 and to apply them to a functional expression in the variable *expr* in order to generate a new **LiftNodeExpr**.

input shape. In order to correctly generate this code or to determine the outcome of the operation, a type inferrence system is required. This system takes a **LiftNode** or a **LiftNodeInputBase** and calculates the type it will generate. The system determines the data type the output would have, given a set of user inputs and the current variable values as stored in the current session. An example of this is the *Shape*[4] operation, which returns the array dimensions of an input. The one problem that arises, is that some operations will have a type that is dependent on runtime values. An example of this is the *Reshape*[5] operation, which takes a shape as its second input and thus the data type is dependent on the value. As such, all type inference is performed on the fly in the context of a **TensorFlowEnvironment** instance, which defines the execution environment.

The following describes how the most important operations are implemented. There are still some further complicated operations such as *DynamicStitch* and *Broadcast-*

[4] https://www.tensorflow.org/versions/r0.9/api_docs/python/array_ops.html#shape

[5] https://www.tensorflow.org/versions/r0.9/api_docs/python/array_ops.html#reshape

GradientArgs, which have been implemented but are not described here. A reference of all operations can be found on the Tensorflow website[6].

REDUCTION NODES. Reduction nodes such as *Sum*, *Prod* and *Max* can be determined by knowing the input dimension and the reduction indices argument. If the input array is an n-dimensional array, the output type is an $n-k$-dimensional array, where k denotes the number of reduction indices. The dimensions along which the input is reduced are described by indices in the reduction indices argument. For example if the input has the shape $[3,7,9,2,4]$ and the reduction indices argument is $[2,4]$, then the output type will have the shape $[3,7,2]$. In order to implement it, the first step is to thus infer the type of the first input, and to use the execution environment to calculate the second. The inference can be done by converting the reduction indices into a boolean set and then recursively going over the input type, always taking the head value of the reduction set, in order to determine if the dimension is kept.

UNARY MATHEMATICAL OPERATIONS. Unary mathematical operations such as *Log* and *Neg* do not perform any changes to the input type. As such the output type can be determined by inferring the input type.

BINARY MATHEMATICAL OPERATIONS. Binary mathematical operations such as *Add* and *Mul* can support broadcasting. This means that it is possible to input two matrices of different input shapes, and to still calculate a result as if the lower dimensional input was tiled along the lower dimensions. For example the input $[2,4]$ can be added to the input $[[3,6],[4,9],[1,2]]$ to produce $[[5,10],[6,13],[3,6]]$. As such, in order to infer correct the output type, it is necessary to infer both of the input types. Of the two input types, the one with the higher rank (dimension count) is the correct one, so it is necessary to calculate and compare the ranks of the two.

CAST OPERATION. The *Cast* operation returns the data in the same shape as the input, but with another underlying data type. As such, in order to infer the output type it is necessary to infer the input type. Given the input type it is necessary to then reconstruct the type with a different underlying scalar type.

CONST OPERATION. The *Const* operation returns a constant with the specified value and dimensions. These are given in the tensorflow node declaration and can be extracted and converted.

[6] `https://www.tensorflow.org/versions/r0.10/api_docs/index.html`

FILL OPERATION. The *Fill* operation takes a shape and a scalar as runtime values and generates an array only consisting out of the scalar value. The output type can be inferred by knowing the runtime value of the input shape. Using the input shape it is possible to construct an array of the underlying type with the dimensions specified in the shape.

MATRIX MULTIPLICATION OPERATION The *MatMul* operation takes two matrices and performs a matrix multiplication on them. It also takes two boolean values, which determine wether or not the input matrices should be transposed. Given two matrices of dimensions $N1 \times M1$ and $N2 \times M2$, then either $N1$ or $M1$ is taken as the first dimension, depending on the first transpose value, and $N2$ or $M2$ taken as the second value, depending on the second transpose value.

SIZE OPERATION. The *Size* operation always returns a scalar integer containing the number of elements. As such it always returns an **Int**, regardless of the input type.

RESHAPE OPERATION. The *Reshape* operation takes an input and reshapes it to the shape specified by the second input. Normally it would have been sufficient to have the runtime value of the shape argument, but a special value of -1 can be passed in the shape argument. If a -1 is encountered, the dimension with that value is automatically determined, so that the shape matches accordingly. As such, it is also necessary to calculate the input size dimension, so that the missing dimension size can be inferred. This is done by calculating the size (number of elements) of the input type by multiplying out its dimensions, and then dividing this number by all other known dimensions. Then the type can be reconstructed with the extracted scalar type and new dimensions.

TILE OPERATION. The *Tile* operation takes an input and tiles it a number of times in each direction as specified by the second parameter. As such it is necessary to calculate the input type as well as the actual value of the second parameter. Then the output type can be constructed by multiplying each dimension's value with the corresponding factor in the second parameter.

In the following section the implementation of the *expr* is explained for the different node types, since this is the only thing missing in order to produce a working functional expression.

4.2 Node Implementation

Section 4.1 explains how to load and execute a Tensorflow graph given implementations for the individual node types. This section describes how some of the individual operations can be converted into functional expressions in the Lift programming language. A description of the operation can be found in the API[7].

4.2.1 Unary Mathematical Nodes

Unary mathematical operations such as *Log* and *Neg*[8] simply require us to apply an expression to a single scalar value. In order to apply a mathematical operation to each scalar value in an array it is necessary to loop over all elements which can be done using the *Map* operation. The *Map* operation takes a lambda function that defines the transformation of each value within the array. An example of performing a logarithm on a one-dimensional array is shown in Listing 17.

```
1   fun(A => Map(fun(cell => log(cell))) \$ A)
```

Listing 17: A one-dimensional *log* operation defined in Lift.

Given a two-dimensional array it is necessary to call *Map* over it twice, since the first *Map*'s function transforms one-dimensional arrays and not the scalar values. An example of a two-dimensional log operation is shown in Listing 18.

```
1   fun(A =>
2     Map(fun(row =>
3           Map(fun(cell =>
4             log(cell)
5           )) \$ row
6     )) \$ A
7   )
```

Listing 18: A two-dimensional *log* operation defined in Lift.

In order to convert an arbitrary dimensional function it is necessary to recursively unpack the input type, and to perform a *Map* for each input dimension. Once the scalar

[7] https://www.tensorflow.org/versions/r0.9/api_docs/index.html
[8] https://www.tensorflow.org/versions/r0.9/api_docs/python/math_ops.html#neg

value has been reached, the actual expression can be applied. This is defined as a recursive function, which applies the map to the resulting expression of the inner array type, as long as an ArrayType is encountered.

4.2.2 Binary Mathematical Nodes

Binary mathematical operations such as *Add* and *Mul* require an elementwise application of an expression between all values of two multidimensional inputs. In general it is possible to apply an operation to each pair of elements in two arrays by mapping over the zipped arrays. The function *noBroadcast* to perform such an operation is shown in Listing 19. *noBroadcast* takes a **Lambda** instance called *sub* which describes an operation to be applied to two elements from each of the arrays. In order to perform an elementwise multiplication of two-dimensional arrays, it is possible to generate the function as:

```
fun((A,B) => noBroadcast(noBroadcast(mul))(A,B)).
```

```
1 def noBroadcast(sub: Lambda) = {
2   fun((a, b) =>
3     Map(fun(c =>
4       sub(Get(c, 0), Get(c, 1))
5     )) $ Zip(a, b)
6   )
7 }
```

Listing 19: A function which applies the operation *sub* to the pairs of elements of two arrays.

