Joint Optimization of Speed, Accuracy, and Energy for Embedded Image Recognition Systems

Duseok Kang, DongHyun Kang, Jintaek Kang, Sungjoo Yoo, Soonhoi Ha[1]
Department of Computer Science and Engineering, Seoul National University, Seoul, Korea,
{kangds0829, kangdongh, zealaton28, yeonbin, sha} at snu.ac.kr

***Abstract*—This paper presents the image recognition system that won the first prize in the LPIRC (Low Power Image Recognition Challenge) in 2017. The goal of the challenge is to maximize the ratio between the accuracy and energy consumption within a time limit of 10 minutes for the processing of 20,000 images. Among three conflicting goals of accuracy, speed, and energy consumption, we considered the trade-off between accuracy and speed first to select Nvidia Jetson TX2 as the hardware platform and Tiny YOLO as the image recognition algorithm. Next, we applied a series of software optimization techniques to improve throughput, such as pipelining, multithreading, Tucker decomposition, and 16-bit quantization. Lastly, we explored the CPU and GPU frequencies to minimize the total energy consumption. As a result, we could achieve an accuracy of 0.24 mAP with energy consumption of 2.08Wh, which corresponds to the score of 0.11931, 2.7 times higher than the winner of LPIRC 2016.**

***Index Terms*—embedded system, energy consumption, deep learning, optimization, neural networks**

I. Introduction

Since AlexNet [1] won the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) [2] in 2012 with more than 10 percentage points ahead of the runner up, deep learning has become the de facto standard technique for image recognition. While the main stream of deep learning researches is pursuing higher performance with new algorithms with ever-increasing demand of higher computational power and memory requirement every year, on-device deep learning is attracting considerable attention recently to make edge devices intelligent without cloud services in various application domains such as autonomous vehicles, wearable computing, and so on. On-device image recognition is very challenging since embedded devices usually have tight constraints on the computational power, memory size, and energy consumption, which prohibits the use of complex state-of-the-art networks.

To encourage researchers to tackle the challenge, the Low-Power Image Recognition Challenge (LPIRC) [3] was started in 2015 as an annual competition that reflects the characteristics of the embedded environment: competition is designed to evaluate the trade-off among speed, accuracy, and energy consumption. As for the trade-off between accuracy and energy consumption, the final score of LPIRC is computed as the ratio of the mean average precision (mAP) over the total energy consumption (Wh). For more details of mAP computation, refer to [3]. As for the speed, there is a time limit of 10 minutes to process 20,000 images. If the number of processed images within the time limit is smaller than 20,000 by $X\%$ of the total number of images, the mAP is reduced by the same $X\%$ and the total energy consumed by the time limit is measured. If a system processes all images earlier than 10 minutes, the energy consumption is measured up to the completion time. Thus, it is critical to complete processing of 20,000 images within the time limit.

There are three possible approaches to tackle this challenge. One is to find an innovative deep learning network that consumes less power and memory space while achieving the similar performance as existent networks. SqueezeNet [4] and Tiny YOLO [5] are two examples of this approach. Another is to develop custom hardware accelerators for deep learning algorithms. TPU [6] from Google and Kirin970 [7] from Huawei are two outcomes of this approach. The third approach is to develop software optimization techniques to reduce the power consumption and memory requirements of a given deep learning network by taking advantage of statistical nature of deep learning algorithms. It is reported that approximate computing based on pruning [8], quantization [9], and low-rank approximation [10] could achieve significantly reduced resource requirements without noticeable accuracy loss.

In this paper, we present a solution that won the first prize in LPIRC 2017, by jointly optimizing speed, accuracy, and energy consumption in a systematic way. Among three conflicting goals of accuracy, speed, and energy consumption, we considered the trade-off between accuracy and speed first to select Nvidia Jetson TX2 as the hardware platform and Tiny YOLO as the image recognition algorithm. Next, we applied the existent software optimization techniques systematically in sequence. To increase the throughput performance and balance the utilization of processing elements, pipelining is first applied to the network. After pipelining, we applied two optimization methods, Tucker decomposition and 16-bit quantization, to improve the GPU performance. To speed up the CPU computation, we parallelized the post-processing task with multi-threading. After all software optimizations are performed, we considered the speed and energy consumption trade-off. To achieve the best score, we explored the operating frequencies of CPU and GPU. At last, we tested our implementation with a similar environment as the on-site competition. In this step, additional optimization was performed to reduce

[1] Corresponding author.

