DeepDSL: A Compilation-based Domain-Specific Language for Deep Learning

Tian Zhao & Xiao Bing Huang
Department of Computer Science
University of Wisconsin – Milwaukee
Milwaukee, WI, USA
`{tzhao,xiaobing}@uwm.edu`

Yu Cao
Department of Computational Neuroscience
The University of Massachusetts, Lowell
Lowell, MA, USA
`ycao@cs.uml.edu`

Abstract

In recent years, *Deep Learning* (DL) has found great success in domains such as multimedia understanding. However, the complex nature of multimedia data makes it difficult to develop DL-based software. The state-of-the-art tools, such as Caffe, TensorFlow, Torch7, and CNTK, while are successful in their applicable domains, are programming libraries with fixed user interface, internal representation, and execution environment. This makes it difficult to implement portable and customized DL applications.

In this paper, we present *DeepDSL*, a *domain specific language* (DSL) embedded in Scala, that compiles deep networks written in DeepDSL to Java source code. Deep DSL provides (1) intuitive constructs to support compact encoding of deep networks; (2) symbolic gradient derivation of the networks; (3) static analysis for memory consumption and error detection; and (4) DSL-level optimization to improve memory and runtime efficiency.

DeepDSL programs are compiled into compact, efficient, customizable, and portable Java source code, which operates the CUDA and CUDNN interfaces running on NVIDIA GPU via a *Java Native Interface* (JNI) library. We evaluated DeepDSL with a number of popular DL networks. Our experiments show that the compiled programs have very competitive runtime performance and memory efficiency compared to the existing libraries.

1 Introduction

Multimedia is increasingly becoming the "biggest big data" as the most important and valuable source for insights and information Chen et al. (2015a). Recently, a new set of machine learning algorithms named "Deep Learning" (DL) LeCun et al. (2015), which aims at learning multiple levels of representation and abstraction that help infer knowledge from multimedia data (e.g. text, image, audio, and video) is making astonishing gains in machine vision, speech recognition, multimedia analysis, and drug designing.

However, current tools, such as Theano Bergstra et al. (2010), Torch7 Collobert et al. (2011), Caffe Jia et al. (2014), *Computational Network Toolkit* (CNTK) Agarwal et al. (2014), and TensorFlow Abadi et al. (2016), while are efficient in their applicable domains, are essentially application libraries with some inherent limitations.

As with all programming libraries, the DL libraries have fixed bindings for key data structures such as tensors and tensor related computations. Users have to adhere to the data structure, which limits their ability to apply application-specific optimization or port it to target runtime platforms. The internal representation of their control flow logic is opaque to users. For example, TensorFlow and CNTK use directed acyclic graphs to represent the DL network computation and generate runtime binaries from the graphs. However, these graphs are not designed for user-level access, which limits the runtime platforms of the DL applications to what the libraries provide.

In general, the current libraries have to be built against specific platforms that they are designed for, which can be difficult for platforms such as Windows. Also, changing the implementation of
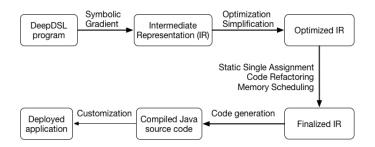
Figure 1: Basic workflow of DeepDSL.

specific type of layers or data structure is very challenging without thorough understanding of the underlying implementation. This limits the portability and reusability of these libraries.

To address these limitations, we present DeepDSL, a domain specific language embedded in Scala, for developing DL applications. DeepDSL allows users to define DL networks as tensor functions. Unlike the existing DL libraries, DSL tensors are not built-in entities. Instead, they are defined as indexed scalar expressions. This exposes tensor related computation at DSL level. As a result, the symbolic gradient derivation of the DL network is fully abstract and the resulting DSL program allows compiler-based optimizations such as code motion and common sub-expression elimination.

DeepDSL compiler translates the optimized DSL program into a Java source program that is compact, efficient, customizable, and portable. The generated Java source only requires a small Java library JCuda [1] that calls the NVIDIA CUDA interface using JNI. Since JVM is supported on all major operating systems, the generated Java source can run on any CUDA enabled platforms. Also, since the generated Java source is compact and human readable, users can customize it easily through an editor or IDE such as eclipse [2]. The generated Java source automatically saves the learned parameters into files after a training period is over. When the user starts the program again (perhaps after adjusting some parameters such as momentum and learning rate), it automatically loads the saved parameters and continues the training from where it stopped at the previous execution. The code also supports loading parameters trained with different data for fine tuning purpose.

