Master's Thesis

OpenDNN: An Open-source, cuDNN-like Deep Learning Primitive Library

cuDNN과 유사한 인터페이스를 갖는 오픈소스 딥 러닝 프리미티브 라이브러리

February 2019

Graduate School of Seoul National University

Computer Science and Engineering

Daeyeon Kim

Abstract

OpenDNN: An Open-source, cuDNN-like Deep Learning Primitive Library

Daeyeon Kim
Department of Computer Science and Engineering
The Graduate School
Seoul National University

Deep neural networks (DNNs) are a key enabler of today's intelligent applications and services. cuDNN is the de-facto standard library of deep learning primitives, which makes it easy to develop sophisticated DNN models. However, cuDNN is a propriatary software from NVIDIA, and thus does not allow the user to customize it based on her needs. Furthermore, it only targets NVIDIA GPUs and cannot support other hardware devices such as manycore CPUs and FPGAs. In this thesis we propose OpenDNN, an open-source, cuDNN-like DNN primitive library that can flexibly support multiple hardware devices. In particular, we demonstrate the portability and flexibility of OpenDNN by porting it to multiple popular DNN frameworks and hardware devices, including GPUs, CPUs, and FPGAs.

Keywords: Deep learning, Library, Open-source, Accelerators, Performance, Portability
Student Number: 2017-23840

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Deep neural networks (DNNs) are a key enabler of today's AI-based applications and services. Unlike the traditional rule-based AI systems, DNNs solve complex problems by stacking various artificial neural network layers. For example, convolution neural networks (CNNs) achieve high accuracy in image classification [14, 16, 22], object detection [23, 28] and image captioning. Besides, Recurrent neural networks (RNNs), long short-term memory (LSTM), and memory augmented neural networks (MANNs) provide superior performance in various natural language processing area like question answering [15, 24, 31]. Furthermore, Google uses multi-layer perceptrons (MLPs) to customize their advertisements.

As the complexity of DNNs grows, the computational requirements for running them also have grown significantly. Generally, because the number of layers heavily affects the DNN performance, today's sophisticated DNNs often employ many layers to spend a lot of time to train and validate them. For example, AlexNet [22] requires 62.38 million parameters and 2.5 giga-operations (GOPs) to classify a single image. VGG-19 [30], one of the most complex CNNs, has 138.36 million parameters and requires 38.5 GOPs to do the same. RNNs and LSTMs also have additional parameters to save cell state and function parameters.

To satisfy this demand GPUs and domain-specific architectures (DSAs) are widely used for both training and inference to provide many computation resources and large

on-chip memory. At the time of this writing GPUs are the de-facto standard for running DNNs. For example, NVIDIA V100 [25] provides not only CUDA cores for general-purpose GPU computing, but also tensor cores to accelerate tensor multiplications. It also has 16GB HBM2 to maximize memory bandwidth, which is essential to provide good DNN performance. Besides, DSAs are also becoming commonplace for both performance and energy efficiency. Google TPUv2 [18] can perform 45 TFLOPS using 16GB HBM with 600GB/s memory bandwidth and uses its own custom floating-point type, called bfloat16. Cambricon-X [32] accelerates DNNs by skipping the zero computation to reduce energy as well as increase the overall speeds as the DNNs are generally sparse and a significant portion of the weights and activations values are zero.

To make DNN programming easier, NVIDIA provides cuDNN, a deep learning primitive library targeting its GPUs. cuDNN offers an easy-to-use API, unified virtual memory, and C++-style standard template library. However, there is no stadard library for DSAs and FPGAs. As a result, DSA developers should not only write custom drivers but also port popular DNNs themselves. This is done in an ad-hoc manner and often not reuseable across multiple devices and frameworks. As for cuDNN it is not a free software, and the user cannot customize it as she wishes.

To address this problem, we propose OpenDNN, an open-source, cuDNN-like primitive library that can flexibly support multiple hardware devices and frameworks (e.g., Caffe [17], TensorFlow [7]) with low programmer effort. OpenDNN API often has one-to-one correspondence to the cuDNN API, to make it easy to port popular DNN frameworks based on cuDNN to the new hardware device. We demonstrate the versatility of OpenDNN by building an end-to-end prototype for three different hardware devices: CPU, GPU, and PCIe-based FPGA. Furthermore, OpenDNN is successfully ported to popular DNN frameworks, including Caffe and TensorFlow. Last but not least, OpenDNN is a free software soon to be open-sourced for the

benefits of the research community.

In summary OpenDNN satisfies the following design objectives:

- *Portability* - The C++ API provides a simple interface that can flexibly support multiple DNN frameworks and hardware platforms.
- *Versatility* - The open source code is released and users can customize the internal structure and algorithm to optimize their own hardware architecture and specific framework.
- *Ease of use* - It requires little programmer effort to port a new DNN framework to OpenDNN. The API format is similar as cuDNN, so programmers can easily modify and use with OpenDNN.

Chapter 2

Background

2.1 Deep Neural Networks

A DNN is an artifcial neural network with multiple layers between the input and output data. There are several kinds of DNNs that performs well in specific area. For example, MLP has been widely used in approximation and recommender algorithm. It is organized with input, output, and hidden layers. Figure 2.1 depicts the structure of common MLP. The circle in the layers is called as node and each circles is mapped to the data. The arrow between the layers named edge and it saves the weight how much does the input data affects the output data. Starting from input layer, the hidden layer takes the input data and sends the output data as the next hiddn layer's input data. In the process, the input data is vector structure and weight is matrix. Therefore, they run the matrix-vector multiplication and return the intermediate vector which the next layer uses. The output layer receives the input that is computed via hidden layers and gives the output that is usually the approximation value such as score or probability. MLP utilizes a supervised learning called backpropagation for training, hence requiring the non-linear activation. As the hidden layer goes deeper, the amount of weight values that stores between for each layers increases and the computation time and data storage cost is also increasd.

In early days, MLP worked well to classify MNIST dataset, a large database of handwritten digits that is commonly used for training various image processing

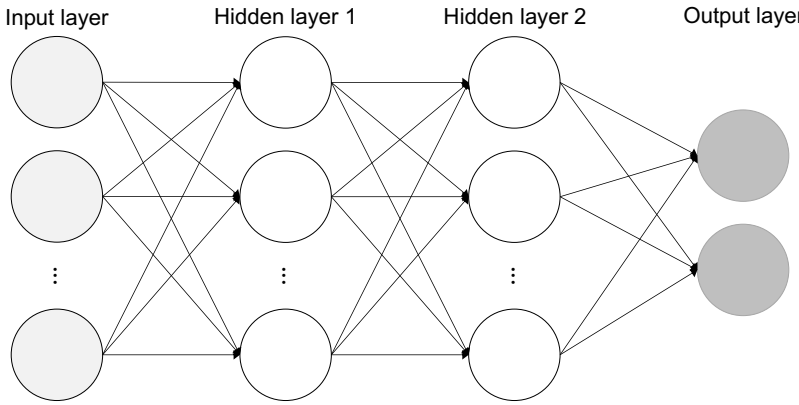


Figure 2.1: Multi-layer perceptron

systems, as it consists of simple black-and-write images of a small size. As the image size is larger and try to colored RGB images, however, MLP is difficult to train the image dataset. As researchers want to increase the accuracy, they stacked additional hidden layers in the MLP and the network size is huge. Also, MLP reconfigures an image as linear vector, which vanishes the relation of image pixels so takes low accuracy in the image datasets. To arrange with the two issues, CNNs, organized by convolution layer and pooling layer mainly, are widely used and success to get high accuracy for the CIFAR-10 [20] and imagenet [9] dataset. Convolution layer is the key factor to get high accuracy in the image dataset as it maintains the characteristics of input image by computing the small kernel filter and configuration of input and output data. Furthermore, the small kernel filter, which is called as weight, can reuses during the convolution layer computation so it reduces the size of weight tremendously. Suppose the $H_{out} \times W_{out} \times C_{out}$ output resulted from multiplication of $H_{in} \times W_{in} \times C_{in}$ input and weight. The size of weight in MLP should be $H_{in} \times H_{out} \times W_{in} \times W_{out} \times C_{in} \times C_{out}$ as it computes general matrix vector multiplication. On the other hand, CNN uses only $F \times F \times C_{in} \times C_{out}$ weight, as the weight replicates and the kernel size F is much smaller than H_{in}, H_{out}, W_{in} or W_{out}. Pooling layer reduces the feature

maps and emphasize the specific data by selecting maximum value (max pooling) or computing average in the windows (average pooling).

