ENERGY EFFICIENCY ANALYSIS AND OPTIMIZATION OF CONVOLUTIONAL NEURAL NETWORKS FOR IMAGE RECOGNITION

by

Xinbo Chen, B.Eng.

A thesis submitted to the Graduate Council of
Texas State University in partial fulfillment
of the requirements for the degree of
Master of Science
With a Major in Computer Science
May 2016

Committee Members:

Ziliang Zong, Chair

Yijuan Lu

Martin Burtscher

FAIR USE AND AUTHOR'S PERMISSION STATEMENT

Fair Use

Duplication Permission

DEDICATION

Dedicated to my parents and my United States family, whose support and encouragement during my two years graduate studies motivated me to complete this thesis.

ACKNOWLEDGEMENTS

I am very appreciated Dr. Ziliang Zong for his guidance and support during my graduate study; he is a great advisor not only in teaching and researching, but also in life principles. I would also like to thank Da Li, Dr. Yijuan Lu, and Dr. Martin Burtscher for their expertise, feedback, and overall support in achieving this thesis work.

TABLE OF CONTENTS

Page

LIST OF TABLES

Table **Page**

LIST OF FIGURES

Figure **Page**

LIST OF ABBREVIATIONS

Abbreviation **Description**

CPU - Central Processing Unit
GPU - Graphic Processing Unit
GPGPU - General-purpose GPU
CUDA - Compute Unified Device Architecture
CNN - Convolutional Neural Network
AI - Artificial Intelligence
FC - Fully Connected Layer
cuDNN - CUDA Deep Neural Network Library

ABSTRACT

In recent years, convolutional neural network (CNN) has been widely used to improve the training time and accuracy of image recognition applications. These CNNs are based on deep learning algorithms, which simulate the learning process of human's brain. After sufficient number of images being used to train the neural networks, high recognition accuracy of images can be achieved. In the past few years, especially after GPUs are utilized to carry heavy-duty computation, the accuracy and training time of machine learning algorithms have been significantly improved (e.g. the error rate has dropped from 28.2% in 2010 to 6.66% in 2014). The newest neural network developed by Microsoft in 2015 has already surpassed human's recognition ability with the error rate less than 5%. This is an exciting achievement but also indicates that the room for accuracy improvement is narrowing. On the other hand, due to the massive volume of training data sets, the increasing complexity of neural network structure, and the significant amount of computation, the training process consumes more and more time and energy. In this thesis, we conduct a comprehensive study on analyzing the performance and energy impact of a variety of outer and inner factors in popular CNN algorithms, which provides detailed workload characterization to facilitate the design of more energy efficient CNN algorithms and training frameworks.

1. INTRODUCTION

In 1958, Frank Rosenblatt proposed the Perceptron concepts and a theory on how do neurons in human brains operate [Rosenblatt, 1958]. This theory created a new field of artificial intelligence, called neural network. In 1989, Yann LeCun et al. applied neural network algorithms to recognize handwritten ZIP codes with the training time to be approximately 3 days. After that, numerous interesting practices such as the wake-sleep algorithm [Hinton, 1995] and vanishing gradient problem [Hochreiter, 1998] appeared. However, the slow speed of training continues to be a key factor to impede the advancement of neural network's algorithms and applications. This situation started to change after 2010 with the high speed GPGPU's being utilized to speedup the training time. For example, a single K20 GPU can achieve 1.17 trillion floating-point operations per second [NVIDIA, 2013] and the new Maxwell GPU Titan X is even faster, which makes it feasible to train complex neural networks with large data sets in reasonable time. In the past few years, we have witnessed the rapid growth of deep learning algorithms (i.e. Learning algorithms with more than one stage of non-linear feature transformation), which have been successfully used to solve challenging problems such as image recognition, speech recognition, and natural language processing. This thesis focus on the image recognition field.

To evaluate a variety of deep learning algorithms for object detection and image classification at large scale, the ImageNet Large Scale Visual Recognition Challenge

(ILSVRC) has been held annually since 2010. The imageNet benchmark with 15 million manual-labeled high-resolution images belonging to roughly 22,000 categories is used to train and evaluate the accuracy of state of art neural network algorithms [Russakovsky, 2015]. This challenge uses the percent of top 5 error rate (i.e. If correct class is not int the top 5 classification recommended by the algorithm, it is counted as an error) as the evaluation metric for accuracy. Table 1-1 shows the error rate of winner algorithms in the past six years, which clearly demonstrates the significant improvement in accuracy [Russakovsky, 2015].

Table 1-1: Recent Year CNNs Top 5 Error

Year	Percent of Top 5 Error
2015	3.57 (Microsoft)
2014	6.66 (GoogLeNet)
2013	13.8 (OverFeat)
2012	16.42 (AlexNet)
2011	25.8
2010	28.2

After Microsoft's algorithm already surpassed human recognition ability (~5%) [He, 2015], neural network research has achieved the goal of matching or beating human's capability on image recognition in terms of accuracy. However, human consumes extremely less amount of energy to recognize a picture. One new direction of neural

network research is to design more energy-efficient deep learning algorithms without compromising accuracy. This is a challenging task because there are too many factors (e.g. hardware configurations, different learning framework, external matrix calculation libraries, neural network structure, configuration setting of each layer inside a network, etc) that can affect the training time, the energy consumption and the accuracy. All such factors have unknown contribution to the total energy consumption. To the best of my knowledge, there is no existing study that comprehensively investigates the energy behavior of neural network, especially CNN based deep learning algorithms.

The major contributions of this thesis are summarized below:

1) It describes a strategy to analyze neural network energy consumption from the high level to the detailed level.

2) It presents a comparative study of performance and energy efficiency of four popular neural network training frameworks and finds the most energy efficient framework.

3) It studies the impact of a variety of external math libraries (for both GPUs and CPUs) on the performance and energy consumption of deep learning algorithms.

4) It presents the decomposed energy consumption of each layer in AlexNet and studies how do different factors affect energy behaviors of each layer.

5) It studies the impact of DVFS on the energy consumption of deep learning algorithms when running them on K20 and Titan X GPUs.

This thesis is organized as follows. In Chapter 4, I discuss the energy efficiency of several popular deep learning frameworks. Since neural network training mostly in GPU, I mainly focus on GPU mode of those frameworks. In Chapter 5, I analyze the network wise energy consumption of neural networks. Particularly, I show the energy percentage of each layer consumes during the total network training time. In Chapter 6, major energy consuming layers are broken down from neural network and analyzed separately. In Chapter 7, I use GPU tuning technology to optimize the energy consumption of neural network training and also compare different hardware.

2. BACKGROUND

In this Chapter, I present the background information and terminologies that help to understand how do deep learning algorithms work and the experimental results shown in later chapters.

The convolutional neural network simulates the way human process and recognize images. It consists of multiple layers. The classic CNNs have three kinds of layers: the convolutional layer, the pooling layer and the fully connected layer. For example, Figure 2-1 shows a simple CNN include all these three layers [LeCun, 1998]. Each layer contains thousands or millions of neurons. A single neuron takes some input, computes the weighted sum of inputs, and sends output to the neurons in the next layer.

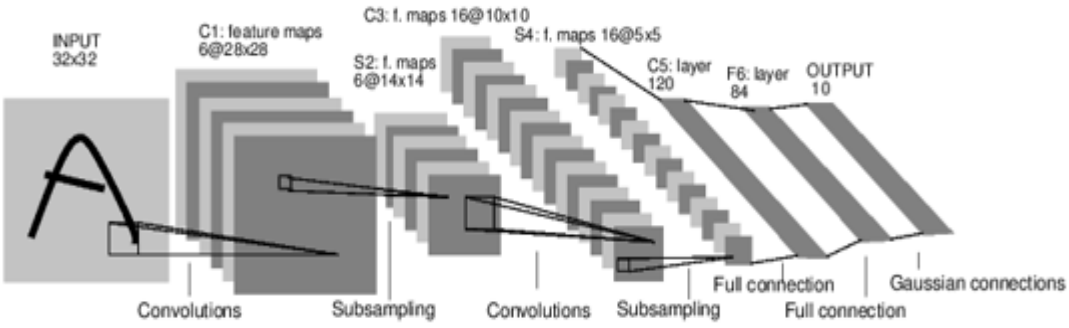

Figure 2-1: LeNet: an algorithm to recognize hand write letter

The following example briefly illustrates how convolution neural network works. CNNs take images as input, for gray image, the channel is 1, for RGB image, the channel is 3. Most current CNNs accept fixed size RGB image as input. If there is a 5 x 5 size gray (channel = 1) image pixel matrix and a 3 x 3 size kernel (or filter). We assume this kernel is used to capture border feature of this image.

$$\begin{bmatrix} 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 \end{bmatrix} \qquad \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix}$$

With the setting of stride = 1, and bias = 1, kernel does inner production with picture matrix, and move one stride, continuing inner production to traverse the whole image matrix, the result is

$$\begin{bmatrix} 4 & 3 & 4 \\ 2 & 4 & 3 \\ 2 & 3 & 4 \end{bmatrix}$$

with (5 - 3 +1) width and (5 - 3 +1) height. This process called convolution and the result called feature map is the following matrix. The kernel is used to detect a specified feature from an image. Each inner production needs to add a bias, so the final feature map matrix is

$$\begin{bmatrix} 5 & 4 & 5 \\ 3 & 5 & 4 \\ 3 & 4 & 5 \end{bmatrix}$$

If we want to capture as many features as possible from this image, we need more different kernels. If there are 100 kernels, then we have 100 feature maps as output after this convolutional layer.

We can use the following two formula to measure the workload of one convolutional layer,

(1) Parameter = (Kernel width x Kernel height + 1) x feature map quantity x channel

(2) Connection = Parameter x Feature map width x Feature map height

Based on the formula, the parameter of this convolutional layer is (3 x 3 +1) x 100 x 1= 1000. The connection quantity is 1000 x 3 x 3 = 9000.

Typically, a threshold function (or activation function) will follow a convolution layer. For example, in AlexNet, the activation function is called a Rectified Linear Unit, or ReLu layer. Its formula is f(x) = max (0, x). The x is the input of a neuron. If x >0, the neuron will be activated and pass value to the next layer (not activated otherwise).

The pooling layers, or down sampling layer, are usually placed between two convolutional layers. It partitions the input feature map into a set of non-overlapping rectangles (the stride size is equal to the filter size). There are mainly two methods of pooling: max pooling and average pooling which will be explained in section 6.3. The pooling layer progressively reduces the amount of parameters as well as controls overfitting. Overfitting means that the training model has higher accuracy in recognizing images of pre-trained data sets, but has lower accuracy in recognizing unseen images. This is usually caused by data noises or limited training data sets.

Finally, after several convolutional and pooling layers, the CNN algorithms finalize the results via fully connected layers. Neurons in a fully connected layer have full connections to all neurons in the previous layer. The output can hence be computed with an one-dimensional matrix multiplication followed by a bias offset. Each value of this

one-dimensional matrix is the classification probability of this image.

Forward propagation refers to the learning process from convolution layers to fully connected layers. Classifying an image only needs forward propagation. Backward propagation is needed to adjust deviation when the training algorithms leverage the temporary training results to verify whether or not the temporary recognition results have a big offset.

3. RELATED WORK

Over the last decade, graphics processing units (GPUs) have been widely used in many fields for application accelerations (e.g. Bioinformatics [Truong, 2014], Graph [Li, 2013], Nested Parallelism [Wu, 2016], Irregular Loops [Li, 2015]). Nowadays, the peak double-precision performance of high-end GPUs from Nvidia is well above 1 teraFLOPS [NVIDIA, 2013]. With the emergence of high speed GPUs and the availability of large data sets for training, we have witnessed the significant improvement of deep learning algorithms in terms of training time and accuracy and the boom of deep learning applications. The Visual Geometry Group (VGG) of University of Oxford has designed a 16 layers model with 7.4% top 5 error and a 19-layer model with 7.3% top 5 error rate [Simonyan, 2014]. In 2015, Microsoft implemented a model with 152 layers - 8x deeper then VGG nets with 3.57 % error [Zhang, 2015]. The fast development not only represents in the structure of neural network, but also reflects in wider scope. For example, The Deep Face deployed at Facebook for Auto tagging [Taigman, 2014]; Estimation of person pose [Tompson, 2014]; Generating a descriptive sentence [Lebret, 2015] etc. In spite of the advancement in developing deeper and more complicated neural network structures, the research on investigating the energy consumption behavior of different neural network and training framework is still in its infancy. To the best of my knowledge, there are no comprehensive studies on analyzing the energy-aware deep learning algorithms. With the size of data sets increase exponentially and the ever

increased energy consumption for training such data sets, it is desirable to design more energy efficient deep learning algorithms. This thesis contributes to this field by presenting the energy behaviors of numerous well-known deep learning algorithms and exploring the impact of various factors on the energy consumption of these algorithms. Although other accelerators like Xeon Phi and FPGA can provide high throughput and are widely used in high performance computing domains including tree/graph traversal [Li, 2014], information fusion [Song, 2011] and deep learning [Lacey, 2016], this thesis focuses on quantifying and studying power and energy behaviors on GPU platforms.

4. ENERGY CONSUMPTION ANALYSIS OF NEURAL NETWORKS IN DIFFERENT LEARNING FRAMEWORKS

In this chapter, I present a comparative study of four popular deep learning frameworks, namely Caffe, Torch, TensorFlow, and MXNet, in terms of performance and energy efficiency. I evaluate these two aspects for both CPU and GPU settings.

4.1 Introduction of Training Framework

In this section, I first briefly introduce each framework and their features.

Table 4-1 shows the number of users in Google groups and the number of open source contributors for each framework in their Github repositories. It is clear that these four frameworks are widely used and supported by the deep learning community.