DeepDSL supports statical analysis of the DSL program to detect network design errors such as mismatching tensor dimensions before compiling the DSL program into Java source. It statically analyzes the memory consumption used at each step of the computation and produces a table detailing the memory usage that would occur at runtime, which includes the memory for feature maps, gradient maps, parameter weights, and convolution workspace. It also uses the static information to reschedule computation so that tensor memory can be freed as early as possible to reduce memory consumption at runtime. Such processing has demonstrated to have great benefit. For example, DeepDSL continues to run well under the GPU memory limit on the testing server with a single GPU when the batch size of ResNet is increased from 32 to 64, while both Caffe and Tensorflow fail due to out of memory exception.

DeepDSL is available at `https://github.com/deepdsl/deepdsl`.

The rest of the paper is organized as follows. We give an overview of DeepDSL in Section 2 and explain the DSL syntax using examples in Section 3. We discuss the intermediate representation in Section 4 and code generation in Section 5. We present details of performance evaluation using DeepDSL in Section 6 and related work in Section 7. We conclude the paper in Section 8.

2 OVERVIEW

DeepDSL directly encodes the mathematical representation of DL networks, where each layer is represented as a tensor function. The entire network is then represented as a composition of these

---

[1]http://www.jcuda.org

[2]http://www.eclipse.org

```scala
1    val K = 10 // # of classes
2    val N = 500; val C = 1; val N1 = 28; val N2 = 28 // batch size, channel, and x/y size
3
4    // Specifying training (and test) dataSet
5    val y = Vec._new(Mnist, "label", "Y", N)          // labels
6    val x = Vec._new(Mnist, "image", "X", N, C, N1, N2) // images
7
8    val cv1 = CudaLayer.convolv("cv1", 5, 20)        // kernel size (5,5), output channel 20
9    val cv2 = CudaLayer.convolv("cv2", 5, 50)
10   val mp = CudaLayer.max_pool(2)                    // max pooling, kernel 2 stride 2
11   val flat = Layer.flatten(4, 1)                    // flatten a 4-D tensor from axis 1 to 3
12   val f = Layer.full("fc1", 500)                    // fully connected layer, output 500
13   val f2 = Layer.full("fc2", K)
14   val relu = CudaLayer.relu(2)                      // 2-D ReLU activation
15   val softmax = CudaLayer.softmax                   // softmax
16
17   // o is a left-associative operator for function composition
18   val network = f2 o relu o f o flat o mp o cv2 o mp o cv1
19
20   val x1 = x.asCuda                                 // load x to GPU
21   val y1 = y.asIndicator(K).asCuda                  // turn each label into an indicator vector
22   val c = (Layer.log_loss(y1) o softmax o network) (x1) // training loss
23   val p = (Layer.precision(y1) o network) (x1)          // test accuracy
24
25   val param = c.freeVar.toList                      // parameters to be trained
26
27   // output file, train and test iteration, learn rate, momentum, decay, gradient cropping (0 means none)
28   val solver = Train("lenet", 1000, 10, 0.01f, 0.9f, 0.0005f, 0)
29
30   val loop = Loop(c, p, (x, y), param, solver)      // training and testing loop
31   cudnn_gen.print(loop)                             // generate Java source program
```

Figure 2: DeepDSL code for training and testing *Lenet*.

functions. DeepDSL symbolically derives the partial derivatives of the tensor functions with respect to tensor variables so that the backward gradients of network parameters are generated automatically.

A high-level overview of DeepDSL is shown in Figure 1. A DeepDSL program is compiled in several stages. At the first stage, the backward gradients of deep networks are derived symbolically to become the intermediate representation (IR). The IR expressions are in turn passed through a series of simplification and optimization at the second stage. At the third stage, DeepDSL compiler performs a SSA (Static Single Assignment) transformation of the optimized IR to break down complex expressions. Redundant computation is eliminated at this stage and the resulting expressions are reordered to optimize memory usage. Memory deallocation and in-place computation are also scheduled at this stage. Lastly, the finalized IR expressions are translated to Java source code.

DeepDSL supports two mode of computation: memory efficient or runtime efficient. In the memory efficient mode, tensor memory in GPU will be dynamically allocated and deallocated, which might decrease runtime performance. In the runtime efficient mode, tensor memory in GPU is reused and not deallocated until the end of the training. In this mode, more memory may be used but with greater runtime performance. To make the switch, the user only needs to switch a flag in the generated Java source. The memory efficient mode can be used for machines with limited GPU memory. Further memory reduction can be achieved by placing a limit on the (convolution) workspace memory.

3 SYNTAX

Figure 2 shows the complete implementation for compiling a program to train and test *Lenet* LeCun et al. (1998). Since DeepDSL is embedded in Scala, the program is in Scala syntax and it can be compiled and executed with a programming tool such as eclipse. This program consists of variable declarations of the form `val x = e`, where `val` starts a declaration for the variable `x` and assigns it with the value of `e`.

Line 5 and 6 declare the tensors that represent labels and images for the training data. We also use the same variables for testing since the DSL compiles the same variables into different code for training and testing.