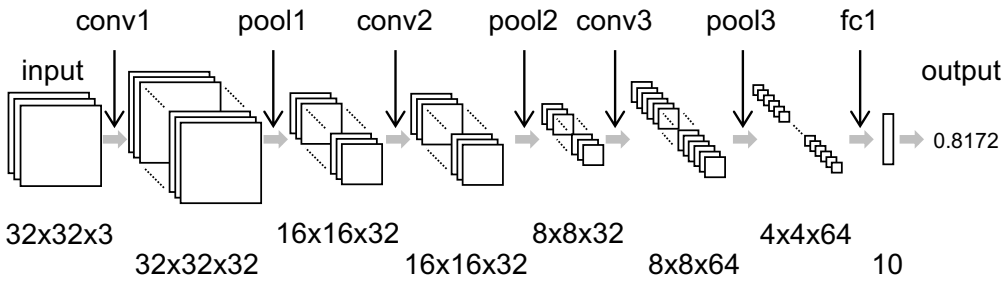


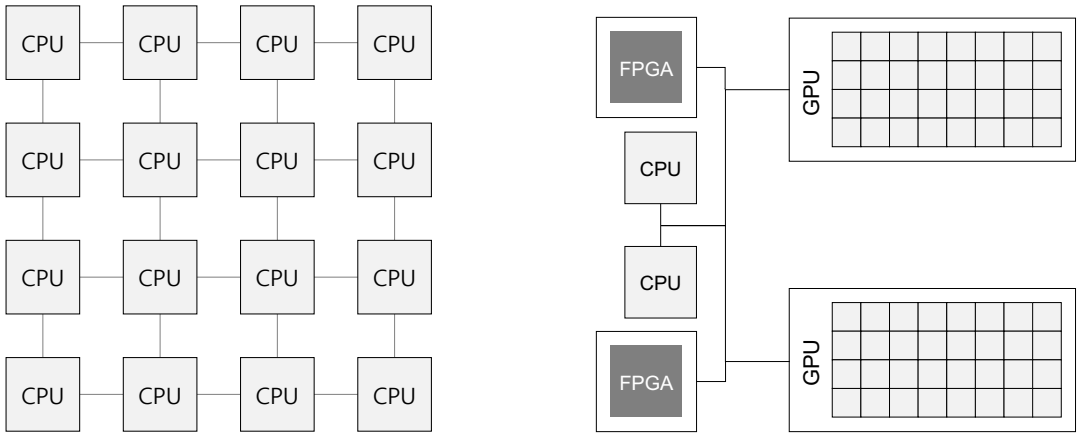
Figure 2.2: The structure of Cuda-convnet [26]

Figure 2.2 shows the structure of Cuda-convnet, which is used for image classification. It takes as input an image file of 32×32 pixels with three input channels (for RGB color components). The image passes through a pipeline of layers, composed of three alternating pairs of convolution and pooling layers, followed by a FC layer at the end. Each layer performs distinct matrix computations to extract more complex features toward the end of the pipeline. Finally, the FC layer is a classifier layer, which computes a vector of the probability for each of the 10 classes. In CNNs the convolution layers are known to be the most compute-intensive (consuming over 90% of total GOPs), while the FC layer accounts for a disproportionally large share of network parameters due to their full connectivity [21].

2.2 Heterogeneous Architectures

In PC era, the processor tried to take high performance by sophiscated and fast clock frequency. After the clock frequency was restricted because of thermal issue, researchers increased the number of cores with proper clock frequency. This concepts were used in the interconnect system or server system. Figure 2.3 (a) shows the conventional homogeneous computers that consists of one kind of processor like CPU.

In this figure, each CPU links anothers as 2-D mesh structure so CPU communicates with the nearest CPUs. For this reason, old-fashioned supercomputers used many CPU and researches about the topology of CPUs to increase the overall performance. Though homogeneous computers had good performance in the general, it spent resources inefficiently to process more than million addition and multiplication because CPU optimized running sophiscated and sequential program, not the parallel program.



(a) Homogeneous architecture (b) Heterogeneous architecture

Figure 2.3: Illustration of homogeneous and heterogeneous architectures

As the demands of deep learning increase significantly, parallel architectures are popular and the heterogeneous computers organized by more than two kinds of architectures are required. People found that the graphic cards for gaming show good performance in the simple computation parallel because they multiply and accumulate a lot of floating point numbers, sends the data to the monitor and scatter it. As the graphic cards use for the general purpose, it spends little time to compute lots of simple computation compared to use priceless and sophiscated processor unit. Figure 2.3 (b) shows the example of deep learning specific heterogeneous computers,

which consists of two CPU, two GPU, and two customizable FPGA. In deep learning working system, GPUs are exploited to process parallel computing, which ecient to executes compute-intensive deep learning like CNNs. On the other hand, FPGAs have an advantage of efficient energy usage and custom design and implementation, hence maximizing the specific DNNs' characters with low energy consumption.

Unfortunately, there is no general deep learning library to support general heterogeneous computers. NVIDIA provides CUDA and cuDNN but they target only NVIDIA GPU and proprietary library not to customizing the internal code. Similarly, Intel is implementing their own library to run the Intel Xeon Phi processor as Intel MPI [5], and AMD recently releases ROCm [6] to support deep learning accelerator based on their heterogeneous computers, which is the coupling of AMD Ryzen and Radeon. These libraries are proprietary so users can neither customize them, or run them on various DNN architectures. For example, to support an FPGA-based accelerator, the developer should implement a software stack from scratch. It is not practical for a model developer to understand the details of a target hardware and port his model to it. OpenDNN serves as an abstraction layer to support a variety of hardware architectures without requiring intrusive changes to the DNN framework. The OpenDNN API has nearly one-to-one correspondence to the industry-standard cuDNN API on which most popular DNN frameworks are based. Thus, it is easy to port an existing DNN framework to OpenDNN to support multiple architectures and/or customize algorithms.

Chapter 3

OpenDNN API

3.1 Overview

Figure 3.1 shows the overall system classified by four layers: application, library, driver, and device. The application layer offers users the comfort interface to model DNNs graph and control the hyper parameters such as learning rate and the number of step. The library layer is arbiter that communicate with hardware devices through driver and sends the computation result by getting the DNNs structure and parameters from the applications. Finally, the device layer runs the core kernel functions defined from library layers with input and weight data, and the driver layer helps to transfer the data and kernel with kernel-specific parameters such as the number of threads and the number of blocks. Our work applies to the library layers. We explain the OpenDNN library and its API in this chapter and the rest will discuss in the next chapters in detail.

Similar as other deep learning primitive libraries, OpenDNN provides the primitive functions of deep learning: convolution, pooling, activation, normalization, softmax, etc. OpenDNN is mainly implemented as OpenCL and C++ to support heterogeneous computers such as Intel Xeon Phi, NVIDIA and AMD GPUs, and FPGA-based accelerator. When the library is built, programmers can control what architecture they target so it supports all hardware with an unique source code. Besides, programmers can easily replace the cuDNN to OpenDNN since the format of OpenDNN API is

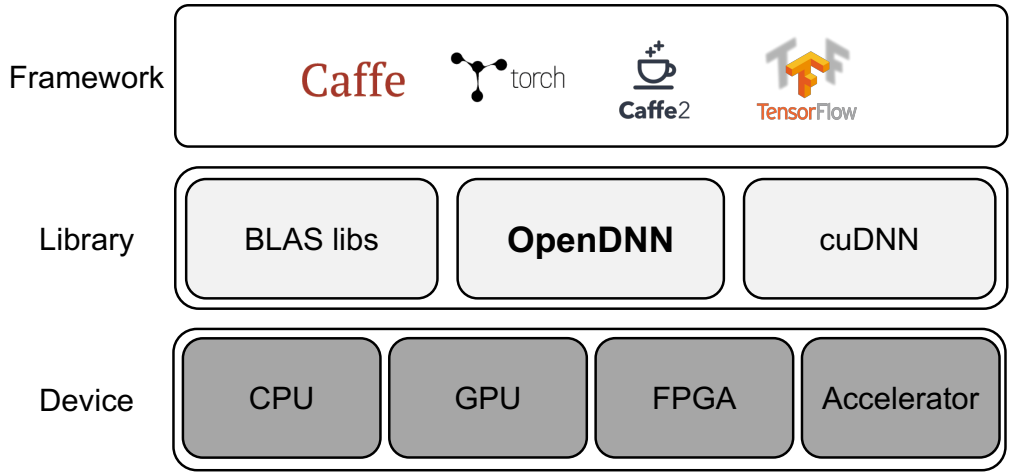


Figure 3.1: Layered structure of DNN systems

similar to cuDNN. It currently supports most of the popular APIs in cuDNN that is frequently used in other frameworks (e.g., Caffe [17] and Tensorflow [7]).

OpenDNN consists of three components for efficient and clear usage during DNN programming. As users run the deep learning framework on the heterogeneous computers usually, we provides the abstraction to manage these system. Also, OpenDNN tries to reduce the number of argument for each layers and executes lots of computation by the kernel functions. We explain the components in the following sections in detail.

3.2 Context Manager

To control and execute the heterogeneous computers, OpenDNN provides the cross-platform abstraction that is used to OpenCL. This constructs when the handler creates. Figure 3.2 shows the components of handler: platform, device, context, etc. The handler first gathers the information of hardware platforms including general CPU. When selecting the platform, handler finds the devices that match the platform to

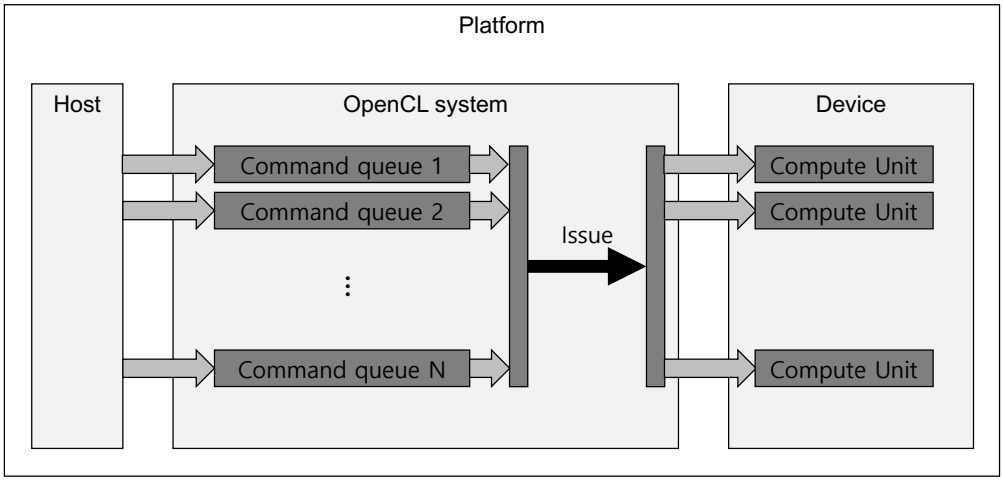


Figure 3.2: Overall structure of context manager

make the context that the environment to execute kernel. Context manages the other objects and system processes memory management and synchronization on the context level. Finally, it makes the queues and compiles the kernel objects that are sent to devices. By doing so, the context manager makes the environment to communicate with devices and reports the device or kernel errors.