978-3-9819263-0-9/DATE18/©2018 EDAA

the overhead incurred in the test server. As a result, we could achieve an accuracy of 0.24 mAP with energy consumption of 2.08Wh in LPIRC 2017, which corresponds to the score of 0.11931, 2.7 times higher than the winner of last year.

II. Hardware Platform

Since there was no specialized neural processor available to us, we compared commercial embedded platforms in terms of maximum performance. Table I compares the peak performance of the GPU subsystem that is the main computing engine in each device.

TABLE I
Characteristics of candidate embedded devices

GPU	Chip	Clock	GFlops
Mali-T628 MP6	Exynos 5422	533MHz	102.4
Mali-G71 MP18	Exynos 8895	546MHz	371.2
Adreno 540	Snapdragon 835	710MHz	567
Maxwell Cores x 256	Tegra X1	1000MHz	512
Pascal Cores x 256	Tegra Parker	1465MHz	750

Odroid XU4 is an embedded device using Exynos5422 that is equipped with a quad-core A15, a quad-core A7, and a Mali-T628 MP6 GPU. Exynos 8895 and Snapdragon 835 are the chipsets used in smart phones where the Mali-G71 MP18 is the GPU on the Exynos 8895, and the Adreno 540 is the GPU on the Snapdragon 835. Jetson TX2 is an embedded AI computing device made by NVIDIA. It has a 2GHz quad-core ARM A57, a dual-core Denver2, and a Tegra GPU with 256 Pascal CUDA cores running at 1301MHz. It shows 1.46 times better performance than its predecessor, Jetson TX1 that was used in the winner of LPIRC 2016.

Based on the comparison result, as Jetson TX2 shows the best peak performance, it was selected as the hardware platform. Another benefit of using Jetson TX2 is the existence of CUDA and cuDNN library. Since main kernels are already optimized in the cuDNN library, it is easy to implement deep learning networks on the device. On the other hand, for the other devices in Table I, it is necessary to develop custom deep learning kernels with OpenCL since there is no available DNN specific library like cuDNN.

III. Deep Neural Network and Software Framework

Image recognition, also known as object detection, is a problem that finds all objects and the associated bounding boxes that contain the objects in a given set of test images. There are several neural networks proposed recently for image recognition. They are largely divided into two approaches. One approach separates the detection of the bounding boxes from the image classification, composing the network into two stages. In the first stage, a CNN is used to extract the features and generate region proposals. Image classification and box prediction in each region are performed in the second stage. Faster R-CNN [11] and R-FCN [12] are two examples.

On the other hand, several networks have been recently proposed to take the other approach that unites the region proposal and image classification in a single sequence of layers. YOLO [13], [14], Tiny YOLO [5], and SSD [15] are popular examples of this approach. These networks are selected as the candidate networks because they promise higher performance without accuracy loss than the first approach [13], [15].

TABLE II
Performance comparison among object detection models for ImageNet Detection Dataset

Model (Framework)	FPS	mAP	Predicted mAP
SSD (Caffe)	3.5	0.43	0.045
YOLOv2 (Darknet)	5.81	0.51	0.087
Tiny YOLO (Darknet)	17.4	0.32	**0.167**

We compared three candidate networks in terms of FPS (Frame Per Second) performance and mAP accuracy. The comparison result is summarized in Table II. Since the challenge uses the training data from ImageNet and the test images are ”ImageNet-like,” the networks were retrained with the ImageNet dataset and evaluation was also conducted with the ImageNet test dataset [16]. The first column lists the name of each network and the associated software framework that runs the network on Jetson TX2. As shown in the second and third columns of Table II, YOLOv2 dominates SSD in both performance and accuracy. However, there is no dominance relation between YOLOv2 and Tiny YOLO. Since there is a time limit in the challenge, the accuracy was re-computed by considering the time limit, which is listed in the last column, named **Predicted mAP**. For instance, YOLOv2 can process $5.81 \times 60 \times 10$ images in 10 minutes, which corresponds to 17.4% of 20,000 images so that the Predicted mAP becomes 0.51×0.17 as the estimated mAP of each algorithm in given 10 minutes. As a result, Tiny YOLO that gives the highest value of Predicted mAP was chosen as the base network.

