MASTER

An efficient end-to-end object detection pipeline on GPU using CUDA

Wang, X.

Award date:
2019

Link to publication
Eindhoven University of Technology

Master Thesis

An Efficient End-to-End Object Detection Pipeline on GPU Using CUDA

Author: Xiaowei Wang
Supervisor: Dr. Andrei Jalba

A thesis submitted in fulfillment of the requirements for the degree of Master of Science in the Department of Mathematics and Computer Science

April 17, 2019
Abstract

Department of Mathematics and Computer Science

Master of Science

An Efficient End-to-End Object Detection Pipeline on GPU Using CUDA

by Xiaowei Wang

Object detection has been the most important task in the field of computer vision. As researchers have proposed many efficient object detection methods, the deployment and application of these methods in an engineering perspective has become a problem. We propose to implement an efficient object detection pipeline on the GPU using CUDA. The pipeline offers an end-to-end solution which can take images as input and output bounding boxes to mark the detected objects. We use PVANet as our object detection method, concatenated with a Non-maximum Suppression (NMS) algorithm to eliminate redundant bounding boxes. We focus on minimize the program latency and footprint by using various CUDA optimization techniques. A new method to implement the NMS algorithm is proposed to improve the algorithm efficiency on the GPU.
# Contents

Abstract iii

1 Introduction 1
  1.1 Context ................................................. 1
  1.2 Motivation ............................................. 2
  1.3 Contribution ......................................... 2

2 Related Work 3
  2.1 Object detection and deep learning ...................... 3
  2.2 Non-maximum Suppression .............................. 6

3 Overview 9
  3.1 CUDA Overview .......................................... 9
    3.1.1 Design of CUDA & GPU ............................. 9
    SIMD & SIMT ........................................... 9
    CUDA Programming Model ............................ 12
    Memory Model ........................................ 12
  3.2 Object Detection Pipeline Overview .................... 14
  3.3 Overview of the PVANet Network ....................... 15
    3.3.1 Convolutional Neural Networks .................... 15
    3.3.2 Pooling and Decovolution ........................ 16
    3.3.3 Detailed Design of PVANet ....................... 17
    C. ReLU .............................................. 17
    Inception ............................................ 19
    Hypernet ............................................ 20
    RPN and ROI Pooling ................................ 20
  3.4 Overview of Non-Maximum Suppression .................. 20

4 Implementation and Optimization 23
  4.1 PVANet .................................................. 23
    4.1.1 TensorRT .......................................... 23
  4.1.2 Construction of PVANet in TensorRT ................ 24
    4.1.3 Customized layers ................................ 25
    Simple Factory Pattern ................................ 26
    Reshape .............................................. 26
    C. ReLU .............................................. 26
  4.2 Non-Maximum Suppression .............................. 27
    4.2.1 Naive Method ..................................... 28
    4.2.2 Map-reduce Method ................................ 28
    4.2.3 Warp-wise Method ................................ 31
  4.3 Optimization .......................................... 33
    4.3.1 Assess ............................................ 34
    4.3.2 Improve .......................................... 34
Memory bound .................................................. 35
Instruction bound .............................................. 37
Latency bound .................................................. 38
4.3.3 Other Techniques for Warp-wise Method ................. 38

5 Results .......................................................... 43
5.1 Results of PVANet ............................................. 43
  5.1.1 Accuracy .................................................. 43
  5.1.2 Time ...................................................... 44
    Customized layers performance ............................ 45
5.2 Results of Non-Maximum Suppression ......................... 46

6 Conclusions .................................................... 51

Bibliography ..................................................... 53
Chapter 1

Introduction

1.1 Context

Object detection refers to a computer vision technology which identifies real-world objects in an image or video sequence. Humans have a natural gift for recognizing objects: a person can identify an object with little effort, despite the fact that an object can have different appearances from different view points, scales, translations or rotations. Sometimes an object is still recognizable even if it is partially obstructed. Our ultimate goal is to perform an objection detection with human-precision as well as computer-speed on a modern computing system. This task still remains a challenge in the field of computer vision.

Inspired by our biological nervous system, Artificial Neural Networks (ANNs) have been proposed to teach computers to learn just like humans. In analogy to biological neural networks, artificial neural networks are structured by individual “neurons” (namely perceptrons) with weights, which model the connection of neurons. The technique of deciding each neuron’s weight is called training, and using a trained neural network to run a computation task is called inference. An ANN is composed of at least two layers, input and output layer, where each layer may contain several perceptrons. The layer between the input and output layer is called “hidden layer”. A deep neural network is defined as an ANN with multiple hidden layers. The method of training deep neural networks to solve problems is called deep learning.

In the modern computer graphics pipeline, two stages are of key importance: rasterization and shading. Rasterization can convert a 3-dimensional object described in a geometric form into a raster image. Shading, on the other hand, depicts the level of darkness in each part of an image. These two techniques combined decide the content and quality of the final output image that we see, and both of them are computationally intensive as the pixel numbers of a high-definition image can be huge. In 1999, Nvidia released the world’s first Graphics Processing Unit (GPU), which is a specialized electronic circuit aiming at accelerating computer graphics algorithms. Different with classical CPU architectures (e.g. x86), one GPU usually has hundreds or even thousands of computing cores. This design is a result of the fact that modern graphics algorithms are extremely suitable for parallelization as each pixel of an image can be computed independently. Since 2005, GPU designers have realized that both rasterization and shading can be reduced to a problem of parallel computing like most of the other computation tasks, thus GPUs have potential for general computing. In 2007, Nvidia launched its parallel computing platform Compute Unified Device Architecture (CUDA) and introduced its General Purpose Graphics Processing Unit (GPGPU). CUDA allows CPUs to be liberated from the
heavy burden of massive computation, such as large matrix operation as well as fast Fourier/wavelet transformations; GPUs are typically ten times even hundreds times faster than CPUs.

1.2 Motivation

With the rapid development of object detection, the commercial value of object detection has been found in many areas, such as massive security surveillance and autonomous driving. Many commercial companies have launched their products or services based on object detection and the market keeps growing. However, an efficient and reliable object detection system is yet to be found on the market. To solve this problem, we decided to build a platform specialized in object detection. Among all the methods of object detection, PVANet caught our eye. PVANet is derived from Faster R-CNN while shortening computation time without sacrificing accuracy. After studying PVANet in depth, we decided to implement the object detection pipeline based on PVANet using Nvidia’s TensorRT [24], which is a platform for high-performance deep learning inference. The advantage of TensorRT is that it provides native support for many popular layers in Deep Neuron Networks (DNNs), such as convolution layer, activation layer, etc., which are essentially CUDA kernels fully optimized by GPU experts at Nvidia. Besides, for customized layers, TensorRT provides a plugin API to allow users to define their own layers.

1.3 Contribution

Our contribution in this project is mainly composed of two parts: implementing an objection detection pipeline based on PVANet using TensorRT, as well as the non-maximum suppression (NMS) algorithm, and optimizing them to the achievable best performance, which is demonstrated through benchmarks and tests. To efficiently implement NMS on GPU in a parallel form, two methods are brought forward. Compared to results in the PVANet paper, our work achieves a speedup ratio of 1.6. Compared to a naive implementation of NMS, our method is almost twice fast.
Chapter 2

Related Work

2.1 Object detection and deep learning

Traditional object detection methods based on manual feature extraction have gone into stagnation since 2010. The traditional method is composed of three stages: region selection, feature extraction and classification. There are two problems coming from this pipeline: a region selecting strategy with proper performance, as well as a simple time complexity, is yet to be found, and the robustness of manual feature extraction cannot be guaranteed. A new method [16] based on Convolutional Neural Networks (CNNs) was introduced in 2012, which started a new era to object detection. CNN offered great improvement on both of the problems, especially for the methods based on region proposal, namely R-CNN. The structure of R-CNN is shown in Fig. 2.1. R-CNN is commonly considered as the ground breaking work for introducing CNNs to object detection for the first time. While this method still has its limitation in today’s view, it increased the detection rate on PASCAL VOC dataset[3] from 35.1% to 53.7%. Since then, the family of objection detection developed into two branches: “one-stage” and “two-stage”. The one-stage methods try to perform regression/classification directly on images, while two-stage methods first generate region proposals, extract features and then perform regression. We will start with two-stage methods, to which the method used in this project belongs.

The whole two-step branch actually originates from R-CNN. Sliding windows are heavily used in traditional objection detection. It is essentially an exhaustive search, whose disadvantages are obvious: too many computations. Also, a sliding window cannot fit in every scale, resulting in a low accuracy on object location. To overcome this problem, R-CNN used a heuristic method called selective search[30]. The idea is to segment an image into small blocks, then apply greedy strategy to combine two neighbour blocks based on their similarity, until only one block is left. These new blocks generated from the process are called region proposals. R-CNN extracts around 2000 region proposals then compute each region’s feature using CNN. These features form a feature map and go through a fully-connected (FC) layer to be classified. Compared to traditional methods, R-CNN greatly improved detection speed as well as accuracy, however, it still has limitations. First, R-CNN only supports fixed-size images. Second, each region much be convoluted (in the CNN) and stored, which requires large memory space and might cause redundancy if regions overlap with each other. One year after R-CNN, SPP-Net[10] was proposed to break these two limitations. As shown in Fig. 2.2, instead of performing convolution on each region, SPP-Net performs convolution on the entire image and then extracts region proposals, i.e., put the CNN layer in front of region proposal extraction. This “simple” modification not only reduces memory space but also speeds up training speed. In addition, a special pooling layer is defined before the fully connected (FC)
layer in SPP-Net, which scale feature maps of arbitrary sizes to a fixed size. In this manner, SPP-Net can handle input images of arbitrary dimensions. The author of R-CNN offered his own improvement on R-CNN as well, called Fast R-CNN[6], which is shown in Fig. 2.3. As its name indicates, Fast R-CNN performed speed optimization on R-CNN. This optimization is done by replacing the original sequential structure with a parallel structure. In R-CNN, before classification the network first determines whether there are objects inside the candidate region, while in Fast R-CNN, the determination and classification run in parallel. Besides, Fast R-CNN also absorbed the design of SPP-Net, improving both inference speed and accuracy.

Until this point, all two-stage methods are of a mixed-structure: a heuristic region proposal algorithm concatenated by a CNN feature extraction. This structure leaves us two problems:

- The robustness of a heuristic region proposal algorithm is random, while two-stage methods rely heavily on this robustness, making performance unpredictable
- The region proposal algorithm runs on CPU, data transfer between CPU and GPU will cost efficiency

Having realized these problems, the author of Faster R-CNN[28] proposed the concept of Region Proposal Networks (RPNs), that is training a neural network to generate region proposals. A neural network can learn features semantically from different levels of abstraction, increasing the robustness of region proposal. Embedding RPN into the original network can also reduce the number of parameters as well as training and inference time. Besides, Faster R-CNN introduced the concept of anchor. Each sliding window in the feature map will generate \( k \) anchors, then determine whether each anchor covers the foreground or the background. At this point, two-stage methods have been unified by neural networks, and entire pipeline runs on GPUs. The initial inference time of a single image using R-CNN was 50 seconds, and eventually dropped to 200 milliseconds using Faster R-CNN, which is about 6 frames per second with high accuracy.

The objection detection pipeline described in this project is based on PVANet, which is derived from Faster R-CNN. The detailed structure of PVANet will be discussed in later chapters. The author of PVANet provided a Python implementation available at [11].