3.3 Descriptor Manager

Generally, each layers of DNNs requires so many arguments that programmers are hard to set the argument, which increases the chance to make a mistake when entering it. For example, traditional convolution layer requires a lot of arguments including 4-D tensor structured data (input, weight, and output) and the characteristics of convolution layers such as stride, padding, and dilation options. If these parameters are managed as single argument, programmers will release from a laborious task.

Figures 3.3 shows the conventional and descriptor-based argument style. We implement OpenDNN's descriptor as following. The `opendnnTensorDescriptor` stores

```
1  void conv2d(
2      int i_n, int i_c, int i_h, int i_w, /* Input Arguments  */
3      int o_n, int o_c, int o_h, int o_w, /* Output Arguments */
4      int f_o, int f_i, int f_h, int f_w, /* Filter Arguments */
5      int p_h, int p_w, int s_h, int s_w, /* Convolution */
6      int d_h, int d_w,                   /* Arguments   */
7      Dtype* i, Dtype* w, Dtype* o        /* Data */
8  );
9
10 void conv2d_desc(
11     opendnnTensorDescriptor i_desc,     /* Input Descriptor  */
12     opendnnTensorDescriptor o_desc,     /* Output Descriptor */
13     opendnnFilterDescriptor f_desc,     /* Filter Descriptor */
14     opendnnConvolutionDescriptor c_desc,/* Convolution Descriptor */
15     Dtype* i, Dtype* w, Dtype* o        /* Data */
16 );
```

Figure 3.3: Convolution forward API with one-to-one argument and descriptor-based argument

the input and output 4-D data structure with its strides that is calculated during setting descriptor. The `opendnnFilterDescriptor` saves the filter structure that consists of kernel size (`f_h` and `f_w`), dimension (`f_i`) and the number of filter (`f_o`). Finally, `opendnnConvolutionDescriptor` records the padding size (`p_h` and `p_w`), kernel stride (`s_h` and `s_w`) and dilation (`d_h` and `d_w`). The descriptor is defined automatically when stacking the layer and the variables use in the computation methods getting from the descriptor.

3.4 Computation Functions

With the abstraction supported by context manager and the layer structure based on the information stored at descriptor manager, the computation kernel runs the feedforward or backpropagation. It is implemented to OpenCL and C++ kernel to run the system not just the heterogeneous system but also the homogeneous one. The name of computation API is similar as cuDNN API like `cudnnConvolutionForward` to reduce the programmer's effort porting at deep learning frameworks. Internally,

the API gets the information from the descriptor received from argument and set the computation environment. After setting, the context manager give the built kernel and dataset to the OpenCL-runtime command queue and it wait issuing to the architecture such as GPU and FPGA-based accelerators. If the computation is finished, the data is sent to the host, synchronized in the host DRAM, and layer deallocates the buffer that mapped to the device.

3.5 Summary

Table 3.1 summarise the OpenDNN API with three categories with its name and description. We skip the return type since all methods return `opendnnStatus` that checks the kernel status such as success, out of memory, and so on. OpenDNN provides the API for CNN mainly and will be implemented for other kinds of DNNs like RNN or LSTM.

API	Explanation
Context Manager Methods	
`opendnnCreate`	Create handler including the status of platform, device and context to communicate with hardware.
`opendnnDestroy`	Free the handler with platform, driver and context information.
Descriptor Manager Methods	
`opendnnCreate[`U`]Descriptor`	Create the descriptor that matches for each parts. For example, pooling descriptor saves the size of window, stride, padding, and the strategy of pooling such as maximum and average.
`opendnnSet[`U`]Descriptor`	Set the parameters to the descriptors when the layer is declared and reshaped. Each parameters receives by I/O file and the APIs parses it and set the parameters declared in the descriptors.
`opendnnGet[`U`]Descriptor`	Get the parameters from the descriptors when they run the computation methods to calcuate the output data. The arguments of APIs are pointer type so users can receive all parameters at one-time.
Computation Methods	
`opendnn[`T`]Forward`	Run the computation methods for each parts. Each methods is implemented by C++ to get the parameters from the descriptor and OpenCL to execute the kernel on the device parallel.

U = `[Convolution|Pooling|Activation|Norm|Tensor|Filter]`

T = `[Convolution|Pooling|Activation|Norm|Softmax|InnerProduct]`

Table 3.1: Summary of OpenDNN API

Chapter 4

Backend Devices

OpenDNN supports various architectures and deep learning frameworks with characteristic programming languages such as C++ and OpenCL. In this section, we introduce what code is run on the each backend hardware.

4.1 CPU

Since processor development is limited due to the power wall, developers try to increase the number of cores for parallel processing. Nowadays, Intel Xeon Phi processor is a bootable host processor that delivers massive parallelism and vectorization to support the most demanding high-performance computing applications. Recent Xeon Phi Processor has 72 cores, 36 MB L2 Cache with 1.7 GHz max turbo frequency, so it is proper to train the DNNs.

Figure 4.1 shows the Intel Xeon Phi coprocessor block diagram. description of Intel Xeon Phi. As the CPU can shows similar performance with GPU, it uses on the server system to training specific DNNs. For example, Facebook AI announces that they train the News Feed ranking and Sigma on the CPU [13]. News Feed ranking algorithms help people see the stories that matter most to them first, every time they visit Facebook. General models are trained to determine various user and environmental factors that should ultimately determine the rank order of content. Later, when a person visits Facebook, the model is used to generate a personalized set of

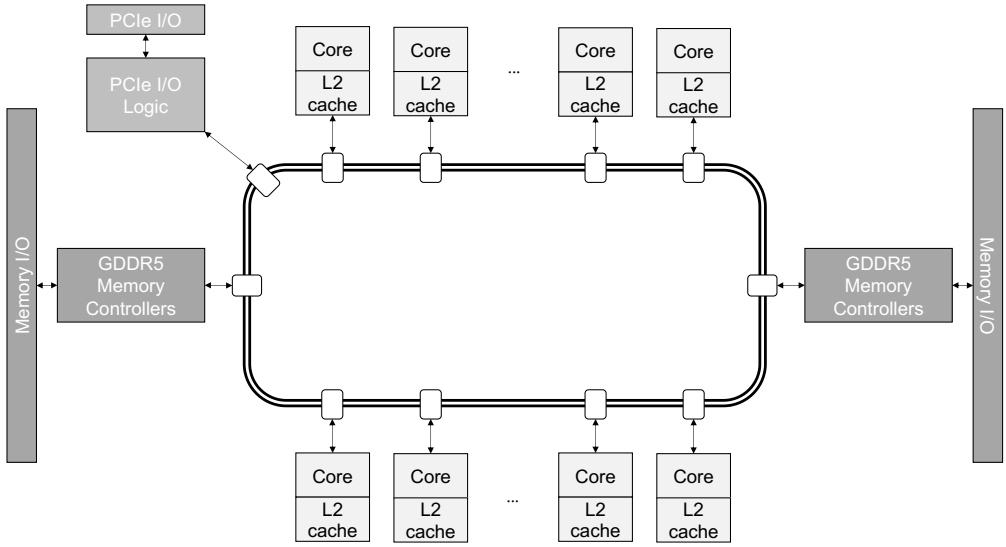


Figure 4.1: Intel Xeon Phi coprocessor block design

the best posts, images, and other. Sigma is the general classification and anomaly detection framework that is used for a variety of internal applications including site integrity, spam detection, payments, registration, unauthorized employee access, and event recommendations. Sigma includes hundreds of distinct models running in production everyday, and each model is trained to detect anomalies or more generally classify content.

```
1  // bot, top, fil, conv: descriptor
2  // The rest arguments: integer pointer
3
4  opendnnGetTensorDescriptor(bot, &in_n, &in_c, &in_h, &in_w, ...);
5  opendnnGetTensorDescriptor(top, &out_n, &out_c, &out_h, &out_w, ...);
6  opendnnGetFilterDescriptor(fil, &fil_o, &fil_i, &fil_h, &fil_w, ...);
7  opendnnGetConvDescriptor(conv, &pad_h, &pad_w, &str_h, &str_w, ...);
```

Figure 4.2: Setting the layer parameters getting from descriptors

When the computation API calls, it first sets the layer parameter from the descriptors received from the argument (depicted as Figure 4.2) and run the kernel.

```
1  // input, filter, output: data
2  for (int c=0;c<out_c;c++) {
3    for (int h=0;h<out_h;h++) {
4      for (int w=0;w<out_w;w++) {
5        float sum=0.f;
6        for (int k=0;k<in_c;k++) {
7          for (int fh=0;fh<fil_h;fh++) {
8            for (int fw=0;fw<fil_w;fw++) {
9              int ih=h*str_h-pad_h+fh;
10             int iw=w*str_w-pad_w+fw;
11             if (iw>=0&&iw<in_w&&ih>=0&&ih<in_h) {
12               sum+=input[k][ih][iw]*filter[c][k][fh][fw];
13             }}}}
14       output[c][h][w]=sum;
15 }}}
```

Figure 4.3: Sequential C convolution kernel code

Figure 4.3 describes the sequential code to run convolution kernel. Suppose that the kernel computes an image. The outer loop (Lines 1-3) indicates the output index to store the accumulated values, while the inner loop (Lines 5-7) indicates the filter and computed input index to multiply filter and input in the kernel windows (`fil_h` × `fil_w`). Programmers can customize the kernel by changing the order of loop, using vector instructions, and multi-processing using OpenMP pragma.