Table 4-1: Community involvements of four deep learning frameworks as of 03/18/2016

Measures	Caffe	Torch7	TensorFlow	MXNet
Number of members in Google groups	4589	2062	780	N/A
Number of contributors in Github	182	84	153	107

4.1.1 Caffe

Caffe, an abbreviation of Convolutional Architecture for Fast Feature Embedding, is a well-known and widely used open source framework that was originally designed by the U.C. Berkeley Vision and Learning Center (BVLC) then co-designed by community contributors. Caffe is written in C++, and supports CUDA for GPU computation. It is designed with expressive architecture and supports open source math libraries to ensure

high performance. In addition, Caffe emphasizes usability by allowing users to configure each layer of a neural network easily without complicated coding [Jia, 2014].

4.1.2 Torch

Torch is a computational framework written in Lua, which is a multi-paradigm scripting language. Compared with C, Lua is more readable and easy to learn. In addition, rich interfaces to C keeps Lua's high performance for large scale applications. Torch currently is used by large tech companies such as Google DeepMind and Facebook, which devote in-house teams to customize their deep learning platforms. In our experiments, we used the newest version Torch 7 to train neural networks.

4.1.3 TensorFlow

TensorFlow is another open source framework which derives from the Google Brain project. It is an interface for expressing machine learning algorithms, and an implementation for executing neural network training algorithms [Abadi, 2016]. TensorFlow is famous for its flexibility to express a wide variety of algorithms, including computer vision, robotics, and natural language processing etc. It also has strong portability to run on desktop, server, or even mobile computing platforms.

4.1.4 MXNet

MXNet is a lightweight, portable and flexible mobile deep learning framework. It supports Python, R, Julia, Go, and JavaScript, which is the framework that supports most programming languages [Chen, 2015].

4.2 Benchmark Setup

There are two types of benchmark in this thesis. The first type of benchmark is the data benchmark, which includes the training data with training setting such as iteration quantity. The second type of benchmark is neural network algorithms. Due to numerous memory occupation during training, the K20 GPU can only fully support AlexNet [Krizhevsky, 2012], and partly support OverFeat [Sermanet, 2013] because some frameworks require more memory to run.

4.2.1 Data Benchmark

The benchmark I select comes from an open source project called Convent Benchmark, which supports most publicly accessible implementation of CNNs under the same data sets entry [Convent-Benchmark, 2015]. In this benchmark, the author, Soumith Chintala, a member of Facebook AI research team, picked popular ImageNet models and clock the time for full forward and backward pass in his machine. However, this benchmark does not measure the energy consumption of each learning framework. I provide valuable extension to this benchmark by comparing the energy consumption of the aforementioned four frameworks. The benchmark parameters are shown in Table 4-2.

Table 4-2: Convent Benchmark parameters setting for four deep learning frameworks

Convent Benchmark Parameter	
Data Set	Random generated data
Training Iteration	10 times

The data sets in Convent Benchmark is randomly generated and able to response to a variety of input size at different layers. The data will be fixed and reused during each iteration's training. I set the training iteration number as 10 to ensure sufficient number of power samples can be collected for energy consumption calculation.

4.2.2 Neural Network Benchmark

I select AlexNet and OverFeat as algorithm benchmarks. Both of them have 128 batch size for each iteration’s input. Other configurations are the same as the published configurations in BVLC’s model zoo [Model Zoo, 2015].

4.2.3 System Setup

All experiments are performed on a single machine running on CentOS 7 with Intel Xeon E5 - 2650 v2 @ 2.6GHz; Nvidia Tesla K20m with 5GB memory; 32GB DDR3 main memory; and 128 GB SSD hard drive. The drivers and libraries used in our experiments include CUDA 7.0, cuDNN v3, OPENBLAS 0.2.16, Caffe (commit ID be163be), Torch 7 (commit ID eb8d7f2), and TensorFlow (commit ID fd464ca), MXNet (commit ID d25053). For the power measurement, the CPU and DRAM power data are collected via the Intel Running Average Power Limit (RAPL) interface [Intel, 2012] and the GPU power is obtained via the Nvidia’s System Management Interface.

4.3 GPU Mode Results and Discussions

4.3.1 Native GPU v.s. GPU with cuDNN Library

In this subsection, I define the GPU without cuDNN library as native GPU. The NVIDIA CUDA Deep Neural Network Library (cuDNN) is a GPU-accelerated library of primitives for deep neural networks. The cuDNN allows deep learning developers and researchers to focus on designing and training neural network models rather than spending their time to tune the low-level hardware performance counters for the best performance [cuDNN, 2013].

Since TensorFlow and MXNet are already embeded with cuDNN when released, Caffe and Torch7 are used to evaluate the impact of cuDNN on performance and energy consumption.

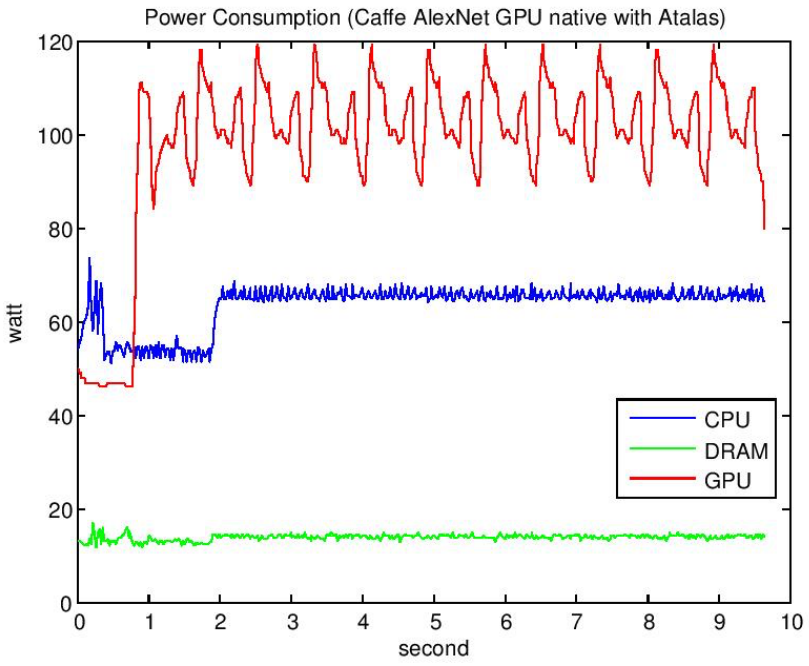

Figure 4-1: Caffe's power consumption with native GPU

Figure 4-1 shows the energy curve of Caffe during benchmark training. As shown in this figure, at the beginning the GPU power (red line) stays idle (< 50W) for around one second. Then after one fluctuation's initialization, ten iterations of similar fluctuations are observed, which represent the ten training iterations. When GPU is idle, CPU is running the benchmark to load and locate the training data, read neural network configuration file, etc. When GPU is doing initialization, CPU goes back to the idle state, followed by CPU's power burst to over 60Wto transfer the data GPU requested for training.

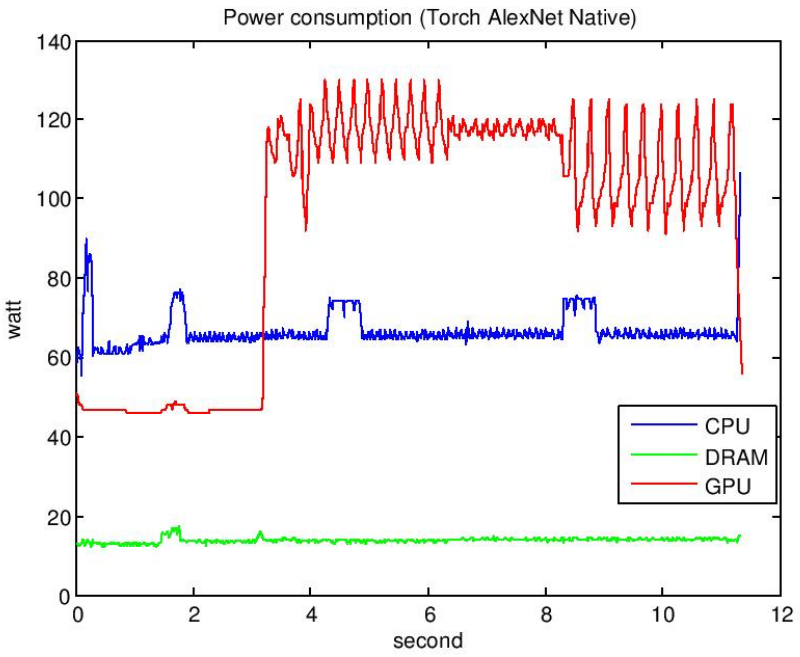

Figure 4-2: Torch's power consumption with native GPU

As we can see from figure 4-2, GPU stays in idle over three seconds, which means there are three seconds of performance loss and energy waste for the CPU waiting for GPU finishing its work. During running, CPU has two power spikes. The first one appears

right after GPU initialization and the second one happens after Torch starts doing backward calculation.

Table 4-3: Caffe and Torch energy and performance comparison in native GPU

Native GPU	Time (s)	CPU average Power (W)	GPU average Power (W)	CPU energy (J)	GPU energy (J)	GPU and CPU energy (J)
Caffe	9.64	63.55	98.47	612.62	949.25	1561.87
Torch7	11.35	66.6	94.19	755.91	1069.06	1824.97

Table 4-3 shows the performance and energy consumption of two frameworks on AlexNet with native GPU. Under these restrictions, Caffe results in the best performance and the lowest energy consumption. Note that in Figure 4-2 of Torch, the power of GPU keeps idle for 3.5 seconds, then GPU instantly goes up to over 120 W. The similar initialization (GPU stays idle) in Caffe only takes 0.9 second. If consider without initialization time, Torch and Caffe have very similar performance and energy consmpution.

To compare whether caffe is always faster than Torch in native GPU, I also executed the same benchmark with the OverFeat neural network.

Table 4-4:Caffe and Torch 7 performance and energy with OverFeat

Native GPU	Time (s)	CPU average Power (W)	GPU average Power (W)	CPU energy (J)	GPU energy (J)	GPU and CPU energy (J)
Caffe	25.82	67.76	109.95	1752	2839	4591
Torch7	28.55	62.96	114.09	1798	3257	5055

Table 4-4 shows that Caffe saves 9% of time and 9.1% of energy when training the OverFeat algorithm than using Torch. Next, I evaluate how can cuDNN accelerate these two frameworks.

Figure 4-3 demonstrates several interesting observations: 1) the total training time drops from 9.64 second to 6.61 second; 2) the average GPU power is much higher than native

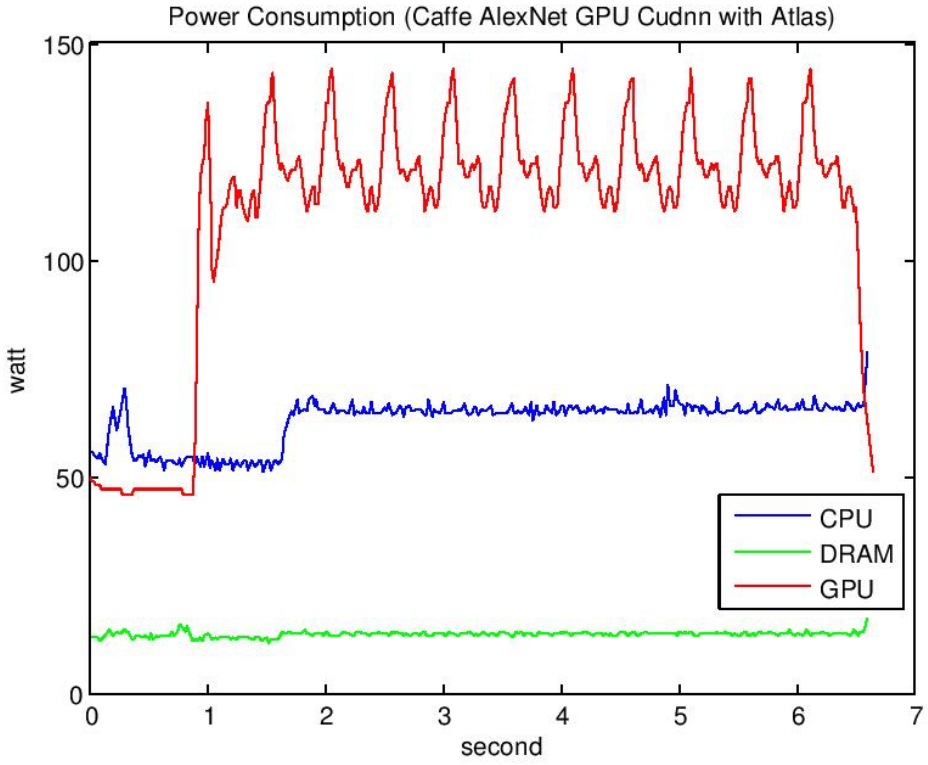

Figure 4-3: Caffe's power consumption in GPU Mode with cuDNN GPU.

Since cuDNN has nothing to do with CPU acceleration, and its optimization does not affect Caffe's architecture, the initialization and CPU power curve doesn't change. However, this three seconds time saving saves energy for both GPU and CPU.