The idea behind one-stage methods is rather simple even “violent”: generating bounding boxes and classes of objects by applying regression directly on an image. One-stage methods essentially rely on the outstanding performance of the CNN classifier, which locates the object through iterations as shown in Figure 2.4. The advantage of one-stage methods is their high speed, at the cost of detection rate. Two representative works of one-stage methods are YOLO [27] and SSD [17].

The basic structure of YOLO is listed in Figure 2.5 which consists of three parts: a preprocessor to resize input images to a fixes size, a CNN to extract features, and a FC layer to apply classification/regression. To speed up the detection procedure, each image is segmented into \( S \times S \) blocks after resizing, and iterations mentioned in the previous paragraph are applied on each block in parallel. The FC layer outputs a bounding box info and a confidence level for each block, indicating the object’s
2.1. Object detection and deep learning

**R-CNN: Regions with CNN features**

![Diagram of R-CNN](image)

**Figure 2.1:** The structure of R-CNN (from [7]). 1. Preprocess the input image; 2. Extract around 2000 region proposals from the input and wrap the regions to a fixed size; 3. Compute CNN features of the regions; 4. Classify the regions based on features.

**SPP-Net**

![Diagram of SPP-Net](image)

**Figure 2.2:** The structure of SPP-Net (from [10]). Instead of generating region proposals, SPP-Net apply convolution layers directly on the input image. The spatial pyramid pooling layer converts the feature map of any size to a fixed-length representation, thus SPP-Net can handle input images of any size. The fully-connected layers are responsible for classifying the features.
predicted name and probability. The final results are processed by non-maximum suppression (NMS) to eliminate redundant results.

In theory, YOLO can achieve a quite fast speed, however in practice, image blocks might be too coarse for a fine detection, leaving small objects undetected. SSD combined the idea of YOLO with the concept of anchor in Faster R-CNN, which achieves similar performance to Faster R-CNN.

There is not a single metric which can tell whether two-step methods are better than one-step methods or vice versa. Both branches try to trade-off between speed and accuracy with different inclinations. Two-step methods take accuracy for the first priority rather than speed, while one-step methods recon speed is more important. Two-step methods represent state-of-the-art accuracy in object detection, while one-step methods are more suitable in real-time scenarios such as video surveillance or high-speed motion capture.

2.2 Non-maximum Suppression

Apart from PVANet, a non-maximum suppression (NMS) algorithm is also required as the final stage of the pipeline: processing the unwanted redundant outputs. Though it might be a rather simple algorithm, NMS has been widely used in many key aspects of computer vision[1, 4, 7]. As it is a necessary stage in many computer vision pipelines, well-known deep learning frameworks have long been considering how to implement NMS in parallel efficiently. However, by now only a naive version is provided [2]. There are also research works focusing on efficient implementation methods [20, 26]. Existing implementations of NMS lack optimization either in an algorithm level or in an engineering level. We use [26] as state-of-the-art of NMS in this project, which was also implemented using CUDA.
2.2. Non-maximum Suppression

Figure 2.4: Iterations of a one-stage classifier. The iterations start from the outer blue box, which has the same size of the image. The box shrinks in each step and finally becomes the red box in the center, which marks the object’s location.

Figure 2.5: Basic Structure of YOLO. 1. All input images are resized to a fixed size; 2. A CNN is applied on the resized images to directly generate results; 3. NMS is applied on the results generated by CNN to eliminate redundant candidates. Unlike two-step methods, there are no separate feature extraction and classification steps.
Chapter 3

Overview

3.1 CUDA Overview

3.1.1 Design of CUDA & GPU

As a general purpose processor, GPGPU has much in common with CPU. We will introduce the design of CUDA and GPU in comparison with CPU for better comprehension. Before the idea of using GPU for general computing, GPUs were specialized hardware which only runs computer graphics algorithms. Through decades of development, a mature and stable rendering pipeline for computer graphics has taken shape. The graphics used in D3D11 by Microsoft is shown as Fig. 3.1. In this pipeline, shaders are of the most importance. At the algorithm level, shading is highly parallel. In the hardware level, shaders are described as programmable processor cores. Nowadays these programmable cores are named CUDA cores.

SIMD & SIMT

Unlike modern multi-core CPUs, GPUs are usually manycore processors, with hundreds even thousands of cores per chip. For instance, as shown in Figure 3.2, an Nvidia GTX 1080 graphics card with a GP104 GPU has 2560 cores while an Intel i7-8700k has 6 cores. These 2560 cores are located in 20 Stream Multiprocessors (SM). Compared to CUDA cores, SMs are more like CPU cores - SMs are equipped with their own register file, cache, scheduler, and dispatch unit. This architectural difference directly reflects on the computation model. The basic model of CUDA is single instruction, multiple thread (SIMT), which is quite similar to single instruction, multiple data (SIMD) in Flynn’s taxonomy[5]. Figure 3.4 depicts both SIMD and SIMT structure. SIMD describes a class of computers which are capable of performing the same instruction on multiple data entries simultaneously in one thread, while SIMT describes a similar computer architecture where different threads perform same instruction on multiple data. To be more specific, in SIMD, each core usually has more than one processing unit and these processing units (PU) form a vector unit (VU). When a thread fetches an instruction, the corresponding VU will be activated, allowing each PU to process one data entry under the same instruction (load, store, add, etc.). In CUDA, each core has only one processing unit - one thread can only perform an operation on one data entry at the same time. However, as mentioned just above, the GPU is a manycore processor, thus one GPU can run many threads simultaneously. In other words, GPU guarantees enough parallelism by launching many threads at the same time.
Figure 3.1: DirectX rendering pipeline (From [19]). The shader stages and rasterizer stage are major components of the pipeline. These stages use highly parallel algorithms.
Figure 3.2: Architecture of the GP104 Stream Multiprocessor. Each SM contains 64 CUDA cores (only 4 is shown in this figure) and each CUDA core has 32 cores which can run 32 threads simultaneously. Each CUDA core is also equipped with 8 load/store (LD/ST) units, 8 special function units (SFUs) and 16384 registers. The L1 cache and shared memory (94KB) is shared by all CUDA cores within a SM.
Chapter 3. Overview

CUDA Programming Model

CUDA is a well-layered software stack which exposes the GPU as a parallel computing platform at different levels. The programming model of CUDA is based on a heterogeneous system architecture: the CPU acts as a host which launches programs on GPU (device) and transfers data from main memory to GPU memory. This processing flow is shown in Figure 3.3. CUDA C is an extension of the C programming language, which allows programmers to define C functions. A function that runs on GPU is called a “kernel”. A kernel uses a <<<...>>> syntax to denote the thread organizations. Each thread, block as well as grid is identified by an index which can be called inside the kernel. CUDA uses “Compute Capability” (CC) to identify the features/instructions supported by various GPU hardware, which is also known as “SM version”. The complete list and specifications of Compute Capability can be found at [21, 22]. The key component to glue hardware and software together is NVCC, the compiler for CUDA C. NVCC can compile both host and device codes and generate a fat binary that can be executed on GPUs with different SM versions.

For instance, GP104 with Compute Capability 6.1 can have 57344 (2048 per SM * 28 SMs) resident threads. CUDA uses a hierarchical model to index threads in multiple dimensions. Threads are organized in “blocks”, a block can be one dimensional, two dimensional or three dimensional. Blocks compose “grids”, which can also have up to three dimensions. The basic scheduling unit of threads is a “warp”. A warp consists of 32 threads - a SM cannot switch threads one by one, but 32 each time. One SM can have up to 4 warps running at the same time.

Memory Model

An x86 CPU uses DRAM as its main memory, so does a GPU. In alignment with the latency oriented architecture, x86 uses DDR SDRAM which has a low latency. GPU, on the other hand, uses Graphics DDR (GDDR) or High Bandwidth Memory (HBM) which has a significant higher throughput than DDR but also higher latency. To overcome this hardware shortage, Nvidia designed a complex hierarchical memory model, including DRAM, L2 cache, shared memory, L1 cache and registers. We will discuss these different types of memory in the order of speed. The closer the memory is to the CUDA cores, the faster the accessing speed will be, and the smaller
3.1. CUDA Overview

**FIGURE 3.4:** Left: SIMD; Right: SIMT. In SIMD, an instruction can process multiple data entries by calling multiple PUs at the same time. In SIMT, the instruction is dispatched to multiple threads and each thread processes one data entry.

the memory size will be. The memory hierarchy of CUDA is illustrated in Fig. 3.5.

- **Registers** In modern CPUs, registers are usually not considered as part of the memory model, however, a GPU can have 64000 32-bit general purpose registers per SM, which equals a 256 KB low-latency memory. A register has an access latency of a few cycles. In CUDA kernels, the compiler will first assign variables to registers as long as no special keywords are used when declaring the variables.

- **L1 Cache** L1 cache is located inside the SM, which means that each SM has its own independent L1 cache. L1 can be by-passed manually using compiler options.

- **Shared Memory** Similar to POSIX shared memory for inter-process communication, CUDA shared memory is used for inter-thread communication. Shared memory is located inside the SM and can be accessed by all threads within a block simultaneously. Shared memory within one block will be divided to 32 banks, one thread can only access one memory bank at one time. Shared memory is not cached by L1 or L2.

- **L2 Cache** L2 cache is also on-chip but outside SMs, all SMs share one L2 cache. Different from L1, L2 cannot be by-passed. L2 only caches DRAM.

- **DRAM** DRAM is an off-chip memory which has the highest latency (around 400 clock cycles per transaction). A DRAM transaction is always 32 byte aligned. If an odd size (not a multiple of 32 byte) or unaligned memory transaction is requested the request will be padded to 32 byte before the transaction.

There are also other two memory types called Texture Memory and Constant Memory, which are not involved in the project described by this thesis, thus beyond the scope of our discussion here.
Chapter 3. Overview

3.2 Object Detection Pipeline Overview

Now we describe an end-to-end deep learning based image object detection pipeline which is illustrated by Figure 3.6. “End-to-end” means that this pipeline takes an image as input and outputs object location on the image with its name and score. The object’s location is marked by a rectangular bounding box. The pipeline is divided into two parts, the PVANet network and a NMS algorithm, both of which run on the GPU.

- **PVANet Network** The PVANet network’s key function is performing object detection on input images. The output of PVANet are bounding boxes and scores. The detailed structure of PVANet will be discussed later. We optimized the inference process of PVANet using NVIDIA TensorRT. The training process is beyond the scope of this work.

- **NMS** The output of PVANet is redundant: one object can be marked by multiple bounding boxes as shown in Figure 3.7a. To obtain a usable result like Figure 3.7b, we must select the best bounding box for each object and eliminate others, which is done by NMS. In this work, we parallelized and optimized the NMS algorithm.

The project described in this paper focuses on the inference process, hence training is beyond the scope of discussion. As the accuracy of a neural network is determined in the training process, we will not discuss the accuracy of this pipeline in depth. Instead, we will concentrate on efficient deployment and optimization.