4.2 GPU

As deep learning requires the ability of large computation in a short time, the demands of parallel architectures like GPUs is significantly increased. NVIDIA, which is already developed programming language for their hardware system, called CUDA, has grown from gaming graphic card company to deep learning architecture company. Recently, they release Tesla V100 showing extreme performance for AI and high performance computer. Figure 4.4 shows the Volta GV100 full with 84 SM units that are 50% more energy efficient than the previous generation Pascal design, enabling major boosts in FP32 and FP64 performance in the same power envelope [25]. Also, NVLink high-

speed interconnect delivers higher bandwidth, more links, and improved scalability for multi-GPU and multi-GPU/CPU system configurations with 16 GB HBM2 memory subsystem.

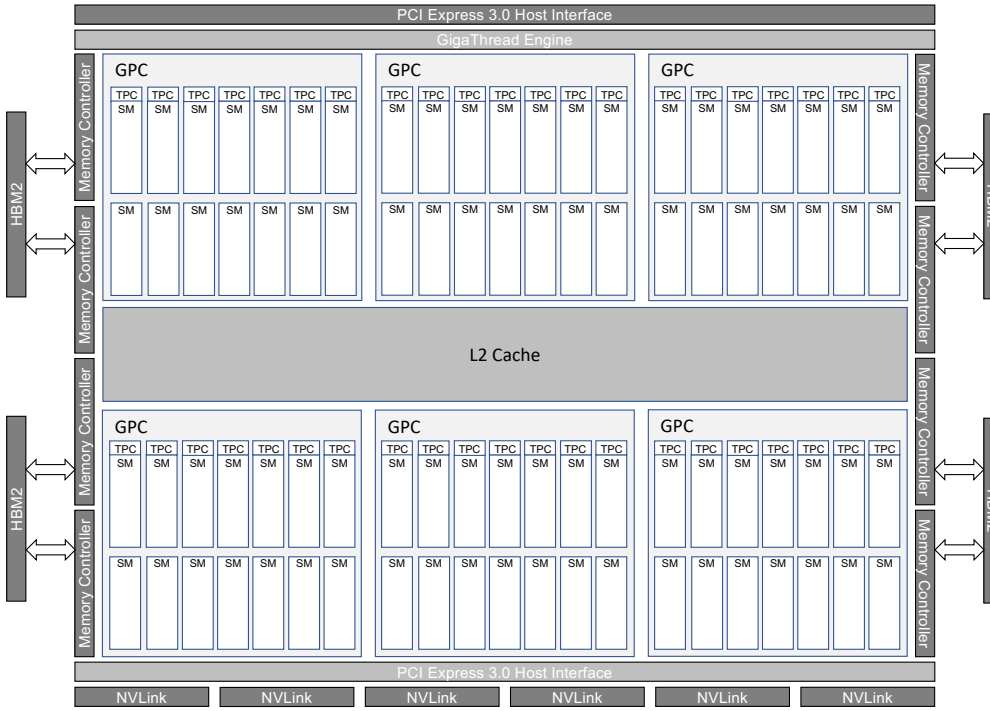


Figure 4.4: NVIDIA Volta 100 arcutectire

Likewise other layers, convolution layer shows the limitation to parallelize if users try to parallel the sequential version code. To maximize the parallelism, it gathers the maximum number of local and global workers and uses them. Especially, convolution layer uses the convolution lowering [8] that transforms the 4-D tensor to 2-D matrix. Figure 4.5 shows the convolution lowering about input, weight and output data. It is done by reshaping the filter tensor F into a matrix F_m with dimensions $K \times CRS$, and gathering a data matrix by duplicating the original input data into a matrix D_m with dimensions $CRS \times NPQ$. The computation can then be performed

with a single matrix multiply to form an output matrix O_m with dimension $K \times NPQ$. Lowering convolutions to matrix multiplication can be efficient, since matrix multiplication is highly optimized. Matrix multiplication is fast because it has a high ratio of floating-point operations per byte of data transferred. This ratio increases as the matrices get larger, meaning that matrix multiplication is less efficient on small matrices. Accordingly, this approach to convolution is most effective when it creates large matrices for multiplication.

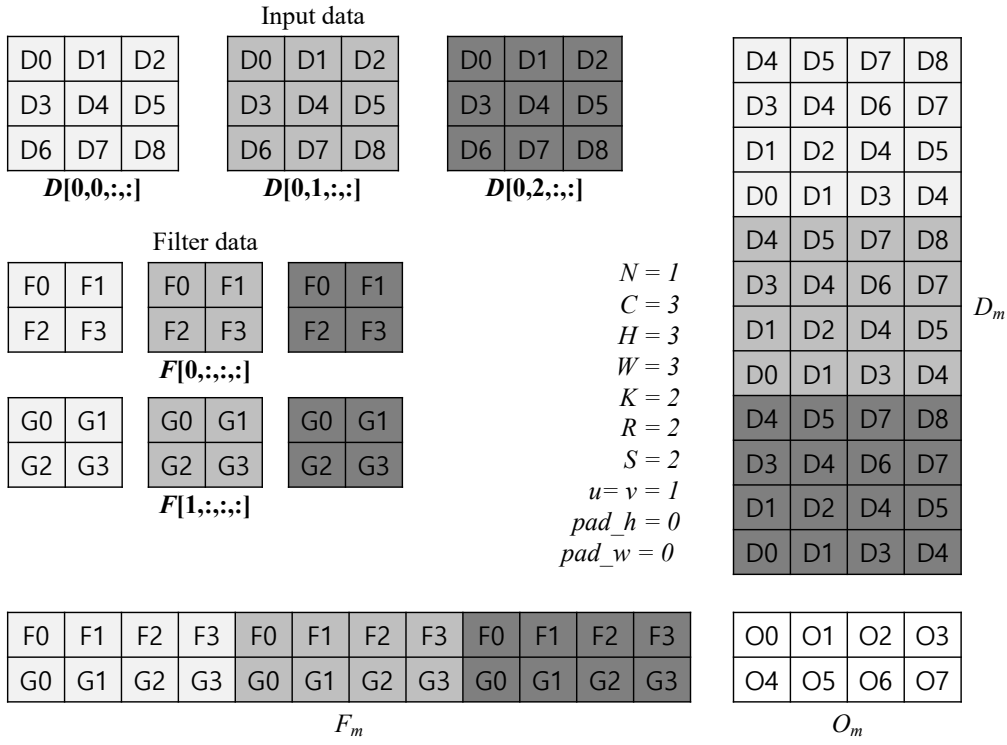


Figure 4.5: Convolution lowering mechanism

Figure 4.6 describes the code of convolution lowering after getting the layer parameters (same as CPU initialization at Figure 4.2). First, OpenDNN sets the im2col function arguments (Lines 2-6) to run the kernel that makes the 3-D and 4-D tensor to 2-D matrix (Lines 7-10). Second, it runs the matrix multiplication function

to compute the output (Lines 17-20) with set arguments based on the convolution lowering parameters (Lines 13-16). When the OpenCL kernel issues from the devices, the devices runs the built kernel code at each processing elements (PE) to maximize the parallelism. NVIDIA has adopted the Winograd algorithm since cuDNN 5 to implement the convolution layer. Winograd reduces the number of 3×3 convolution operations, thus significantly improving performance. Currently, OpenDNN does not have this feature yet to show much lower performance (some 10× slowdown over cuDNN), but can be improved in the future without breaking the software stack above.

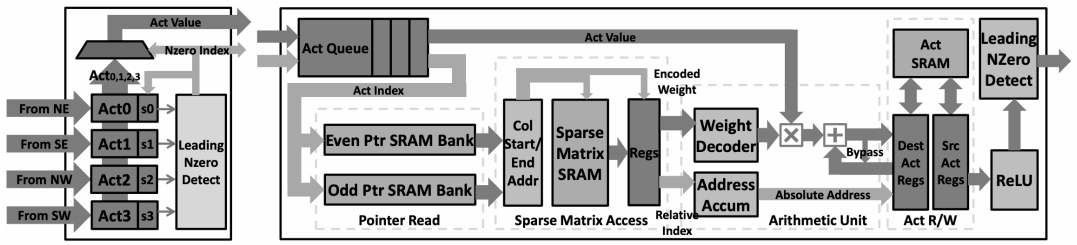
```
1  // im2col
2  int height_col=(in_h+2*pad_h-fil_h)/str_h+1;
3  int width_col=(in_w+2*pad_w-fil_w)str+w+1;
4  int num_kernels=in_c*height_col*width_col;
5  const int local=1024;  // max num.of work items per group
6  const int global=(num_kernels+local-1)/local*local;
7  q->enqueueNDRangeKernel(im2col,
8      cl::NullRange,
9      cl::NDRange(global),
10     cl::NDRange(local));
11
12 // matmul
13 const int M = out_c;
14 const int N = out_h*out_w;
15 const int K = in_c*fil_h*fil_w;
16 const int B = sqrt(local);
17 q->enqueueNDRangeKernel(matmul,
18     cl::NullRange,
19     cl::NDRange((N+B-1)/B*B,(M+B-1)/B*B,1),
20     cl::NDRange(B,B,1));
```