Table 4-5: Caffe and Torch7 energy efficiency comparison in GPU with cuDNN

GPU with cuDNN	Time (s)	CPU average Power (W)	GPU average Power(W)	CPU energy (J)	GPU energy(J)	GPU and CPU energy(J)
Caffe	6.61	62.99	111.67	416.4	742.61	1159.01
Torch7	18.47	55.64	87.51	1027.67	1616.31	2643.98

In addition, there are huge difference in Torch diagram. From Figure 4-4, we can observe

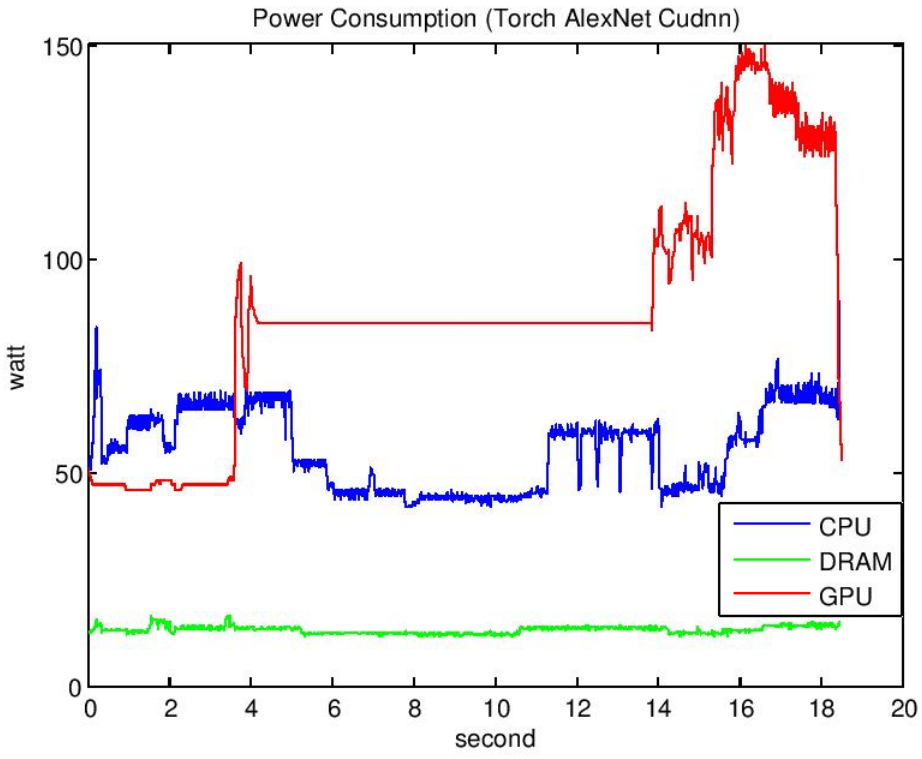

Figure 4-4: Torch's AlexNet power consumption in GPU Mode with cuDNN

a long straight power line (at 85 W) that lasts over ten seconds after GPU's idle state and

before the training starts.

On the contrary, based on the execution log of Torch, it finishes its training process in 2.58 second, which is the same duration for forward and backward propagation in Figure 4-4 (From the highest point to the end of the red line). However, the time recorded by our power meter reflects the true execution time of the entire benchmark script, which means that Torch starts its own timer right after the first forward propagation and ignored the time before training. To figure out this conflict, I check the source code of Torch's benchmark. The crucial portion is marked in Figure 4-5.

```
23 print('Running on device: ' .. cutorch.getDeviceProperties(cutorch.getDevice       ()).name)
24 print('cuDNN version: ' .. cudnn.version)
25
26 steps = 10 -- nb of steps in loop to average perf
27 nDryRuns = 50
28
29 function makeInput(config, size)
30       local layout = config[4]
31       local osize
32       if layout == 'BDHW' then
33            osize = size
34       elseif layout == 'DHWB' then
35            osize = {size[2],size[3],size[4],size[1]}
36       elseif layout == 'BHWD' then
37            osize = {size[1], size[3], size[4], size[2]}
38       end
39       return torch.randn(torch.LongStorage(osize))
40 end
41
42 for i=1,#nets do
43       for j=1,#libs do
```

Figure 4-5: Torch benchmark script file source code

```
44          collectgarbage()
45          local model,model_name,size = nets[i](libs[j])
46          model=model:cuda()
47          local input = makeInput(libs[j],size):cuda()
48          local lib_name = libs[j][5]
49          print('ModelType: ' .. model_name, 'Kernels: ' .. lib_name,
50                  'Input shape: ' .. input:size(1) .. 'x' .. input:size(2) ..
51                      'x' .. input:size(3) .. 'x' .. input:size(4))
52          -- dry-run
53          -- for i=1,nDryRuns do
54            --    model:zeroGradParameters()
55             -- local output = model:updateOutput(input)
56            --    local gradInput = model:updateGradInput(input, output)
57            --    model:accGradParameters(input, output)
58            --    cutorch.synchronize()
59            --    collectgarbage()
60        -- end
61
62          local tmf, tmbi, tmbg
63          sys.tic()
64          for t = 1,steps do
65              output = model:updateOutput(input)
66          end
67          cutorch.synchronize()
68          tmf = sys.toc()/steps
69          print(string.format("%-30s %25s %10.2f", lib_name, ':updateOutput():',        tmf*1000))
70
71          collectgarbage()
72          sys.tic()
73          for t = 1,steps do
74              model:updateGradInput(input, output)
75          end
76          cutorch.synchronize()
77          tmbi = sys.toc()/steps
78          print(string.format("%-30s %25s %10.2f", lib_name, ':updateGradInput()       :', tmbi*1000))
79
80          collectgarbage()
81          sys.tic()
```

Figure 4-5: Continued

```
82          local ok = 1
83          for t = 1,steps do
84              ok = pcall(function() model:accGradParameters(input, output) end)
85          end
```

Figure 4-5: Continued

In Figure 4-5, line 52 to line 60 is the Dry-run code, which is used to prepare for training. To verify whether the straight line is related to the Dry-run code, I first set up independent timer to measure duration of each portion of code. Initialization takes 4.2 second, Dry-run portion takes near 12 seconds, forward and backward propagation takes 2.58 second, which altogether represent the energy curve in Figure 4-4. Additionally, I change the value of nDryRuns from 1 to 50 and retrain again, the result shows in Figure 4-5. The initialization time in Figure 4-5 stays the same (4.2 second), the Dry-run portion takes 25.1 second, and the forward and backward propagation remains the same (2.58 second). Therefore, its clear that the straight line is related to the Dry-run code.

Lua is known as one of the fastest scripting languages and very popular in game industry, which requires low latency [Lua Docuement, 2016]. This delay phenomenon in Torch should not relate to which programming language the framework choose.

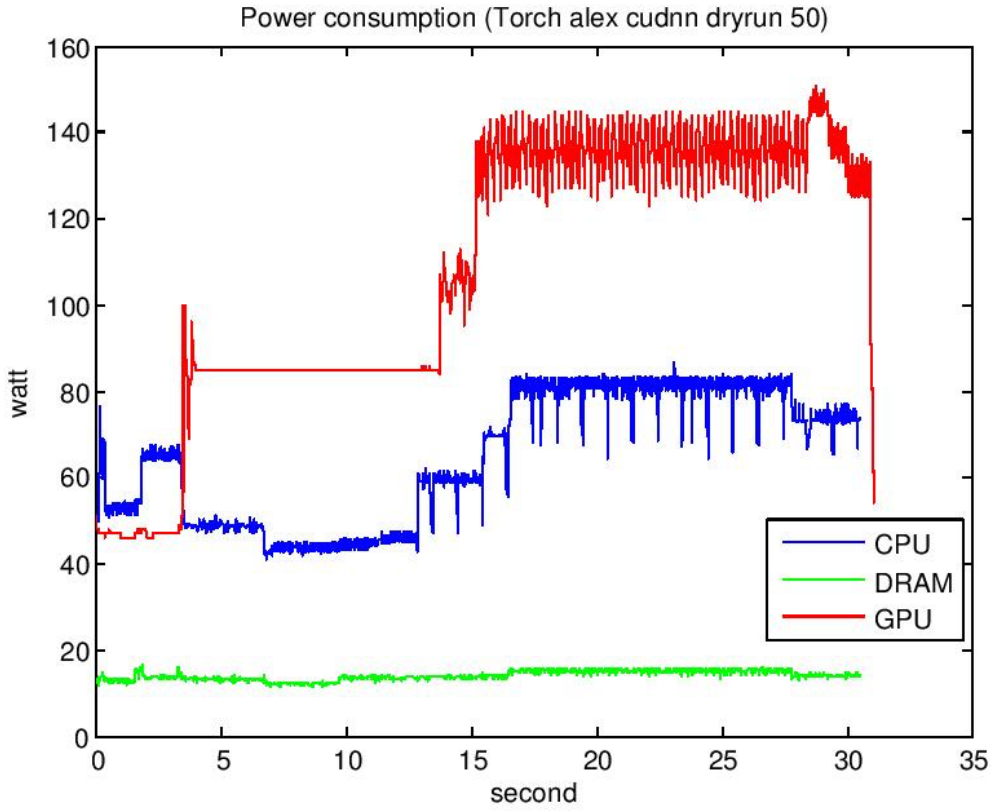

Figure 4-6: Torch's power consumption in GPU Mode with cuDNN in 50 Dry-run

Then I decide to comment the whole Dry-run code and execute again. We surprisingly find that the forward and backward propagation time increases from 2.58 second to 14.67 second, while Dry Run time becomes 0 second. It means the over ten seconds straight line still exists. Therefore the Dry-run is not meaningless and its the true initialization for forward and backward propagations. If such initialization time is included in the actual training time, Torch will be much slower and less energy efficient than Caffe.

Note that Caffe also has near one second initialization time on GPU, I would like to see how much energy and time are actually used to train the neural network. I cut off both

Caffe and Torch's initialization time. The results are shown in Figure 4-7 and Figure 4-8.

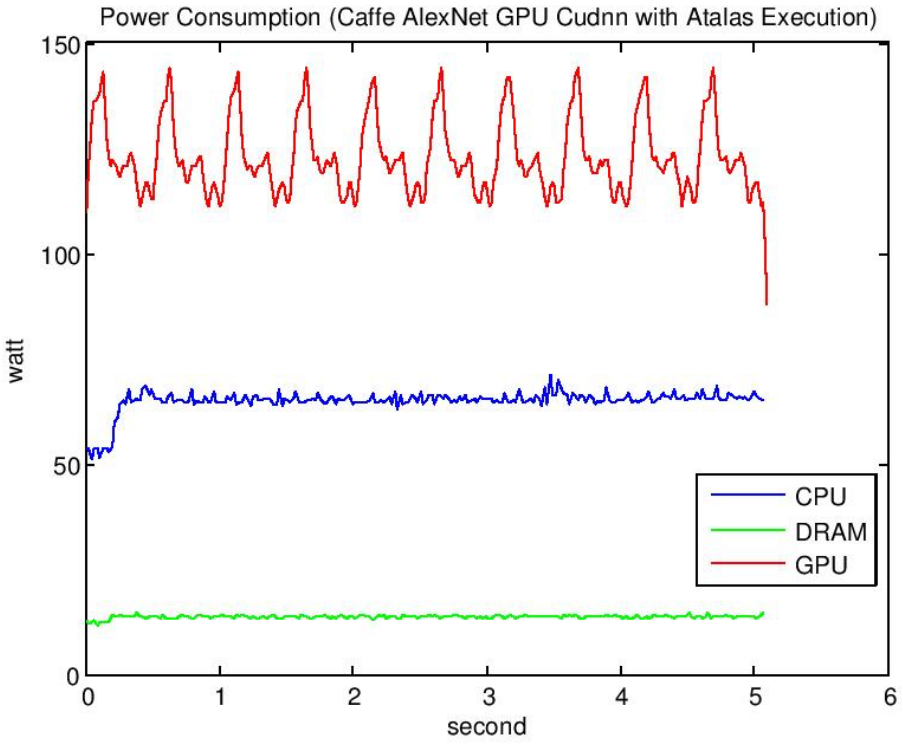

Figure 4-7: The actual execution time and power consumption of Caffe

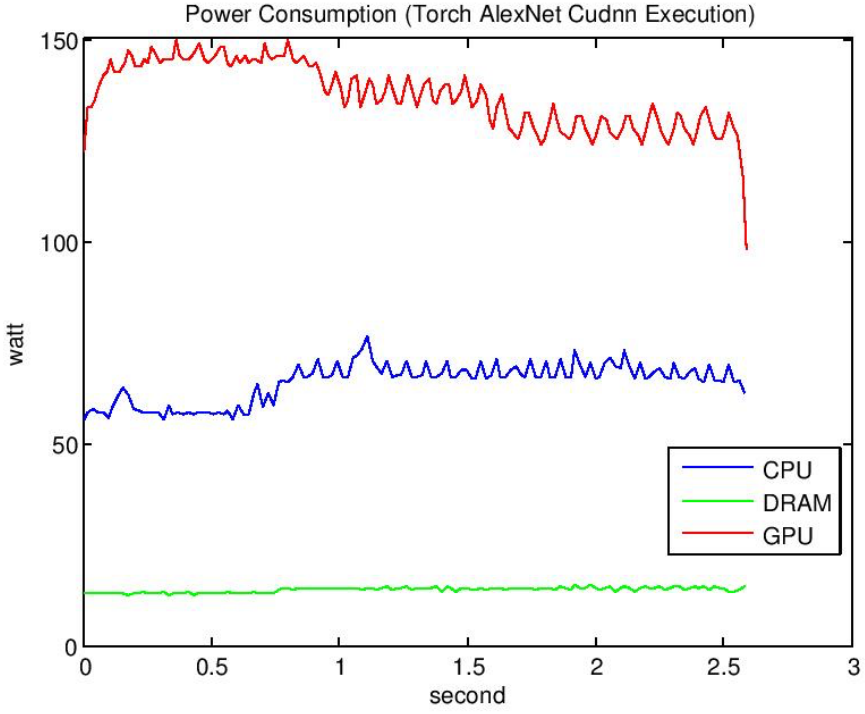

Figure 4-8: The actual execution time and power consumption of Torch

Table 4-6: Caffe and Torch7 actual execution time and energy efficiency comparison in GPU with cuDNN

GPU	Time (s)	CPU average Power (W)	GPU average Power(W)	CPU energy (J)	GPU energy(J)	GPU and CPU energy(J)
Caffe-cudnn	5.06	65.01	122.78	328.95	621.27	950.22
Caffe-native	7.99	65.13	103.04	520.39	823.29	1343.68
Torch7-cudnn	2.58	65.13	135.9	168.04	350.62	518.65
Torch7-native	7.35	67.2	112.89	493.92	829.74	1323.66

Table 4-6 shows the energy consumption and performance caused by the training code. To analyze how cuDNN promote the training process and ensure the results are comparable, I also isolate the actual execution of Caffe and Torch with native GPU. With the help of cuDNN, Caffe trains the same data sets with 1.58 time faster and consumes 1.41 time less energy. Torch achieves 2.85 times of speedup and saves 2.55 times of energy.