Binary mathematical operations also support broadcasting. This means that it should be possible to multiply a one-dimensional array with a two-dimensional array, by duplicating the one-dimensional array along the lowest dimensions. In order to enable this, two further functions are defined. The first called *broadcastRight* applies the operation sub to every pair of the entire second input and each element in the first input. The second function called *broadcastLeft* does the same except for the fact that it takes the entire first input and applies it to each element of the second input. Given the additional broadcast functions it is possible to define an elementwise multiplication

between a one-dimensional array A and a two-dimensional array B as:
`fun((A, B) => broadcastLeft(noBroadcast(mul))(A,B))`.

```
def broadcastRight(sub: Lambda) = {
  fun((a, b) =>
    Map(fun(c =>
      sub(c, b)
    )) $ a)
}

def broadcastLeft(sub: Lambda) = {
  fun((a, b) =>
    Map(fun(c =>
      sub(a, c)
    )) $ b)
}
```

Listing 20: The function required to broadcast the second input along the first dimension of the first input (*broadcastRight*), as well as the complementary *broadcastLeft* function.

In order to determine the number of broadcasting operations that are necessary it is possible to calculate the difference of the rank[9] between the two matrices, $rankDiff := rank(t2) - rank(t1)$. If $rankDiff > 0$ (the second input has a higher rank), then it is necessary to first perform the *broadcastLeft* operation $rankDiff$ times. If $rankDiff < 0$ (the first input has a higher rank), then it is necessary to first perform the *broadcastRight* operation $-rankDiff$ times.

Furthermore it is possible for an input dimension in the one array to be of length 1. This requires that dimension be broadcast along the other array. In order to perform an elementwise multiplication between array of shape $[3,1]$ and $[3,20]$ one could generate the correct kernel using the code shown in Listing 21. The basic idea is to map over the one-dimensional kernel and to left broadcast that result onto the second array. Finally the result has to be joined, since otherwise an additional dimension will be introduced due to the two nested map expressions, whereby the one dimension would be one-dimensional.

[9]The rank operation in this context is defined as the number of dimension of an input, which is different to the mathematical definition of a rank of a matrix.

```
noBroadcast(fun((A,B) =>
  Join() o Map(fun(C =>
    broadcastLeft(mul)(C, B)
  )) $ A
))
```

Listing 21: The functional expression required to broadcast along a one-dimensional array.

These constructs are then combined to produce a recursive function which takes a binary expression, two input types and a $rankDiff$ in order to produce the correct functional expression. If the function encounters a scalar type in either of the types it performs a broadcast of the scalar argument. If the function encounters a scalar in both of the types it returns the underlying binary expression. When two array types are passed to the function, it first checks the rank difference $rankDiff$ and performs a broadcasting operation if it is nonzero. Should the rank difference $rankDiff$ be zero and one of the input dimensions one-dimensional, then it broadcasts that dimension as well. Otherwise it simply uses the *noBroadcast* function.

4.2.3 MatMul Node

The MatMul node takes two two-dimensional matrices and performs a matrix multiplication between the two matrices. It can also transpose the matrices before multiplication using two additional boolean values supplied to the node. Listing 22 shows the functional representation of how to perform a matrix multiplication, under the assumption that B is already transposed. Thus in order to perform a matrix multiplication, if the node requires the first input to be transposed, then the first input has to be transposed. If the node requires the second node to not be transposed, then it is necessary to transpose the second node.

4.2.4 Reduction Nodes

Reduction nodes such as *Sum* take an n-dimensional input array and then perform the specified reduction across the dimensions described by the second input, called the reduction indices. Performing a reduction among all dimensions of an input is fairly triv-

```
val expr = fun(
  (A, B) => {
        Map(fun(aRow =>
          Join() o Map(fun(bCol =>
                Reduce(add, 0.0f) o Map(fun(x =>
                  mult(Get(x, 0), Get(x, 1))
            )) $ Zip(aRow, bCol)
          )) $ B
        )) $ A
  })
```

Listing 22: The *MatMul* function, which assumes that B is already transposed.

ial, since the pattern used would be to nest `fun(A => Map(Reduce(expr)(init)) $ A)` for each dimension (the final dimension would not need a *Map* and due to issues outlined in section 4.3.1, additional *Join* statements would be necessary). Assuming a two dimensional array is passed as the input, along with $[0]$ as the reduction index, performing this reduction becomes slightly more complicated. There are two options for performing this reduction. The first is to perform a reduction with an array as the accumulator, which is shown in Listing 23. This option is viable, but is slightly inefficient at the moment, because reductions are harder to parallelise.

```
fun(A =>
  Reduce(fun((Arow, accum) =>
    Map(fun(pair =>
      sum(Get(pair, 0), Get(pair, 1))
    )) $ Zip(Arow, accum)
  )) $ A
)
```

Listing 23: One method of performing a reduction on the first dimension of a two-dimensional array.

Instead, it is better to *Transpose* the input, since this means a rewrite of the order internally. The *Transpose* operation also works on higher order inputs, only swapping the first two dimensions. So given the same two-dimensional matrix example, the better method is to perform a *Transpose* first, and to then *Map* over each row and perform

a *Reduce* over it. This is shown in Listing 24. In order to swap higher dimensions it is necessary to map over the input first. For example if one would like to reduce the second dimension of a three-dimensional input, apply `Map(Transpose)` to the input first. After that it is possible to perform a reduction on the final dimension.

```
1 fun(A =>
2   Map(fun(Acol =>
3     Reduce(add)(0) $ Acol
4   ) o Transpose() $ A
5 )
```

Listing 24: The better method of performing a reduction on the first dimension of a two-dimensional array.

It is thus necessary to find a method in order to correctly transpose an arbitrary input, so that all the dimensions to be reduced are remapped to be the highest dimensions. For this a boolean array of the reduction indices is generated, which has a boolean value for each input dimension, indicating wether or not it should be reduced (`true`) or not (`false`). So, given a four-dimensional array where the first and third dimensions should be reduced, it would be `Array(true, false, true, false)`. The method of correctly rearranging the input dimensions is to essentially perform a bubble sort, starting with the highest dimensions. This way, each time the combination `(true, false)` is encountered, the two dimensions are be swapped. This sorting is repeated until no more swaps have been made. Each swap operation updates the accumulator with a transpose:
`acc = fun(A => Map(acc) o Transpose() $ A)`.

Every other operation simply updates the accumulator with a map operation:
`acc = fun(A => Map(acc) $ A)`.

These steps are then recursively repeated, until no more transforms have been applied. The resulting transformation steps are just composed behind each other. For the given example, the sorting steps would be as follows:

1. `Array(true, false, true, false)`
2. `Array(false, true, false, true)`
   ⇒ `Map(Map(Transform())) o Transform()`

3. `Array(false, false, true, true)`
   ⇒ `Map(Transform())`
4. `Array(false, false, true, true)`

And thus the final resulting transformation code would be:
`Map(Transform()) ∘ Map(Map(Transform())) ∘ Transform()`.

Once the array has been retransformed so only the highest dimensions contain the dimensions to be reduced, it is possible to perform the actual reduction. For this the number of input dimensions is determined as *arrayDepth*, along with the number of dimensions to reduce *reductionDepth*. Then a *Map* operation is applied to the input $arrayDepth - reductionDepth$ times, in order to iterate over an array that has to be completely reduced. Using the array that has to be reduced the *Reduce* operation is applied to the output of a *Map* operation, which recursively reduces the inner dimension. As soon as the scalar value is encountered, the scalar value is simply returned. Due to the bug described in Section 4.3.1, some special cases are added so that the additional Array layer is removed by using a *Join* operation.

4.2.5 Reshape Node

The reshape node takes an input array and changes it into another array of the same size but different dimensions, according to the second input. It also supports a special value of -1 for one of the target shape dimensions, which indicates that the value should automatically be determined in order to correctly build an array of the same size. The approach taken here in order to reshape the array is to first flatten it and to then split it according to the second input.