Figure 4.6: Host code that calls the OpenCL kernel function

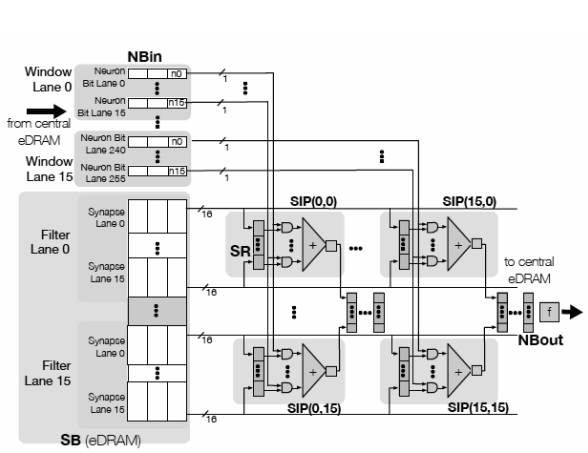
4.3 FPGA

FPGA-based accelerator provides users to customize architecture with the characteristics of DNNs, even though the performance is generally low compared to the GPU. Also, the energy consumption is significantly low that it is very efficient when

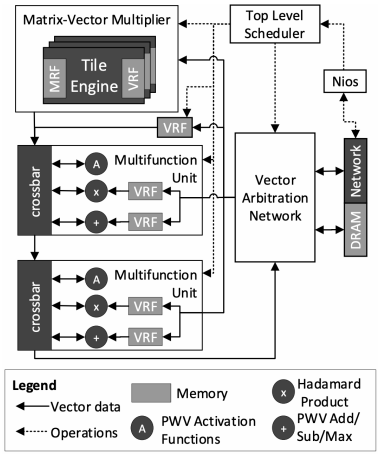
measuring performance per watt. For example, EIE [12] takes the sparsity of DNNs and skips the zero-related computation (depicted as Figure 4.7 (a)) to speed up the overall data process, which is called pruning. On the other hand, Stripes [19] shows that there is little accuracy loss but achieves high speeds with low precision data type and bit-serial multiplication PE called SIP (4.7 (b)). Microsoft Brainwave (BW) team announces that they train the overall networks using FPGA-based accelerator [10] as you see in the Figure 4.7 (c) with customizable number type called *ms-fp-8* with multi function units and vector arbitration networks.



(a) The architecture of Leading non-zero detection node and PE [12]



(b) Stripes Tile [19]



(c) BW microarchitecture [10]

Figure 4.7: Efficient FPGA-based accelerators

OpenDNN supports the FPGA-based accelerator for two ways. First is that user

customize their own architecture manually and port it on the OpenDNN. Programmers implement it using hardware-level languages such as VHDL or Verilog and link following the layer parameters. The other method is using automatically generated code that is built by OpenCL and compiler provided by vendors. For example, Xilinx supports SDAccel program that makes the bitstream running on the Xilinx FPGA and driver linking host and devices automatically. In this case, we explain the interconnected code that used to SDAccel compiler and OpenDNN API.

Figure 4.8 (a) shows the kernel code optimized by loop unrolling with the pragma option. With this option, SDAccel automatically unrolls the loop and maximize the number of PE that determines the parallelism and performance. Figure 4.8 (b) shows the host code that uses loop tiling optimization automatically that devides whole loop iteration into small size windows to match the size of cache. In this code, the offset is set (Lines 2-4) and data is allocated as a degree of offset in the FPGA memory (Lines 10, 11 and 15). While the kernel is built on run-time in GPU, it is built on compile time when using SDAccel because it should also make the drivers automatically.

```
/* The buffer is tiled */
__kernel __attribute__ ((opencl_unroll_hint(4)) void compute (
  global const float* f_buf,
  global const float* i_buf,
  global float* o_buf,
  const int str_h, const int str_w,
  const int Tr_i, const int Tc_i,
  const int Ti, const int Tr,
  const int Tc, const int To, const int K) {
  for (int fh=0;fh<K;++fh) {
    for (int fw=0;fw<K;++fw) {
      for (int h=0;h<Tr;++h) {
        for (int w=0;w<Tc;++w) {
          for (int c=0;c<To;++c) {
            for (int k=0;k<Ti;++k) {
              o_buf[c][h][w]+=
                i_buf[k*Tc_i*Tr_i+(h*str_h+fh)*Tc_i+w*str_w+fw]*
                f_buf[c][k][fh][fw];
}}}}}}}
```

(a) FPGA kernel that runs with loop unrolling

```
/* To, Ti, Tc, and Tr are customizable value */
size_t f_size = To*Ti*K*K;
size_t i_size = Ti*Tr_i*Tc_i;
size_t o_size = To*Tr*Tc;

for (int row=0;row<out_h;row+=Tr) {
  for (int col=0;col<out_w;col+=Tc) {
    for (int to=0;to<out_c;to+=To) {
      for (int ti=0;ti<in_c;ti+=Ti) {
        load_mem_input(i_buf, input, isize);
        load_mem_filger(f_buf, filter, fsize);
        ...
        q->enqueueTask(kenl_conv);
        ...
        store_mem_output(output, o_buf, osize);
}}}}
```

(b) FPGA host code calling kernel with loop tiling

Figure 4.8: FPGA convolution layer code

Chapter 5

OpenDNN-enabled DNN Frameworks

5.1 Caffe

Caffe [17] is a deep learning framework made with expression, speed, and modularity in mind. It is developed by Berkeley AI Research (BAIR) and by community contributors. As a C++-based deep learning framework, the speed is faster than other framework so it was widely used in research experiments. Also, the extensible code forsters active development such as Intel Caffe, which is optimized to run on their Xeon Phi, and Caffe-Ristretto [11] that emulates hardware-oriented approximation of quantized DNN. We port OpenDNN in the Caffe easily with a few code modification.

```
1  # In Makefile.config
2  OPENDNN := 1
```

(a) Caffe `Makefile.config`

```
1  # In Makefile
2  ifeq ($(OPENDNN), 1)
3    LIBRARIES += opendnn OpenCL
4    COMMON_FLAGS += -DOPENDNN
5  endif
```

(b) Caffe `Makefile`

Figure 5.1: Addition code at Caffe `Makefile` and its configuration

To link the OpenDNN on the Caffe, programmers should add the Figure 5.1 (a) code in the `Makefile.config` that uses `Makefile`. Also, they add the Figure 5.1 (b) code in the `Makefile` to link the OpenDNN and OpenCL library and set the `OPENDNN` flag that is used in the Caffe layer functions such as convolution layer, pooling layers,

and so on.

```
1  /* API added on conventional layer methods */
2  ...
3  #include <opendnn.h>
4  ...
5  template <typenape Dtype>
6  class CuDNNConvolutionLayer : public ConvolutionLayer<Dtype> {
7    ...
8    opendnnHandle_t* handle_;
9    vector<opendnnTensorDescriptor_t> bottom_descs_, top_descs_;
10   opendnnFilterDescriptor_t filter_desc;
11   vector<opendnnConvolutionDescriptor_t> conv_descs_;
12   ...
13 }
```

Figure 5.2: Header file modification to run OpenDNN in Caffe

In this example, we explain the OpenDNN-based convolution layer function that is portion of 80 $\sim$ 90% of CNNs. Figure 5.2 shows the additional code to declare and define the OpenDNN API. Users do not have to use `#ifdef` macro if you use only OpenDNN API; In other words, you can use both OpenDNN and other library like cuDNN.

Figure 5.3 shows the difference between original code and OpenDNN-ported one (the grey box). As you see, the OpenDNN try to maintain the cuDNN format for easy programming and portability. There is some difference on the API. First, we remove the `one` and `zero` arguments since they are almost constant values. Second, OpenDNN runs the group loop in the API that is exposed in the cuDNN API (Line 7 in Figure 5.3). In this code, programmers can use OpenDNN API with the effort of removing loop iteration. This method is same as other layers such as pooling, activation, and so on.

```
1  void CuDNNConvolutionLayer::Forward_gpu(
2    vector<Blob<Dtype>* >& bottom, vector<Blob<Dtype>* >& top) {
3    Dtype* weight = this->blobs_[0]->gpu_data();
4    for (int i = 0; i < bottom.size(); ++i) {
5      Dtype* bottom_data = bottom[i]->gpu_data();
6      Dtype* top_data = top[i]->mutable_gpu_data();
7      for (int g = 0; g < this->group_; g++) {
8        cudnnConvolutionForward(handle_[g],
9          cudnn::dataType<Dtype>::one,
10         bottom_descs_[i], bottom_data + bottom_offset_ * g,
11         filter_desc_, weight + weight_offset_ * g,
12         conv_descs_[i],
13         fwd_algo_[i], workspace[g], workspace_fwd_sizes_[i],
14         cudnn::dataType<Dtype>::zero,
15         top_descs_[i], top_data + top_offset_ * g);
16     }
17     // Skip
18 }}}
```

(a) Original Caffe code

```
1  void CuDNNConvolutionLayer::Forward_gpu(
2    vector<Blob<Dtype>* >& bottom, vector<Blob<Dtype>* >& top) {
3    Dtype* weight = this->blobs_[0]->gpu_data();
4    for (int i = 0; i < bottom.size(); ++i) {
5      Dtype* bottom_data = bottom[i]->gpu_data();
6      Dtype* top_data = top[i]->mutable_gpu_data();
7        opendnnConvolutionForward(handle_,
8          bottom_descs_[i], bottom_data,
9          filter_desc_, weight,
10         conv_descs_[i], workspace,
11         top_descs_[i], top_data);
12     }
13     // Skip
14 }}}
```

(b) Modified Caffe code

Figure 5.3: The difference between original and OpenDNN code.

5.2 TensorFlow

TensorFlow (TF) [7] is an open-source software library for high performance numerical computation. Its flexible architecture allows easy deployment of computation across a variety of platforms (CPUs, GPUs, TPUs), and from desktops to clusters of servers to mobile and edge devices. Also, TF includes powerful linear algebra compiler called XLA which helps TF code on the embedded processor and other hardware platforms as fast as possible. In contrast to Caffe, the interface language is python so users can start the framework easily. However, the core functions are designed by C++ and CUDA and the python package is built by google internal build program called bazel.