While many deep neural networks have one or more fully connected layers, Tiny YOLO is a fully convolutional network (CNN) that consists of 9 convolution layers and 6 max pooling layers without the fully connected layer. Table III shows the structure of Tiny YOLO. The output tensor from the last layer includes all predictions of objects and bounding boxes. Since some predictions are not accurate, the post-processing step, called non-maximum suppression (NMS), is supposed to be run on the CPU side, which selects meaningful ones with some threshold values.

Note that the baseline Tiny YOLO is included in the Darknet framework [17]. The mAP accuracy and the FPS performance shown in Table II were obtained by running the Darknet on Jetson TX2. Since it is an open source framework written in C, we could modify the Darknet directly to apply the software optimization techniques to the network. For example,

TABLE III
TINY YOLO LAYER INFORMATION

Type	Size/Stride	# of In/Out Channels	Output Size
Convolution	3x3 / 1	3 / 16	416 x 416
Maxpool	2x2 / 2	16 / 16	208 x 208
Convolution	3x3 / 1	16 / 32	208 x 208
Maxpool	2x2 / 2	32 / 32	104 x 104
Convolution	3x3 / 1	32 / 64	104 x 104
Maxpool	2x2 / 2	64 / 64	52 x 52
Convolution	3x3 / 1	64 / 128	52 x 52
Maxpool	2x2 / 2	128 / 128	26 x 26
Convolution	3x3 / 1	128 / 256	26 x 26
Maxpool	2x2 / 2	256 / 256	13 x 13
Convolution	3x3 / 1	256 / 512	13 x 13
Maxpool	2x2 / 1	512 / 512	13 x 13
Convolution	3x3 / 1	512 / 1024	13 x 13
Convolution	3x3 / 1	1024 / 1024	13 x 13
Convolution	1x1 / 1	1024 / 1025	13 x 13

we inserted some codes for performance profiling of each layer. To accurately measure the time spent in each layer, *cudaDeviceSynchronize()* API is used to isolate the executions between layers. Also, the time spent was measured using the *clock_gettime()* API provided by the standard C *time* library.

IV. SOFTWARE OPTIMIZATION TECHNIQUES

The overall flow of the proposed methodology is shown in Figure 1. After the hardware platform and the neural network were selected in the first two steps, we applied a sequence of software optimization techniques from step 3 to step 6.

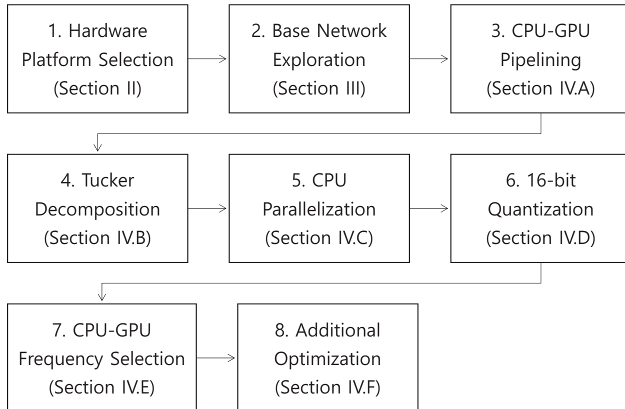

Fig. 1. Overall flow of the proposed optimization methodology

To improve the throughput performance, we first pipelined the network, aiming to overlap the CPU and GPU operations maximally. Afterwards, other optimization techniques were performed to reduce the bottleneck time based on the profiled execution times of CPU and GPU. To reduce the GPU execution time, we applied two techniques, Tucker decomposition, and 16-bit quantization. To reduce the CPU execution time, we parallelized the CPU computation by multi-threading.