Line 8–15 declare the tensor functions that represent the layers in the network. Most of the layers are self-explanatory except `val flat = Layer.flatten(4, 1)`, which is used to convert the 4-D tensor returned by the last pooling layer into a 2-D layer for the next fully connected layer.

Line 18 constructs the network as function compositions using the operator `o`, which is left associative. For example, `f2 o relu o f` should be read as `(f2 o relu) o f`. A composed function such as `network` is still a function.

Line 22 defines the expression that represents the loss of the network when applied to the training data. Line 23 defines the testing accuracy of the trained network.

Line 25 extracts the parameters such as weights and biases from the loss expression. Line 28–31 defines the solver object, passes it to the loop object for training and testing, and then generates the Java source code.

Layer reuse Since each layer is a tensor function, for the layers such as ReLU and pooling that do not contain parameters, we can simply reuse them in a network. For example, in the following definition for *Alexnet*, `relu2` (2 dimensional), `relu` (4 dimensional), `pool` (max pooling), `drop` (drop out), and `lrn` (local response normalization) are reused.

```
1    val network =        full8 o
2        drop o relu2 o full7 o
3        drop o relu2 o full6 o flat o
4         pool o relu o cv5 o
5                 relu o cv4 o
6                 relu o cv3 o
7    pool o lrn o relu o cv2 o
8    pool o lrn o relu o cv1
```

Layer function reuse simplifies the definitions of deep networks. For *Alexnet*, only 5 convolution layers and 3 fully connected layers need to be defined separately. Note that the above definition can be written in just one line and the line breaks are only for clarity.

Network reuse For complex network such as *Googlenet*, we can define reusable subnet to achieve compact definitions. For example, the Scala method `inception` below returns a tensor function that represents an inception subnet in *Googlenet*.

```
1    val w = Param.xavier                    // Xavier initialization for weight
2    val b0 = Param.const(0, 2, 0)           // constant 0 for bias, learn rate/decay multiplier 2 and 0
3    val b02 = Param.const(0.2f, 2, 0)       // constant 0.2 for bias
4    val ipool = CudaLayer.max_pool(3, 1, 1) // max pooling kernel size, stride, and padding
5
6    def inception(n: Int) = {
7      // convolution name, kernel size, channel, stride, padding, weight and bias configuration
8      val icv1 = CudaLayer.convolv(s"cv${n}1", 1, 64, 1, 0, w, b02)
9      val icv2 = CudaLayer.convolv(s"cv${n}2", 1, 96, 1, 0, w, b02)
10     val icv3 = CudaLayer.convolv(s"cv${n}3", 3, 128, 1, 1, w, b02)
11     val icv4 = CudaLayer.convolv(s"cv${n}4", 1, 16, 1, 0, w, b02)
12     val icv5 = CudaLayer.convolv(s"cv${n}5", 5, 32, 1, 2, w, b02)
13     val icv6 = CudaLayer.convolv(s"cv${n}6", 1, 32, 1, 0, w, b02)
14
15     val p = Vec._new(4)                              // a 4-dimensional tensor variable
16
17     // a tensor function with parameter p
18     VecFun(p, CudaLayer.concat( (relu o icv1)(p),   // concatenation of 4 subnets connected to p
19                 (relu o icv3 o relu o icv2)(p),
20                 (relu o icv5 o relu o icv4)(p),
21                     (relu o icv6 o ipool)(p) )
22     )
23   }
```

Using the `inception` method, we can define three subnets that are used to define the test accuracy `p` (line 6 below) of the main branch of *Googlenet*.

```
1    val network3 = full7 o flat o drop o pool7 o inception(9) o inception(8) o pool o inception(7)
2    val network2 = inception(6) o inception(5) o inception(4)
3    val network1 = inception(3) o pool o inception(2) o inception(1) o
4                         pool o lrn o relu o cv3 o relu o cv2 o lrn o pool o relu o cv1
5
6    val p = Layer.precision(y1)(network3(network2(network1(x1))))  // accuracy at main branch
```

4

The three subnets are also used to define the training loss `c` (line 16 below) that adds up the losses of the three branches of *Googlenet*.

```
1    def branch(n: Int) = {      // a subnet reused in the two side branches of Googlenet
2      val cv = CudaLayer.convolv(s"b${n}cv", 1, 128, 1, 0, w, b02)
3      val f1 = Layer.full(s"b${n}fc1", 1024, w, b02)
4      val f2 = Layer.full(s"b${n}fc2", K, w, b0)
5      f2 o drop2 o relu2 o f1 o flat o relu o cv o bpool
6    }
7    val stage2 = {              // Vec2ScalarFun defines a function from tensor to scalar
8      val p = Vec._new(4)
9      Vec2ScalarFun(p, softmax_loss(network3(p)) + softmax_loss(branch(2)(p)) * Real(0.3f, "loss2"))
10   }
11   val stage1 = {              // Real(0.3f, "loss1") is a named constant of value 0.3
12     val p = Vec._new(4)
13     Vec2ScalarFun(p, stage2(network2(p)) + softmax_loss(branch(1)(p)) * Real(0.3f, "loss1"))
14   }
15
16   val c = (stage1 o network1)(x1)  // training loss of the three branches
```

Other than some definitions of shared layers such as activation, pooling, normalization, drop out, and softmax loss, this is the complete definition of *Googlenet*.