The flattening task is quite trivial for an n-dimensional array. The array can be flattened using $n-1$ *Join* operations in order to concatenate all the data. So flattening a 3 dimensional array can be done using the following function:
`fun(A => Join() ∘ Join() $ A)`.

There is still the special case that it is possible to reshape a scalar value. Should a scalar value be encountered, then a method is required to pack it into a one dimensional array. This can be done by mapping over a one dimensional constant, with a function that returns the actual scalar value to be used. An example of this and how to fix it is shown in Listing 25.

```
// encountered problem
val original = \(ArrayType(Float, 1), A =>
  logIt(A)
)

// how to fix
val fixed = \(ArrayType(Float, 1), A =>
  Map(\(scalar => logIt(scalar))) $ A
)
```

Listing 25: An example of a function call expecting a scalar value, but receiving a one-dimensional array and how to fix it.

Before the flattened array can be reshaped into the correct dimensions, it is necessary to determine the actual dimensions. This is done in order to replace the dimension with a value of -1 with an actual number that reflects the correct target dimension, so that the sizes of the input and output match. This value can be calculated by taking the size of the input array (the total number of elements) and dividing it by the product of all other dimensions.

Next, code is needed in order to take a flattened array and to convert it into an array of a specific shape. The Lift framework offers a *Split* operation, which takes an input and splits it into groups of the size given as a parameter. Importantly, it is necessary to perform the *Split* operations in the correct order, starting from the highest dimension first. Furthermore, the outermost dimension has to be ignored, since this is already automatically implied. Consider the example of a flattened array of size 24 and a target shape of [2, 3, 4]. The first step is to split it by 4, which would produce an array of shape [6, 4]. Then the array is split by 3, producing the target array of [2, 3, 4]. As such the function required would be:
`fun(A => Split(3) o Split(4) o A.`

4.3 Implementation Workarounds

In its current state the Lift framework still has some issues which cause problems with code produced as described in this chapter. The first problem is that Lift doesn't

correctly support unpacking scalar types, and some operations such as *Reduce* return a single element array instead of a scalar value. A workaround for this is described in Section 4.3.1. Furthermore, the Lift language does not support zero length memory types. The workaround for the zero length memory problems is described in Section 4.3.2.

4.3.1 Incorrectly supported scalar Types

The Lift programming language currently has the problem that it is not possible to unpack an array that only contains one value. An example of such a function is shown in Listing 26. This is especially problematic since Reduce functions always only reduce down to a single element array, instead of returning a scalar value. The only solution to this problem is to *Map* over the result and perform the *add* operation within the *Map* lambda. This however, again returns a single element array. An example of this is shown in Listing 27.

```
1 fun(ArrayType(Float, 5), A =>
2         add(5, ReduceSeq(add, 0.0f) $ A)
3 )
```

Listing 26: Code that cannot be compiled in Lift, because there is no way to convert the result of *ReduceSeq* into a scalar as required for the *add* user function.

```
1 fun(ArrayType(Float, 5), A =>
2         Map(fun(x =>
3                 add(5, x)
4         )) o ReduceSeq(add, 0.0f) $ A
5 )
```

Listing 27: A workaround for the problem shown in Listing 26.

Furthermore, there is the special case where such a single element type is wrapped within a Map function, causing the creation of a two-dimensional array instead of a one-dimensional array. The solution to this is to perform a *Join* after the outer *Map*. This is shown in Listing 28

```
fun(ArrayType(ArrayType(Float, 5), 5), A => {
        Join() o Map(fun(Arow =>
                Map(fun(x =>
                        add(5, x)
                )) o ReduceSeq(add, 0.0f) $ Arow
        )) $ A
)
```

Listing 28: A *Join* operation is required in order to produce the expected output.

In order to perform this workaround, code which traverses the function tree and tries to find functions that require a scalar value but are given a single element array was written. This function also needs to perform type inferrence going upwards, since it is necessary to be able to see if the type is a single element array where a scalar value is expected. These functions are then wrapped in a *Map* function, and then the single dimensional scalar value is fed in as input. The function to perform this workaround works directly on the expressions, and the main function is called *traverse* and takes an **Expr** object. An **Expr** object can either be a **Param** or a **FunCall**. If it is a **Param** it should be returned as is with the type attached to the parameter.

If the **FunCall** is a *Map* function call, the first requirement is to call *traverse* on the argument in order to both infer its type, and to fix any potential calls within it. Once the type is known, it is possible to set the type of the parameter for the inner lambda function. Using this type, it is then possible to traverse the body of the inner lambda of the *Map* operation. Using this corrected lambda expression, the *Map* operation is recreated. If the traversal of the inner lambda signals that the output type of itself is a single element array instead of being a scalar, a *Join* operation is prepended to the output first in order to fix this.

If the **FunCall** is a *UserFun* call, it is necessary to first traverse all argument expressions and to determine which arguments are arrays that need to be swapped out. For this, the *traverse* function is called on each argument expression and the type is simultaniously determined. If the *UserFun* expects a **ScalarType** and the expression argument is a single element array, then the expression is replaced with a new **Param**. Furthermore, if it is replaced with a new **Param**, then an entry in a map is created pointing the new **Param** to the **Expr** for the fixed expression. Then for each entry in the map, a new *Map* function call is created, which uses the given parameter and takes

the **Expr** that it maps to as its argument.

Finally, if multiple parameters have been replaced, this causes the element to be converted into a multi-dimensional single-element array and it is possible to get rid of one of the additional dimensions. This is done by calling *Join* $n-1$ times, if n parameters have been replaced. If any expressions have been replaced, then it is signaled to the next function, that a *Join* should be used after any subsequent calls to *Map*.

If a *Reduce* is encountered, the arguments are traversed in order to fix any calls within them and to infer their types. Then the types of the **Param** of the lambda function are set using this information. The first argument type of the lambda expression is the accumulator and has the same type as the initial value, which is the first argument passed to **Reduce**. The second argument type of the lambda expression is equivalent to an element of the array to reduce over, which is the inner type of the second argument passed to **Reduce**. Once the types have been set it is possible to call *traverse* on the body of the lambda expression passed to *Reduce*.

Any other **FunCall** requires the arguments to be traversed, and a new **FunCall** to be generated with the traversed argument.

4.3.2 Unsupported zero length Memory Types

The other problem that can occur is the fact that Lift currently does not support the allocation of zero length memory types. This can occur when the function has an argument of length zero, or when a reduction should actually return its initial value. The solution to this problem is to find all reductions, which have a zero length input and to replace them with the initial value. Furthermore, all parameters which are of zero length should be removed from the function. Instead of writing a completely new tree traversal, an addition is added to the *traverseReduce* function code, which is described in section 4.3.1. The *Map* function helps to convert a functional expression into a single element array in order to comply with Lift.

```
1 fun(ArrayType(Float,0), A =>
2         Reduce(add, 0.0f) $ A
3 )
```

Listing 29: A code example that would require the allocation of 0 bytes.

Chapter 5

Evaluation

The goal of this project is to improve performance of existing machine learning applications written in Google Tensorflow. This is done by taking the Tensorflow data flow graph and converting sections of it into executable Lift code, which can then be compiled into OpenCL code. The previous chapter explains how to convert subgraphs of a Google Tensorflow graph into the functional programming Language Lift and how to execute them. The main performance metric that is of interest, is the execution time of graphs and how they compare to the original Google Tensorflow application. Section 5.1 explains how the work presented here is evaluated. Section 5.2 contains some actual results that were measured using these techniques.

5.1 Methodology

This section contains an explanation of what methods were used in order to test the work presented here. The first Subsection 5.1.1 explains the problem which has to be solved. The second Subsection 5.1.2 explains how the subgraph to test was chosen, and how it was prepared for evaluation. Finally the performance benchmarking techniques are explained in Subsection 5.1.3.