After software optimization was performed, we explored the CPU and GPU frequencies to minimize the energy consumption of the system in step 7. At last, we emulated the end-to-end test scenario to check if the proposed solution performs as expected. During this final test, we identified some unexpected bottlenecks and so applied an additional optimization to solve the problems.

Table IV summarizes the performance improvement from each optimization step. In the rest of this section, software optimization techniques are explained in detail.

TABLE IV
STEP-BY-STEP PERFORMANCE IMPROVEMENT RESULTS

Step	Running Time	FPS	Improvement ratio
Baseline	1150s	17.4	1.0x
Pipelining	660s	30.3	1.74x
Tucker	540s	37.0	
Selective-Tucker	530s	37.7	1.25x
CPU Parallelization	502s	39.8	1.06x
16-bit Quantization	502s	39.8	
Selective-Quantization	460s	43.5	1.09x

A. CPU-GPU Pipelining

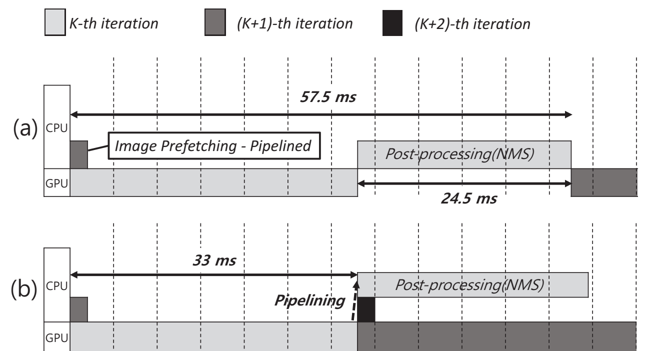

Fig. 2. Illustration of the CPU-GPU pipelining procedure

While the CNN structure of Table III represents the main algorithm, an image recognition system has a pre-processing step that fetches an input image from the disk sequentially, and a post-processing step, NMS step, as explained in the previous section. Both steps are performed on the CPU side. In the baseline Darknet implementation of Tiny YOLO, the pre-processing step and the other two steps are pipelined as shown in Figure 2 (a). As the first step of optimization, we added another pipeline stage between the second and third steps to overlap the CPU and GPU operations as illustrated in

Figure 2 (b), To this end, we modified the Darknet by adding a buffer between these two steps.

Table V displays the profiled information on the execution times on CPU and GPU. By simply pipelining the second and third steps, we could reduce the inference time from 1150 sec to 660 sec (42.6%) since the post-processing step takes a significant amount of execution time.

TABLE V
EXECUTION TIME COMPARISON BETWEEN THE BASELINE AND THE PIPELINED NETWORK

	Inference Time	CPU Time	GPU Time
Baseline	1150 sec	488 sec	656 sec
Pipelined	660 sec	488 sec	656 sec

B. Tucker Decomposition

Since GPU is the bottleneck in the pipelined network, it was necessary to reduce the execution time of the CNN. Since convolution layers are the most time consuming and require large memory space, several approximate computing methods have been developed to reduce the computation time and memory requirements. Tucker decomposition is one of such methods we adopted in the proposed solution [10].

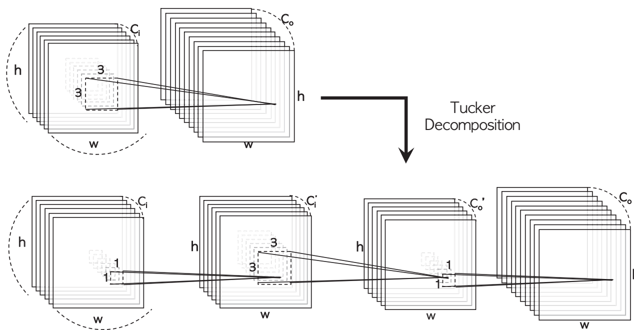

Fig. 3. Tucker Decomposition

Figure 3 shows how one convolution layer with 3×3 kernel size, C_i input channels, and C_o filters can be decomposed to three small convolution layers that include two 1×1 convolution layers and one 3×3 convolution layer by Tucker decomposition. The 1×1 convolution reduces the number of input feature maps from C_i channels to $C_i^{'}$ channels. The 3×3 convolution takes the output from the previous convolution as input and generates the output feature maps with $C_o^{'}$ channels. Finally, the 1×1 convolution increases the number of feature maps to C_o channels, which is equal to the original number of output channels. The number of multiplications for each pixel in the original convolution layer is $3 \times 3 \times C_i \times C_o$, which is reduced to $C_i \times C_i^{'} + 3 \times 3 \times C_i^{'} \times C_o^{'} + C_o^{'} \times C_o$ after Tucker decomposition.