As we aim to build an efficient object detection pipeline, performance is the top priority and detection accuracy is not our concern. This does not mean that our platform performs poorly, on the contrary, we believe that PVANet represents state-of-the-art. We only concentrate on methods which improve speed rather than accuracy.
3.3 Overview of the PVANet Network

3.3.1 Convolutional Neural Networks

Before diving right into PVANet, we must explain concrete details of CNN. The basic unit of a neural network is called “perceptron” (shown in Fig. 3.8), defined as

$$ f_{\text{perceptron}}(x) = \sum_{i} w_i \ast x_i + b \quad (3.1) $$

where $x_i$ are the inputs, $w_i$ are weights of the inputs and $b$ the bias. As we can see, a perceptron is essentially a linear classifier, and we can combine several perceptrons together to form a neural network. A CNN is a NN that contains convolution layers. A convolution layer is made up of one or several 2 dimensional filters (also called convolution kernels), as shown in Figure 3.9. The process of a convolution layer is quite similar to image convolution. A formal definition of convolution is

$$ f_{\text{conv}}(x_{i,j}) = \frac{1}{n \ast n} \sum_{p} \sum_{q} w_{p+i,q+j} \ast x_{i,j} \quad (3.2) $$

where $n$ is the size of the convolution kernel. As shown in Figure 3.9, for each input, the convolution layer slides through the input image, each element of the output is the weighted average of the input within the slide window. The weights are the parameters of the convolution kernel.

Each kernel can have a weight and a bias, also called parameters of the network. The processing of determining parameters is called training. A convolution layer can have more than one kernel (called multiple channels). A multi-channel layer
Chapter 3. Overview

Figure 3.8: Left: A perceptron; Right: The plot of the perceptron function. The blue area marks the region where the output of the perceptron is positive.

Figure 3.9: Convolution Layer. $I$ is the width of the input image, $K$ is the width of the filter (shaded as blue), and $I - K + 1$ is the width of the layer’s output. The values in the filter are the weights of the filter. When the window hits the border, it starts from a new line for the next round. If we use the upper left element of the window as the anchor point, after the anchor point hits $I_{0,4}$, it will start from $I_{1,0}$ in the next round and continue sliding.

produces a multi-channel output, referred to as “feature maps”. Feature maps can also be the input of a convolution layer, each kernel will be performed on all channels and the output of each channel will be added together as the output of that kernel.

3.3.2 Pooling and Decovolution

Pooling and decovolution are other two layers that will also be involved. A max pooling operation is shown in Figure 3.10. As we can see, a $NN$ max pooling divides a $M \times M$ input into $N \times N$ areas, each area of size $\frac{N}{M} \times \frac{N}{M}$. The output is the max value of each area. Besides max pooling, there is also average pooling, which replaces the max value in the output with average value. Pooling can not only reduce the data size but also prevent over-fitting due to its translation invariant property.

Deconvolution is somehow a misleading name “transposed convolution” is more appropriate\(^1\). If we represent convolution in a matrix form, then convolution can be

\(^1\)Google’s TensorFlow framework uses the name “transposed convolution” for deconvolution layer: https://www.tensorflow.org/api_docs/python/tf/nn/conv2d_transpose
3.3. Overview of the PVANet Network

C. ReLU

To understand C. ReLU, we must understand ReLU first. As we just mentioned, the perceptron is the basic unit of a neural network. Although we can combine many perceptrons together, no matter how many perceptrons we use, the combination of linear functions can only be a linear function, if we want neural network to fit a non-linear function, non-linearity must be introduced to the network. One solution is to append a non-linear function to the perceptron, and such functions are called “activation functions”. ReLU is a certain type of activation function, defined as

\[ f_{ReLU}(x) = x^+ = \max(0, x) \]  

ReLU is believed to have both biological and mathematical justifications [8, 9], it outperforms other activation functions and is now the most popular activation function.
The inspiration of C. ReLU comes from an observation of life that edges of objects tend to exist in pairs, e.g. canthi (corners of the eye) of humans, edges of tables, etc. Thus in CNN, one node’s activation can be the opposite of another’s. Here, “activation” means output of the node. The “concatenation” in C. ReLU means concatenating the activation by its negation before ReLU. The definition of C. ReLU is

\[ f_{C.ReLU}(x) = \max(0, x|\neg x) \]  

where | denotes concatenation. Assume \( x = \{12, 20, 8, 10\} \), then

\[ x|\neg x = \{(12, 20, 8, 10), (-12, -20, -8, -10)\}^T \]  

Using equation 3.5, we have

\[ f_{C.ReLU}(x) = \max(0, x|\neg x) \]  

In addition, a scaling and bias layer are also added after the concatenation, allowing each node (perceptron) to have its own slope and activation threshold.

\[ f_{C.ReLU}(x) = \max(0, ax - b) \]  

C. ReLU acts as the first building block in feature extraction. By using the parity of features, C. ReLU reduces the number of output channels by half, achieving a 2x speedup without losing accuracy.
3.3. Overview of the PVANet Network

**Inception**

Inception is the second building block in feature extraction. Inception is a DNN structure introduced in [29]. A most intuitive way of improving a network’s performance is to increase its depth. However, several problems will follow:

- A deeper network means more parameters, overfitting is quite likely to happen if the dataset is not large enough
- More parameters also mean higher computation complexity
- Deep networks are prone to gradient vanishing

A natural way to solve these problems is reducing the number of parameters while keeping the network deep: use sparse connection instead of full connection. In other words, each perceptron does not need to connect with every perceptron in the next layer. However, in modern computers, sparse and dense matrices do not make much difference in terms of computation time. The idea of Inception is to design a sparse network which can generate dense data. Instead of increasing the depth of the network, Inception increases the width by stacking multiple filters into one layer. The Inception structure used by PVANet is composed of 3 different sized kernels, 1x1 and 3x3 (shown in Figure 3.12).

![Inception structure in PVANet](image)

**Figure 3.12:** Left: Inception structure in PVANet; Right: Inception for reducing feature map size by half. From [14]

The left side of Figure 3.12 shows a Inception structure for a full-size feature map. It should be noted that each path contains a 1x1 convolution kernel, which seems trivial but actually reduces parameter numbers. Output of each path will be concatenated together. The right side of Figure 3.12 is a structure with a stride of 2 and a pooling layer. This structure reduces the size of the feature map by half.

What if we lose the 1x1 convolution kernel in each path? Suppose we have an input of 100 × 100 × 128, after a 256-channel of 5 × 5 layer, the output feature map will be 100 × 100 × 256. The layer will have 128 × 5 × 5 × 256 parameters (128 input channels, 5 × 5 parameters each kernel, 256 kernels). If we apply a 32-channel 1 × 1 layer before the 5 × 5 layer the final output will still be 100 × 100 × 256, however the parameters will be cut down to 128 × 1 × 1 × 32 + 32 × 5 × 5 × 256, which is about 1/4 of the original.
Hypernet

Objects in nature exist in different sizes, which requires measurements in different scales. For instance, we measure a building in meters while we measure a cell in micrometers. Different scales represent different levels of abstraction. Imagine a digital map, larger scale can allow us to see the “bigger picture” and think globally while smaller scale can reveal fine-grained details. In object detection, an input image can contain various objects in different sizes, hence in order to achieve a better detection rate and accuracy, multi-scale representation was introduced to object detection. A naive approach to generate multi-scale representations is to upscale and downscale the image and concatenate the results to the original image. Hypernet[15] uses this approach to generate multi-scale features. In Hypernet, max pooling layer is used to downscale images and a deconvolution layer is used to upscale images.

In PVANet, Hypernet is the second part of feature extraction. The PVANet designed one upscale layer and one downscale layer. The author of PVANet believes that computational cost will be unproportional to performance if there are too many layers. The output of these two layers are concatenated together to form the final feature map.

RPN and ROI Pooling

At this point feature extraction is completed and region proposals should be generated for classification. As mentioned above, region proposals are generated by RPN. The inputs of the RPN are the first 128 channels of the feature map, and the outputs are the region proposals. Based on the region proposals, Region of Interests (ROI) Pooling layer generates the final bounding boxes and scores.

- **RPN** The structure of RPN is shown in Figure 3.13a. As mentioned in the previous chapter, anchor plays an important role in RPN. Anchors are boxes, as shown in Figure 3.13b. By default anchors can have 9 different sizes. RPN predicts the possibility of an anchor being background or foreground. Region Proposal Layer is quite like a convolution layer which uses anchors as convolution kernels: each anchor slides through the feature map and computes weighted averages as outputs.

- **ROI Pooling** ROI Pooling is essentially a max pooling performed on region proposals. ROI Pooling uniforms the region proposals into a fixed size by reducing the data size, which is more efficient in computation.

3.4 Overview of Non-Maximum Suppression

The complete process of NMS is shown in Figure 3.14. We use Figure 3.7 as our input and desired output. The idea of NMS is quite straightforward: if an object is marked by more than one bounding box, only the box with the highest score is kept and others are discarded. How do we determine whether two boxes mark the same object? We compute a variable named Intersection over Union (IoU) of two boxes, which is defined as

\[
\text{IoU} = \frac{p_{01}}{p_0 + p_1 - p_{01}}
\]  

(3.10)
3.4. Overview of Non-Maximum Suppression

Figure 3.13: (A): Structure of RPN; (B): Anchors of different sizes in RPN. RPN determines whether the area marked by an anchor is the foreground or background. If an anchor marks the background then it will be ignored when extracting features. Anchors of different sizes are more flexible and improves detection accuracy.

where $p_{01}$ stands for the two boxes' overlap area and $p_0, p_1$ stand for the area of the two boxes. If IoU of two boxes is higher than a certain threshold, then we believe the two boxes belong to the same object. Now we go through the NMS algorithm step by step.

Figure 3.14: Non Maximum Suppression Algorithm for the example shown in Fig. 3.7

1. NMS takes the output of the PVANet (bounding boxes and scores) as its input.

2. All the bounding boxes within an image are sorted according to their scores into an array $B = \{b_i | i \in [1, n]\}$. Each score is the probability of the respective box containing an object of a certain class.

3. Select the bounding box $b_1$ with the highest score and compute the IoU $p_{1j}$ between $b_1$ and $b_j, j \in [2, n]$. If $p_{1j}$ is larger than the threshold $P$, $b_j$ is removed from $B$.

4. Move $b_1$ from $B$ to the output set $B'$.

---

$^2$Figure from https://cdn-images-1.medium.com/max/1600/1*IS_9HnkfDdF00nID6xxF_A.png
5. If $B \neq \emptyset$, repeat step 2 with the first element in $B$. Otherwise output $B'$

At first glance, NMS may seem to be a rather simple algorithm. However, it is difficult to parallelize. IoU is not transitive: suppose there are three bounding boxes $a, b, c$, if $a$ overlaps with $b$ and $b$ overlaps with $c$, we cannot deduce that $a$ overlaps with $c$. It is entirely possible that $a$ and $b$ mark two different objects which are close to each other. Thus we must go through the loop of step 2 & 3 in the order of scores. This causes a data dependency. For optimization, we need to dismantle the data dependency or speed up the loop. Either way, the implementation method needs to be carefully designed.
Chapter 4

Implementation and Optimization

In this chapter we will first discuss how the PVANet is constructed using TensorRT and how NMS is implemented using CUDA. Then we will go through the optimization techniques involved in this project. Detailed results and benchmarks will be discussed in the next chapter.

4.1 PVANet

The entire PVANet is accelerated by Nvidia’s TensorRT platform. Hence to introduce the implementation of PVANet, TensorRT must be introduced first.

4.1.1 TensorRT

TensorRT is designed to work in a complementary fashion with existing deep learning frameworks such as TensorFlow. The purpose of TensorRT is to provide a programmable inference accelerator which can help companies deploy efficient deep learning applications in a massive manner. The workflow of deploying an inference application using TensorRT is shown in Fig 4.1. Once a network is trained, it can be stored in a certain format such as caffemodel [12] or ONNX [25]. TensorRT is capable of decoding such format and reconstruct the network. Besides, TensorRT also provides APIs for programmers to reconstruct a network using C++/Python codes. After reconstructing the network, TensorRT will perform optimization on the network’s topology structure. TensorRT will perform graph optimization on the topology of the network and generate a runtime engine, which can run the inference procedure.