5.1.1 The Test Problem

Evaluation of the work presented here is done using a practical example. In this case the simple MNIST handwriting recognition example was taken from the Google Ten-

sorflow website[1]. It offers a single layer neural nework with a softmax activation layer. The neural network is shown in Figure 5.1. The mathematical definition of the network is shown in Equation 5.1.

$$y = \text{softmax}(Wx + b) \tag{5.1}$$

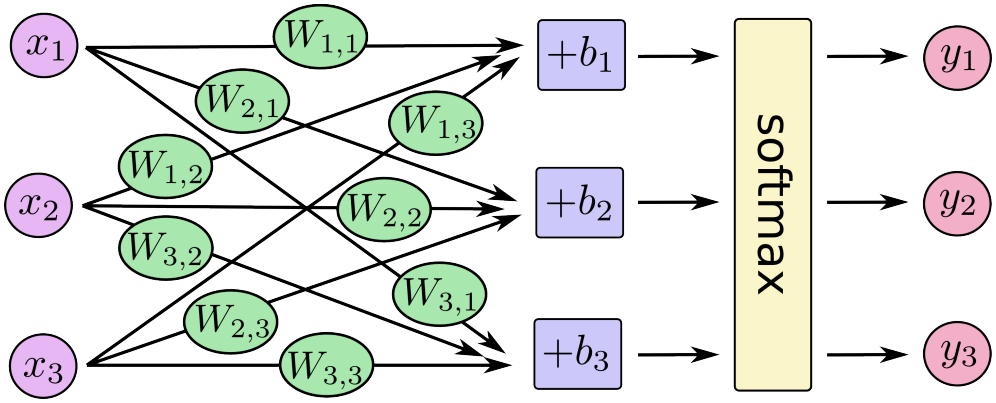


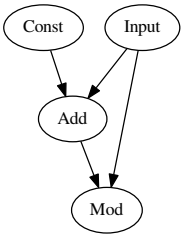
Figure 5.1: The simple neural network, which is used as an example. The image is taken from the Google Tensorflow website.

5.1.2 Preparing a Subgraph

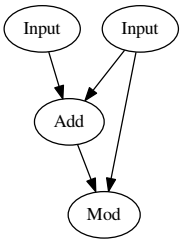
The first 6 subgraphs can be converted into Lift and run without any problems. These subgraphs are shown in Figure 5.2. Unfortunately, most of these subgraphs are either very small (e.g Figure 5.2a and Figure 5.2b) or they do not use the large rows of data such as Figure 5.2e, making them unsuitable for actual performance tests. As such a graph was needed, which contains a suitable collection of nodes that work on the bulk data that needs to be processed.

Even with the bug fixes presented in Section 4.3, a further bug prevents the entire graph from running. This has to do with the way that Lift handles private memory. If an array is encountered in private memory, then Lift tries to completely flatten this array into individual scalar values in order to ensure fast performance on graphics cards, since scalar values can be translated into a register. This requires the array to not only have a static length, but also for the scalar value which will be accessed to be statically

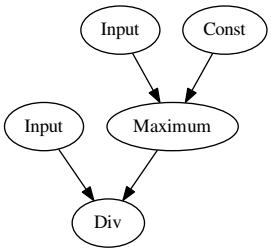
[1] https://www.tensorflow.org/versions/r0.10/tutorials/mnist/beginners/index.html



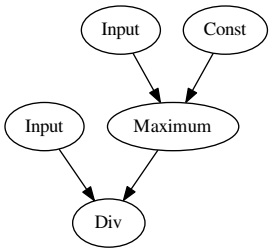
(a) Graph *gradients/Sum_grad/mod*



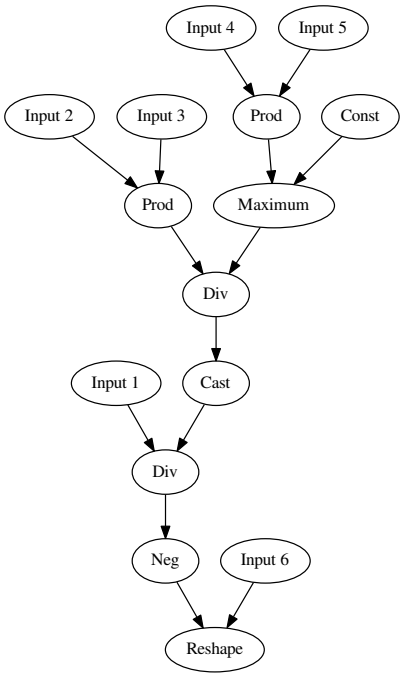
(b) Graph *gradients/Mean_grad/mod*



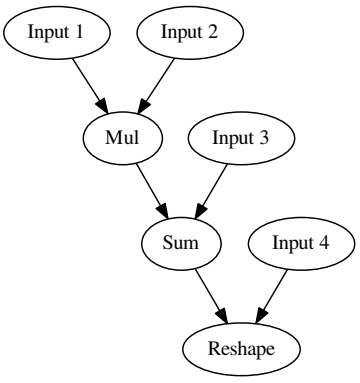
(c) Graph *gradients/Mean_grad/floordiv*



(d) Graph *gradients/Sum_grad/floordiv*



(e) Graph *gradients/Sum_grad/Reshape*



(f) Graph *gradients/mul_grad/Reshape_1*

Figure 5.2: The subgraphs which can be converted into Lift and executed without any issues.

```
import tensorflow as tf

x = tf.placeholder(tf.float32, [None, 784])

W = tf.Variable(tf.zeros([784, 10]))
b = tf.Variable(tf.zeros([10]))

y = tf.nn.softmax(tf.matmul(x, W) + b)

y_ = tf.placeholder(tf.float32, [None, 10])

cross_entropy = tf.reduce_mean(-tf.reduce_sum(y_ * tf.log(y),
  reduction_indices=[1]))
train_step = tf.train.GradientDescentOptimizer(0.5).minimize(
  cross_entropy)

init = tf.initialize_all_variables()

sess = tf.Session()
tf.train.write_graph(sess.graph_def, 'models/', 'simple_mnist.pb',
  as_text=False)
```

Listing 30: The code used to generate the data flow graph used to perform analysis.

known. The Tensorflow code presented here produces graphs which requires slight modifications before they can be run.

A suitable subgraph for evaluation is the graph *gradients/mul_grad/Reshape*. This subgraph contains lots of different nodes including nodes that are typical for machine learning applications such as *Softmax*. The subgraph is shown in Figure 5.3.

The *gradients/mul_grad/Reshape* graph has some issues during compilation however. The code presented here is able to generate the functional expression, which is shown in Listing 31. It is then necessary to apply **Lower**.*mapCombinations* to the function which produces the code shown in Listing 32. This code fails during compilation due to the array unflattening issue as described above. The solution to this is to introduce *toGlobal* operations which cause the result to be written to global memory after calcu-

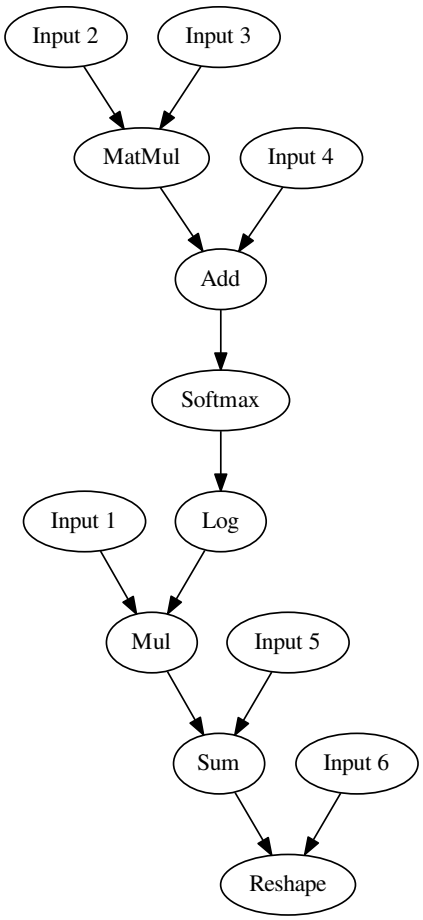


Figure 5.3: The representative subgraph used to evaluate performance.

lating the intermediate result, which prevents the use of private memory. Additionally some of the *MapSeq* calls were converted into *MapGlb* calls, in order to make use of the additionally available paralellism.