This compact style of definition is similar to that of Theano, Tensorflow, Torch, and Mxnet. In the example, we used two types of functions `VecFun` and `Vec2ScalarFun`, which model computation that takes a tensor as input and returns a tensor or scalar respectively. These functions can be composed or applied to arguments. When applied, they are similar to functions in Theano, Tensorflow, and Mxnet. When composed, they are similar to the sequential container of Torch.

4 INTERMEDIATE REPRESENTATION

The unique advantage of DeepDSL is that it is entirely high-level so that it permits static analysis of the deep networks for error checking, memory analysis, optimization, and code generation.

While DeepDSL compiler is implemented in Scala, it has no runtime dependency on code in Scala at all. The whole purpose of using Scala as the host language for DeepDSL is that Scala is a strongly typed language with flexible syntax. As a result, the syntax of DeepDSL can resemble that of a standalone DSL without having a parser. After taking symbolic gradients, a DeepDSL program is immediately evaluated to intermediate representation (IR), which is essentially an *abstract syntax tree* (AST). DeepDSL compiler analyzes its IR expressions by performing a series of optimization and simplification steps. During this process, DeepDSL checks the compatibility of the layers, infers concrete dimensions for variables, removes duplicated computation, and optimizes IR expressions for code generation.

The IR expressions of DeepDSL are also abstract and human readable. For example, Figure 3 shows a portion of the IR expressions for Lenet, where the first column shows an IR expression that represents a single-step computation, the second column shows the dimensions of the tensor being computed if applicable, the third column shows the memory usage of that tensor, the fourth column shows the current memory consumption if memory is dynamically allocated and deallocated, and the last column shows the memory consumption if memory is reused instead of deallocated.

IR expression such as Line 14 is for GPU memory deallocation. DeepDSL compiler analyzes the dependencies of the IR expressions, reorders them, and determines the earliest point where a tensor can be freed. For example, the last use of the tensor `X18` in at line 13, it can be freed next. The tensor `X8` cannot be freed until much later since it is used at line 26.

If we compile IR expressions such as line 14 to actual memory deallocation, then the maximum dynamic memory consumed is peaked at line 27, which is about 59 MB. However, frequent memory allocation and deallocation in NVIDIA GPU reduces runtime performance. Therefore, DeepDSL runtime library (implemented in Java) supports memory reuse instead of deallocation. DeepDSL runtime maintains a pool of allocated memory blocks and when a tensor is freed, its memory is returned to the pool and when a tensor is allocated, the runtime tries to find a suitable block in the pool first. With memory reuse, the memory consumption always peaks at the last line, which is about 77 MB. Note that the above memory figure is for storing intermediate results such as gradients; the static memory allocated for parameters and convolution workspace are calculated separately.

```
1  IR expression                            Dimensions   Current mem    Total    w/o dealloc
2  --------------------------------------------------------------------------------------
3  val X7 = Cuda(X)                         500 1 28 28    1.568000    1.568000    1.568000
4  val X8 = Convolv(1,0)(X7,cv1_W,cv1_B)    500 20 24 24  23.040001   24.608000   24.608000
5  val X9 = Pooling(2,2,0,true)(X8)         500 20 12 12   5.760000   30.368000   30.368000
6  val X10 = Convolv(1,0)(X9,cv2_W,cv2_B)   500 50 8 8     6.400000   36.768002   36.768002
7  val X11 = Pooling(2,2,0,true)(X10)       500 50 4 4     1.600000   38.368000   38.368000
8  val X12 = (X11[1><3])(i | @) * (fc1_W)(j | @) 500 500   1.000000   39.368000   39.368000
9  val X14 = (X12 + (i) => fc1_B)           500 500        0.000000   39.368000   39.368000
10 val X15 = ReLU()(X14)                    500 500        0.000000   39.368000   39.368000
11 val X16 = (X15)(i | @) * (fc2_W)(j | @)  500 10         0.020000   39.388000   39.388000
12 val X18 = (X16 + (i) => fc2_B)           500 10         0.000000   39.388000   39.388000
13 val X19 = Softmax()(X18)                 500 10         0.020000   39.408001   39.408001
14 Dealloc(X18)                                           -0.020000   39.388000   39.408001
15 val X20 = Cuda(Indicator(Y, 10))         500 10         0.020000   39.408001   39.408001
16 val X21 = Log X19.copy                   500 10         0.020000   39.428001   39.428001
17 val X52 = 1/(X19.copy)                   500 10         0.020000   39.448002   39.448002
18 Print(((0 - (X20 . X21)) / |500|))                      0.000000   39.448002   39.448002
19
20 ................. 30 lines omitted ......................................................
21
22 cv2_B <~~ X71 * d_Convolv(1,0)()/d_cv2_B                0.000000   36.768002   48.448002
23 val X72 = X71 * d_Convolv(1,0)(cv2_W)/d_X9  500 20 12 12  5.760000  42.528000   54.208000
24 cv2_W <~~ X71 * d_Convolv(1,0)(X9)/d_cv2_W             0.000000   42.528000   54.208000
25 Dealloc(X71)                                           -6.400000   36.127998   54.208000
26 val X74 = X72 * d_Pooling(2,2,0,true)(X9,X8)/d_X8
27                                          500 20 24 24  23.040001   59.167999   77.248001
28 Dealloc(X72)                                           -5.760000   53.408001   77.248001
29 Dealloc(X9)                                            -5.760000   47.647999   77.248001
30 Dealloc(X8)                                           -23.040001   24.608000   77.248001
31 cv1_B <~~ X74 * d_Convolv(1,0)()/d_cv1_B               0.000000   24.608000   77.248001
32 cv1_W <~~ X74 * d_Convolv(1,0)(X7)/d_cv1_W             0.000000   24.608000   77.248001
33 Dealloc(X74)                                          -23.040001    1.568000   77.248001
34 Dealloc(X7)                                            -1.568000    0.000000   77.248001
```