The process of reconstructing a neural network, performing optimizations and generating the inference engine is called “build phase”. The build phase is quite similar to the process of compiling a program. TensorRT parses the network definition using similar mechanisms of the parse in compiler, and looks for potential optimization policies. In the build phase, TensorRT performs optimizations by combining layers and optimizing kernel selection:

- Eliminate layers whose outputs are never used
- Fuse convolution and ReLU operations
- Aggregate operations with the same inputs (i.e. the 1x1 convolution layer in Inception)
- Select kernels which are optimized specifically for different hardware platforms
Chapter 4. Implementation and Optimization

Figure 4.1: Deploying a neural network inference using TensorRT\(^1\). TensorRT first reads the trained neural network model and applies graph optimization on the model, such as layer fusion and dead layer elimination. Based on the optimized model, TensorRT creates a runtime engine which runs the inference process.

Combining layers can reduce the depth of a neural network, thus lowering computational cost. TensorRT provides APIs for common layers such as convolutions and activation functions, which are essentially CUDA kernels optimized by GPU experts. For unsupported layers, TensorRT provides an interface for programmers to define their own layers, which we will discuss later in this chapter.

4.1.2 Construction of PVANet in TensorRT

The process of constructing PVANet in TensorRT is shown in Fig. 4.2. The PVANet is trained by Caffe [12] and TensorRT provides native support for parsing trained models. In Fig. 4.2, each component’s name starts with an “i”, which indicates that it is an interface. The interface is implemented as a C++ abstract class. Here, we list the detailed steps of the reconstruction procedure.

- To begin with, a logger is build. The logger handles all information output.
- Using the logger, we can create a builder object. The builder is responsible for creating the network definition. The network definition contains the network topology and each layer’s type.
- Once the network definition is in place, we can fill in the network with parameters. In our case, the parameters are loaded from files, thus we need a parser to decode the model file.
- After the parameters are loaded, the builder can use the network definition to create a CUDA engine.
- The CUDA engine finally creates an execution context, which is used to perform inference.

It should be pointed out that the purpose of the CUDA engine is reuse. As we can see, the process of constructing a network in TensorRT is complex; if we want to deploy a network on multiple servers and reuse it, we need to save the reconstructed network into files so that we do not need to reconstruct from the beginning.

\(^1\)Figure from https://docs.nvidia.com/deeplearning/sdk/tensorrt-developer-guide/index.html
every time. The execution context is highly dependent on the hardware’s runtime information. Saving hardware runtime informations hardly makes sense. Thus a CUDA engine is designed to contain all the information needed to create an execution context. The CUDA engine can be serialized and stored on disk or sent to other processes.

### 4.1.3 Customized layers

However, not all the layers in PVANet can be transformed to TensorRT directly. Reshape and C.ReLU haven’t been implemented in TensorRT. So we need to write customized implementation using the plugin interfaces of TensorRT. The interfaces are designed in a “simple factory pattern”, which we will discuss shortly.

As we have described C.ReLU in the previous chapter, now we discuss the definition of Reshape. The Reshape layer reorganizes the input data’s dimension. For instance, if a tensor $t$ is defined as

$$ t = \{1, 2, 3, 4, 5, 6, 7, 8, 9\} $$

then we say the tensor has a shape of 9. If we would like to reshape $t$ to shape $(3 \times 3)$, then

$$ \text{reshape}(t, (3, 3)) = \{(1, 2, 3), (4, 5, 6), (7, 8, 9)\}^T $$

It should be noted that the element number of a tensor should be the same before and after reshape.
Simple Factory Pattern

In object oriented programming, a layer should be abstracted as a class. To allow programmers to define any layer they want, TensorRT uses an abstract class “iPlugin” to provide the interfaces for customized layers. Every class of a customized layer must inherit iPlugin class and override the virtual functions. To build and manage multiple types of customized layers, TensorRT adopts the simple factory pattern. The computation logic and configuration of the layer is enclosed in the layer class, while the instantiation and management of these layers are handled by the factory class (named “iPluginFactory”) in TensorRT.

The main advantage of the simple factory pattern is clear responsibility and simple management. When the number of customized layers goes up, it can be tiring to instantiate each layer one by one. The factory class provides an uniform interface to manage all customized plugins. In addition, the simple factory pattern offers flexibility to modifications. For instance, if we have defined Reshape and C. ReLU in TensorRT and now we would like to add more of these two layers in the network to improve performance, we only need to generate a new model file for TensorRT; no modification on source code is required.

Reshape

As we known, in C/C++, the memory layout of a static multi-dimension array is actually the same as a one-dimension array. For instance, both

```c
int array1[3][2] = {{0, 1}, {2, 3}, {4, 5}};
```

and

```c
int array2[6] = {0, 1, 2, 3, 4, 5};
```

are the same in memory, which looks like

```
{0 1 2 3 4 5}
```

Thus a Reshape operation can be implemented by altering the data dimensions: copy the data from input to output and set the new dimensions. TensorRT uses “tensors” to represent the input and output data of layers. A tensor has a member variable to describe its dimensions. We use asynchronous memory copy in CUDA so that these memory transactions can happen concurrently whenever it is possible and reduce overall latency of data transfer.

C. ReLU

As we have shown in the previous context, C. ReLU can be assembled by three layers: negation, concatenation and ReLU. A negation layer is virtually a multiplication layer with a multiplier of $-1$. TensorRT provides native support for all these three layers. Hence we have two ways of implementing C. ReLU:

- Modify the model file generated in the training process directly. We can manually split the C. ReLU into three layers. However, modifying a automatically generated file is not a smart choice as the file is complex and lacks efficient debugging tools.
Implement C. ReLU using TensorRT’s API. As mentioned before, TensorRT provides APIs to build and modify the network definition. Compared to modifying the model file, this is clearly a much better approach.

However, the second approach is still not perfect. First, using APIs to modify the network definition is not agile, if any changes were to be made to the network, the source code has to be changed as well. Second, benchmarking indicates that the performance is not satisfying. To overcome these two shortcomings, we decided to implement C. ReLU as a TensorRT plugin.

Each plugin is actually executed via a CUDA kernel. The plugin is created in the plugin factory as mentioned above and the factory is passed to the parser which needs the plugins to parse the model file. The CUDA is designed in a element-wise fashion: each thread processes one data entry in the feature map. The code of the C. ReLU kernel is shown in Algorithm 1.

**Algorithm 1 C. ReLU Kernel**

- **Input:** inTensor, inTensorSize and coef
- **Output:** outTensor

1: // blockIdx, blockDim and threadIdx are used to calculate thread indices
2: tid ← blockIdx.x * blockDim.x + threadIdx.x
3: input ← inTensor[tid]
4: // determine the sign of the input and apply the coefficient
5: if input > 0 then
6:   input ← input * coef
7: else
8:   input ← 0
9: end if
10: // output the result and its negation
11: outTensor[tid] ← input
12: outTensor[tid+inTensorSize] ← -input

This kernel exactly implements Equation 3.9. Each thread reads an element of the input tensor and multiply the element with the coefficient according to the sign of the input. Then both the element and its negation are output to a new tensor. Benchmark shows that the plugin approach performs better than the API approach, which we will discuss in details in the next chapter.

### 4.2 Non-Maximum Suppression

By now, we have ported most layers to TensorRT. However, one important step in this network cannot be run in parallel: Non-Maximum Suppression. Due to its high data dependency, by now there is still no efficient parallel algorithm. The implementation of non-maximum suppression is more complex compared to the construction of PVANet. We tried a naive method and a map-reduce based method (which we considered as state of the art). To improve performance, we bring forward our warp-wise method as well.

The whole NMS stage consists of three kernels: a bounding box transformation kernel, a transpose kernel and a NMS kernel. The first two kernels are preprocessing kernels. The bounding boxes generated by PVANet are actually based on the
Chapter 4. Implementation and Optimization

feature map. Thus we need to map the boxes from the feature map back to the image. This is done by the bounding box transformation kernel. The behavior of the transpose kernel is the same as transposing a matrix. This operation is required by an optimization technique called “coalesced memory access”. The NMS kernel actually applies the NMS algorithm. For all three methods mentioned above, the two preprocessing kernels are the same, only the NMS kernel varies.

4.2.1 Naive Method

The naive method is quite straightforward, and it incorporates an element-wise method: one thread handles one bounding box. Inter-thread communication is completed via shared memory. The data layouts of bounding box and score are the same, [batchSize, NMS_MAX_OUT, OUTPUT_CLS_SIZE], where:

- batchSize indicates how many images are processed simultaneously. This variable can be set using command-line arguments when launching the program.
- NMS_MAX.OUT is the maximum number of candidates in each class that NMS will finally output. This is a static constant defined at compile time.
- OUTPUT_CLS_SIZE is called the class number or the category number. The “class” refers to the PASCAL VOC dataset [3] class list², which defines 21 real-life objects contained in the dataset. The value of OUTPUT_CLS_SIZE is defined by a macro definition. Each class is represented by a thread block in CUDA.

The data flow of this method is shown in Fig. 4.3. One thread represents a bounding box and one block represents an object class. The kernel is shown in Algorithm 4.2. __Shared__ is the keyword which declares an array to be allocated in shared memory; threadIdx is a C struct which contains the index of each thread. As the thread block can have up to 3 dimensions, the threadIdx has 3 members: x, y and z. The inputs of the kernel are scores and bboxes, and the output is the index of the result bboxes; __syncthreads() is a block-level synchronization barrier: all threads within a block will wait until every thread has reached this barrier.

4.2.2 Map-reduce Method

The map-reduce method is based on [26], and here, we provide several improvements to this method. To begin with, the algorithm described in [26] requires a sorted input while our method accepts inputs in arbitrary order. Second, [26] violates the non-transitive property that we mentioned previously, and we rectify this problem.

We start from the algorithm in [26]. The process is shown in 4.4. Suppose we have a bit matrix \( M \) of size \( N \times N \) for each object class. The matrix is defined as

\[
M_{i,j} = \begin{cases} 
1 & \text{if } i > j \text{ and } \text{IOU}(i,j) > \text{threshold} \\
0 & \text{otherwise}
\end{cases}
\]  

²The list can be found at https://github.com/NVIDIA/DIGITS/blob/master/examples/semantic-segmentation/pascal-voc-classes.txt
Algorithm 2 NMS Naive Method

**Input:** scores and bbox  
**Output:** indexOut

1: __shared__ score[NMS_MAX_OUT] ← scores[threadIdx.x] //Score array  
2: __shared__ bits[NMS_MAX_OUT] ← 1 //Bit array, initialized to 1  
3: thread_score ← score[threadIdx.x]  
4: thread_index ← threadIdx.x  
5: //Sort the bounding boxes by their scores  
6: BlockRadixSort(thread_score, thread_index)  
7: bbox1 ← bbox[thread_index]  
8: __syncthreads() //Synchronize all threads  
9: //Compute IoU and eliminate redundant bboxes  
10: for i: 1 → NMS_MAX_OUT do  
11:     bbox2 ← bbox[score[i]]  
12:     IoU ← computeIoU(bbox1, bbox2)  
13:     if IoU > IoU_threshold then  
14:         bits[threadIdx.x] ← −1 //mark the eliminated box as −1  
15:     end if  
16:     __syncthreads() //Synchronize all threads  
17: end for  
18: indexOut[threadIdx.x] ← bits[threadIdx.x] ∗ thread_index

---

Figure 4.3: Data flow of the naive method. Each dark grey box represents a block and light grey box a thread. The numbers within the thread are the bounding boxes’ score-index pairs. The bounding boxes are sorted by their scores. The sorted result is placed in the shared memory. Each thread fetches the bounding box with the highest score in the memory and computes the IoU for elimination.
Figure 4.4: The map-reduce method described in [26]. (a). Box $d_1$, $d_2$ and $d_3$ are the desired results and all the other boxes are redundant. First we map the bit matrix using equation 4.3; (b). Elements with value 0 are white and elements with value 1 black. The result matrix is symmetric; (c). Shade half of the symmetric white elements, excluding the diagonal; (d) If row $M_i$ of the matrix contains a grey element then eliminate $d_i$.