5.1.3 Benchmarking

The code in Listing 33 was then used in order to run benchmarks. The Lift framework includes a function in order to perform benchmarking. It works by executing the function a number of times, while timing the kernel execution time on each run. From the different execution times, the median of the kernel execution times is taken as the execution time. The functional expression is benchmarked against a number of different input row sizes. The number of rows is defined by the *num* variable.

```
fun((p117,p132,p78,p37) => Split(10) o Join() o Map(\(p27 =>
    Map(\(p93 =>
      p93
    ))(p27)
  )) o Map(\(p107 =>
    Map(\(p13 => mult(Get(0)(p13), Get(1)(p13)))
    ) $ Zip(2)(Get(0)(p107), Get(1)(p107))
  )) $ Zip(2)(p117, Map(\(p158 =>
    Map(\(p225 =>
      logIt(p225)
    ))(p158)
  )) o Map(\(p103 =>
    Join() $ Map(\(p248 =>
      Map(\(p114 =>
        divide(p248, p114)
      )) $ Reduce(\((p249, p0) => add(p249, p0)),0.0f)(p103)
    ))(p103)
  )) o Map(\(p234 =>
    Map(\(p39 =>
      expIt(p39)
    ))(p234)
  )) o Map(\(p7 =>
    Map(\(p67 =>
      add(Get(0)(p67), Get(1)(p67))
    )) $ Zip(2)(p7, p37)
  )) $ Map(\(p112 =>
    Join() o Map(\(p134 =>
      Reduce(\((p92, p173) => add(p92, p173)), 0.0f)
             o Map(\(p208 =>
        mult(Get(0)(p208), Get(1)(p208))
      )) $ Zip(2)(p112, p134)
    )) $ Transpose()(p78)
  ))(p132))
)
```

Listing 31: The functional expression produced for the graph shown in Figure 5.3 including the workarounds described in the previous section.

```
1  fun((p208, p91, p244, p186) =>
2    MapGlb(0,\(p251 =>
3      toGlobal(\(p68 =>
4        MapSeq(\(p250 =>
5          mult(Get(0)(p250), Get(1)(p250))
6        ))(p68)
7      )) $ Zip(2)(Get(0)(p251), \(p175 =>
8        MapSeq(\(p254 =>
9          logIt(p254)
10       )) o Join() o MapSeq(\(p179 =>
11         MapSeq(\(p242 =>
12           divide(expIt $ add(Get(0)(p179), Get(1)(p179)), p242)
13         )) $ ReduceSeq(\((p22, p105) =>
14           add(p22, expIt $ add(Get(0)(p105), Get(1)(p105)))
15         ))(idfloat(0.0f), Zip(2)(Join() o MapSeq(\(p154 =>
16           ReduceSeq(\((p145, p116) =>
17             add(p145, mult(Get(0)(p116), Get(1)(p116)))
18           ))(idfloat(0.0f), Zip(2)(p175, p154))
19         )) $ Transpose()(p244), p186))
20       )) $ Zip(2)(Join() o MapSeq(\(p154 =>
21         ReduceSeq(\((p145, p116) =>
22           add(p145, mult(Get(0)(p116), Get(1)(p116)))
23         ))(idfloat(0.0f), Zip(2)(p175, p154))
24       )) $ Transpose()(p244), p186)
25     ) $ Get(1)(p251))
26   )) $ Zip(2)(p208, p91)
```

Listing 32: The output of calling **Lower**.*mapCombinations* on the functional expression in Listing 31.

```
fun(
      ArrayType(ArrayType(Float, 10), num),
      ArrayType(ArrayType(Float, 784), num),
      ArrayType(ArrayType(Float, 10), 784),
      ArrayType(Float, 10),
      (p208, p91, p244, p186) =>
      MapGlb(0,\(p251 =>
        toGlobal(\(p68 =>
          MapSeq(\(p250 =>
            mult(Get(0)(p250), Get(1)(p250))
          ))(p68)
        )) $ Zip(2)(Get(0)(p251), \(p175 =>
          MapSeq(\(p254 =>
            logIt(p254)
          )) o Join() o MapGlb(1,\(p179 =>
            toGlobal(MapSeq(\(p242 =>
              divide(expIt $ add(Get(0)(p179), Get(1)(p179)), p242)
            ))) $ ReduceSeq(\((p22, p105) =>
              add(p22, expIt $ add(Get(0)(p105), Get(1)(p105)))
            ))(idfloat(0.0f), Zip(2)(Join() o MapSeq(\(p154 =>
              toGlobal(MapSeq(\(A => idfloat(A)))) $
              ReduceSeq(\((p145, p116) =>
                add(p145, mult(Get(0)(p116), Get(1)(p116)))
              ))(idfloat(0.0f), Zip(2)(p175, p154))
            )) $ Transpose()(p244), p186))
          )) $ Zip(2)(Join() o MapGlb(1,\(p154 =>
            toGlobal(MapSeq(\(A => idfloat(A)))
            ) $ ReduceSeq(\((p145, p116) =>
              add(p145, mult(Get(0)(p116), Get(1)(p116)))
            ))(idfloat(0.0f), Zip(2)(p175, p154))
          )) $ Transpose()(p244), p186)
        ) $ Get(1)(p251))
      )) $ Zip(2)(p208, p91)
    )
```

Listing 33: The manually corrected functional expression of the functional expression shown in Listing 32.

In order to compare the kernel to the original Tensorflow with NVIDIA cuBLAS library, it was necessary to recreate the graph using just the elements in the subgraph. This subgraph executes on variables representing the intermediate results. It is important to execute the functions on variables and not placeholders, as placeholders are loaded from the main memory and have to be copied to the device. Variables on the other hand are stored on the graphics card, so the performance determined is much closer to the actual performance of Google Tensorflow. For timing the execution time a call to **time**.*time()* is used. The data required for the python function is randomly initiated, so long as it does not affect the amount of data to be processed. Both calculations are run on the same machine, making use of an NVIDIA Quadro K21 GPU.

Per row it can be said that the amount of data in Bytes processed for n number of rows is given by Equation 5.3. This is because each input row contains 784 cells, and the amount of input data that is processed is equivalent to the combined amount of data the input is comprised of. So in the case of 32768 rows $\approx 99.31MB$ of input data is processed and for 65536 rows of data it is $\approx 198.56MB$.

$$D(n) = (n*10 + n*784 + 784*10 + 10 + 10*784)*4[Byte] \tag{5.2}$$

$$D(n) = 3176*n + 62720[Byte] \tag{5.3}$$

After performing some initial benchmarks, an attempt was made in order to increase the execution performance of the data flow graph. This includes splitting the original subgraph shown in Listing 31 into two subgraphs. The first containing only the matrix multiplication and the second containing the other remaining operations. Then the matrix multiplication subgraph is replaced with an optimized matrix multiplication function taken from the Lift framework in order to try to improve performance.