4.3.2 Overall Framework Energy Consumption and Performance Comparison

In this experiment, I only evaluate the impact of cuDNN on the TensorFlow and the MXNet (results are shown in Figure 4 -9, Figure 4 -10 and Table 4 – 7).

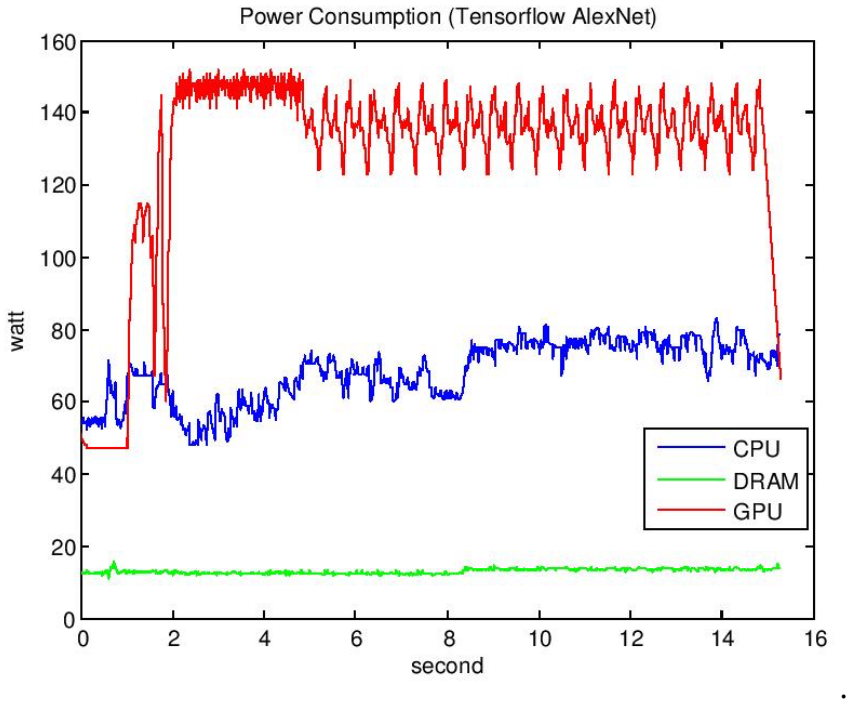

Figure 4-9: TensorFlow's power consumption in GPU Mode with cuDNN

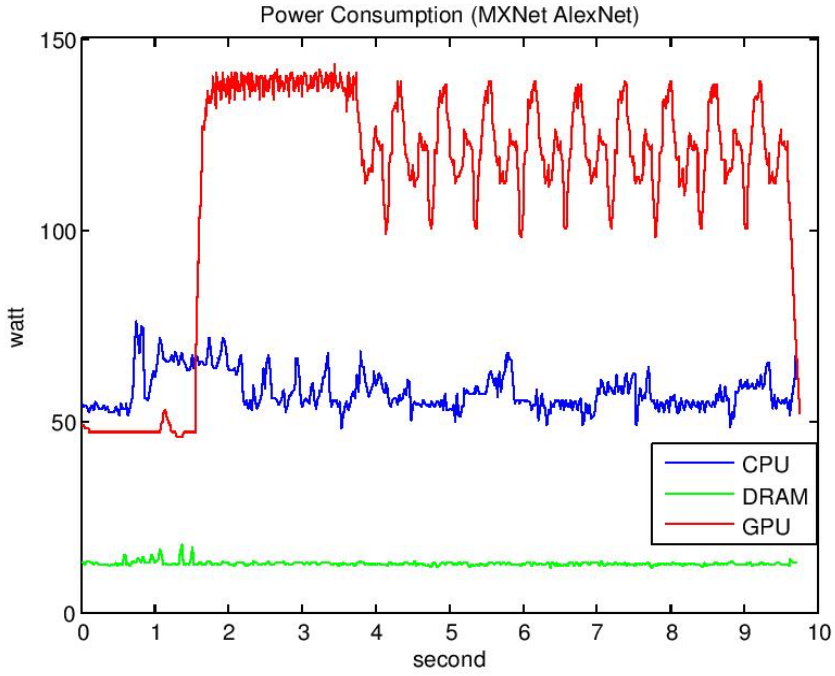

Figure 4-10: MXNet's power consumption in GPU Mode with cuDNN

Table 4-7: Four framework's overall energy consumption and performance

GPU	Time (s)	CPU average Power (W)	GPU average Power(W)	CPU energy (J)	GPU energy(J)	GPU and CPU energy(J)
Caffe	6.61	62.99	111.67	416.4	742.61	1159.01
Torch7	18.47	55.64	87.51	1027.67	1616.31	2643.98
TensorFlow	15.21	68.26	130.41	1038.23	1983.54	3021.76
MXNet	9.71	57.9	112.37	562.2	1091.11	1653.31

Table 4-7 shows the total energy consumption and performance of four deep learning frameworks, which include their initialization and actual training process. Overall, Caffe has the best performance and energy efficiency among these four benchmarks.

4.4 CPU Mode Results and Discussions

4.4.1 Math Libraries for CPU

The Basic Linear Algebra Subprograms, or BLAS libraries, are used to support CPU performing linear algebra operations such as matrix operations. Since Caffe provides flexible interfaces to incorporate different external libraries, I choose Caffe to compare three popular BLAS libraries: Atlas, OpenBLAS, and MKL. Atlas is the abbreviation of Automatically Tuned Linear Algebra Software, which provides C and Fortran interfaces to a portable and efficient BLAS implementation. OpenBLAS is an open source project of BLAS with many optimizations for specific processor type. MKL, or Math Kernel Library, is the commercialized BLAS of Intel Corporation, which has been specially optimized for Intel CPUs. All of these three BLAS libraries support multiple threads.

4.4.2 Caffe - CPU Mode Energy Efficiency Analysis with BLAS

It's predictable that CPU will train neural network much slower than GPU. Therefore, I only select AlexNet to reduce the time of generating experimental results.

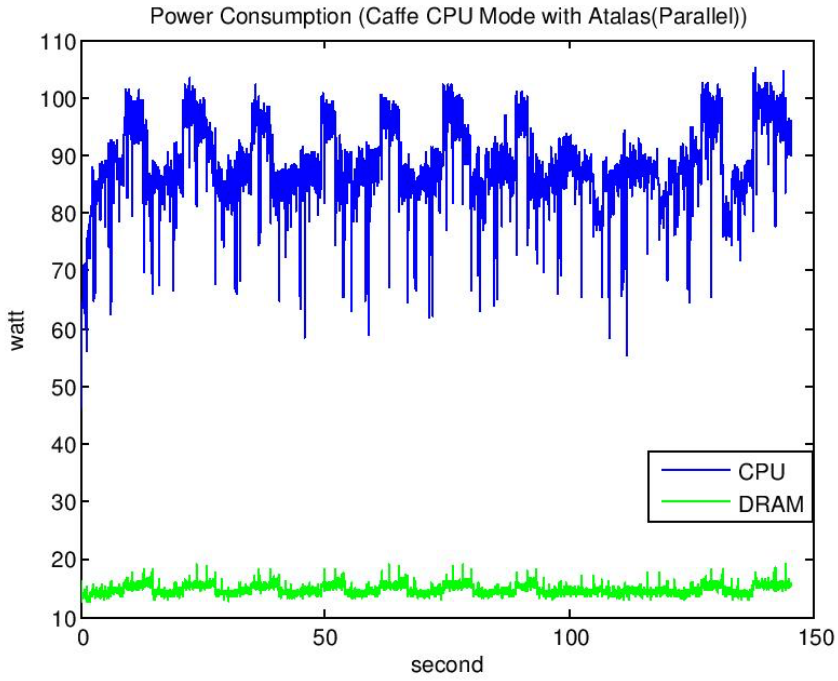

Figure 4-11: Caffe - CPU mode with Atlas energy consumption

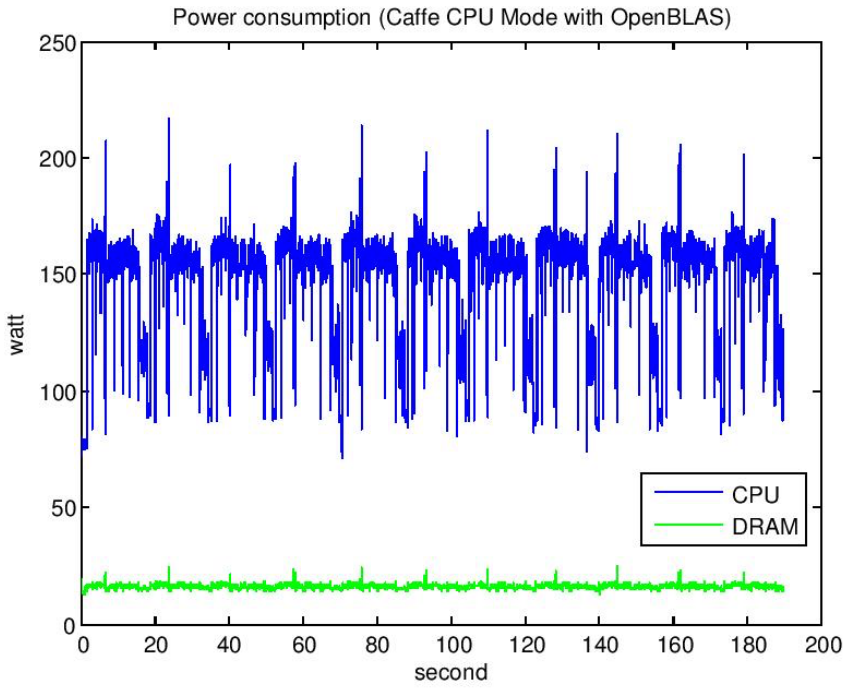

Figure 4-12: Caffe - CPU mode with OpenBLAS energy consumption

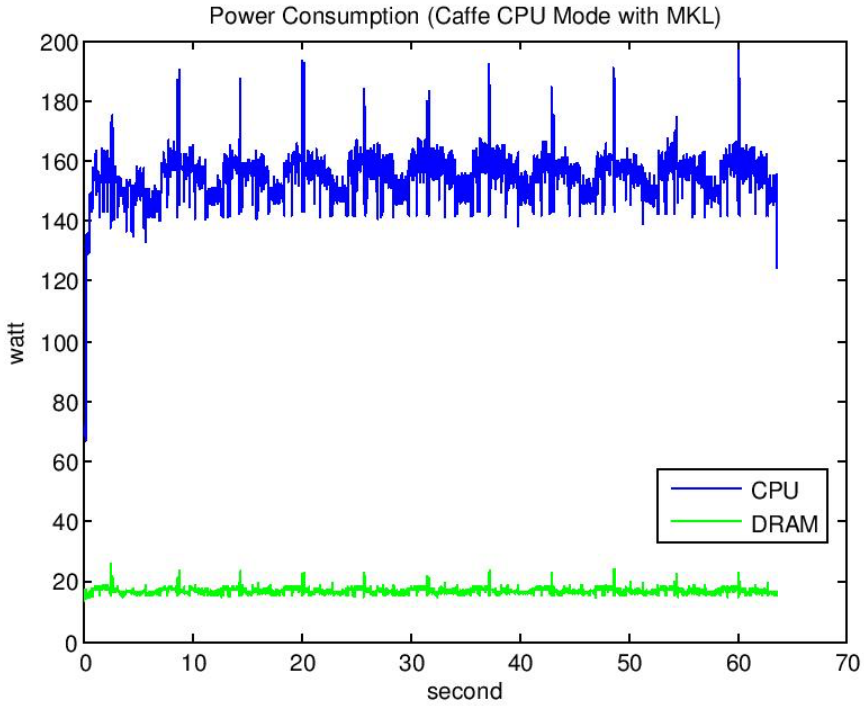

Figure 4-13: Caffe - CPU mode with MKL energy consumption

Table 4-8: BLAS library energy efficiency and performance comparison in Caffe

Caffe -CPU	Time (s)	CPU average Power (W)	CPU energy (J)
Atlas	145.35	88.08	12802
OpenBLAS	189.82	148.86	28257
MKL	63.7	154.51	9842

From Figure 4-11, 4-12, 4-13, and Table 4-8, we can observe that MKL helps CPU running in a high average power with less fluctuation. Fluctuated utilization of hardware is normal in deep learning training process because of its iterate characteristics. From the Atlas and OpenBLAS energy figures we can see that after each iteration's training, power drops to the same or similar level of the idle state, which is the first point drawing in the blue curve. This means after every training iteration, the CPU takes a break and needs a short amount of time to warm up again before training next iteration. However, MKL keeps CPU in high power after one iteration, which helps CPU better utilize its resources during training time. Compared to Atlas, MKL achieves 2.28 times of speedup and saves1.3 times of energy.

4.5 Conclusion

In this chapter, I compare the performance and energy behavior of four learning frameworks in the GPU mode, and then discuss the impact of three math libraries on performance and energy in the CPU mode. For the overall training process, Caffe is the best choice because of its good training performance and short initialization time. For

only training portion, Torch 7 is the fastest one. Since the initialization step is unavoidable for each framework, so Caffe is the best choice both in performance and energy saving. In the CPU mode, MKL shows much better acceleration of Intel CPUs than other two libraries. Therefore, in the experiments shown in Chapters 5 and 6, I chose Caffe to train the neural network with the cuDNN and MKL acceleration libraries enabled for GPU and CPU respectively.