As we can see in Fig. 4.4b, $M_{i,j}$ is a symmetric matrix, as $IOU(i,j) = IOU(j,i)$. We mark all the 1 elements (the white block in Fig. 4.3b) in the lower triangle as -1 (the gray block in Fig. 4.3c). This is the map stage. The matrix is reduced row by row: if a row of the matrix contains a -1 element, then output 1, else output 0. The final output is a mask array shown in Fig. 4.3d.

In our method we modify the definition of $M$ to further utilize the lower triangle of the matrix:

$$
M_{i,j} = \begin{cases}
1 & \text{if } i > j \text{ and } IOU(i,j) > \text{threshold} \\
1 & \text{if } i < j \text{ and } \text{score}(i) < \text{score}(j) \\
0 & \text{otherwise}
\end{cases}
$$

(4.4)

The upper triangle of the matrix records whether two bounding boxes overlap with each other, and the lower triangle of the matrix shows the score relation of the two bounding boxes. Using the lower triangle, we can sort the bboxes by counting sort: suppose we have 10 bounding boxes with scores $[s_1 ... s_{10}]$; if there are 4 bounding boxes whose scores is higher than $s_4$, then we know that $s_4$ is the 5th higher score. Counting sort is a time-stable algorithm, which has a time complexity of $O(n)$. Once we have the sorted array, the redundant bboxes can be easily eliminated using the upper triangle.

As we introduced the counting sort into the method, the input data no longer needs to be sorted. The elimination is done in the same way as in the naive method, only that the information of overlap has been recorded previously in the upper triangle of $M$. Thus this method satisfies the non-transitive property.

Unfortunately, the map-reduce method is not work-efficient. As the bit matrix must be placed in shared memory, adding more processors will not help. The overall space complexity is $O(n^2)$ as it requires a $N \times N$ matrix to store the map. After sorting the bounding boxes, the original address order of the boxes will be broken, resulting in a random memory access pattern. A random access pattern is very inefficient in CUDA. We will discuss this in detail in the optimization section of this chapter.
4.2.3 Warp-wise Method

As we have mentioned in Chapter 2, a warp is the basic unit of GPU scheduling. If a thread block contains more than 32 threads then it will be split into several warps. If the number of threads in a block cannot be divided by 32, then the thread number will be “padded” to a multiple of 32. For instance, if a block contains 200 threads, then it will be split to 7 warps, and the last 24 threads in warp 7 will be inactive throughout the lifespan of the kernel. If we design the kernel based on warp rather than thread blocks, the kernel will have better execution efficiency. As a matter of fact, CUDA provides warp-level primitives for warp-level programming.

The first warp-level primitive we would like to introduce is \( \_\_\text{syncwarp}() \). As we have shown above, the naive method uses \( \_\_\text{syncthreads}() \) quite a few times. We all know that synchronizations will cost performance as some of the threads maybe idle until everyone reaches the barrier. It is hardly possible that all threads in a block march in a uniform stride as they are divided to different warps. The threads in a warp, however, march more uniformly as they are scheduled together every time. Similar to \( \_\_\text{syncthreads}() \), \( \_\_\text{syncwarp}() \) provides a warp-level synchronization barrier, which is much faster.

In the naive method, we also used shared memory for inter-thread communication. Although shared memory is on-chip, it is still not the fastest method. A GPU with Compute Capability 3.0 or higher provides an instruction named “shuffle” which allows threads within a warp to access each others’ register directly. If two threads exchange data through shared memory, then it requires one load instruction, one store instruction and an extra register to hold the address, while it only requires one load using shuffle.

![Warp-wise Method Diagram](image)

**Figure 4.5:** Data-thread mapping of warp-wise method. Suppose \( NMS\_\text{MAX\_OUT} = 192 \), then each thread holds \( 192/32 = 6 \) bounding boxes.

In the warp-wise method, each warp represents a class of objects (shown in Fig. 4.5). Each thread has an array \( A \) of size \( \frac{NMS\_\text{MAX\_OUT}}{32} + 1 \) to hold the bounding boxes, e.g. if \( NMS\_\text{MAX\_OUT} = 200 \) then each thread holds 7 bounding boxes. Fig. 4.6 shows the process of the warp-wise method. Each thread first sorts \( A \) independently, then each thread in a warp outputs the top1 element in \( A \) to form a new
array $B$ which is of size 32. Next, find the top1 $b$ in $B$ and compute the IoU of $b$ and all the other elements in $A$. Finally, eliminate the elements whose IoU is higher than the threshold. This process is repeated until all the elements are either eliminated or kept as results. The entire kernel is shown in Alg. 3.

`laneid` is a keyword to access the thread index of a warp, the range of laneid is $[0, 31]$. A warp has 32 lanes and one thread occupies one lane. $d_{scores}$ and $d_{bbox}$ are the input arrays which contain the scores and bboxes. `__shfl_down_sync()` is the function which calls the shuffle instruction mentioned above. There are 3 arguments in `__shfl_down_sync()`: mask, val and offset. `Mask` is a binary value that marks which lanes are active and which lanes are left out, e.g. a mask of $0b00000001$ indicates that only lane 0 actually performs this instruction. A thread at lane X gets the value of `val` from the thread at lane $X + offset$ of the same warp. A modulo operation is performed on $X + offset$ to ensure that it ranges from 0 to 31.

**Algorithm 3 NMS Warp-wise Method**

**Input:** scores and bboxes  
**Output:** indexOut

1: arraySize $\leftarrow$ \textit{NMS\_MAX\_OUT}/32 $+$ 1 
2: \textbf{for} $i : 1 \rightarrow$ arraySize \textbf{do}  
3: \hspace{1em} //laneid is given by a special register, which marks  
4: \hspace{1em} //the index of each thread within a warp  
5: \hspace{1em} index[i] $\leftarrow$ $i \times 32 + \text{laneid}$ \hspace{1em} //i*32 is the warp index  
6: \hspace{1em} score[i] $\leftarrow$ scores[$i \times 32$]  
7: \textbf{end for}  
8: //any sort with 0(nlogn) complexity will do as  
9: //score is very small  
10: sort(score, index)  
11: \textbf{for} $i : 1 \rightarrow$ arraySize \textbf{do}  
12: \hspace{1em} \textbf{for} $j : 1, 2, 4, 8, 16$ \textbf{do}  
13: \hspace{2em} //find the bounding box with the highest score  
14: \hspace{2em} /\_\_\_shfl\_down\_sync is used to access registers within a warp  
15: \hspace{2em} top1Index $\leftarrow$ max(scores[i], \_\_\_shfl\_down\_sync(FULL\_MASK, score[i, j]))  
16: \hspace{1em} \textbf{end for}  
17: \hspace{1em} \textbf{for} $j : 1 \rightarrow i$ \textbf{do}  
18: \hspace{2em} overlap $\leftarrow$ IOU(bboxes[top1Index], bboxes[$j\times32+\text{laneid}$])  
19: \hspace{2em} \textbf{if} overlap $>$ threshold \textbf{then}  
20: \hspace{3em} indexOut[j] $\leftarrow$ $-1$  
21: \hspace{2em} \textbf{end if}  
22: \hspace{2em} \_\_\_syncwarp()  
23: \hspace{1em} \textbf{end for}  
24: \textbf{end for}
4.3 Optimization

To build an efficient CUDA computing program, proper optimizations are indispensable. The general path to an optimized program is in an iterative fashion: assess, improve and repeat. The terminating condition of this iteration is when the program is sufficiently close to hardware limit or theoretical peak performance. “Sufficiently” means the gap between program’s actual performance and maximally achievable performance can be flexible. In practical software development, reward does not grow linearly with effort. As a matter of fact, the effort-reward ratio can be similar to Fig. 4.8, when the performance is high enough, a little improvement may cost huge amounts of time. Thus in realistic development, we must make a trade off between program performance and development cost. Usually, we consider 70% of peak performance as “sufficiently close”. The value 70% is an empirical value which is summarized from daily development [13, 18]. This does not necessarily means that we do not pursue the best performance, only that the best performance comes with a high price. For most applications, 70% is the neutral ground where the cost and gain meet. In our project, we also choose 70% as our goal of optimization. In next chapter, we will use the roofline model[31] to demonstrate our results.

Line 7-10 in Alg. 3 describes a top-1 algorithm. The algorithm proceeds as in Fig. 4.7. This algorithm is implemented using the shuffle instruction. Each round one thread compares the input value with another thread to find the bigger and half of the threads are finished. The algorithm is completed when there is only one thread left. It should be noted that \texttt{__syncewarp()} is not needed at the end of each round as \texttt{__shfl_down_sync()} ensures synchronization already.

Our benchmark (shown in Table 5.8 of Chapter 5) shows that the warp-wise method is much faster than the naive method, let alone the map-reduce method. The warp-wise method has a space complexity of $O(n)$. As a $O(n\log n)$ sort is required and the elimination process is only $O(n)$, the overall time complexity is $O(n\log n)$.
4.3.1 Assess

Before applying any optimization technique, we must first measure the program’s performance in a scientific way, such that we can figure out where the bottleneck is. A kernel’s performance can be bounded in three ways: memory bandwidth, instruction throughput and latency. Memory bandwidth refers to the hardware speeds of memories used by GPU, usually measured in Gigabytes per second (GBps). The GDDR6 DRAM used by Tesla P4 has a bandwidth of 192 GBps. Instruction throughput is defined as how many instructions are run by the processor per unit time. Different instructions may have different throughputs, for instance, Tesla P4 has a single-precision performance of 5.5 TFLOPS, which means the processor can execute $5.5 \times 10^{12}$ 32-bit floating number operations per second. Latency is defined as the time to complete a certain task which has units of time. The NVIDIA Visual Profiler (NVVP) is the profiling tool which can measure a CUDA program’s performance in all kinds of ways. It can tell us the program’s achieved bandwidth and throughput as well as the theoretical hardware limit. Generally, we can tell whether a kernel is instruction bound or memory bound by its instruction-to-memory ratio. This metric is called “arithmetic intensity”. If the arithmetic intensity of a kernel is approximately equal to the theoretical value, then we consider the kernel is well-balanced. For instance, Tesla P4 has an instruction throughput of 275 GFLOPS (FP32) and a memory bandwidth 192 GBps. Thus the theoretical instruction-memory ratio is $5443/192 \approx 28.87$. If a kernel’s achieved ratio is lower than 1.43 then we consider it is memory bound, and instruction bound otherwise. If both bandwidth and throughput are far from the hardware limit then we should consider the issue of latency.