Figure 3: A portion of the IR expressions and memory information compiled from Lenet

DeepDSL compiler generates Java source code for each of the IR expressions. For example, line 3 loads a batch of images into GPU memory. Line 4 and line 5 perform forward convolution and pooling computation respectively. Line 18 prints out the training loss. Line 22 updates of the bias of the second convolution layer with its gradient.

Some computation (e.g. Log) is always in-place. Therefore we make a copy of a tensor if it is passed to such computation (e.g. Log X19.copy). Gradient update such as cv1_W <~~ X74 * d_Convolv(1,0)(X7)/d_cv1_W may be implemented as in-place computation as well by directly updating the tensor cv1_W when computing the backward filter gradient of the convolution layer cv1.

5 COMPILATION

A DeepDSL program compiles to a Java source program, which uses a small Java library JCuda to call CUDA and CuDNN via a JNI wrapper. The compiled Java code does not depend on DeepDSL compiler or Scala, which makes it more portable and easier to integrate with other applications.

Most of the current tools use platform dependent programming languages such as C, Python, and Lua, which compiles to specific binary for each installation. Since our compiled program is Java, it runs directly on any platforms that support JVM. Compilation of Java is trivial for most computing platforms. For example, the Java source generated by DeepDSL on a Windows laptop can run on a Linux server without any modifications. While it takes efforts to install tools like Tensorflow, Caffe, or Torch on machines with different system architectures, running the Java code generated from DeepDSL require very little effort.

Gradient Derivation and Optimization   The gradient derivation and optimization are implemented by the Loop class called in code below:

```
1  val loop = Loop(loss, accuracy, (x, y), param, solver)
```

To derive the gradient of a scalar expression `loss` with respect to a tensor variable `p`, we can write `val grad = loss.grad(p)`, which evaluates to a tensor expression. The gradient updates are formed by expressions of the form $\text{Update}(p, \text{grad}, \alpha, \beta)$, which represents the computation $p = \beta * p + \alpha * \text{grad}$.

The gradient updates of all parameters together with the loss expression are then passed to optimization functions to obtain a list of IR expressions ready for code generation. The optimization functions implement simplification, loop merging, code motion, vectorization, SSA transformation, common sub-expression elimination, inlining, tensor deallocation, and code scheduling.

Generated code   The compiled Java code includes just one class. The class fields include the objects that handle computations such as convolution and activation and the objects that store tensors such as parameters and gradients. The class includes one method for the training loop and one method for testing.

The generated code includes the corresponding IR expressions in the comments to improve readability. For example, the code below shows the Java statements generated for the forward inference of max pooling. Note that the variable name in the comments has no relation to the variable names in the code as they are independently generated.

```
1    // val X9 = Pooling(2,2,0,true)(X8)
2    JCudaTensor x16;
3    JCudaTensor x17;
4    x17 = x9;
5    x16 = x18.forward(x17);
```

It is easy to perform some customization of the generated code such as changing the number of training iterations or reducing learning rate at a specified interval. User can also use the generated code as a component of another application.