Finally the execution performance of both Lift and cuBLAS using the NVIDIA K21 are compared to an AMD R280 (Radeon 7970) graphics card.

In the next section the results for this setup are shown and examined.

```
def benchmark(nrows):
  with tf.Session() as sess:
    ncols = 784
    in1 = tf.Variable(tf.random_normal([nrows,10]), name='in1')
    in2 = tf.Variable(tf.random_normal([nrows,ncols]), name='in2')
    in3 = tf.Variable(tf.random_normal([ncols,10]), name='in3')
    in4 = tf.Variable(tf.zeros([10]), name='in4')
    in5 = tf.Variable([], name='in5', dtype=tf.int32)
    in6 = tf.Variable([nrows,10], name='in6')

    matmul = tf.matmul(in2, in3)
    add = tf.add(matmul, in4)
    softmax = tf.nn.softmax(add)
    log = tf.log(softmax)
    mul = tf.mul(in1, log)
    sm = tf.reduce_sum(mul, in5)
    rshape = tf.reshape(sm, in6)

    init_op = tf.initialize_all_variables()
    sess.run(init_op)

    for i in range(0,50):
      start = time.time()
      sess.run(rshape)
      performance[i] = (time.time() - start) * 1000

    return np.median(performance[2:])
```

Listing 34: The function used to benchmark the subgraph shown in Figure 5.1 for a given number of rows.

5.2 Results

This section contains the performance results from the experiments described in the previous chapter.

Running the described setup with variable amount of input rows produces the graph shown in Figure 5.4. As can be seen the Tensorflow implementation making use of cuBLAS still outperforms the code presented here by a large margin. It is also important to note that the Tensorflow tests contain a little more overhead than the Lift tests, since the Lift test only determines the kernel execution time performance. In order to have a slightly better understanding of the overhead, the NVIDIA profiler was used to measure kernel execution times on the routine for 6400 rows. The result of this is shown in Table 5.1. Summing up all the routines except for the init routines and the memory copy routines, the average time per call adds up to 552.871us, in comparison to the roughly 1.491ms required for the complete call. However even though the total kernel execution time is lower, one has to consider, that not all of the overhead can be ignored. Some of the overhead is due to the multiple callings of the many different kernels, and as such is an inherent problem that the Lift framework can circumvent.

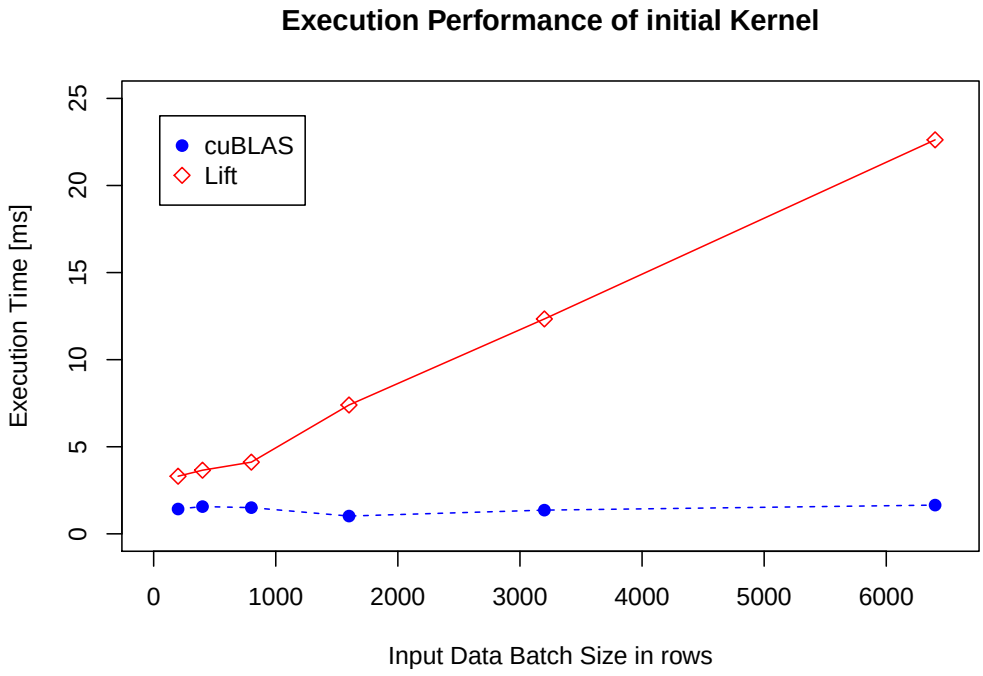


Figure 5.4: A comparison of Google Tensorflow performance using CUDA with the Lift programming language.

Time(%)	Time	Calls	Avg	Min	Max	Notes
73.56%	22.854ms	50	457.08us	453.32us	502.41us	matr. mult.
6.94%	2.1558ms	50	43.115us	40.097us	55.328us	memcopy
4.25%	1.3219ms	50	26.438us	26.112us	27.584us	
4.09%	1.2694ms	50	25.388us	25.024us	27.968us	
2.81%	874.25us	50	17.485us	16.992us	17.856us	
2.12%	657.45us	3	219.15us	13.793us	620.81us	init routine
1.16%	361.38us	50	7.2270us	6.8480us	13.024us	
1.08%	334.09us	50	6.6810us	6.4640us	6.8480us	
1.02%	315.81us	50	6.3160us	5.8240us	6.6880us	
1.01%	312.71us	50	6.2540us	6.0800us	7.9360us	
0.95%	296.58us	3	98.860us	4.0000us	287.46us	init routine
0.95%	296.20us	3	98.732us	3.8080us	287.27us	init routine
0.03%	10.305us	4	2.5760us	1.6960us	3.8090us	init routine
0.02%	5.2800us	1	5.2800us	5.2800us	5.2800us	init routine
0.01%	3.5520us	4	888ns	864ns	896ns	init routine

Table 5.1: The NVIDIA profiler applied to the test Google Tensorflow graph with an input size of 6400 rows.

Looking at Table 5.1, it can be seen that a lot of the computation time is due to the matrix multiplication operation. And furthermore the generated code shows that the matrix multiplication is performed twice due to the implementation of **Lower**.*mapCombinations*. Matrix multiplications have been largely studied by Dubach et. al and it has been shown, that it is possible to achieve much higher performance on the given platform [17].

An attempt was made to split the generated Lift expression into two parts, separating the matrix multiplication from the other operations. Benchmarking these subkernels shows that the execution performance can be increased greatly by splitting these kernels. The curve titled ”Lift Separated” in Figure 5.5 shows the execution time curve is much flatter than the original curve’s calculation titled ”Lift”. In fact for a large portion of the graph it outperforms the Tensorflow cuBLAS implementation. The small peak in the cuBLAS implementation for small dataset sizes is due to the fact that the GPU probably isn’t stressed out enough in order to run at its full clock speed. The separated Lift kernels make the execution on much larger datasets much more feasible than the

the kernel in Listing 33.