In this technique, speed and accuracy trade-off can be adjusted by two variables, $C_i^{'}$ and $C_o^{'}$. As a rule of thumb, we set $C_i^{'}$ and $C_o^{'}$ to the half of C_i and C_o, respectively, as depicted in Table VI except two layers, Conv2 and Conv3. Since the number of input channels is small in Conv2 layer, we omit the first 1x1 convolution in Tucker decomposition.

TABLE VI
PARAMETERS FOR TUCKER DECOMPOSED LAYERS

Layer	C_i	C_o	C_i'	C_o'
Conv2	16	32	16	20
Conv3	32	64	20	32
Conv4	64	128	32	64
Conv5	128	256	64	128
Conv6	256	512	128	256
Conv7	512	1024	256	512
Conv8	1024	1024	512	512

TABLE VII
LAYERWISE PROFILE RESULT FOR EACH OPTIMIZATION TECHNIQUE

	Baseline	Tucker	Selective-Tucker	FP16	Selective-FP16
Conv1	120	120	120	148	120
Conv2	62	77	62	76	62
Conv3	34	44	34	35	34
Conv4	26	25	25	19	19
Conv5	23	21	21	13	13
Conv6	25	18	18	12	12
Conv7	90	46	46	31	31
Conv8	171	72	72	48	48
Conv9	27	26	26	17	17

With Tucker decomposition, we could reduce the total inference time to process 20,000 images from 660 sec to 540 sec as shown in the fourth row of Table IV, which is smaller than the time limit of 10 minutes.

From the layer-level profiling of execution times, it was observed that Tucker decomposition is not always beneficial. As shown in Figure 4, three convolution layers from Conv2 to Conv3 become slower if Tucker decomposition is applied. Thus rather than applying Tucker decomposition to all convolution layers, we selectively applied Tucker decomposition to the layers. As a result, we could reduce the execution time to 530 seconds with the selective-Tucker method as reported in the fifth row of Table IV. From the profiling, however, we observed that the post-processing step, NMS step, became the performance bottleneck that limits the speed improvement only by 10 seconds after the selective Tucker is applied. Thus, we moved to the next optimization step of CPU parallelization.

C. CPU Parallelization

Algorithm 1 shows the pseudo code of the NMS step where the main loop is applied to each object class (line 2), which gives an opportunity of parallelization. Among predictions made from the CNN, it first invalidates predictions whose confidence value is smaller than a threshold (line 4 to 6). Then redundant predictions are removed (line 7 to 9). IOU (intersection of union) is a metric to quantify how the proposed bounding box is close to the ground truth bounding box [3]. Finally, the remaining predictions are valid ones.

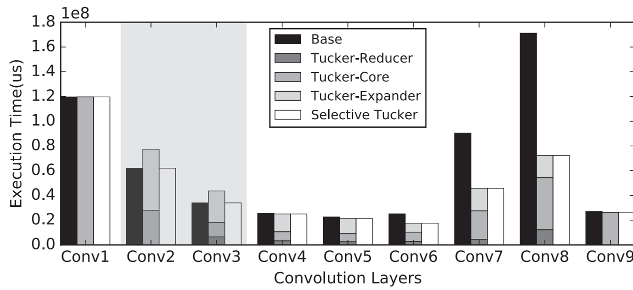

Fig. 4. Execution time profile of convolution layers before and after Tucker decomposition

Algorithm 1 Non-Maximum Suppression in Tiny YOLO network

```
1:  procedure NMS(classes, predictions, thrs)
2:      for c in classes do
3:          t ← predictions[c]
4:          for each p in t do
5:              if p.conf < thrs.conf then
6:                  remove p
7:          for each (p1, p2) : p1.conf > p2.conf in t do
8:              if IOU(p1, p2) > thrs.iou then
9:                  remove p2
10:         print each element of t
```

Since Jetson TX2 platform has a multi-core CPU, we parallelized the NMS step by using the OpenMP library [18]. In consequence, the CPU execution time of the NMS step was greatly reduced the total inference time from 530 sec to 502 sec and GPU became the performance bottleneck again.