4.3.2 Improve

In this section we will introduce optimization techniques regarding the three bounds mention above. These techniques are universal and can be applied to all kernels involved in this project. Then we will discuss special techniques which are designed exclusively for some kernel in this project. The discussion will focus on the warp-wise method as it is the best method in the algorithmic level.
4.3. Optimization

Memory bound

A kernel is said to be memory bound if the achieved memory bandwidth is much lower than the peak. Global memory has an access latency of 400-500 clock cycles which is huge, thus memory is the biggest enemy against performance. To eliminate the memory bound, we must make sure that the program only access necessary data. There are two major techniques: improving the access pattern to reduce wasted transactions and reducing redundant access. The first one is achieved using “coalesced access” and the last one shared memory.

Access pattern refers to the mapping between thread address and memory address. There is another metric of memory speed called “transactions per second” or “transfers per second”. This metric is determined by the memory clock speed. Let us still take Tesla P4 as example, P4 has a DRAM clock speed of 1500 MHz, it can complete 4 memory transactions per clock cycle since using GDDR5 DRAM. The memory bus is 256-bit wide. Thus the bandwidth can be calculated as $1.5 \times 4 \times 256/8 = 192$ GBps. However, full bandwidth can only be achieved if the GPU fetches 256 bits of data in each transaction. Thus we need to “fill” each transaction as fully as possible. In CUDA, global memory access from a warp can be coalesced into a single transaction. The criterion is to make request from a warp fall into one L1 cache line. In other words, the number of transactions should equal to the number of L1 line accessed. As a matter of fact, L1 cache can be manually switched on and off by the programmer. We will discuss coalesced memory access in two situations: caching and non-caching.

The advantage of using cache is obvious: it helps on misaligned and randomized access. The ideal pattern of a coalesced access is then all threads in a warp request 32 aligned and consecutive 4-byte words (4 bytes is the length of a single precision floating number), shown as Fig. 4.9a. In this case, the warp requests 128 bytes and 128 bytes of data go through the bus. Thus the bus utilization is 100%. Now if a warp requests 32 aligned but permuted 4-byte words shown as Fig. 4.9b, the bus utilization would still be 100%. When L1 is turned on, the memory transaction size is fixed to 128 bytes. When a warp launches a memory request, the request will first go through the L1 cache, and 128 bytes of data will be fetched from global memory and stored in the cache. Threads in the warp can find the data they need in the cache as all 128 bytes are already loaded in the cache.

However, caching is not always perfect. As we have mentioned before, the transaction size is fixed to 128 if L1 is turned on. Now consider all threads within a warp requests the same 4-byte word. A broadcasting mechanism will be triggered and all 32 threads will receive their requested data through a single broadcast. If the L1 cache is turned on, 128 bytes of data go through the bus while we only need 4 bytes of them. Now the bus utilization has dropped to 3.125%. If we turn off the L1 cache, then the transaction size can be reduced to 32 bytes, resulting in a bus utilization of 12.5%. Another scenario is that a warp requests 32 misaligned (“align” means 128-byte aligned) but consecutive 4-byte words. The addresses will fall on 2 cache lines in this case and 256 bytes of data will go through the bus, resulting in a bus utilization of 50%. When L1 is turned off, the 128 bytes of data needed by the warp will fall within 5 32-byte segments at most and 4 at least. This will give us a bus utilization of at least 50%.
The worst memory access pattern is when a warp requests 32 scattered 4-byte words. Depending on how the data is scattered, the addresses may fall within $N$ segments. The bus utilization will be $\frac{128}{N} \times 128$ with L1 turned on and $\frac{128}{N} \times 32$ with L1 turned off. This is the case we should definitely avoid.

To avoid the worst case, we added a transpose kernel before the NMS kernel so that all threads can access consecutive memory addresses. There are random access patterns in the naive method as after sorting the bounding boxes, the original order will be broken. As a result we choose to leave L1 cache on to help improve the performance. However, the random access pattern does not need to be worried as randomness within a warp does not affect performance.

Compared to global memory, shared memory has a latency of just a few clock cycles. The bandwidth of shared memory is also high which is around 1TBps per GPU. Tesla P4 has 96 KB shared memory on each SM. These features make shared memory a user-managed cache which can reduce redundant global memory access and avoid non-coalesced access. For instance, in the NMS kernel, both bounding boxes and scores are used multiple times, thus we loaded them into shared memory.
to reduce global memory access. The transpose kernel also guaranteed coalesced access in both reading and writing to memory using shared memory.

Although shared memory does not require coalesced access to achieve peak performance, it does have a drawback called “bank conflict” which we will describe later in this chapter. An alternative to shared memory is using registers, which represents the fastest memory type. There are no rules on using registers, only that the size of registers is much lower compared to shared memory.

Instruction bound

A kernel is said to be instruction bound if the instruction throughput is much lower than the peak. Usually instruction bound is considered after memory bound as instruction is less likely to happen. Even a computation-heavy kernel can be memory bound if not programmed carefully. The idea to break instruction bound is quite straightforward: reduce the instruction count without changing the task. More specifically, we should use instructions with high throughput and eliminate wasted instructions.

There is a technique called “vectorized memory access” in CUDA, which is quite similar to the AVX instruction in GPU. Using vector load and store instructions can access multiple data entries with one instruction, increasing the throughput. In the NMS kernel, bounding boxes are marked by 4 floating point coordinates which are loaded and stored like the following code:

```c
for (int j = 0; j < NMS_MAX_OUT * 4; j += 4)
{
    float bbox0 = bboxes[j];
    float bbox1 = bboxes[j + 1];
    float bbox2 = bboxes[j + 2];
    float bbox3 = bboxes[j + 3];
}
```

Using vector load and store, the code can be written like this:

```c
for (int j = 0; j < NMS_MAX_OUT; j++)
{
    float4 bbox = *(float4*)&bboxes[4 * j];
}
```

As we can see by using vector load and store the four coordinates of a bounding box can be accessed with one instruction. A large memory access granularity can also help improve memory bandwidth.

Divergence is another obstacle to performance. Now consider the following code:

```c
if (__laneid < 16)
    output[__laneid] = 0;
else output[__laneid] = 1;
```

The behaviors of threads in a warp are different when executing this code: half of the threads write 0 to the output array while the other half write 1. This is called divergence. When the first half warp is writing 0 to output, the other half warp will be predicated off and become inactive, waiting for the first half to finish their task.
Apparently, divergence will lower execution efficiency. However, divergence happening on different warps has no impact on performance.

**Latency bound**

A kernel is latency bound when both memory bandwidth and instruction throughput are far from the hardware limit. A thread is blocked when the resource it requests is not ready. For instance, if a thread requests data from global memory then it has to wait for the data to arrive. To utilize this waiting time, the scheduler will switch to other threads thus the latency of the blocked thread is hidden by other threads. As long as the thread number is large enough, the latency can always be hidden. The block size should be a multiple of 32 which is the size of a warp. A metric to measure how the latency is hidden is “occupancy”. In Pascal architecture (the architecture that Tesla P4 used), each SM can hold 64 resident warps at the same time. Hence the occupancy is defined as

\[
\text{occupancy} = \frac{\text{number of active warps}}{64}
\] (4.5)

Occupancy should be high enough for the latency to be well-hidden. Assume a global memory access takes 400 cycles to finish and an arithmetic operation takes 2 cycles to finish. To hide the latency, we need \(400/2 = 200\) arithmetic operations for each global memory access. This ratio is too high for actual codes to achieve. Now suppose we have 8 arithmetic instructions for one global memory access, then we need \(200/8 \approx 26\) warps to hide the latency. The occupancy in this situation is 40.625\%. Occupancy can also be limited by resources on the SM. For instance, on Tesla P4, there are 64000 registers per SM, if each thread in the kernel we just discussed uses 80 registers then a SM can only hold \(64000/80 \times 32 = 25\) active warps, which is lower than 26.

The occupancy of the warp-wise NMS kernel is quite low, which is only about 5\%. This is a result of low warp numbers. As one warp represents one object class, there are only 21 warps in this kernel according to the PASCAL VOC dataset. If we use another dataset with more classes, the occupancy will be much more improved. The \(\text{NMS\_MAX\_OUT}\) variable will also affect the occupancy as we put all data in registers. If \(\text{NMS\_MAX\_OUT}\) is too large then the register problem we just discussed may happen.

### 4.3.3 Other Techniques for Warp-wise Method

The NMS kernel is a memory-intensive task, thus our optimization focuses on improving memory bandwidth. The code of the kernel is as follows:

For simplicity we assume \(\text{NMS\_MAX\_OUT}\) is a multiple of 32 and one block contains 32 threads. The code for writing the results back is also omitted. As we can see the code first loads data from global memory using coalesced access. Vectorized memory access (float4) is also used when loading bounding boxes. Function \(\text{top1\_thread()}\) is the top-1 algorithm. The access pattern is already coalesced and we only need to worry about the efficiency of the top-1 algorithm now. The code of the top-1 kernel is as follows:
**Algorithm 4 NMS_warpwise0**

**Input:** scores and bboxes  
**Output:** indexOut

1: // Declare shared memory and register arrays
2: __shared__ sdata[32]
3: rdata[NMS_MAX_OUT/32+1]
4: // Load data to registers
5: tid ← threadIdx.x
6: index ← blockIdx.x * blockDim.x + threadIdx.x
7: // Load bounding boxes into registers with coalesced access
8: for i : 0 → NMS_MAX_OUT/32 do
9:  rdata[i] ← bboxes[index + i * 32]
10: end for
11: __sincwarp()
12: // Sort the boxes according to their scores
13: sort(rdata, scores)
14: sdata[tid] ← rdata[0]
15: // Find the top 1 in the shared memory
16: top1 ← top1_thread(sdata)
17: // Compute IoU
18: for i : 0 → NMS_MAX_OUT do
19:  bbox1 ← getBbox(sdata[0])
20:  bbox2 ← getBbox(sdata[index + i * 32])
21:  iou ← computeIoU(bbox1, bbox2)
22: // Output the eliminated boxes
23:  if iou > threshold then
24:     indexOut[sdata[0]] ← −1
25: end if
26: end for

**Algorithm 5 top1_thread0**

**Input:** scores and bboxes  
**Output:** indexOut

1: laneid ← getLaneID()
2: for i : 1 → 32; i* = 2 do
3:  if laneid%(2 * i)is0 then
4:     input[laneid] ← max(input[laneid], input[laneid+i])
5: end if
6: end for
7: __sincwarp()
This code exactly describes the process of Fig. 4.7. It should be noted that the volatile keyword is necessary for the code to generate correct results. volatile prevents compiler to perform optimization on input which may alter the behavior of the code. The first problem with this code is the if branch. As we have mentioned before, divergence within a warp is highly inefficient. Besides, the modulo operation \( \% \) is also relatively slow. The solution is to replace with a new branch code:

**Algorithm 6** top1_thread1

**Input:** scores and bboxes  
**Output:** indexOut  
1: laneid ← getLaneID()  
2: for \( i : 1 \rightarrow 32; i* = 2 \) do  
3:   laneid ← 2*i*laneid  
4:   if laneid < 32 then  
5:     input[laneid] ← max(input[laneid], input([laneid+i]))  
6:   end if  
7: end for  
8: __syncwarp()

Now the new problem is bank conflicts in shared memory. The shared memory is divided into 32 banks, each time one bank can only handle one request. In the situation shown in Fig. 4.10, the shared memory with address 0 and 32 both belong to bank 0. If a thread \( a \) in warp 0 requests shared memory 0 and a thread \( b \) in warp 1 shared memory 32, it will result a two-way bank conflict, the request of \( a \) and \( b \) will be serialized. One exception is that if all threads in a warp requires shared memory of the same address, the data will be broadcast to all threads in the warp.