5. NETWORK WISE ENERGY CONSUMPTION ANALYSIS

In Chapter 4, I investigate the energy efficiency of different training framework and conclude that Caffe shows good energy efficiency and performance cross two neural networks. This Chapter focuses on studying the energy efficiency of the neural network itself. Since Caffe is both user friendly and highly efficient, I choose Caffe in this Chapter as the training framework to further explore the impact of neural network inner structure on performance and energy efficiency. Because of the time consuming of disassembling a neural network, I only disassemble AlexNet in this Chapter. I analyze the energy consumption distribution of all major layers and study the impact of batch size on energy consumption.

5.1 Each Layer Energy Occupation

To split an integrated neural network into separate layers, the key point is acknowledgment of each layers input and output data size. For example, the previous layers output size should perfectly match the next layer’s input size.

It is difficult to disassemble the layers of a neural network. Fortunately, AlexNet has relatively simpler structure and I select it to perform the studies of this Chapter . The primary difference of an integrated neural network and separate layers collections are data. For neural network, I need to input data in one time, then data will transmit between layers and generate output. However, separate layers need execute respectively with unrelated data. Since different data will not affect the energy consumption of each layer,

only data input size matters. As a result, I do not need to use actual data calculated by previous layer as the next layer's input data. Rather, I can use benchmark data with specified input size each time when I target only on one layer.

If this break-down approach succeeds, the total breakdown time and accumulated energy consumption of each layer should be equal or similar to the time and energy consumption of the integrated neural network. Since some layers run faster, in order to collect sufficient number of power samples for accurate energy calculation, I use benchmark data with 100 iterations to train each layer as well as the whole AlexNet.

Table 5-1: Separate Layer Time and Energy Consumption

Layer	Time (Second)	Energy Consumption (Joules)
Conv1	7.325	1307.073
Conv2	14.291	2626.11416
Conv3	6.988	1336.87428
Conv4	9.583	1769.50095
Conv5	6.468	1149.81636
conv1-relu	0.154	20.41424
conv2-relu	0.113	15.18833
conv3-relu	0.064	7.6608
conv4-relu	0.047	5.20995
conv5-relu	0.046	5.21318
pooling 1	0.761	124.85727
pooling 2	0.497	79.31126
pooling 5	0.181	25.53729
Fully connected 6	2.111	332.35584
Fully connected 7	1.022	153.69858
Fully connected 8	2.642	411.25372
Accumulations	52.293	9370.079
Integrated AlexNet	52.28	9409.35

From the last two rows in Table 5-1, the accumulation of each layers time is 52.293 seconds, and the total energy is 9370.079 Joules. These two data are very close to the whole AlexNet training time and energy consumption, which means our break-down approach is acceptable. To analyze which layer consumes more energy, I present each

layer's percentage of total energy in Figure 5-1 and Figure 5-2.

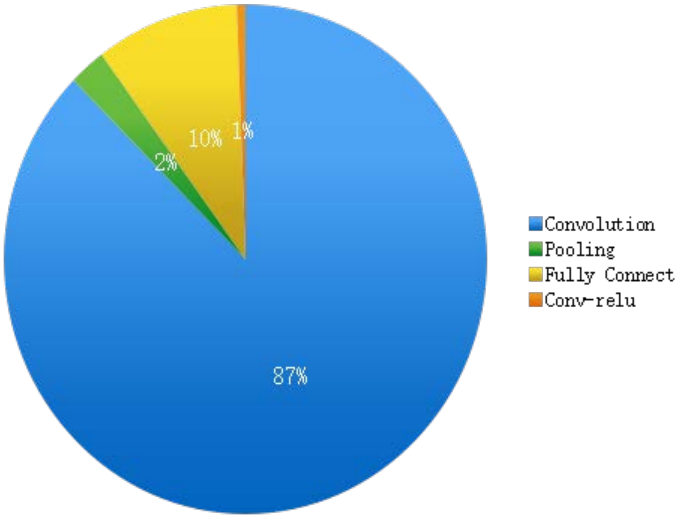

Figure 5-1: Time Occupation percentage of layers

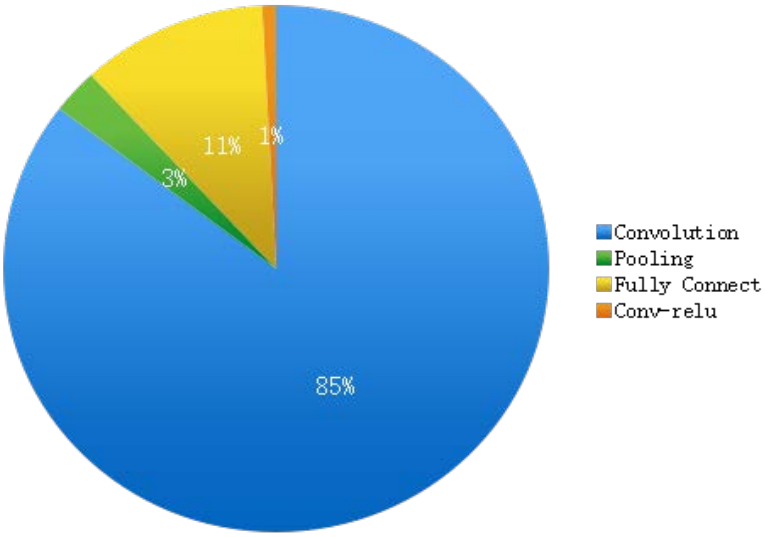

Figure 5-2: Energy Occupation percentage of layers

Figures 5-1 and 5-2 demonstrate that convolutional layer uses 85% of total time and 87%

of the total energy. The fully connected layer accounts for 11% of total time and 10% of total energy. The pooling layer is responsible for 3% of time and 2% of energy consumption.

Based on the break-down approach of AlexNet and the energy distribution of each layer, researchers will be able to get more knowledge about which layer should put more concentration in order to reduce the overall energy consumption of neural networks.

5.2 Batch Size Factor of Neural Network

Batch size is an important setting of neural network but unrelated to neural network structure. A bigger batch size indicates that the data can better represent the full data sets' feature. However, loading in a bigger batch requires more GPU memory thereby may consume more energy.

Table 5-2: AlexNet Energy Consumption with different batch size

Batch Size	Time (s)	CPU average Power (W)	GPU average Power(W)	CPU energy (J)	GPU energy(J)	GPU and CPU energy(J)
16	1.935	55.76	80.672	107.9	156.1	264
32	2.603	61.93	90.745	161.2	236.21	397.41
64	3.976	61.52	101.5	244.6	403.56	648.16
96	5.319	62.386	106.23	331.8	565.04	896.87
128	6.641	63.455	111.22	421.4	738.61	1160.01

In this section, I evaluate integrated AlexNet and OverFeat with batch size ranging from 16 to 128 to in terms of training time and energy consumption.

Table 5-2 shows that the training time increases proportionally with batch size. In

addition, I find that the average power of both CPU and GPU goes up as batch size increases. Figures 5-3 and 5-4 reveal the linear growth of training time and energy consumption with batch size.

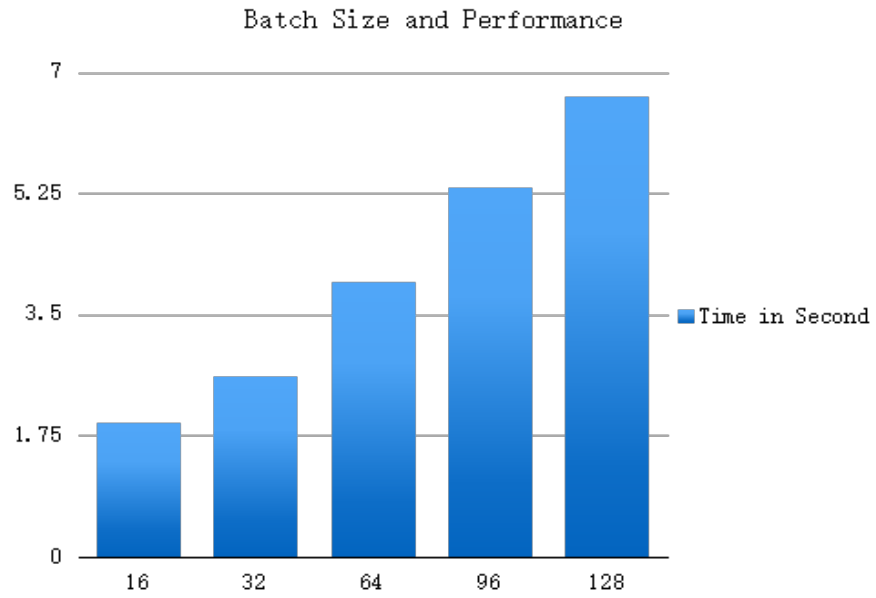

Figure 5-3: Batch size parameter influence of performance

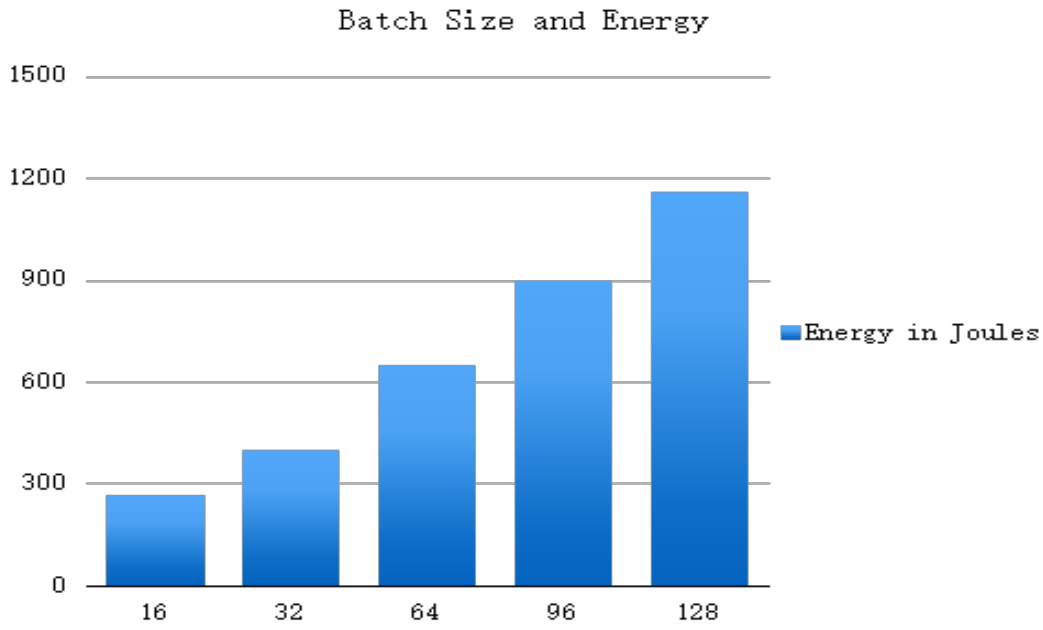

Figure 5-4: Batch size parameter influence of energy

In addition, I use the same strategy to analyze the batch size effect on OverFeat, Table 5-3 shows that OverFeat's performance and energy consumption both grow linearly with the batch size.

Table 5-3：OverFeat Energy Consumption with different batch size

Batch Size	Time (s)	CPU average Power (W)	GPU average Power(W)	CPU energy (J)	GPU energy(J)	GPU and CPU energy(J)
16	4.262	61.2	97.48	260.83	415.46	676.29
32	6.44	62.81	108.36	404.49	697.83	1102.33
64	10.76	64.13	118.98	690.04	1280.22	1970.26
96	15.217	64.62	123.12	983.32	1873.52	2856.84
128	19.46	64.71	125.22	1259.26	2436.78	3696.04

6. LAYER WISE ENERGY CONSUMPTION ANALYSIS

In this chapter, I focus on analyzing power and energy behaviors in each layer. Based on our analysis in previous sections, there are main types of layers to a neural network: 1) Convolutional Layer, 2) Fully-Connected Layer; and 3) Pooling Layer. I use these three layers to analyze layer wise power / energy consumption. Since I already conclude that GPU is more energy-efficient than CPU, I only concentrate on GPU energy analysis in this chapter.

6.1 Convolutional Layer

6.1.1 Common Convolutional Layers

In our AlexNet analysis, I find that convolutional layer consume over 85% of total energy in training. This is important to understand which factors contribute to the energy consumption most for convolutional layers. First of all, I conduct the measurement on six most commonly seen convolutional layers adapted from [Convent Benchmark, 2015]. These layers are frequently used in these championship neural network of ILSVRC in recent years. The following table shows the configuration of six layers.

Table 6-1: Convolution Layer Specification

Convolution Layer #	Input Size	Batch Size	Feature Map (Channel->Kernel number)	Kernel Size	Stride
L1	128 x 128	128	3 -> 96	11 x 11	1 x 1
L2	64 x 64	128	64 -> 128	9 x 9	1 x 1
L3	32 x 32	128	128 -> 128	9 x 9	1 x 1
L4	16 x 16	128	128 -> 128	7 x 7	1 x 1
L5	13 x 13	128	384 -> 384	3 x 3	1 x 1
L6	27 x 27	128	192 -> 192	5 x 5	1 x 1

In this experiment, I run each layer with 100 iterations with the same benchmark settings to collect results.