Bazel is the open-source tool that build and test software of any size, quickly and reliably. It only rebuilds what is necessary and supports for a lot of languages such as Java, C++, Android and runs on Windows, macOS, and Linux. As the TF source code is large, it was difficult to find the proper configuraion setting on the overall code but we success to build it. We explain how to set the build configuration for the Bazel. The gray box of Figure 5.4 (a) shows the BUILD file modification and the one of Figure 5.4 (b) shows the bazel configuration file that is similar as python format. With the compiled *libopendnn*, bazel starts to build the OpenDNN-ported TF and spends a long time to build it.

After setting the build configuration, we change the TF core function to connect OpenDNN. Figure 5.5 shows the macro code that wrapping the OpenDNN on the TF API. To enter this code, TF can show the encapsulated code to use only in TF, hence guarantee of tight security. This is common procedure to port the customized library on the TF.

Figure 5.6 describes the ported code of convolution layer in TF that uses OpenDNN API. First, it creates the handler (Lines 4-5) and set the input, output, filter, and convolution descriptors (Lines 6-34). Next, it runs the computation function (Lines

36-40) that defined on the OpenDNN API. In this code, you can see the `wrap` scope that is declared by the Figure 5.5.

```
# tensorflow/stream_executor/BUILD
# at Line 77
cc_library(
    "//tensorflow/core:cuda",
    ...
    "@local_config_cuda//cuda:opendnn",
    "@local_config_cuda//cuda:cudnn",
    ...
)

# third_party/gpus/cuda/BUILD.tpl
cc_libary(
    name="opendnn",
    srcs=["cuda/lib/%{opendnn_lib}"],
    data=["cuda/lib/%{opendnn_lib}"],
    includes=[
        ".", "cuda/include",
    ],
    linkstatic=1,
    visibility=["//visibility:public"],
)
```

(a) BUILD file modification

```
# third_party/gpus/cuda_configure.bzl
# at line 588
def _find_libs(repository_ctx, cuda_config):
    "opendnn": _find_cuda_lib(
        "opendnn", repository_ctx, cpu_value,
        cuda_config.cudnn_install_basedir, cuda_config.cudnn_version),
    ...

# at line 963
def _create_local_cuda_repository(repository_ctx):
    "%{opendnn_lib}": cuda_libs["opendnn"].file_name,
    ...
```

(b) Bazel configuration file modification

Figure 5.4: Parameters to build OpenDNN-ported TensorFlow

```
/* tensorflow/stream_executor/cuda/cuda_dnn.cc */
...
#include "cuda/include/opendnn.h"
...
#define OPENDNN_DNN_ROUTINE_EACH_ARC(__macro)                    \
  __macro(opendnnSetStream)     \
  __macro(opendnnCreateTensorDescriptor)     \
  __macro(opendnnSetTensor4dDescriptor)     \
  __macro(opendnnCreateFilterDescriptor)     \
  __macro(opendnnSetFilter4dDescriptor)     \
  __macro(opendnnCreateConvolutionDescriptor)     \
  __macro(opendnnSetConvolution2dDescriptor)     \
  __macro(opendnnGetConvolutionForwardWorkspaceSize)     \
  __macro(opendnnConvolutionForward)
OPENDNN_DNN_ROUTINE_EACH_ARC(PERFTOOLS_GPUTOOLS_OPENDNN_WRAP_ARC)
...
```

Figure 5.5: Macro setting to build OpenDNN with bazel

```
1  // Define the openDNN parameter and API
2  bool CudnnSupport::DoConvolveImpl(...) {
3    ...
4    opendnnHandle_t handle;
5    opendnnCreate(&handle);
6    // Set input parameters
7    std::vector<int64> input_dims =
8      batch_descriptor.full_dims(dnn::DataLayout::kBatchDepthYX);
9    opendnnTensorDescriptor_t input_desc;
10   wrap::opendnnCreateTensorDescriptor(parent_, &input_desc);
11   wrap::opendnnSetTensor4dDescriptor(parent_, input_desc,
12     input_dims[0], input_dims[1], input_dims[2], input_dims[3]);
13   // Set output parameters
14   std::vector<int64> output_dims =
15     output_descriptor.full_dims(dnn::DataLayout::kBatchDepthYX);
16   opendnnTensorDescriptor_t output_desc;
17   wrap::opendnnCreateTensorDescriptor(parent_, &output_desc);
18   wrap::opendnnSetTensor4dDescriptor(parent_, output_desc,
19     output_dims[0], output_dims[1], output_dims[2], output_dims[3]);
20   // Set filter parameters
21   opendnnFilterDescriptor_t filter_desc;
22   const auto& spatial_dims = filter_descriptor.input_filter_dims();
23   wrap::opendnnCreateFilterDescriptor(parent_, &filter_desc);
24   wrap::opendnnSetFilter4dDescriptor(parent_, filter_desc,
25     filter_descriptor.output_feature_map_count(),
26     filter_descriptor.input_feature_map_count(),
27     spatial_dims[0], spatial_dims[1]);
28   // Set convolution parameters
29   opendnnConvolutionDescriptor_t conv_desc;
30   const auto& strides = convolution_descriptor.strides();
31   const auto& padding = convolution_descriptor.padding();
32   wrap::opendnnCreateConvolutionDescriptor(parent_, &conv_desc);
33   wrap::opendnnSetConvolution2dDescriptor(parent_, conv_desc,
34     padding[0], padding[1], strides[0], strides[1], 1, 1);
35   ...
36   wrap::opendnnConvolutionForward(parent_, handle,
37   input_desc, (float*)input_data.opaque(),
38   filter_desc, (float*)filter_data.opaque(), conv_desc,
39   (float*)scratch.opaque(), scratch.size(),
40   output_desc, (float*)output_data->opaque());
41   ...
42 }
```

Figure 5.6: Source code modification to run OpenDNN in TensorFlow

5.3 DarkNet

DarkNet [29] is an open-source neural network framework written in C and CUDA. It is fast, easy to install, and supports CPU and GPU computation. Based on this projects, many works are designed in the computer vision and deep learning area. For example, YOLO [28] project that is already widely used in image detection is improved recently. Also, XNOR-Net [27], reducing lots of multiplication by using a few XNOR logical computation, is DarkNet-based project. As DarkNet is based on C language, we add `extern C` code in the OpenDNN to interconvert it.

```
# In Makefile
OPENDNN=1
ifeq ($(OPENDNN), 1)
  COMMON+= -DOPENDNN
  CFLAGS+= -DOPENDNN
endif
```

(a) DarkNet `Makefile`

```
/* convolutional_layer.h */
void opendnn_convolutional_setup(layer *l);

/* darknet.h */
#ifdef GPU
  ...
  #include "opendnn.h"
  ...
#endif
...
opendnnHandle_t opendnn_handle;
opendnnTensorDescriptor_t srcTensorDesc;
opendnnTensorDescriptor_t dsrcTensorDesc;
opendnnTensorDescriptor_t dstTensorDesc;
opendnnTensorDescriptor_t ddstTensorDesc;
opendnnTensorDescriptor_t normTensorDesc;
opendnnFilterDescriptor_t weightDesc;
opendnnFilterDescriptor_t dweightDesc;
opendnnConvolutionDescriptor_t convDesc;
...
```

(b) DarkNet framework header file

Figure 5.7: `Makefile` and header file modification to port OpenDNN in DarkNet

In DarkNet `Makefile` we add some code as Figure 5.7 (a) to set the flags. Also, we declare the OpenDNN API on the `darknet.h` as you see in the Figure 5.7 (b). The variables starting as `d` mean the differentiation and they are used to back propagation.

Figure 5.9 describes the initialization of convolution layers using OpenDNN API.

```
1  /* convolutional_kernels.cu */
2  void forward_convolutional_layer_gpu(
3    convolutional_layer l, network net) {
4    ...
5    opendnnConvolutionForward(l.opendnn_handle,
6      l.srcTensorDesc,
7      net.input_gpu,
8      l.weightDesc,
9      l.weights_gpu,
10     l.convDesc,
11     l.dstTensorDesc,
12     l.output_gpu,
13     net.index);
14   ...
15 }
```

Figure 5.8: Convolution forward running OpenDNN-ported DarkNet

The handler is initialized at GPU-related code (Lines 2-9) and other elements such as tensors, filter, and convolution are initialized on the convolution layer code (Lines 16-24). After creating and initializing, the handler and elements are set from the convolution setup functions (Line 25) that is defined on the OpenDNN API (Lines 29-46). This process is same as constructor and reshaping. DarkNet uses this defined convolution layer to execute the convolution forward function as you see in the Figure 5.8 that runs on the GPU.

```
/* cuda.c */
opendnnHandle_t opendnn_handle() {
  static int init = 0;
  static opendnnHandle_t handle;
  if(!init) {
      opendnnCreate(&handle);
      init = 1;
  }
  return handle;
}

/* convolutional_layer.c */
convolutional_layer make_convolutional_layer (...) {
  convolutional_layer l;
  ...
  l.opendnn_handle = opendnn_handle();
  opendnnCreateTensorDescriptor(&l.normTensorDesc);
  opendnnCreateTensorDescriptor(&l.srcTensorDesc);
  opendnnCreateTensorDescriptor(&l.dstTensorDesc);
  opendnnCreateFilterDescriptor(&l.weightDesc);
  opendnnCreateTensorDescriptor(&l.dsrcTensorDesc);
  opendnnCreateTensorDescriptor(&l.ddstTensorDesc);
  opendnnCreateFilterDescriptor(&l.dweightDesc);
  opendnnCreateConvolutionDescriptor(&l.convDesc);
  opendnn_convolutional_setup(&l);
  ...
}
void opendnn_convolutional_setup(layer *l) {
  opendnnSetTensor4dDescriptor(l->dsrcTensorDesc,
    l->batch, l->c, l->h, l->w);
  opendnnSetTensor4dDescriptor(l->ddstTensorDesc,
    l->batch, l->out_c, l->out_h, l->out_w);
  opendnnSetTensor4dDescriptor(l->srcTensorDesc,
    l->batch, l->c, l->h, l->w);
  opendnnSetTensor4dDescriptor(l->dstTensorDesc,
    l->batch, l->out_c, l->out_h, l->out_w);
  opendnnSetTensor4dDescriptor(l->normTensorDesc, 1, l->out_c, 1, 1);
  opendnnSetFilter4dDescriptor(l->dweightDesc,
    l->n, l->c/l->groups, l->size, l->size);
  opendnnSetFilter4dDescriptor(l->weightDesc,
    l->n, l->c/l->groups, l->size, l->size);
  opendnnSetConvolution2dDescriptor(l->convDesc,
    l->pad, l->pad, l->stride, l->stride, 1, 1);
  opendnnSetConvolutionGroupCount(l->convDesc, l->groups);
}
```