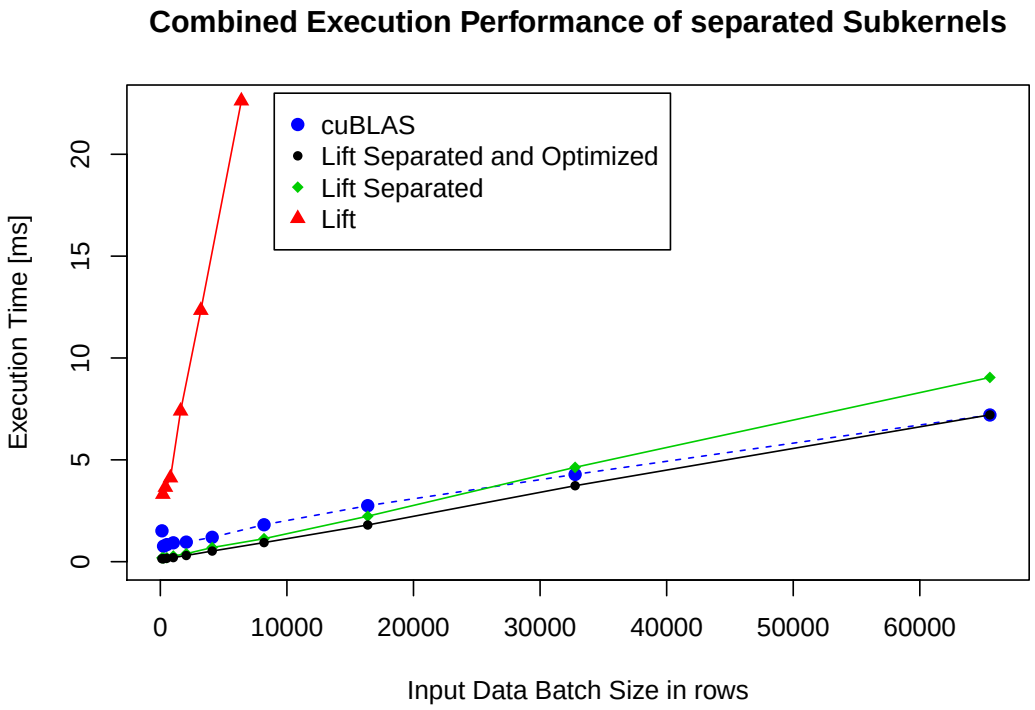


Figure 5.5: Performing the same calculation in two different kernels.

The next step was to apply some rewrite rules to the kernels in order to try and increase the execution performance by altering the way they are calculated. This was done by hand in order to demonstrate that some performance improvements are possible, but does not demonstrate the full optimization potential. Since the kernels are seperated, optimization of both kernels would be required in order to achieve best performance.

Performing some optimizations to the other kernel by hand yielded little results, since it is already quite efficient. Figure 5.6 compares the execution time of the one kernel to its Tensorflow (”cuBLAS”) equivalent. It can be seen that the kernel is able to outperform the Tensorflow cuBLAS implementation for all data batch sizes. At a batch size of 65536 rows the Lift implementation needed only 1.36*ms* in comparison to the 2.69*ms* required by Tensorflow. This means that by combining these operations it was possible to reduce much of the overhead required for calling all the kernels.

It was still possible to replace the matrix multiplication kernel with one that performs tiling. This allows for an even better total execution performance, as shown in Figure 5.5. The line titled ”Lift Separated and Optimized” shows the execution performance

for the other matrix multiplication operation combined with the kernel. With the other matrix multiplication operation, the kernel outperforms the cuBLAS implementation for all batch sizes except 65536. The original Tensorflow implementation took 7.20ms for 65536 rows, while the optimized version took 7.21ms, which is 1.83ms less than the unoptimized version. At lower batch sizes such as 8192 rows, the optimized version needs only 0.94ms compared to the 1.81ms required by tensorflow, making it nearly double as fast. The only issue is that the new matrix multiplication kernel performs a matrix multiplication which requires the second matrix to have an input shape of $784 \times \mathbf{16}$ instead of $784 \times \mathbf{10}$ and the additional overhead for padding and removing the additional dimensions has not been accounted for.

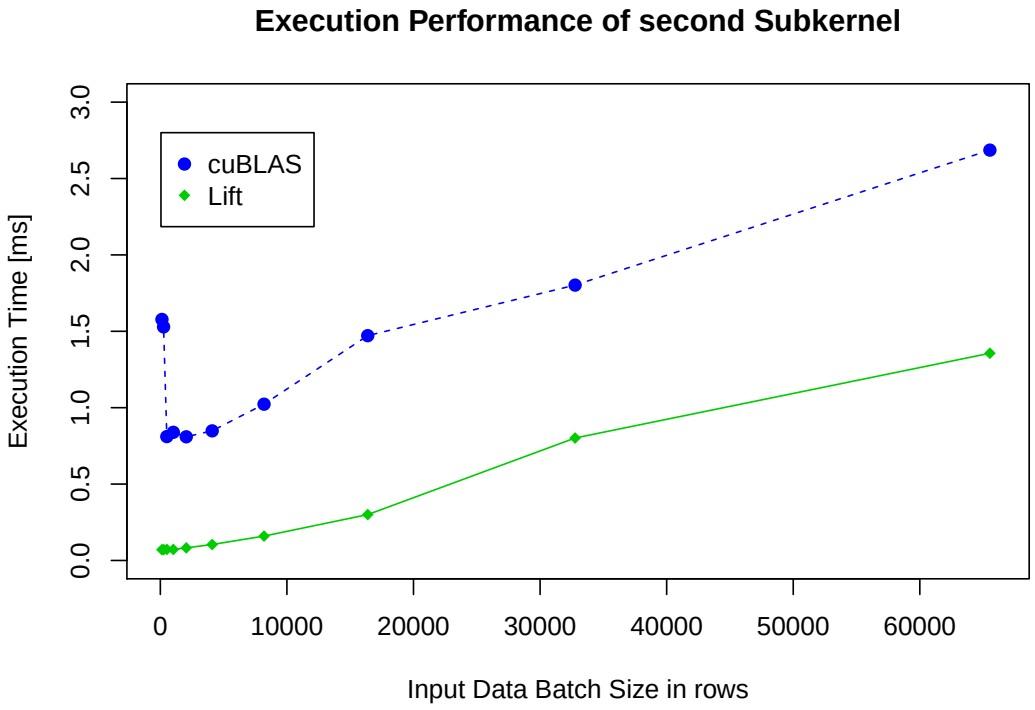


Figure 5.6: Performance optimizations using rewrite rules on one of the kernels.

To show that the software is portable it was run on an AMD R280 (Radeon 7970). Figure 5.7 shows how the performance compares to the NVIDIA K21 graphics card. Since cuBLAS only runs on NVIDIA graphics cards it is not possible to perform a direct comparison between the original Tensorflow implementation and the one implemented here. The AMD graphics card is able to outperform both the original cuBLAS implementation as well as the implementation presented here by nearly a factor of two, showing that the code presented here is portable accross different devices. At 65536

rows, the R280 only needed 2.82*ms* in comparison to 7.21*ms* for the K21 running the Lift implementation and 7.20*ms* for the cuBLAS implementation.

Figure 5.7: Comparison of execution performance on different platforms.

Chapter 6

Conclusion

6.1 Summary

The goal of this project is to be able to improve portable machine learning performance of existing Google Tensorflow data flow graphs. The idea is to convert subgraphs of the dataflow graph into a functional expression that can be compiled into OpenCL code. In order to optimize the performance of the generated OpenCL code, it is possible to apply rewrite rules to the functional expression in order to improve its performance without changing the correctness of the expression.

The experiments show that it is possible to generate the functional expression for relevant subgraphs within an existing tensorflow application. It is shown that some of these expressions can be compiled without assistance, while others only need minor tweaking in order to correctly compile. The small corrections required for expressions to compile are due to current limitations in the Lift framework.

Finally the evaluation shows that by applying rewrite rules it is possibe to change and also improve the execution performance. The execution performance of the split kernels show promising results and indicate that by reducing overhead it is possible to compete with and sometimes even beat the highly optimized cuBLAS library. It is notable that by combining many specific types of operations it is possible to reduce overhead compared to the cuBLAS implementation, while other combinations may yield poor results.

6.2 Limitations and Future Work

After being able to show that it is possible to execute parts of Google Tensorflow graphs in Lift, it is necssary to discuss the limitations of the project and what still needs to be done.