Table 6-2: The performance comparison of six convolutional layers

Convolution Layer #	Performance (Second)
L1	40.802
L2	114.82
L3	43.07
L4	5.518
L5	8.021
L6	30.75

Table 6-3: The GPU energy comparison of six convolutional layers

Convolution Layer #	GPU Average Power(Watt)	GPU energy (Joules)
L1	111.11	4534
L2	121.78	13983
L3	119.10	5130
L4	118.4	653.33
L5	120.04	962.8
L6	114.69	3527

Based on Table 6-2, 6-3, I can conclude that L1-L6 lead to similar average GPU power regardless the various kernel sizes and input sizes. Compare the layer with the largest power (L2, 12q.78 Watt) with the layer with the smallest power (L1, 111.11 Watt), L2 only leads to 9% more power.

Since each layer has more or less similar average power, the differences in total energy consumption are determined by the processing time. It is intuitive that layers with larger kernels and larger inputs would requires longer processing time, thus would consume more energy. Our initialized thinking is that the total number of kernels parameters and total size of input data determine the processing time and energy. For convolutional layer, numbers of kernels parameters are calculated using the following formula:

Kernel parameter = (kernel width x kernel height + 1) x channel x output feature map number

Total size of input data is quantified using following rule:

Total size of input data = input width x input height x batch size x output feature map

number

Based on above formulas, I calculate numbers in Table 6-4.

Table 6-4: Kernel and input size parameter of six convolutional layer

Convolution Layer #	Kernel Parameter	Input Size Parameter
L1	35136	201326592
L2	671744	67108864
L3	1343488	16777216
L4	819200	4194304
L5	1474560	8306688
L6	958464	17915904

At the first glance, it seems that there is not any obvious pattern for the energy consumption for each layer if we only consider single factor between number of kernel parameter and the size of input data. L2 consumes the most energy, but its total number of kernel parameters is smaller than L3 and L5. But if we take both factors into consideration, we can easily see that why L2 consumes more power. Although L2's number of kernel parameters is roughly half of those in L3 and L5, its input has roughly 4 times compared to L3 and L5. So the total energy consumption of L2 is more than L3 and L5.

From Table 6-4, L1 has the smallest kernel parameter, but L1 isn't the fastest one. Also L1 has similar performance with L3, but L3's kernel parameter is 38 times of L1's parameter. So it's clear the kernel parameter can't determine a convolutional layer's performance and energy consumption.

Although all six layers with the same batch size and stride, with the integrated influence of input size, kernel number, and kernel size, it's hard to find obvious principles to explain how one layer has better performance and energy saving than a another one. Because the data input size, kernel size, and kernel quantity, all play an important role in determining workload of each layer, they all affect performance and energy consumption on GPU. Thus, it's hard to figure out any patterns based on these commonly used convolutional layers unless I conduct a serial of controlled experiments. So in the following subsections, I only change one factor while fix all the other factors, which decouple these factors and is much easier to find the rules of energy consumption.

6.1.2 Data Input Size

In this subsection, I keep the kernel size and kernel quantity, but vary the size of input feature map. To best model power behavior of real-world neural network, I carefully select the layer configurations and input data size. The kernel size I use is 3 x 3, which is the most commonly seen kernel size among all ImageNet winner models; The number of input feature map is 96 and the number of output feature map is also set to 96.

Table 6-5: Convolutional layer with different data input size

Input Size	Performance (second)	GPU Average Power(Watt)	GPU energy (Joules)
13 x 13	0.937	105.59	98.94
16 x 16	1.433	108.31	155.21
27 x 27	3.939	112.01	441.21
56 x 56	17.964	115.58	2076.28
128 x 128	107.141	112.21	12022.29
224 x 224	GPU out of memory		

In the current setting, the input data size is the only changing factor. From Table 6-5, we can see that when the input data size is small, increasing input data size will increase both the GPU average power and total energy consumption. However, when the input data size is bigger than 56 x 56, the power will be stable while the energy consumption continues to increase. Figure 6-1 shows how the energy changes with input data size. The x-axis is the input size and the y-axis is the energy. I can conclude that with the data input size increased, the energy of training growing

linearly.

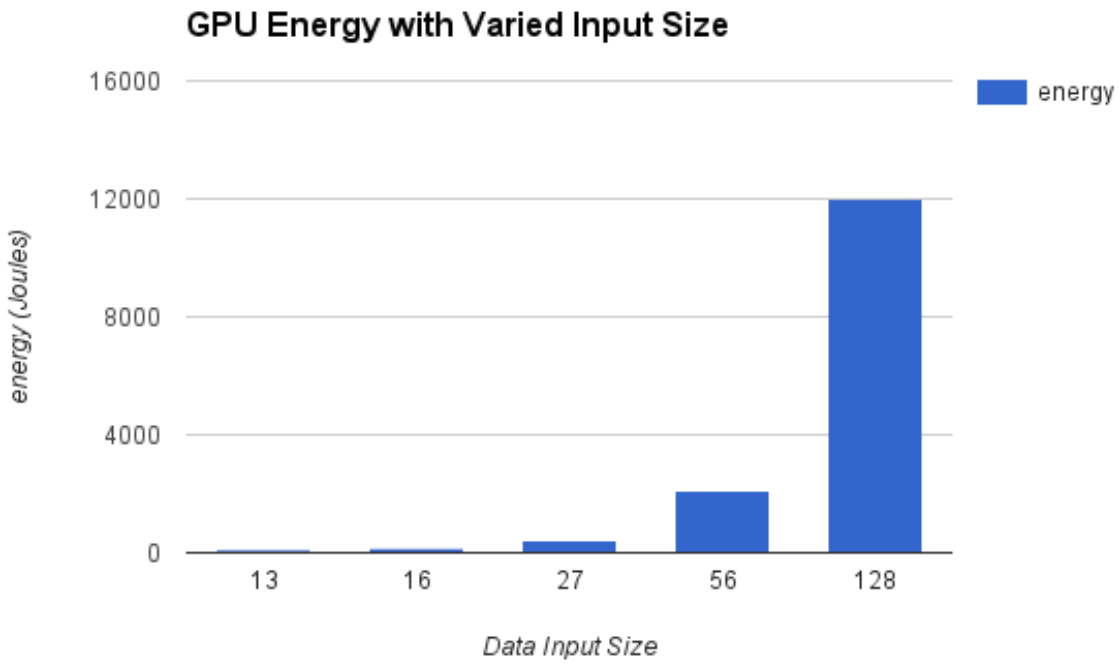

Figure 6-1: Convolutional layer GPU energy with varied data input size

6.1.3 Kernel Size

In this section, I want to analyze the influence of kernel size on energy. I configure the input data in the size of 64 x 64. The number of input feature map is set to 128 and the number of output feature map is also set to 128. So the total number of kernel for each covolutional layer is 128 x 128.

Table 6-6: Convolutional layer with different kernel size

Kernel Size	Performance (second)	GPU Average Power(Watt)	GPU energy (Joules)
3 x 3	33.38	119.2	3979.56
5 x 5	84.93	122.51	10404.77
7 x 7	149.71	124.31	18610.45
9 x 9	228.34	124.53	28435.18
11 x 11	317.76	124.55	39577.01

Kernel size is extremely important to reduce the quantity of parameters. The larger kernel size means one nerve cell can learn more features in bigger region of pictures. But too big kernel size will affect neural network's accuracy. From the Table 6-6 we can clearly see that for small kernels, increasing kernel size will lead to increase of both power and energy consumption. However, once the kernel size is relative big (e.g. 7x7), increasing kernel size will not affect power while the energy consumption always increases.

6.1.4 Kernel Number

In the last experiment, I want to see the effect on energy when total number of kernels in a convolutional layer is changing. To achieve this goal, I keep the input feature map size to 64x64, and use kernels in the size of 3x3.

Table 6-7: Convolutional layer with different kernel number

Kernel Number (input -> output)	Performance (second)	GPU Average Power(Watt)	GPU energy (Joules)
3 -> 96 (288)	1.97	114.45	225.12
64 -> 128(8192)	17.34	117.49	2037.16
128 -> 128(16384)	33.37	117.58	3923.53
128 -> 384(49152)	90.79	117.21	10640.9
384 ->384(147456)	265.7	120.62	32048.5

From Table 6-7, we can see that the performance and energy consumption of one convolutional layer is strongly related to the multiplication of input and output kernel number, except of the 3->96 layer, start from 64->128 to 384->384 these four layers, the training time consuming increase in the same speed of the multiplication of kernel number. For example, 128 -> 128 is two times of 64->128, the time also increased near two times. This phenomenon also reflects in GPU energy increment.

6.1.5 Conclusion

From the previous three sections analysis of convolution layer's performance and energy consumption in input size, kernel size, and kernel number three aspects. I summarize the impact of each factor of one convolution in Table 6-8 based on the gap between minimum and maximum performance and energy consumption with three factors' increasing.

Table 6-8: The variety impact of performance and energy of convolutional layer

	Time Increment	Energy Increment
Input Size increase 297 x	100 x	121 x
Kernel Size increase 13 x	100 x	10 x
Kernel number increase 512 x	132 x	142 x

From Table 6-8, the setting of these three factors has different influence of performance and energy increment. First of all, all three factors increasing will result in both performance and energy go up. Secondly, input size and kernel number has relatively weak affect of time and energy. Thirdly, kernel size has lower affect of energy increment but stronger impact of performance.

6.2 Fully Connected Layer

In this subsection, based on the number of the layer's parameter, I choose three fully connected layer (large, median, and small) to test their energy consumption. The quantity of parameter of one fully connected layer is calculated by following formula:

Fully connected parameter = (feature map number × input × input + 1) x output.

I want to see how the parameter quantity affect fully connected layer's performance and energy consumption.

Table 6-9: The configuration setting of three fully connected layers

	Input Size	Feature Map (Kernel Size)	Output Size	Parameter Quantity
Large	6×6	256	4096	37,752,832
Median	4096×1	1	4096	16,781,312
Small	4096×1	1	1000	4,100,096

The following table shows the performance of three fully connected layers. From Small to Median, the parameter quantity increase 4.09 times, which is similar to the 3.89 times performance drop. In the same manner, Large has 2.25 times parameter number than Median, and also 2.06 times slower than Median.

Table 6-10: The performance of three fully connected layer

	Performance (Second)
Large	2.107
Median	1.022
Small	0.263

Table 6-11: The energy consumption of three fully connected layer

	GPU Average Power (Watt)	GPU Energy (Joules)
Large	103.51	218.1
Median	93.73	95.79
Small	83.95	22.16

Table 6-11 shows, the GPU average power increase with the parameter becomes bigger. The GPU energy also shows linearly increasing with parameter.

Therefore, based on the analysis of this subsection, both performance and energy

consumption has linear relation with fully connected layer parameters.

6.3 Pooling Layer

The pooling layer is used to progressively reduce the spatial size of the representation to reduce the amount of parameters and computation in the network.

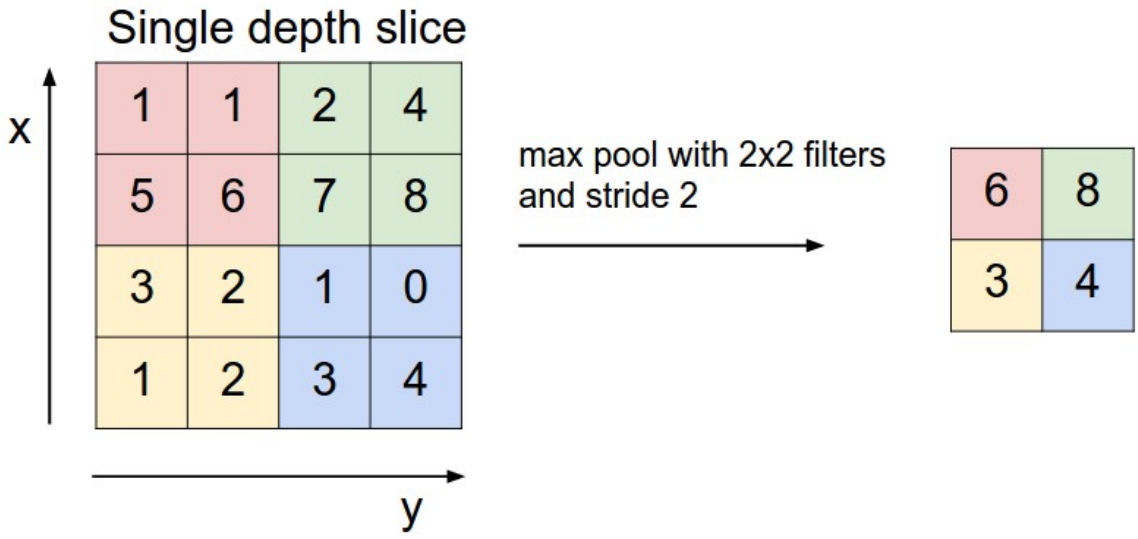

Figure 6-2: One pooling method: max pooling

For example, the figure 6-2 shows how the pooling layer with max method works. With the 2 × 2 size of filter kernel, the max-valued pixel is picked up to represent the 4 pixels area. After this process, 75% area was discarded and the depth dimension remains unchanged.

Besides max pooling, there are another popular pooling methods is average pooling, which calculate average of all pixel value inside filter as the activation in each pooling region, but not the biggest one.

In this subsection, I analyze the variables of pooling layer, I find the variables can be

from: (1) input size, (2) filter kernel size, (3) pooling method. Based on these three portions, I modify one variable while keeping other two fixed.

6.3.1 Input Size

In the first experiment, I choose three different sizes of input and keep filter kernel size as 3 × 3, and also keep pooling method as max pooling. These three pooling layers come from AlexNet, but I set the feature map all the same to maintain the impartiality of experiments.