Persistency   The compiled Java source includes code to save the trained parameters into files after the training is complete. When the same program or another program compiled from the same network starts, it can load the same parameters to resume training or for forward testing.

Workspace   The convolution layers in the compiled Java source share the same workspace. Thus, users can place a limit on the total workspace by making one change. By reducing workspace and using memory efficient mode, users may reduce memory consumption to fit into a particular GPU.

6   PERFORMANCE

The primary compilation target of DeepDSL is a Java program that runs on Nvidia GPU through its CUDA/CuDNN library[3]. DeepDSL can encode well-known networks such as Alexnet, Overfeat, GoogleNet, Vgg, and Deep Residual Networks (Resnet). In this section, we evaluate the performance of DeepDSL with Caffe and Tensorflow using these networks. To be consistent, DeepDSL, Caffe, and Tensorflow tests all follow the same Caffe prototxt definitions. Specifically, for Alexnet and GoogleNet, we followed the prototxt from Caffe's website[4]; for Vgg (Vgg-16), we followed the prototxt from this link[5]; for Overfeat, we followed the prototxt from IntelLabs[6]; and for Deep Residual Network (ResNet-50), we followed the prototxt from the author's website[7]. The Tensorflow implementation of these networks are either modified from versions of convnet-benchmarks[8] or created from scratch. Note there are a couple of differences between the tests of Tensorflow and those of DeepDSL and Caffe. The training data in the Tensorflow tests is generated from random data in memory while DeepDSL and Caffe tests load real images from the Lmdb database. Also,

---

[3]DeepDSL has limited support for CPU with features sufficient to implement Lenet.

[4]`github.com/BVLC/caffe/tree/master/models`

[5]`github.com/ruimashita/caffe-train/blob/master/vgg.train_val.prototxt`

[6]`github.com/IntelLabs/Latte.jl/blob/master/benchmarks/overfeat/overfeat.prototxt`

[7]`github.com/KaimingHe/deep-residual-networks/blob/master/prototxt/ResNet-50-deploy.prototxt`

[8]`github.com/soumith/convnet-benchmarks`

the GoogleNet test of Tensorflow only includes the main branch of the GoogleNet while DeepDSL and Caffe train with the full network. All our tests are trained with ImageNet images that have been resized to 224 by 224 (though DeepDSL do support random cropping of images when their sizes are larger than specified dimensions).
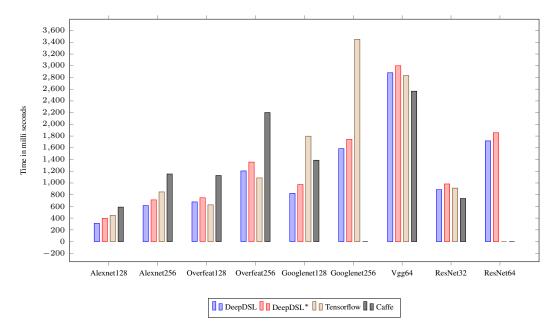
Figure 4: Runtime performance of DeepDSL, Tensorflow, and Caffe (1 forward/backward iteration). DeepDSL and DeepDSL* are performance in runtime-efficient and memory-efficient mode respectively. The names of the networks are followed by the batch size. Caffe failed to run GoogleNet (batch 256) and ResNet (batch 64) and Tensorflow failed to run ResNet (batch 64) due to exhaustion of GPU memory.

The tests are run on a server with a single NVIDIA Tesla K40C GPU equipped with 12 gigabytes of memory. The server runs the CentOS 7 Linux distribution. DeepDSL uses the JCuda 0.8.0RC binding that runs against CUDA 8.0.27[9]. DeepDSL programs are publicly available[10].

The runtime performance of DeepDSL, Tensorflow, and Caffe is compared in Figure 4, where DeepDSL has significant advantage over Caffe in Alexnet, Overfeat, and Googlenet while only marginally slower than Caffe in Vgg and ResNet (Deep Residual Network). DeepDSL is also faster than Tensorflow in Alexnet, Googlenet, and ResNet while slightly slower in Overfeat and Vgg.

The memory consumption of the DeepDSL, Tensorflow, and Caffe is compared in Figure 5, where DeepDSL uses less memory in Alexnet, Googlenet, and ResNet while Caffe uses less memory in Overfeat and Vgg. DeepDSL uses significantly less memory for Googlenet and ResNet where Caffe runs out of memory for Googlenet at batch size 256 and ResNet at batch size 64. DeepDSL uses less memory than Tensorflow in all tests except Vgg. Tensorflow also ran out of memory for ResNet at batch size 64. It is unclear why Tensorflow uses similar amount of memory for Overfeat with batch size 128 and 256.