A common way to solve bank conflict is to declare one more column of shared memory, which will make the banks “skewed”. However, in our project, this issue is trivial if we use the warp-level primitives. After using the \_shfl_down_sync() instruction the kernel is no different than Algo. 6, except that we do not need shared memory anymore.

The input of the function is no longer an array but a register. The shuffle instruction spares the trouble of loading and storing data to shared memory, lowering the required memory bandwidth. Since the shuffle instruction is synchronized, __syncwarp() is no longer needed. As shared memory is no longer needed, the NMS kernel should also be updated:

```c
__global__ void NMS_warpwise1(int *g_iscore){
  // Declare register arrays
  int rdata[NMS_MAX_OUT/32+1];
  // Load data to registers
  unsigned int tid = threadIdx.x;
  unsigned int index = blockIdx.x*blockDim.x + threadIdx.x;
  for (int i = 0; i < NMS_MAX_OUT/32+1; i++){
    rdata[i] = g_iscore[index + i * 32];
  }
  __syncwarp();
  sort(rdata);
  // perform top−1 in registers
  top1_thread(rdata[0]);
  // Compute IoU
  for(int i =0; i < NMS_MAX_OUT; i++){
    float4 bbox1 = getBbox(__shfl_down_sync(FULL_MASK, rdata[0], offset));
    float4 bbox2 = getBbox(rdata[i]);
  }
}
```
4.3. Optimization

**Figure 4.10:** Bank conflict. Address 0 and address 32 both belong to bank 0. If a thread in warp 0 accesses address 0 and a thread in warp 1 accesses address 1, then a two-way bank conflict is formed and the two accesses will be serialized.

```c
float iou = computeIoU(bbox1, bbox2);
}
}
```

The for loop in `top1_warp0` actually runs only 5 iterations, which is a small number. Short loops can usually be unrolled to improve performance:

**Algorithm 7 top1_warp1**

**Input:** scores  
**Output:** scores (in-place function)

```c
1: input ← max(input, __shfl_down_sync(0xffff, input, 1))  
2: input ← max(input, __shfl_down_sync(0xffff, input, 2))  
3: input ← max(input, __shfl_down_sync(0xffff, input, 4))  
4: input ← max(input, __shfl_down_sync(0xffff, input, 8))  
5: input ← max(input, __shfl_down_sync(0xffff, input, 16))
```

`NMS_warwipse1` and `top1_warp1` are the final optimized kernels of the NMS algorithm. For each thread, sorting the boxes has a time complexity of $O(n \log n)$, and eliminating the redundant boxes has a time complexity of $O(nm)$. Thus the overall time complexity of the warpwise method is $O(n \log n + nm)$. When the size of the data (the number of classes and number of candidates) is fixed, adding more processors (SMs) does not necessarily speedup the program. As the number of warps is equal to the number of classes, as long as all these warps can be held by the SMs, adding more SMs will not help. For instance, suppose we have 64 classes, which means we need 64 warps, and all the 64 warps can run on one SM at the same time. Adding a second SM will not reduce the overall latency in this situation. However, if a SM can only hold 32 warps or the number of classes goes up to 128, then adding another SM will reduce the overall latency by 50% compared to using one SM. We
will also use the roofline model to demonstrate the performance of the warp-wise method.
Chapter 5

Results

In this chapter, we present the benchmark results of our pipeline. The result of the PVANet will be first given, then we will discuss the performance of the NMS algorithm.

5.1 Results of PVANet

5.1.1 Accuracy

The detection accuracy of a neural network is determined by its structure and the training process. Since we focus on efficient inference, we do not aim to improve the accuracy of PVANet. However, we still need to compare the accuracy of our reconstructed PVANet with the accuracy in the original paper of PVANet [14], so that we can show that accuracy is not sacrificed to trade for speed.

There are two metrics to measure accuracy: precision and recall. The correctness of each output bounding box can be divided into four classes: true positive, true negative, false positive and false negative. The precision is defined as:

$$\text{Precision} = \frac{TP}{TP + FP}$$  \hspace{1cm} (5.1)

where $TP$ and $FP$ denotes the number of true positives and false positives respectively. The recall is defined as:

$$\text{Recall} = \frac{TP}{TP + FN}$$  \hspace{1cm} (5.2)

where $FN$ denotes the number of false negatives. The precision tells us how many bounding boxes marks the correct positions of target objects among the outputs, i.e. the correctness of the output bounding boxes. The recall represents how many target objects can be captured in the output. As objects are divided into multiple classes in datasets, another metric “mean average precision” (mAP) is defined to show the overall precision:

$$\text{mAP} = \frac{1}{|\text{classes}|} \sum_{c \in \text{classes}} \text{Precision}(c)$$  \hspace{1cm} (5.3)

where $|\text{classes}|$ denotes the number of classes. mAP is more commonly used than precision. We run an experiment to test recall and mAP of our PVANet. The performance of the original PVANet is given in the paper by the author. The experiment runs on the PASCAL VOC 2007 dataset, same as the paper of PVANet. The recall and mAP comparison between our accelerated PVANet and the original PVANet is
Table 5.1: The comparison of our accelerated PVANet with the original paper. Experiments are run under PASCAL VOC 2012 dataset. “Proposals” means the number of bounding boxes generated in the final output of PVANet. The middle column shows that in different proposals, our accelerated PVANet is as accurate as the original paper. The right column shows the speed of the original PVANet and our accelerated PVANet.

<table>
<thead>
<tr>
<th>Model</th>
<th>Proposals</th>
<th>Recall (%)</th>
<th>mAP (%)</th>
<th>Time (ms)</th>
<th>FPS</th>
<th>Speed up (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original PVANet</td>
<td>300</td>
<td>99.2</td>
<td>84.4</td>
<td>48.5</td>
<td>20.6</td>
<td></td>
</tr>
<tr>
<td></td>
<td>200</td>
<td>98.8</td>
<td>84.4</td>
<td>42.2</td>
<td>23.7</td>
<td></td>
</tr>
<tr>
<td></td>
<td>100</td>
<td>97.7</td>
<td>84.0</td>
<td>40.0</td>
<td>25.0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>50</td>
<td>95.9</td>
<td>83.2</td>
<td>26.2</td>
<td>37.8</td>
<td></td>
</tr>
<tr>
<td>PVANet in our pipeline</td>
<td>300</td>
<td>99.0</td>
<td>83.9</td>
<td>36.8</td>
<td>27.2</td>
<td>32.4</td>
</tr>
<tr>
<td></td>
<td>200</td>
<td>98.9</td>
<td>84.1</td>
<td>33.2</td>
<td>31.2</td>
<td>31.8</td>
</tr>
<tr>
<td></td>
<td>100</td>
<td>97.8</td>
<td>83.3</td>
<td>30.6</td>
<td>32.7</td>
<td>30.8</td>
</tr>
<tr>
<td></td>
<td>50</td>
<td>95.6</td>
<td>83.0</td>
<td>20.7</td>
<td>48.3</td>
<td>27.8</td>
</tr>
</tbody>
</table>

Table 5.2: The computational cost of each stage of the PVANet. (Unit: GMAC)

<table>
<thead>
<tr>
<th></th>
<th>Feature Extraction</th>
<th>RPN</th>
<th>Classifier</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Computation cost</td>
<td>7.9</td>
<td>1.4</td>
<td>18.5</td>
<td>27.8</td>
</tr>
</tbody>
</table>

shown in the middle column of Table 5.1. “Proposals” means the number of bounding boxes generated in the final output of PVANet. As the middle column shows, our PVANet is almost the same with the original PVANet. We believe the slight difference is caused by the different hardware (Titan X vs. Tesla P4).

5.1.2 Time

In the original paper of PVANet, the author gives detailed computation cost of PVANet. As Table 5.2 shows, each image requires 37 GMACs. “GMAC” is the abbreviation of “Giga Multiplier–Accumulator”. A MAC is defined as:

\[ a \leftarrow a + (b \ast c) \]  

(5.4)

In GPUs, MAC is implemented via fused multiply–add (FMA) operations. For performance purposes, the GPU only performs one rounding operation when dealing with a floating-point FMA. This means the multiplication and addition can be done with a single instruction. Thus 1 GMAC = 1 GFLOPS, and the required instruction throughput for each image is 37 GFLOPS. The author of PVANet uses Titan X to run the benchmarks, which has a single-precision performance of 6700 GFLOPS. Thus the theoretical peak performance of PVANet on Titan X should be 6700/27.8 = 241 frames per second (FPS).

The performance comparison between the PVANet in our pipeline and in the original paper is shown in the right column of Table 5.1. The performance in the original paper is evaluated on a Titan X GPU with an Intel i7-6700K CPU, while our performance is evaluated on a Tesla P4 GPU with an Intel E5 CPU. As the CPU does take part in computation tasks, we only consider the performance of GPU. As
5.1. Results of PVANet

As we can see, our pipeline has a calculation efficiency almost 6% higher, which is 1.55 times than the original. Thus we can conclude that the pipeline achieves a higher FPS with fewer computation power, without lossing accuracy.

<table>
<thead>
<tr>
<th>Model</th>
<th>Time (ms)</th>
<th>FPS</th>
<th>Calculation efficiency (%)</th>
<th>mAP (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original PVANet</td>
<td>42.2</td>
<td>23.7</td>
<td>9.8</td>
<td>84.4</td>
</tr>
<tr>
<td>Our PVANet</td>
<td>33.2</td>
<td>30.1</td>
<td>15.2</td>
<td>84.1</td>
</tr>
</tbody>
</table>

Table 5.4: The time to run C. ReLU on a 264x160x64 feature map. The advantage of using plugin over APIs is that each layer only needs one kernel instead of several. Thus the overhead of launching kernels can be minimized.

<table>
<thead>
<tr>
<th>Method</th>
<th>Time (ms)</th>
<th>API</th>
<th>Plugin</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>0.019</td>
<td>0.0093</td>
</tr>
</tbody>
</table>

mentioned just above, the theoretical peak performance on Titan X is 181 FPS, while the actual performance is 23.7 FPS (we use 200 proposals as reference). We define a metric called “Calculation efficiency” to represent how efficiently the GPU is used:

\[
\text{Calculation efficiency} = \frac{\text{actual performance}}{\text{theoretical peak performance}} \tag{5.5}
\]

The calculation efficiency in the original paper is \(\frac{23.7}{241} = 9.8\%\). The Tesla P4 has a single-precision performance of 5500 GFLOPS, thus the peak performance is \(\frac{5000}{27.8} = 197.8\). Then the calculation efficiency in our PVANet would be 15.2%. As shown in Table 5.3, we can see that our pipeline has a calculation efficiency almost 6% higher, which is 1.55 times than the original. Thus we can conclude that the pipeline achieves a higher FPS with fewer computation power, without lossing accuracy. It should be pointed out that this calculation efficiency is calculated under the condition that only one pipeline is run on one GPU. If we run multiple pipelines on one GPU the utilization should go up. However, the GPU memory on Tesla P4 cannot hold another pipeline thus this argument is yet to be proved.

Customized layers performance

In the previous chapter we described two methods of implementing C. ReLU in TensorRT: assemble using the APIs or implement as a plugin. We have mentioned that implementing as a plugin is the faster method and now we give the performances of both methods.