Table 6-12: Different input size of pooling layer

	Input Size	Feature Map	Kernel Size	Pooling Method
Pooling 1	55 × 55	128	3 × 3	Max
Pooling 2	27 × 27	128	3 × 3	Max
Pooling 3	13 × 13	128	3 × 3	Max

Table 6-13: Different input size of pooling layer performance and energy

	Performance (Second)	GPU Average Power (Watt)	GPU Energy (Joules)
Pooling 1	1.504	111.43	167.6
Pooling 2	0.338	95.33	32.22
Pooling 3	0.096	66.92	6.424

From Table 6-12, 6-13, they directly show input size is positively correlated to performance, GPU average power, and energy. Based on previous chapter data, pooling layer only occupy 2% of total training time and energy of neural network, so the base energy is relatively small. Compared with the input size affect energy exponentially, the input size in pooling layer has linear relation with energy.

6.3.2 Filter Kernel Size

In this subsection, I reset the feature map as the original number of AlexNet to reflect real world neural network. Since recent imageNet winners commonly choose 3×3 or 2×2 kernel size, I design the experiments with these two size. Noted I only compare the kernel size influence between each pooling layer, so the difference of feature map will not affect experiment result.

Table 6-14: Configuration setting of three pooling layers

	Input Size	Feature Map	Kernel Size	Pooling Method
Pooling 1	55 × 55	64	3×3 / 2×2	Max
Pooling 2	27 × 27	192	3×3 / 2×2	Max
Pooling 3	13 × 13	256	3×3 / 2×2	Max

For the performance I can see in Table 6-15, the kernel size does not has big impact of performance. Although I average the time of 5 runs, the difference already in millisecond level. So I bet there is not big influence of performance between these two sizes of kernels.

Table 6-15: Different kernel size of pooling layer performance

Performance (second)		
	3×3 Kernel	2×2 Kernel
Pooling 1	0.759	0.629
Pooling 2	0.491	0.464
Pooling 3	0.178	0.184

Table 6-16: Different kernel size of pooling layer GPU energy

GPU Energy (Joules)		
	3×3 Kernel	2×2 Kernel
Pooling 1	81.32	64.15
Pooling 2	49.96	46.05
Pooling 3	14.83	14.64

Both performance and energy difference between two kinds of kernel becomes bigger,

For small input size, choosing 3×3 kernel size has not much difference of choosing 2×2 from energy aspect. For larger input size, 2 × 2 can save more time and energy.

From this subsection experiment we know, although small kernel size means the kernel sliding window need to move more time than bigger kernel to traverse the whole input data, small kernel also save time to do every down sampling than big kernel size and finally save energy.

6.3.3 Pooling Method

In this subsection, I keep input size and kernel size, and change pooling method to compare how energy efficient of these two methods in poling layer.

Table 6-17: The pooling method configuration of pooling layers

	Input Size	Feature Map	Kernel Size	Pooling Method
Pooling 1	55 × 55	64	3×3	Max/Average
Pooling 2	27 × 27	192	3×3	Max/ Average
Pooling 3	13 × 13	256	3×3	Max/ Average

Table 6-18: Pooling layers performance with various pooling method

Performance(second)		
	Max Pooling	Average Pooling
Pooling 1	0.759	1.275
Pooling 2	0.491	0.953
Pooling 3	0.178	0.346

Table 6-19: Pooling layers energy consumption with various pooling method

GPU energy (Joules)		
	Max Pooling	Average Pooling
Pooling 1	81.32	112.57
Pooling 2	49.96	81.79
Pooling 3	14.83	26.3

From table 6-18, 6-19, max pooling have better performance and also more energy efficiency than average pooling in each layer. Since Max pooling only pick the max value of sliding window, which need less calculation than calculating average of all the pixel value that helping to reduce the workload of GPU, then save more energy. In here, with the input size increasing, the ratio of average pooling energy and max pooling becomes smaller from 1.77 to 1.38.

6.4 Conclusion

In this chapter, I systematically analysis the major layers. Then I discuss main factors of each layers influence of layer performance and energy consumption. In convolutional layer, I discuss three different factor's contribution of layer. For fully connected layer, one factor used to modify to see how it affect the layer. In pooling layer, I visualize three kinds of setting 's effect of pooling. Based on all the results, we can see how single factors affect one layer, then affect the whole neural network in both performance and energy consumption field.

7. HARDWARE TUNING

In previous chapters, I analyze how the frameworks and neural network itself will affect energy consumption of training process. But I also want to save more energy by tuning hardware. Because most of the training job is responsible for GPU, so this tuning mainly focus on GPU tuning. In our work, I experimentally study the impacts of DVFS about neural network training performance and energy efficiency in Nvidia K20m GPU. Besides, I compare the K20m and Titan X GPU performance and energy consumption difference with the same benchmark.

7.1 DVFS

Dynamic Voltage and Frequency Scaling (DVFS) is an advanced power-saving technology whose aim is to lower a component's power state while still meeting the performance requirement of the running workload [Ge, 2013]. The K20m GPU I used is Nvidia Kepler architecture. It includes total 2496 CUDA cores, six 64-bit memory controllers, and 5 GB global memory. The GPU cores and memory are capable of DVFS and support the clock frequency shown in Table 7-1.

Table 7-1: Support memory frequency and core frequency options in K20m GPU

Memory Frequency (MHz)	GPU Core Frequency(MHz)
2600	758 705 666 640 614
324	324

To control the clock frequency of the K20m GPU, I used *nvidia-smi* utility or Nvidia System Management Interface. This interface is a command line utility that uses the Nvidia Management Library (NVML) for management and control of Nvidia devices [Price, 2014].

With the help of GPU boost commands on k20m GPU, I test six experiments which include all the pair combinations from table 7-1. The benchmark for this experiment is based on previous Convent Benchmark but with 100 iteration. The experiment performance results shows in Figure 7-1 with 100 times forward and backward propagation.

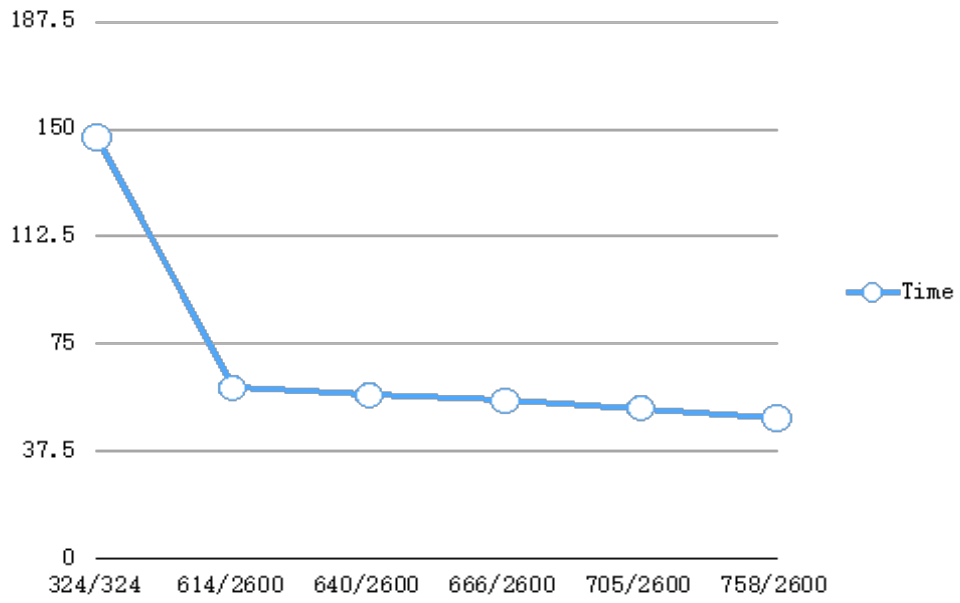

Figure 7-1: Performance impact of DVFS in K20m

Figure 7-1 shows how big influence of GPU frequency affect the performance. The best performance is 48.95 second with core speed with combination of 758 MHz and memory with 2600 MHz, which is 3 times faster than GPU with 324 MHz core speed and 324 MHz memory speed.

In addition, when the memory frequency is fixed in 2600 MHz, the performance going better with the increasing of GPU core frequency. But this increase is limited. When core speed from 614 MHz to Max 758 MHz, the time saving is 10.57 second. The average time saving prompted 2.6 second each change when memory fixed in 2600 MHz.

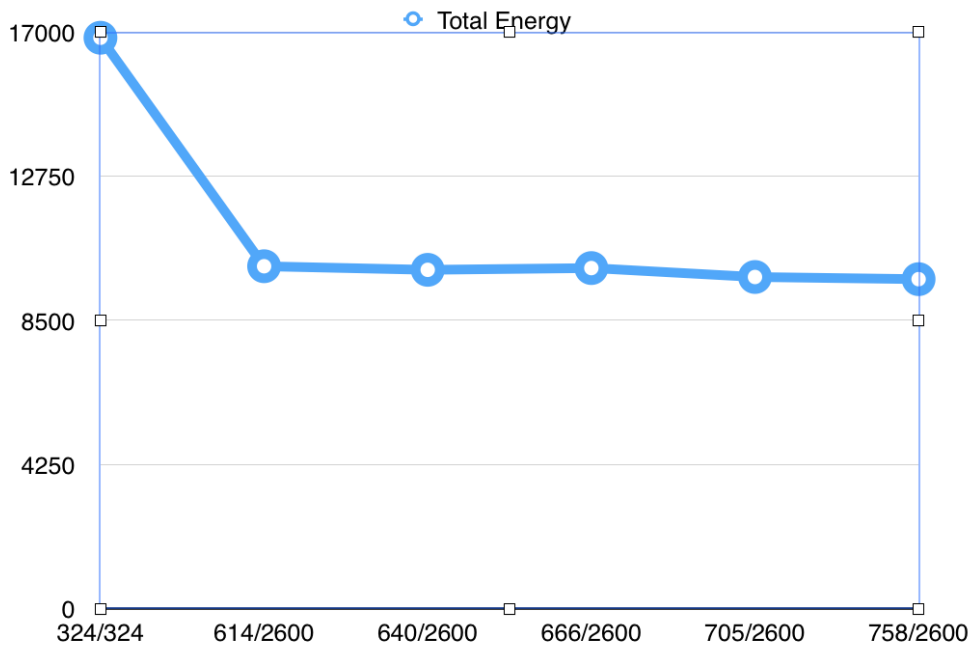

Figure 7-2: Total system energy impact of DVFS

The Figure 7-2 shows the system energy of the whole training process. The system energy in here includes CPU and GPU energy consumption. From our data, CPU portion of average power doesn't affected too much during GPU tuning and keep in 63 ~ 67 Watt. So that means when GPU can run faster with higher frequency, CPU energy portion will drop certainly. Figure7-3 shows the average power of GPU under different frequency pair.

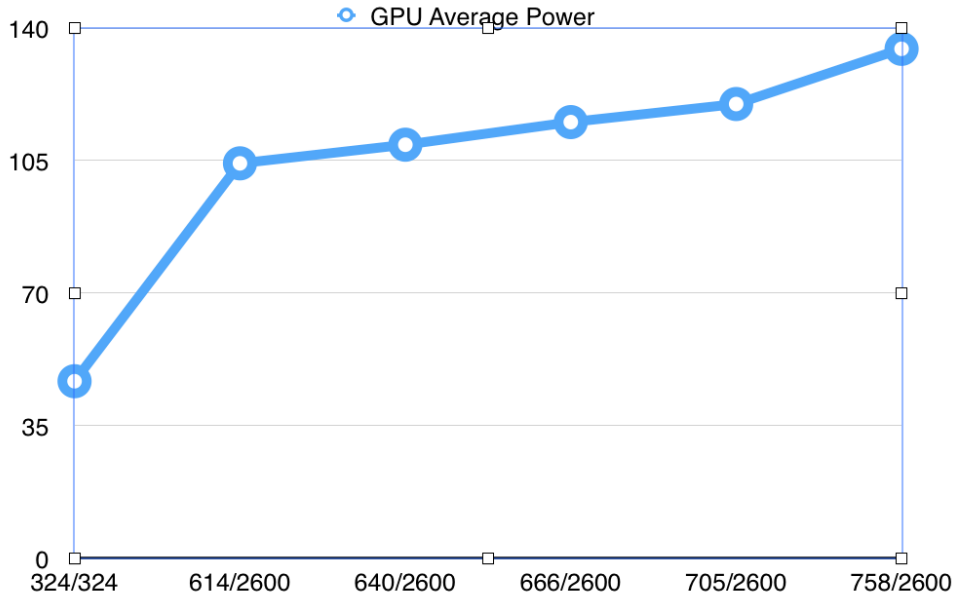

Figure 7-3: GPU Average Power with the impact of DVFS

In Figure 7-3, both memory and core frequency with 324 MHz has 46.75 Watt average power, which slightly higher than 46-watt idle power of K20m. It means the utilization of GPU is very low under this frequency pair. Then the GPU power go up with the core frequency increased and arrives the top when core speed is 758 MHz with 134.5 Watt.