In the tests, DeepDSL programs are run with runtime efficient mode which caches tensor objects and with memory efficient mode (denoted by DeepDSL*) which deallocates tensor objects as soon as possible. DeepDSL* uses 10 to 30% less memory with similar percentage of runtime overhead except Vgg and Googlenet where runtime overhead is relatively smaller than memory saving.

DeepDSL also lets CUDNN to pick the convolution algorithms with max performance. In Overfeat (batch size 128), out of the 4290 megabytes of GPU memory consumed, more than 2700 megabytes are for convolution workspace. While Caffe uses less memory in this test, it also runs much slower.

---

[9]Note previous CUDA versions such as 6.5 or 7.x can also be used with minor modifications.
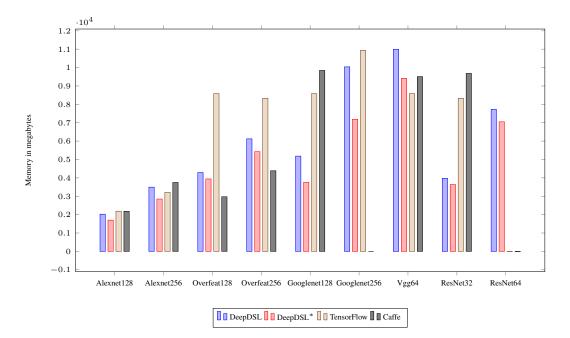
[10]github.com/deepdsl/deepdsl

8

Figure 5: Peak GPU memory use of DeepDSL, Tensorflow, and Caffe during training. DeepDSL and DeepDSL* are performance in runtime-efficient and memory-efficient mode respectively. Caffe ran out of GPU memory for Googlenet (batch 256) and ResNet (batch 64). Tensorflow ran out of memory for ResNet (batch 64).

Among all tests, DeepDSL either outperforms Caffe by a large margin or uses significantly less memory with Vgg being the only exception where Caffe uses slightly less time and memory. DeepDSL also has competitive runtime performance when compared with Tensorflow.

As a side note, while running DeepDSL requires little setup, installing libraries such as Caffe and Tensorflow requires a list of dependencies and long compilation sessions. Consequently, we skipped testing with Torch 7 due to time limitation.

7 RELATED WORK

In this section, we review some popular tools: Torch7, Theano, Caffe, TensorFlow, and CNTK, and newer ones such as Chainer Tokui et al. (2015) and MXNet Chen et al. (2015b).

Torch7 Collobert et al. (2011) uses Lua language for integration with C program and achieves C-like performance. It has a large set of optimized routines to support CPU, GPU, mobile and FPGA backends. Theano Bergstra et al. (2010), hosted in Python, allows users to define symbolic variables and functions (using NumPy van der Walt et al. (2011)) to encode DL networks and compiles the symbolic expressions to C. Theano performs optimization such as normalizing mathematical expressions, numerical stabilization, and code specialization during compilation and the target code can run on CPU or GPU devices. Caffe Jia et al. (2014) constructs a graph for DL network by connecting the layers with the 4D arrays that store tensors. Caffe separates its DL network model representation (using ProtocolBuffers Google) from the actual model parameter calculations. With its layered structure, Caffe computes the memory needed for each layer and reserves memory accordingly. TensorFlow Abadi et al. (2016) shares largely common design paradigms as that of Caffe. Its core is written in C++ and its computation graph is described with a graph where tensors and layers are alternatively arranged. Unlike Caffe's tensor, TensorFlow's tensor is a typed multi-dimensional array and is persistent mutable. Like TensorFlow and Caffe, CNTK describes a network with a configuration file. CNTK can encode arbitrary computational network and it can map computation onto multiple GPUs across multiple machines by assigning each computation node to a particular CPU/GPU device.

Comparing to the "define-and-run" paradigm (adopted by Torch7, Theano, and Caffe), Chainer Tokui et al. (2015) follows a "define-by-run" pattern, which essentially allows modifying the control flow during the execution of a computational graph. MXNet Chen et al. (2015b) provides both declarative and imperative programming styles and multiple language supports by embedding into multiple host languages and unifying the execution with one backend engine.

The major difference between DeepDSL and the above tools is that DeepDSL is fully abstract until code generation. This means that DeepDSL's intermediate representation can be compiled to different languages or to run on different platforms. While the current compilation target of DeepDSL is Java, targeting a different language mainly involves building an interface library to call CUDA routines while the optimization components of DeepDSL remain the same. This separation between optimization and code generation also means that we can apply generic optimization techniques at IR level without worrying about the underlying data structure such as the representation of tensors or how the layers are connected. In fact, the optimization of DeepDSL involves nothing specific to deep neural networks since they are mostly compilation techniques.

Note that while Theano and DeepDSL have similarity in the way that DSL expressions are optimized and transformed, there are two important differences that make DeepDSL more efficient and flexible.