Figure 5.9: OpenDNN API usage in DarkNet convolution layer

Chapter 6

Evaluation

6.1 Programmable Effort

Framework	Modification of Existing Code				LoC for Conv.Layers	LoC for Framework
	Added	Modified	Deleted	Total		
Caffe	26	4	1	31	380	63,733
TensorFlow	64	22	196	296	4454	1,889,608
DarkNet	43	4	1	48	380	25,144

Table 6.1: Modified code lines for each framework

We try to reduce the code lines in original framework as modifying many lines requires programmer's efforts. Table 6.1 shows the count of lines to run the OpenDNN's convolution API for each framework. Caffe and DarkNet require 31 and 48 code revision at the convolution layer code, while TF shows much code modification as 296. For the percentage of code, however, OpenDNN provides the environment that modify a few code for all frameworks. Especially, the modified TF code lines is 296, which is 6.65% of convolution layer code lines and only 0.016% of total framework that is approximately 1.9 million lines.

Table 6.2 shows the library coverage of OpenDNN and cuDNN. OpenDNN supports six descriptors, which is except for RNN and dropout descriptor from cuDNN. Since we does not support these descriptor and forward/backward specific algorithm,

Library	Num.of Descriptor	Num.of Datatype	Num.of API
cuDNN	8	36	124
OpenDNN	6	10	28

Table 6.2: API Coverage comparison OpenDNN with cuDNN

the number of data type is 10, but cuDNN supports 36 data type including algorithm settings such as Winograd and FFT. Also, the number of OpenDNN API is 28, but the number of cuDNN API is 124. Because above 80 percent of DNNs are CNNs, OpenDNN is concentrated to implement CNN-specific API. MLP, another kinds of DNNs, does not require complex API as it is calcuated by inner product function. For this reason, current OpenDNN does not support RNN-based DNNs. Besides, the three frameworks use cuDNN library only when they run the CNNs, hence no needs to implement the rest APIs. We will implement other APIs for latest DNNs such as MANN and LSTM in the future.

6.2 Performance

We evaluate the OpenDNN performance with different architectures (CPU, GPU, FPGA). Since convolution layer spends dominant computation time in CNNs, we evaluate only convolution layer. The CPU runs the sequential code, GPU does the parallel code implemented by convolution lowering, and FPGA does the loop tiling code with accelerator compiled by OpenCL kernel. Table 6.3 describes the archtectures environment that OpenDNN is measured on.

Before evaluating the performance, we profile the GPU kernel and found that the `im2col` is 10 times slower than the one designed by NVIDIA CUDA because of the OpenCL address space issues. We internally tested to use CUDA `im2col` and OpenCL `matmul` function as usual, and the performance is much higher than previous

Hardware	Environment	Note
CPU	Intel ®Core^{TM}i7-7700K CPU @ 4.20GHz	32GB DDR4, use single-core
GPU	NVIDIA Titan Xp @ 1405MHz	12GB GDDR5, 3840 core
FPGA	Xilinx Kintex UltraScale FPGA @ 250MHz	16GB on-board DDR4

Table 6.3: Experiment environment

one. We test spended time to test the overall testset of three popular DNNs (LeNet, Cuda-convnet, AlexNet). As you see in the Figure 6.1, GPU shows best performance in the three architectures with more than 10x faster than CPU, while FPGA has low performance even though it is designed to process data parallel. In FPGA, its bitstream is built using the general-purpose kernel that runs GPU. Since the kernel is too small to maximize FPGA utilization, SDAccel makes the bitstream, which includes a tiling PE, small buffer to store the tiled data, and low bandwidth utility to transport data. The performance would be increased if we implement specific kernel that targets only FPGA with complex pragma options.

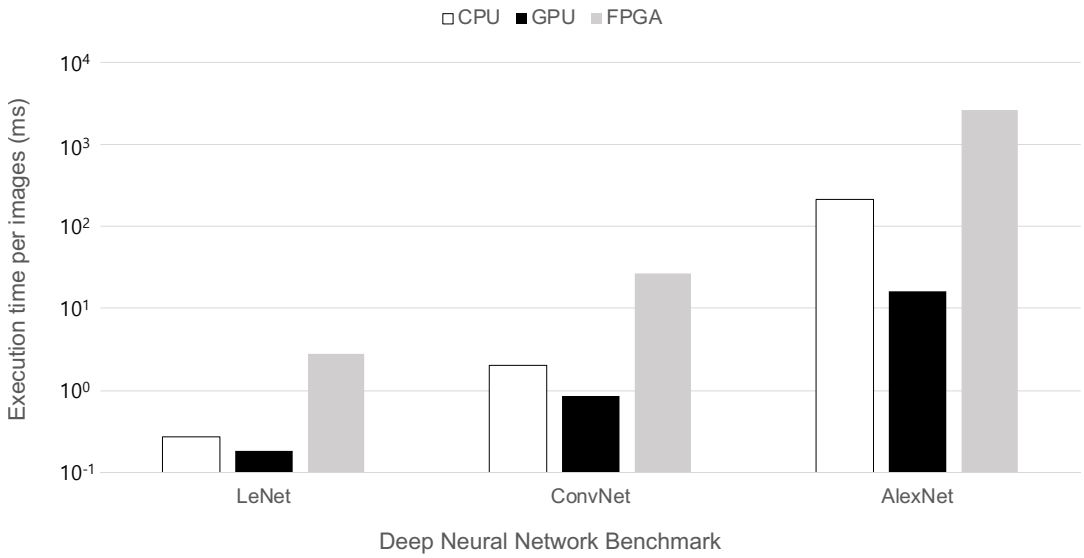


Figure 6.1: Performance of CPU, GPU, and FPGA with different DNNs

Figure 6.2 shows the time portion to run the DNNs as a single batch. Note that CPU and GPU does not require additional command to write opcode to compute input-weight vector multiplication, while FPGA needs it to do. Generally, the time to write and read the data on the CPU is similar to the time on GPU. The big difference is that execution time is reduced more than 10 times on GPU. As the time is decreased, the ratio is changed, but the absolute time is static. In FPGA, 46% of time is used to run the hardware, and the rest time is used to read and write the data. As the hardware executes a single PE the performance could be improved if parallel PEs are implemented on the FPGA. Also, about the half of total time is used to transfer weight, input, and output data. OpenDNN allocates the memory and moves the data from the host to device. This is inefficient compared to the general framework that allocates the overall workspace at one-shot. Therefore, the performance would be increased if we add additional API to allocate the data memory at one-shot.

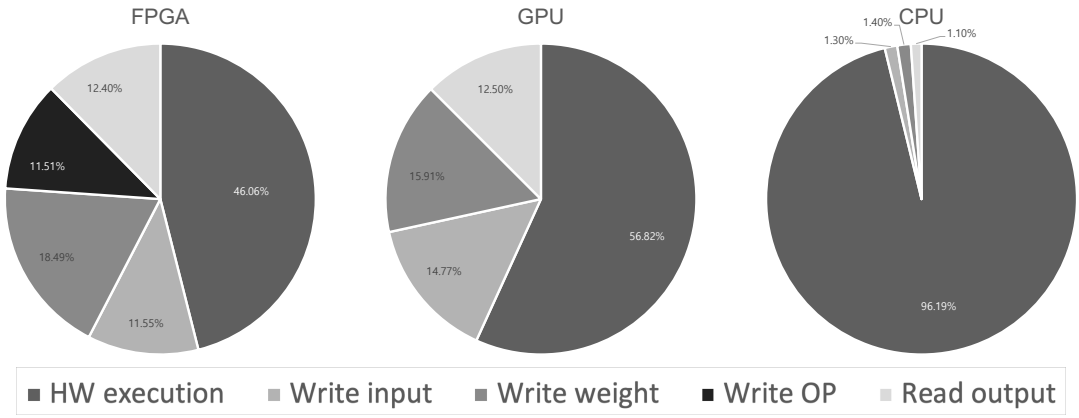


Figure 6.2: Time portion to run one batch on CPU, GPU, and FPGA

Chapter 7

Related Work

Nowadays, a lot of open-source projects for DNNs are released and have implemented by vendors to selling their architecture and server system. NVIDIA CUTLASS [2] is a set of CUDA C++ template abstractions to implement high performance matrix multiplication at all levels and scales within CUDA. It includes concepts for hierarchical decomposition and data transformation similar to those used to implement cuBLAS. The difference is that it decomposes these moving parts into reusable components abstracted by C++ template classes and can be specialized and tuned by tiling sizes, data types, and other algorithmic policy. AMD MIOpen [3] targets for CNN acceleration built to run on top of the ROCm software stack. It suuports for OpenCL and HIP enabled frameworks. With ROCm, it optimized convolutions including Winograd,FFT transformations, and GEMM. Intel clDNN [1] is an open-source performance library for deep learning applications intended for acceleration of deep learning inference on Intel HD Graphics Driver and Intel Iris graphics. clDNN includes highly optimized building blocks to implement CNNs with C and C++ interfaces. Intel MKL-DNN [4] is capable of programming from general Intel CPU to their manycore architecture such as Intel Atom and Xeon Phi series and programmed as OpenCL. While these open-source projects target their own devices, OpenDNN supports all these devices without hardware discriminations, needless to say the superior compatibility of deep learning framework.