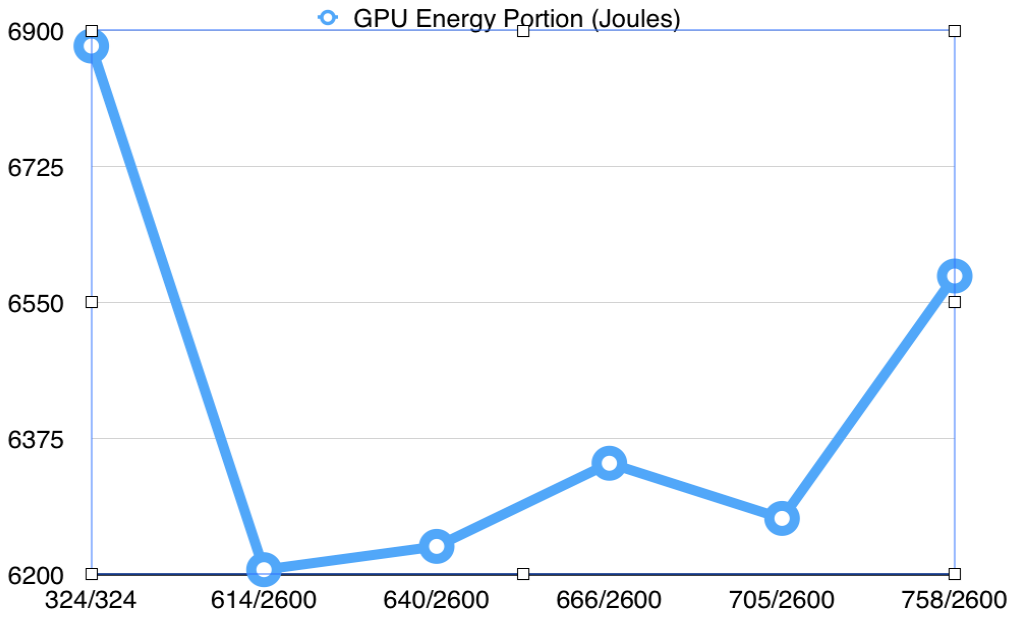

Figure 7-4: GPU energy portion with the impact of DVFS

Compared with Figure 7-2 (total energy) with Figure 7-4 (GPU energy), I find that the most energy saving comes from CPU but not GPU. Even though GPU run faster with DVFS tuning, the average power goes high, so the energy of GPU doesn't improve much. From Figure 7-4, when K20m GPU is in 614/2600 pair, it achieves the best GPU energy efficiency with 6206 Joules, then with the faster core speed, K20m consume more energy generally. When K20m GPU with 758/2600 frequency, the second highest GPU energy consumption can not stop this pair becomes the best system energy efficiency option because of the saving in CPU make up the extra lost in GPU.

To sum up, DVFS has pros (better performance) and cons (higher average power) of GPU running in neural network training, but DVFS helps GPU runs faster, CPU can save more energy to make up the shortage of DVFS, the total energy can drop slightly.

7.2 K20m and Titan X GPU Comparison

When my work going to the end, I got Titan X GPU from Nvidia's donation, so I add this subsection to make a brief comparison of Titan X and K20m GPU's performance and energy efficiency with the same benchmark. Titan X is the newest and strongest GPU in current market. From table 7-2, Titan X has 3072 CUDA cores, and a total 12 GB global memory. Also, Titan X is MaxWell architecture that each CUDA core is stronger than cores with Kepler architecture of K20m. Based on such specifications, I can assume that Titan X has better performance than K20m, but I want to quantify the result and I also interested in whether Titan X can be more energy efficient than K20m GPU.

Table 7-2: Titan X and K20m hardware comparison [Nvidia, 2012; Nvidia 2014]

	Chip	CUDA Core	single-precision floating point performance	Max Power
Titan X	GM200 (MaxWell)	3072	7 TFLOPS	250 W
K20m	GK100 (Kepler)	2496	3.52 TFLOPS	225 W

I used the Convent Benchmark with 100 iterations and AlexNet running in Caffe framework to compare two GPUs.

Table 7-3: Titan X GPU and K20m GPU comparison

	Time	CPU Average Power	GPU Average Power	Total Energy
Titan X	13.283	63.14	213.3	3671.95
K20m	52.28	59.61	120.07	9409.35

From Table 7-3, Titan X has 3.9 times faster than K20m. With almost two time higher

average GPU power then K20m, Titan X still can achieve 2.56 times energy efficiency.

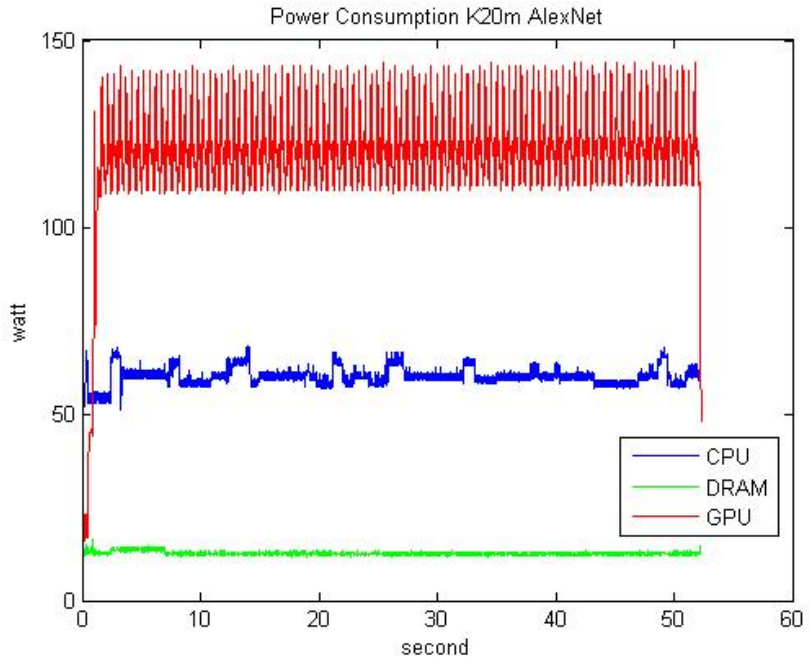

Figure 7-5: K20m energy curve of Caffe with AlexNet

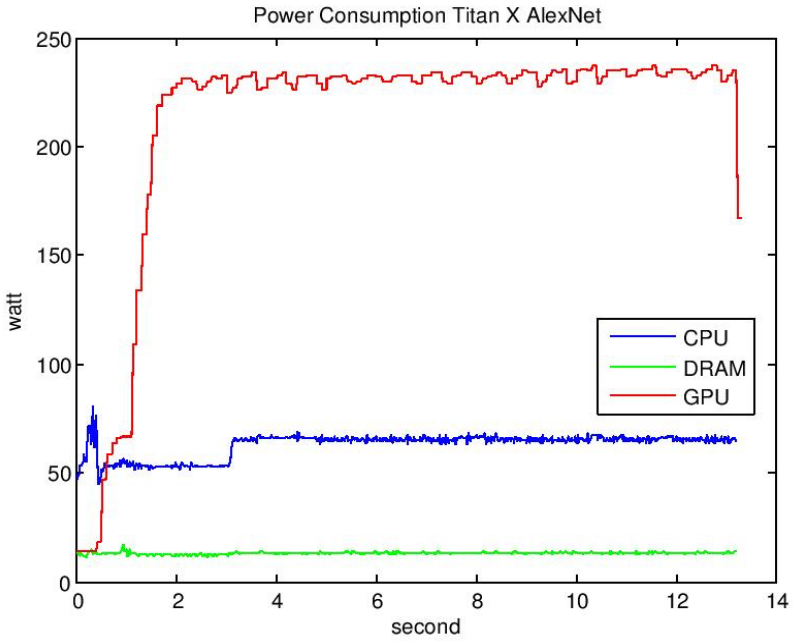

Figure 7-6: Titan X energy curve of Caffe with AlexNet

From Figure 7-5, 7-6, compared with K20m, Titan X shows more stable power fluctuation when doing recurrent iteration training. In addition, Titan X doesn't drop the power too much between iterations to keep in high power level during the whole training process, which save plenty of time to cool down and warm up hardware.

7.3 Conclusion

In this chapter, I compare the DVFS tuning influence of neural network training in K20m and also compare energy consumption between two GPUs. DVFS with high core and memory frequency can help current GPU have better performance with more cost of GPU energy, and lower total energy. This strategy can be used in other GPU heavy workload but CPU low workload application. Titan X achieved both performance and energy saving goals has been verified in neural network training. With the comparison of both Titan X and K20m energy curve, Titan x has more stable curve and maintain in high power stage under the same neural network, benchmark, and training framework of K20m.

8. CONCLUSION

In this paper, I analyze many aspects related to neural network training from performance and energy efficiency, includes hardware, framework, library, neural network level, layer level, even data input size. Following this analysis strategy can cover all potential factors of neural network. With massive comparison in various conditions, I find the most energy efficient training framework and library, also based on this framework, Caffe, I expand more details about neural network, how each factor affect one specific layer, and finally affect the whole neural network. Since now researcher plan to implement deeper neural network models with more complicated layers, I think my research can help them to avoid high energy consuming and low performance factors and provide them ideas to design energy efficient layers, energy efficient framework, and finally green neural network.

9. FUTURE WORK

In the future, I will expand our experiments in Titan X with bigger memory to handle deeper neural network like GoolgeNet and Oxford Net. In addition, since some advanced frameworks didn’t support Kepler Architecture, Titan X with new Maxwell architecture will support more frameworks such as Nirvana. With wider range of experiments, I will find the energy consumption model of neural network and design a green networks based on the best energy saving principles from network wise and layer wise.

REFERENCES

Rosenblatt, F. (1958). The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, *65*(6), 386.

Hinton, G. E., Dayan, P., Frey, B. J., & Neal, R. M. (1995). The" wake-sleep" algorithm for unsupervised neural networks. *Science*, *268*(5214), 1158-1161.

Hochreiter, S. (1998). The vanishing gradient problem during learning recurrent neural nets and problem solutions. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, *6*(02), 107-116.

Deep Learning Concept in Wiki *https://en.wikipedia.org/wiki/Deep_learning*

CNN wiki, 2016: *https://en.wikipedia.org/wiki/Convolutional_neural_network*

Convent-Benchmark (2015) Github Repository:

https://github.com/soumith/convnet-benchmarks

Model Zoo, 2015: *https://github.com/BVLC/caffe/wiki/Model-Zoo*

cuDNN 2013 : *https://developer.nvidia.com/cudnn*

MXNet 2015, Github Repository: https://github.com/dmlc/mxnet

Intel RAPL:*https://01.org/rapl-power-meter*

Chen, T., Li, M., Li, Y., Lin, M., Wang, N., Wang, M., ... & Zhang, Z. (2015). MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems. *arXiv preprint arXiv:1512.01274*.

He, K., Zhang, X., Ren, S., & Sun, J. (2015). Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In*Proceedings of the IEEE International Conference on Computer Vision* (pp. 1026-1034).

Caffe Deep Learning Framework: *http://caffe.berkeleyvision.org*

Ge, R., Vogt, R., Majumder, J., Alam, A., Burtscher, M., & Zong, Z. (2013, October). Effects of dynamic voltage and frequency scaling on a k20 gpu. In *Parallel Processing (ICPP), 2013 42nd International Conference on* (pp. 826-833). IEEE.

Price, D. C., Clark, M. A., Barsdell, B. R., Babich, R., & Greenhill, L. J. (2015). Optimizing performance-per-watt on GPUs in high performance computing. *Computer Science-Research and Development*, 1-9.

Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems* (pp. 1097-1105).

Sermanet, P., Eigen, D., Zhang, X., Mathieu, M., Fergus, R., & LeCun, Y. (2013). Overfeat: Integrated recognition, localization and detection using convolutional networks. *arXiv preprint arXiv:1312.6229*.

NVIDIA (2011). Nvidia management library. *http://developer.nvidia.com/nvidia-management-library-nvml.*

NVIDIA (2012). Fermi compute architecture whitepaper.

NVIDIA(2013). K20m Accelerator, *http://www.nvidia.com/content/pdf/kepler/tesla-k20-passive-bd-06455-001-v07.pdf*

NVIDIA (2014). Titan X User Guide http://www.nvidia.com/content/geforce-gtx/GTX_TITAN_X_User_Guide.pdf

http://www.nvidia.com/content/PDF/fermi white papers/ NVIDIA Fermi Compute Architecture Whitepaper.pdf.

CUDA (2013). Cuda SDK. *https://developer.nvidia.com/cuda-toolkit*.

Bahrampour, S., Ramakrishnan, N., Schot. Lukas., Shah, & Mohak.(2016). Comparative Study of Deep Learning Software Frameworks *arXiv preprint arXiv: 1511.06435*

Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., ... & Ghemawat, S. (2016). TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems. *arXiv preprint arXiv:1603.04467*.

Russakovsky, O., Deng, J., Su, H., Krause, J., Satheesh, S., Ma, S., ... & Berg, A. C. (2015). Imagenet large scale visual recognition challenge. *International Journal of Computer Vision*, *115*(3), 211-252.

Mei, X., Yung, L. S., Zhao, K., & Chu, X. (2013, November). A measurement study of GPU DVFS on energy conservation. In *Proceedings of the Workshop on Power-Aware Computing and Systems* (p. 10). ACM.

Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE, november 1998*

Taigman, Y., Yang, M., Ranzato, M. A., & Wolf, L. (2014). Deepface: Closing the gap to human-level performance in face verification. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (pp. 1701-1708).

Tompson, J. J., Jain, A., LeCun, Y., & Bregler, C. (2014). Joint training of a convolutional network and a graphical model for human pose estimation. In*Advances in neural information processing systems* (pp. 1799-1807).

Lebret, R., Pinheiro, P. O., & Collobert, R. (2015). Phrase-based image captioning. *arXiv preprint arXiv:1502.03671*.

Li, D., & Becchi, M. (2012, November). Multiple Pairwise Sequence Alignments with the Needleman-Wunsch Algorithm on GPU. In *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:* (pp. 1471-1472). IEEE.

Truong, H., Li, D., Sajjapongse, K., Conant, G., & Becchi, M. (2014). Large-scale pairwise alignments on GPU clusters: exploring the implementation space. *Journal of Signal Processing Systems*, *77*(1-2), 131-149.

Wu, H, Li, D., & Becchi, M. (2016, May). Deploying graph algorithms on gpus: An adaptive solution. In *Parallel & Distributed Processing (IPDPS), 2016 IEEE 30th International Symposium on*. IEEE.

Li, D., & Becchi, M. (2013, May). Deploying graph algorithms on gpus: An adaptive solution. In *Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on* (pp. 1013-1024). IEEE.

Li, D., Wu, H., & Becchi, M. (2015, February). Exploiting Dynamic Parallelism to Efficiently Support Irregular Nested Loops on GPUs. In *Proceedings of the 2015 International Workshop on Code Optimisation for Multi and Many Cores* (p. 5). ACM.