D. 16-bit Quantization

The next optimization method was to reduce the precision of data representation from 32-bit floating point to 16-bit floating point, which is denoted as 16-bit Quantization. While aggressive quantization has been extensively researched in the design of specialized neural processors, there is no benefit of using smaller bit widths than 16-bit in Jetson TX2 in the computation time.

Jetson TX2 supports 16-bit data type called fp16 and fp16 data type should be used with the fp16 APIs, including the math function APIs and basic operation APIs such as addition, subtraction, multiplication, and division. While the cuDNN library supports 16-bit precision in Jetson TX2, we had to re-implement the GPU kernels that are directly implemented by the Darknet with fp16 APIs for 16-bit Quantization. Also, the input feature maps and the filter weights must be converted into fp16 precision.

Since quantization reduces the bit width of data only, it was expected that it is beneficial in all convolution layers. Thus we applied fp16 operations to all Tiny YOLO layers. Interestingly, as can be seen in the seventh row '16-bit quantization' of Table IV, no speed-up was obtained. From layer-level profiling, it was observed that the first three convolution layers became slower after 16-bit quantization as shown in Figure 5,

To find the cause of this unexpected result, we examined the profiling data obtained by the *nvprof* profiler provided by NVIDIA for the 1st and the 8th convolution layers which showed two largest differences between fp16 and float operations. Table VIII summarizes some data obtained from the *nvprof* profiler. The number cache accesses are decreased with fp16 operations in both layers as expected since the same cache line contains twice more data with fp16 types. The profiling result revealed that the performance largely depends on the number of instructions. At the first convolution layer, the fp16 kernel executes 2.5× more instructions than the 32-bit kernel, which overshadowed the benefit of reduced cache access in the execution time. On the other hand, there is a small difference at the 8th convolution layer. Thus the reduced cache access results in the execution time reduction. It seems that the cuDNN library provides more optimized kernels for 32-bit float data than the kernels for fp16 data.

TABLE VIII
PROFILING RESULTS FOR TWO CONVOLUTION LAYERS BEFORE AND AFTER 16-BIT QUANTIZATION

	Conv1		Conv8	
	FP16	**Float**	**FP16**	**Float**
L2 Cache Read	419 K	858 K	2,247 K	5,109 K
L2 Cache Write	173 K	346 K	16 K	58 K
Total Number of Instructions	30 M	12 M	48 M	43 M

Hence, to optimize the overall network, we applied the fp16 operation selectively from the fourth convolution layer and the inference time was shortened from 502 sec to 460 sec (9.4%).

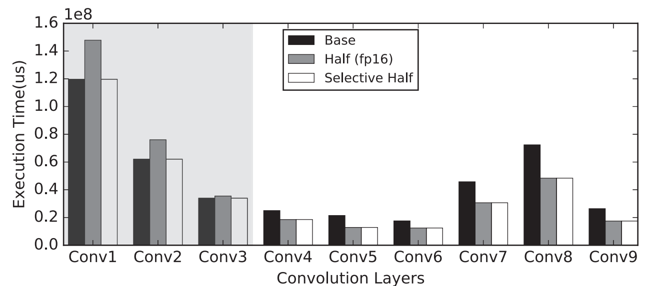

Fig. 5. Execution time profile of convolution layers before and after 16-bit quantization

E. CPU-GPU Frequency Selection

After a sequence of software optimizations, the total inference time is reduced to 460 seconds, which means that we could finish 140 seconds earlier than the time limit. So we could use the spare time to minimize the energy consumption further by adjusting the operating frequencies of processing elements. If we decrease the frequency, it will increase the execution time, but decrease the power consumption. Since the energy is the integral of the power consumption until the

TABLE IX
CPU-GPU FREQUENCY EXPLORATION RESULTS

CPU Freq.	GPU Freq.	Exec. Time(s)	Energy Consumption(Wh)
1,114MHz	1,122MHz	58	0.180682
1,728MHz	1,122MHz	52	0.184183
1,728MHz	944MHz	59	0.184728
1,114MHz	1,301MHz	55	0.192547
1,728MHz	1,301MHz	49	0.195594
2,035MHz	1,122MHz	52	0.196636
2,035MHz	944MHz	58	0.196706
2,035MHz	1,301MHz	48	0.205224

completion time, we may reduce the total energy consumption by lowering the frequency of processing elements, particularly if a processing element is under-utilized.