The main limitations of the current state of work are the inability of all kernels to compile and execution performance. The first problem of not being able to compile all kernels has two causes. The first cause is that not all kernels are implemented. This would just require any further kernels that are required to be implemented. If they cannot be implemented in Lift an alternative implementation would be necessary. The other cause preventing kernels from being compiled is due to the Lift framework. This is either due to not being able to correctly rewrite *Map* operations so that memory allocation happens correctly, or because the OpenCL generator cannot correctly handle the memory allocation. Arguable, the exact fix to this depends on a design decision of the Lift framework. The problem could however also be fixed by having the performance optimization done on the functional expression.

The performance issues are due to a multitude of causes. First of all, the memory is currently handled in main memory instead of on the graphics card. While experimenting with Google Tensorflow it was noticed that the copying of inputs to the graphics card could easily account for nearly 20 times the execution time of the kernels required to process the data. This memory overhead was not examined though, since it is not relevant to the hypothesis of this thesis, which focuses on the execution performance. This means that it would ideally be necessary to ensure that all arternative implementations are performed on the graphics card instead of on the CPU, as this requires accessing the data from the main memory.

The other big performance issue is due to a lack of optimization of the kernels. The Lift framework aims to have a method of automatic optimization based on rewrite rules that produce identical code [8]. Determining which rules to apply is a complicated topic of its own, potentially requiring machine learning and other methods for determining what works best. Furthermore many of the rewrite rules still have to be discovered, which will in time lead to the increase of execution performance as more are found.

As can be seen by the automated combination of Lift nodes, not all combinations lead to increased execution performance. It would need to be determined how well certain

operations can be combined and optimized in the future.

Finally the solution presented does not do a good job of reusing kernels. For a performant execution of code it would be necessary to have the kernels precompiled and reusable during the entire application execution duration. Currently an intermediate representation of the functional expression is cached. It is still necessary to ensure that the input types of the kernel have not changed however, since the current implementation still relies on fixed kernels sizes.

6.3 Critical Analysis

The main issue with the work presented here is that it is problematic to determine comparable performance statistics with the used methods. It is difficult to determine the exact comparable execution time of the Tensorflow graph segment, since some of the additional overhead has to be considered between the internal kernel calls, while using **time**.*time()*. As such it is possible to assume a lower bound of roughly 0.5ms and an upper bound of roughly 1.5ms for a 6400 row input size, but difficult to determine an exactly comparible time. Ideally it would be better to perform some analysis on how much overhead is involved with different function calls.

The Lift programming language on the other hand only allows the timing of either the kernel execution time or the entire function call including the flattening of memory and copying it to the device. This allows for either a measurement which is lower or higher than that of the Google Tensorflow measurement. This can especially be seen by the fact that the Google Tensorflow execution performance is a lot less proportional to the input batch size than that of Lift, as can be seen in Figure 5.6. Due to the fusion of many of the kernels, a lot of the overhead is also reduced in comparison to using the Tensorflow implementation, making the lower figure somewhat justified. Despite this the execution performance demonstrated here is still promising and still has room for further improvements.

Another big issue is the fact that the project demonstrated here defines a new runtime in Scala that allows exported Tensorflow graphs to be executed. It would potentially have been a nicer solution to try and integrate the compilation and execution of Lift code within the Google Tensorflow framework. This way it would not be as serious if the implementation of certain nodes were missing, since they could simply be reused

from the Google Tensorflow framework. Furthermore it would have made it easier to integrate Lift into existing applications if the Lift framework had been used. Unfortunately integrating Lift directly into the Tensorflow framework would have complicated development and potentially have made it impossible within the given time constraints. As such, it was decided that the conversion of Tensorflow subgraphs in a simplified execution environment within Lift should be demonstrated, even if it is not the optimal environment in which to do it.

As mentioned in the previous section, it was also decided to have the intermediate results stored in main memory, because the alternative implementations for missing GPU kernels are performed in Scala. Requiring memory to be copied from the GPU to the main memory is very costly and inefficient and should be avoided as much as possible. It would then ahve been required to implement the alternative implementations on the graphics card, but this would have again been outside of scope for this project.

Finally a lot of the work execution performance here is evaluated on larger batch sizes, which may not always be relevent in machine learning applications. Yet for applications with smaller batch sizes the proportion of overhead per calculation becomes much higher. This makes the approach of generating very efficient kernels that perform more steps benificial, since even if the execution performance is slightly higher, the loss may be smaller than the overhead.

Bibliography

[1] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein *et al.*, "Imagenet large scale visual recognition challenge," *International Journal of Computer Vision*, vol. 115, no. 3, pp. 211–252, 2015.

[2] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2015, pp. 1–9.

[3] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," *arXiv preprint arXiv:1512.03385*, 2015.

[4] T. Tamasi, "Evolution of computer graphics," *Proc. NVISION*, vol. 8, 2008.

[5] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin *et al.*, "Tensorflow: Large-scale machine learning on heterogeneous distributed systems," *arXiv preprint arXiv:1603.04467*, 2016.

[6] A. Chrzeszczyk and J. Chrzeszczyk, "Matrix computations on the gpu," 2013.

[7] J. E. Stone, D. Gohara, and G. Shi, "Opencl: A parallel programming standard for heterogeneous computing systems," *Computing in science & engineering*, vol. 12, no. 1-3, pp. 66–73, 2010.

[8] M. Steuwer, C. Fensch, S. Lindley, and C. Dubach, "Generating performance portable code using rewrite rules: from high-level functional expressions to high-performance opencl code," in *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*. ACM, 2015, pp. 205–217.

[9] *Introduction to OpenCL Programming, Training Guide*. Advanced Micro Devices Inc., 2010.

[10] N. Jouppi, "Google supercharges machine learning tasks with tpu custom chip," https://cloudplatform.googleblog.com/2016/05/Google-supercharges-machine-learning-tasks-with-custom-chip.html, accessed: 2016-08-08.

[11] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer, "cudnn: Efficient primitives for deep learning," *arXiv preprint arXiv:1410.0759*, 2014.

[12] L. Dagum and R. Menon, "Openmp: an industry standard api for shared-memory programming," *IEEE computational science and engineering*, vol. 5, no. 1, pp. 46–55, 1998.

[13] N. Lopes and B. Ribeiro, "Gpumlib: An efficient open-source gpu machine learning library," *International Journal of Computer Information Systems and Industrial Management Applications*, vol. 3, pp. 355–362, 2011.

[14] R. Collobert, K. Kavukcuoglu, and C. Farabet, "Torch7: A matlab-like environment for machine learning," in *BigLearn, NIPS Workshop*, no. EPFL-CONF-192376, 2011.

[15] S. Lee and R. Eigenmann, "Openmpc: Extended openmp programming and tuning for gpus," in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE Computer Society, 2010, pp. 1–11.

[16] R. Miceli, G. Civario, A. Sikora, E. César, M. Gerndt, H. Haitof, C. Navarrete, S. Benkner, M. Sandrieser, L. Morin *et al.*, "Autotune: A plugin-driven approach to the automatic tuning of parallel applications," in *International Workshop on Applied Parallel Computing*. Springer, 2012, pp. 328–342.

[17] C. Dubach, M. Steuwer, and T. Remmelg, "Matrix multiplication beyond autotuning: Rewrite-based gpu code generation."

[18] S. Cadambi, I. Durdanovic, V. Jakkula, M. Sankaradass, E. Cosatto, S. Chakradhar, and H. P. Graf, "A massively parallel fpga-based coprocessor for support

vector machines," in *Field Programmable Custom Computing Machines, 2009. FCCM'09. 17th IEEE Symposium on*. IEEE, 2009, pp. 115–122.