Table 5.4 shows the performance of running C. ReLU on a 264x160x64 feature map. The plugin method is over 50% faster than the API method. The advantage of using plugin over APIs is that each layer only needs one kernel instead of several. Thus the overhead of launching kernels can be minimized.
Chapter 5. Results

Table 5.5: CPU and GPU performances of NMS. The naive method can guarantee a speed-up ratio of at least 27 with batch size 1 and 200 proposals (NMS_MAX_OUT). Note that this benchmark is based on the code that has not been optimized by the various techniques mentioned in the previous chapter.

<table>
<thead>
<tr>
<th>Batch Size</th>
<th>CPU (ms)</th>
<th>GPU Naive (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2.7</td>
<td>0.0966</td>
</tr>
<tr>
<td>2</td>
<td>5.6</td>
<td>0.0984</td>
</tr>
<tr>
<td>4</td>
<td>10.6</td>
<td>0.101</td>
</tr>
<tr>
<td>8</td>
<td>17.5</td>
<td>0.103</td>
</tr>
<tr>
<td>16</td>
<td>34.9</td>
<td>0.185</td>
</tr>
</tbody>
</table>

5.2 Results of Non-Maximum Suppression

First we show the speed-up ratios of the NMS algorithm compared to the CPU. We use a E5-2680v2 CPU and a Tesla P40 to run the test. Each test is run with 200 proposals (NMS_MAX_OUT) in each classes, 20 classes in each image and averaged over 100 iterations. On the CPU, as data between classes is completely independent, the inputs can be divided by classes for parallelization. Thus, if we use all cores of the CPU, the latency $T_p$ would be

$$T_p = T_1 \cdot \left\lceil \frac{N_c}{N_t} \right\rceil$$  \hspace{1cm} (5.6)

where $N_c$ and $N_t$ denotes the number of classes and CPU cores respectively. $T_1$ denotes the latency of processing 1 class using 1 thread. $T_1$ can be calculated as:

$$T_1 = \frac{T}{N_c}$$  \hspace{1cm} (5.7)

For simplicity, we only show the latency of using one CPU core$^1$. The test result is shown in Table 5.5. A more intuitive figure is given in Fig. 5.1. The naive method can guarantee a speed-up ratio of at least 27 with batch size 1. The time of the naive GPU method does not grow linearly with the batch size, on the contrary, the time is almost the same when the batch size is smaller than 16. The reason is that the overhead of the kernel launch takes up most of the time when the batch size is smaller than 16. When the batch size goes up, thread-level parallelism is also increasing as more threads are launched, thus it is possible to complete more computations within the same time. When the batch size is equal or higher than 16, the number of threads launched may exceed the number of threads that a GPU can run at the same time, resulting in a longer execution time. We will use the performance of the naive method as the baseline in the later tests.

If the NMS is running on the CPU, when the batch size is 1 and number of proposals is 200, it takes 2.7 ms for each image. In Table 5.1, our pipeline takes 33.2 ms to complete an inference on an image. Hence the NMS would take up 8% of the total time if using CPU. When using the GPU, the NMS will be finished in less than 0.01 ms, which can be ignored in the whole pipeline. This is a significant performance

$^1$The data within each class has high dependencies, making vectorization very hard. Thus SSE instructions do not help in this case.
5.2. Results of Non-Maximum Suppression

As we have discussed in Chapter 3, several preprocessing kernels are needed before applying the NMS algorithm. In Table 5.6, the detailed time of each kernel is given. “Vanilla” and “opt” represent whether or not the kernels are optimized by the techniques described in Chapter 4. A transpose kernel is added in the opt kernels such that coalesced memory access can be performed. Although a new kernel is added, the overall time of opt is still at least 20% shorter than vanilla. Note that from now on we use the performance of opt as the performance of the naive method.

**Table 5.6:** The performance of all kernels of the naive method with different batch sizes. Batch size indicates how many images are processed at the same time. As we can see after optimization, despite the fact that a transpose kernel is added for coalesced access, the overall latency is reduced by approximately 20%.

<table>
<thead>
<tr>
<th>Batch size</th>
<th>Scale &amp; clip</th>
<th>NMS</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Batch size=1</td>
<td>0.025</td>
<td>N/A</td>
<td>0.071</td>
</tr>
<tr>
<td>Batch size=2</td>
<td>0.029</td>
<td>N/A</td>
<td>0.074</td>
</tr>
<tr>
<td>Batch size=4</td>
<td>0.030</td>
<td>N/A</td>
<td>0.076</td>
</tr>
<tr>
<td>Batch size=8</td>
<td>0.045</td>
<td>N/A</td>
<td>0.085</td>
</tr>
<tr>
<td>Batch size=16</td>
<td>0.071</td>
<td>N/A</td>
<td>0.12</td>
</tr>
<tr>
<td>Opt</td>
<td>Scale &amp; clip</td>
<td>Transpose</td>
<td>NMS</td>
</tr>
<tr>
<td>Batch size=1</td>
<td>0.022</td>
<td>0.014</td>
<td>0.044</td>
</tr>
<tr>
<td>Batch size=2</td>
<td>0.025</td>
<td>0.017</td>
<td>0.046</td>
</tr>
<tr>
<td>Batch size=4</td>
<td>0.027</td>
<td>0.018</td>
<td>0.049</td>
</tr>
<tr>
<td>Batch size=8</td>
<td>0.022</td>
<td>0.018</td>
<td>0.056</td>
</tr>
<tr>
<td>Batch size=16</td>
<td>0.029</td>
<td>0.017</td>
<td>0.087</td>
</tr>
</tbody>
</table>

**Figure 5.1:** NMS speed-up ratio with different batch sizes.
Table 5.7: Performances of the warp-wise method and map-reduce method, compared to the naive method. The warp-wise method is the fastest method, at least twice as fast as the naive method. Note that when proposals are more than 200, there is not enough shared memory on each SM to run the map-reduce method.

<table>
<thead>
<tr>
<th>Proposals</th>
<th>50</th>
<th>80</th>
<th>100</th>
<th>200</th>
<th>400</th>
<th>800</th>
</tr>
</thead>
<tbody>
<tr>
<td>Naive</td>
<td>0.057</td>
<td>0.051</td>
<td>0.070</td>
<td>0.095</td>
<td>0.19</td>
<td>0.71</td>
</tr>
<tr>
<td>Warp-wise</td>
<td>0.018</td>
<td>0.028</td>
<td>0.031</td>
<td>0.032</td>
<td>0.074</td>
<td>0.38</td>
</tr>
<tr>
<td>Map-reduce</td>
<td>0.071</td>
<td>0.091</td>
<td>0.11</td>
<td>0.40</td>
<td>Not enough smem</td>
<td></td>
</tr>
</tbody>
</table>

Table 5.8: Speed-up ratio of the warp-wise method compared to the naive method, with different proposal numbers.

<table>
<thead>
<tr>
<th>Proposals</th>
<th>50</th>
<th>80</th>
<th>100</th>
<th>200</th>
<th>400</th>
<th>800</th>
</tr>
</thead>
<tbody>
<tr>
<td>Speed-up</td>
<td>3.17</td>
<td>1.82</td>
<td>2.26</td>
<td>2.97</td>
<td>2.56</td>
<td>1.87</td>
</tr>
</tbody>
</table>

Figure 5.2: The roofline model of Tesla P4. The horizontal axis represents the arithmetic intensity (FLOP:Byte ratio) and the vertical axis gives the attainable instruction throughput. The warp-wise kernel only uses one SM of P4, and the peak instruction throughput of a SM on P4 is 275 GFLOPS. If all instructions on P4 are floating point operations, the peak instruction throughput would be 275 GFLOPS. However, in our warp-wise method, the kernel only contains 13% floating point instructions, thus the throughput in this situation is 39.3 GFLOPS. The blue dot marks the performance of the warp-wise kernel, which is 30.4 GFLOPS with a 1.38 arithmetic intensity. Thus we achieved 77.4% of the peak performance, which is beyond our goal of 70%.

The performances of the map-reduce and warp-wise methods are shown in Table 5.7. The map-reduce method is much slower than the warp-wise method, even
5.2. Results of Non-Maximum Suppression

slower than the naive method.

Now we compare the warp-wise method with the naive method. Table 5.8 shows the speed-up ratio of the warp-wise method compared to the naive method. When there are less than 200 proposals, the time grows linearly with the number of proposals. However, when there are 800 proposals, the time increases almost 20 times. This is caused by the large number of registers used. When NMS_MAX_OUT is 800, each thread holds 800/32 = 25 bounding boxes, which requires 25 registers to hold them. Register spill will happen when the required number of registers of a thread exceeds a certain limit. This limit can be manually set to avoid register spill, however, if there are too many registers used per thread, the occupancy will be low. Either case, lowers the runtime performance.

Fig. 5.2 shows the roofline model of Tesla P4. The horizontal axis represents the arithmetic intensity (FLOP:Byte ratio) and the vertical axis gives the attainable instruction throughput. The warp-wise kernel only uses one SM of P4, and the peak instruction throughput of a SM on P4 is 275 GFLOPS. If all instructions on P4 are floating point operations, the peak instruction throughput would be 275 GFLOPS. However, in our warp-wise method, the kernel only contains 13% floating point instructions, thus the throughput in this situation is 39.3 GFLOPS. The blue dot marks the performance of the warp-wise kernel, which is 30.4 GFLOPS with a 1.38 arithmetic intensity. Thus we achieved 77.4% of the peak performance, which is higher than our goal of 70%.

We can see that our implementation for both the PVANet and NMS have a higher performance with respect to previous methods. Compared to the original PVANet, our pipeline gives a speedup of 30% in terms of FPS without sacrificing accuracy. Our warp-wise method for implementing the NMS algorithm is not only fast but also efficient. Compared to the naive method on the GPU, the warp-wise method is almost 3 times faster. Compared to the map-reduce method, the warp-wise method requires less shared memory such that it is able to process more proposals within one class.
Chapter 6

Conclusions

In this thesis we presented an efficient pipeline for object detection. By careful implementation using CUDA, the performance of the pipeline was dramatically increased. The pipeline offers an end-to-end solution to enterprises which is suitable for deployment. Our pipeline balances the development cost as well as performance. To build such a pipeline, we optimized the PVANet in an engineering way and proposed a new method to implement the NMS algorithm. Especially, our work on the NMS implementation has been accepted by the GPU Technology Conference (GTC) 2019 as a 50-minute talk. We believe that warp-wise should be a heuristic to all subtle algorithms run on GPU.

However, there are limitations of our work. To begin with, the memory consumption of the PVANet is not solved. Memory consumption is a common problem when deploying deep learning applications on the GPU. The Tesla P4 can only run one instance of the pipeline at the same time. Secondly, the pipeline’s inference speed can be furthered improved by using fixed precision number representation. That is, instead of using 32-bit single-precision floating point numbers for tensors, 8-bit integers may be a better choice for inference. Experiments in [23] have shown that lower precision and dynamic range of tensors does not cause obvious damage to the neural network’s performance. As long as calibration is properly done, using 8-bit integers can achieve similar performance as when using 32-bit floating points. If we replace 32-bit tensors in the inference process with 8-bit tensors, both the speed and memory consumption can be dramatically improved. In the future, we will work on mixed precision inference to bring the performance of our pipeline to a new level.
Bibliography


[17] Wei Liu et al. “SSD: Single Shot MultiBox Detector”. In: Lecture Notes in Computer Science (2016), 21–37. ISSN: 1611-3349. DOI: 10.1007/978-3-319-46448-0_2. URL: http://dx.doi.org/10.1007/978-3-319-46448-0_2.