Chapter 8

Conclusion

This thesis introduces OpenDNN, an open-source library with cuDNN-like API supporting multiple hardware devices. cuDNN offers high performance for DNNs, but it is not open-source and targets NVIDIA GPUs only. By using OpenDNN, users can easily develop and accelerate DNNs on the CPUs, GPUs and FPGAs. Furthermore, DNN experts can customize it as they wish, to improve performance. OpenDNN is successfully ported to popular deep learning frameworks on different architectures. Even though the performance is not good compared to hardware-specific libraries, we have an opportunity to improve the performance with different optimization methodology. We leave such optimization as future work, including Winograd convolution and n-bit quantization.

Bibliography

[1] Compute library for deep neural networks (cldnn). https://github.com/intel/clDNN, 2018.

[2] Cuda templates for linear algebra subroutines. https://github.com/NVIDIA/cutlass, 2018.

[3] Amd's library for high performance machine learning primitives. https://github.com/ROCmSoftwarePlatform/MIOpen, 2018.

[4] Intel(r) math kernel library for deep neural networks (intel(r) mkl-dnn). https://github.com/intel/mkl-dnn, 2018.

[5] Deliver flexible, efficient, and scalable cluster messaging on intel architecture. https://software.intel.com/en-us/mpi-library, 2018.

[6] Rocm, a new era in open gpu computing. https://rocm.github.io/, 2018.

[7] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: a system for large-scale machine learning. In *OSDI*, 2016.

[8] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cudnn: Efficient primitives for deep learning, 2014.

[9] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. ImageNet: A Large-Scale Hierarchical Image Database. In *CVPR09*, 2009.

[10] J. Fowers, K. Ovtcharov, M. Papamichael, T. Massengill, M. Liu, D. Lo, S. Alkalay, M. Haselman, L. Adams, M. Ghandi, et al. A configurable cloud-scale dnn processor for real-time ai. In *Proceedings of the 45th Annual International Symposium on Computer Architecture*, 2018.

[11] Philipp Gysel. Ristretto: Hardware-oriented approximation of convolutional neural networks, 2016.

[12] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A. Horowitz, and William J. Dally. Eie: Efficient inference engine on compressed deep neural network. In *Proceedings of the 43rd International Symposium on Computer Architecture*, 2016.

[13] K. Hazelwood, S. Bird, D. Brooks, S. Chintala, U. Diril, D. Dzhulgakov, M. Fawzy, B. Jia, Y. Jia, A. Kalro, J. Law, K. Lee, et al. Applied machine learning at facebook: A datacenter infrastructure perspective. In *2018 IEEE International Symposium on High Performance Computer Architecture*, 2018.

[14] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition, 2015.

[15] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 1997.

[16] Gao Huang, Zhuang Liu, Laurens van der Maaten, and Kilian Q. Weinberger. Densely connected convolutional networks. In *The IEEE Conference on Computer Vision and Pattern Recognition*, CVPR'17, 2017.

[17] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22Nd ACM International Conference on Multimedia*, MM '14, 2014.

[18] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. In-datacenter performance analysis of a tensor processing unit. In *Computer Architecture (ISCA), 2017 ACM/IEEE 44th Annual International Symposium on*, 2017.

[19] P. Judd, J. Albericio, T. Hetherington, T. M. Aamodt, and A. Moshovos. Stripes: Bit-serial deep neural network computing. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture*, 2016.

[20] Alex Krizhevsky. Learning multiple layers of features from tiny images, 2009.

[21] Alex Krizhevsky. cuda-convnet: High-performance c++/cuda implementation of convolutional neural networks. `https://code.google.com/p/cuda-convnet/`, 2012.

[22] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems 25*. Curran Associates, Inc., 2012.

[23] Wei Liu and Dragomir Anguelov et al. Ssd: Single shot multibox detector. In *Computer Vision - ECCV 2016 - 14th European Conference*, 2016.

[24] Tomáš Mikolov, Martin Karafiát, Lukáš Burget, Jan Černockỳ, and Sanjeev Khudanpur. Recurrent neural network based language model. In *Eleventh Annual Conference of the International Speech Communication Association*, 2010.

[25] Nvidia^{TM}. Nvidia tesla v100 tensor core gpu. `http://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf`, 2017.

[26] Young H. Oh, Quan Quan, Daeyeon Kim, Seonghak Kim, Jun Heo, Sungjun Jung, Jaeyoung Jang, and Jae W. Lee. A portable, automatic data qantizer for deep neural networks. In *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques*, 2018.

[27] Mohammad Rastegari and others. Xnor-net: Imagenet classification using binary convolutional neural networks. *CoRR*, 2016.

[28] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi. You only look once: Unified, real-time object detection. In *2016 IEEE Conference on Computer Vision and Pattern Recognition*, CVPR'17, 2016.

[29] Joseph Redmon. Darknet: Open source neural networks in c. `http://pjreddie.com/darknet/`, 2013–2016.

[30] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition, 2014.

[31] Sainbayar Sukhbaatar, arthur szlam, Jason Weston, and Rob Fergus. End-to-end memory networks. In *Advances in neural information processing systems*. 2015.

[32] Shijin Zhang, Zidong Du, Lei Zhang, Huiying Lan, Shaoli Liu, Ling Li, Qi Guo, Tianshi Chen, and Yunji Chen. Cambricon-x: An accelerator for sparse neural networks. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*, 2016.

국문초록

심층 신경망은 오늘날의 지능형 어플리케이션과 서비스의 핵심 요소로 각광받고 있다. NVIDIA에서 개발한 cuDNN은 딥 러닝 프리미티브 라이브러리의 표준으로, 정교한 심층 신경망 모델을 쉽게 개발하도록 돕는다. 그러나, cuDNN은 NVIDIA의 특허 소프트웨어로 유저들이 자신들의 요구에 맞게 제작하는 것을 허용하지 않는다. 게다가 NVIDIA GPU만을 지원하기 때문에 멀티코어 CPU나 타 FPGA를 지원하지 않는다. 이 논문에서는 다양한 하드웨어 장치를 유연하게 지원하고, cuDNN과 유사한 인터페이스를 가진 딥 러닝 프리미티브 라이브러리인 OpenDNN을 소개한다. 특히, 다양한 심층 신경망 프레임워크와 CPU, GPU, 그리고 FPGA와 같은 하드웨어 장치들에 연동하여 OpenDNN의 이식성과 유연성을 입증한다.

주요어: 딥 러닝, 라이브러리, 오픈소스, 가속기, 성능, 이식성
학번: 2017-23840

Acknowledgements

길게만 느껴지던 2년의 대학원 석사과정이 어느덧 끝이 났습니다. 2년의 시간동안 많은 사람들의 도움이 있었기에 지금 이 학위논문을 쓰고 있는게 아닐까 싶습니다.

우선 대학원 과정 기간 중에서 가장 많은 도움을 주신 제 지도교수님인 이재욱 교수님께 감사를 표합니다. 저에게 적합한 연구 주제와 과제를 배정해 주신 것은 물론, 여러 사람들과 함께 아이디어를 논의해야 하는 팀 워크에 대해서 깨달음을 주시고, 학자와 교육자의 태도가 무엇인가에 대해 보여주셔서 많은 것을 배울 수 있는 기회를 얻었습니다. 또한 약간의 부족함이 있는 저를 이끌어주셔서 감사합니다.

다음으로 뉴럴 네트워크 팀원들에게 감사를 표합니다. 팀장인 영환이형은 많은 과제들이 있는 팀에서 홀로 박사과정을 진행하면서 많이 힘들었을 텐데, 그 와중에도 불평없이 팀원들을 케어해 주어서 감사했습니다. 같은 팀원인 천이 누나와 성학이, 그리고 성준이형도 팀에서 각자의 몫을 맡아 주셨고, 늦게 팀에 합류했지만 뛰어난 능력으로 연구의 방향을 제시해 주신 함태준 박사님에게도 감사를 표합니다.

이외에도 아키텍쳐와 프로그래밍에서 뛰어난 재능을 보유한 찬노형, 주말에도 나오셔서 열심히 하는 재영이형, 술을 자주 마시던 학범이형과 문경누나, 신이라 불리는 종현이형, 같이 서버를 담당한 준이형, 먼저 졸업한 상진이 형과 기태형, 정말 하고 싶은 공부를 시작하는 초석사 남호형, 마지막 학기 같이 조교를 한 예진이, 현업에서 종사하고 오셔서 새로운 시야를 깨워주신 김신 책임님과 강석용 책임님, 그리고 꼼꼼한 행정업무를 해 주시는 최미림 선생님께 감사드립니다.

마지막으로, 대학원 생활동안 항상 응원하고 잘 끝낼 수 있도록 도와주신 아버지와 어머니, 그리고 동생에게 감사하다는 말을 전합니다.