The first is that while Theano expressions are treated as graphs during optimization, DeepDSL expressions are optimized in two phases. The first phase is at expression level where the training loss and the parameter gradients go through the process of simplification, loop merging, code motion, and vectorization. In the second phase, DeepDSL expressions are reduced to *static single assignment* form for additional optimization such as common subexpression elimination, code scheduling, inlining of in-place computation, and tensor deallocation.

The second is that DeepDSL generates target code using a single-pass generator (about 1200 lines of code) that prints Java source code as strings to a file. The input of the generator is DeepDSL expressions, which are completely independent from the generated code. The generated Java code is high-level and human readable with a simple Java API that allows customization. This clean separation between DSL expression and target code also allows independent evolution of DSL optimization and target-code generation. In contrast, the code generation of Theano is embedded in its functions for low-level computation and is tied to C code that is not readable to users.

8 CONCLUSION

We have developed a domain specific language DeepDSL that compiles to Java source program for deep learning. The compiled DeepDSL programs are very easy to use and extend as its primary dependencies are just JCuda and CUDA libraries. DeepDSL programs are also efficient and its runtime performance and memory consumption are significantly better than Caffe and Tensorflow in some DL networks. DeepDSL performs static analysis for early error detection and provides readable intermediate representation and memory consumption analysis. DeepDSL allows compact encoding of complex networks and since it is based on a strongly typed language Scala, writing DeepDSL programs is less error prone than dynamic languages such as Python.

While the compiled DeepDSL programs are efficient, DeepDSL itself is not optimized. Though compiling simpler networks such as Alexnet takes a few seconds, the compilation of complex networks such as ResNet can take a few minutes. As the future work, we plan to optimize DeepDSL to improve the compilation efficiency. Also while the memory efficient mode of DeepDSL can reduce GPU memory consumption, it may not be enough for memory intensive networks such as Vgg. As future work, we plan to implement GPU memory virtualization by paging out tensors that are not immediately needed.

REFERENCES

M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mane, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Vie-

gas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng. TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems. *ArXiv e-prints*, March 2016.

Amit Agarwal, Eldar Akchurin, Chris Basoglu, Guoguo Chen, Scott Cyphers, Jasha Droppo, Adam Eversole, Brian Guenter, Mark Hillebrand, Ryan Hoens, Xuedong Huang, Zhiheng Huang, Vladimir Ivanov, Alexey Kamenev, Philipp Kranen, Oleksii Kuchaiev, Wolfgang Manousek, Avner May, Bhaskar Mitra, Olivier Nano, Gaizka Navarro, Alexey Orlov, Marko Padmilac, Hari Parthasarathi, Baolin Peng, Alexey Reznichenko, Frank Seide, Michael L. Seltzer, Malcolm Slaney, Andreas Stolcke, Yongqiang Wang, Huaming Wang, Kaisheng Yao, Dong Yu, Yu Zhang, and Geoffrey Zweig. An introduction to computational networks and the computational network toolkit. Technical Report MSR-TR-2014-112, August 2014. URL `http://research.microsoft.com/apps/pubs/default.aspx?id=226641`.

James Bergstra, Olivier Breuleux, Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, Guillaume Desjardins, Joseph Turian, David Warde-Farley, and Yoshua Bengio. Theano: a CPU and GPU math expression compiler. In *Proceedings of the Python for Scientific Computing Conference (SciPy)*, June 2010. Oral Presentation.

Shu-Ching Chen, Ramesh Jain, Yonghong Tian, and Haohong Wang. Guest editorial multimedia: The biggest big data. *Multimedia, IEEE Transactions on*, 17(9):1401–1403, 2015a.

Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *Neural Information Processing Systems, Workshop on Machine Learning Systems*, 2015b.

R. Collobert, K. Kavukcuoglu, and C. Farabet. Torch7: A matlab-like environment for machine learning. In *BigLearn, NIPS Workshop*, 2011.

Google. Protocol buffers. `http://code.google.com/apis/protocolbuffers/`.

Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. *arXiv preprint arXiv:1408.5093*, 2014.

Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521(7553):436–444, 2015.

Seiya Tokui, Kenta Oono, Shohei Hido, and Justin Clayton. Chainer: a next-generation open source framework for deep learning. In *Proceedings of Workshop on Machine Learning Systems (LearningSys) in The Twenty-ninth Annual Conference on Neural Information Processing Systems (NIPS)*, 2015. URL `http://learningsys.org/papers/LearningSys_2015_paper_33.pdf`.

Stéfan van der Walt, S. Chris Colbert, and Gaël Varoquaux. The numpy array: a structure for efficient numerical computation. *CoRR*, abs/1102.1523, 2011. URL `http://arxiv.org/abs/1102.1523`.