Table IX shows the measurement results on the execution time, and energy consumption consumed to process 2,000 images with various pairs of CPU and GPU frequencies. Note that the table includes the pairs of frequencies only that can meet the time limit (60 sec). Energy consumption was measured with the WT310E powermeter in Table IX and inference is performed with local images to exclude the network delay. From this exploration of frequencies, the selected frequencies for CPU and GPU are 1,114MHz and 1,122MHz, respectively.

F. Additional Optimization

After finishing all optimizations, we tested the server-client program used in the challenge. To reduce the communication counts between the device and the server, we sent one packet of detection results for a group of 1000 images at once. It was observed that the server took about 15 seconds to process one packet. To solve this problem, We increased the confidence threshold from 0.01 to 0.1 in the NMS module to reduce the number of valid predictions, which reduced the size of the packet and so the processing time of the server.

V. Conclusion and Future Work

In this paper, we presented our solution that won the first prize in LPIRC 2017 by jointly optimizing speed, accuracy, and energy consumption systematically, resulting in the score of 0.11931, which is 2.7 times better than the winner of LPIRC 2016. We used a NVIDIA Jetson TX2 board as the hardware platform and Tiny YOLO as the image recognition network, respectively, by considering the trade-off between the speed and accuracy. We applied a sequence of optimization methods to reduce the total execution time, applying pipelining, Tucker decomposition, NMS multi-threading, and 16-bit quantization step by step. In addition, we searched for the optimal pair of CPU and GPU frequencies to minimize the energy consumption, maintaining the same accuracy level.

There are more rooms for further performance improvement.

1) Design Space Exploration for Tucker Decomposition: As stated above, in Tucker decomposition, speed and accuracy trade-off is determined by two variables, $C_i^{'}$ and $C_o^{'}$. Since it took a few days to optimize the network for a given pair of values, We chose them by a rule of thumb without exploring the full design space. It is left as a future work to determine the optimal value pair.

2) Merging Batch Normalization into Weights: When processing images, the normalization process is performed after the convolution operation to make the output feature map have a unit variance, and zero mean. Since normalization is a linear transformation, it can be merged into the convolution operation by changing the filter weights. With this technique, the overall network execution time can be further reduced.

3) Image Size Reduction: By reducing the size of the input feature map, we can reduce the computation complexity as well as the memory size, but paying the cost of accuracy loss. The tradeoff between the performance improvement and the accuracy loss needs to be investigated further.

Acknowledgment

This research was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government(MSIP) (No. NRF-2016R1A2B3012662

References

[1] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Proceedings of NIPS*, 2012, pp. 1097–1105.
[2] O. Russakovsky *et al.*, "Imagenet large scale visual recognition challenge," *IJCV*, vol. 115, no. 3, pp. 211–252, 2015.
[3] K. Gauen *et al.*, "Low-power image recognition challenge," in *Proceedings of ASP-DAC, IEEE*, 2017, pp. 99–104.
[4] F. N. Iandola *et al.*, "Squeezenet: Alexnet-level accuracy with 50x fewer parameters and¡ 0.5 mb model size," *arXiv preprint 1602.07360*, 2016.
[5] J. Redmon, "Yolo: Real-time object detection," http://pjreddie.com/darknet/yolo/, 2013–2016.
[6] N. P. Jouppi *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *Proceedings of ISCA*. ACM, 2017, pp. 1–12.
[7] Huawei announces the kirin 970 – new flagship soc with ai capabilities. [Online]. Available: https://www.androidauthority.com/huawei-announces-kirin-970-797788/
[8] H. Li *et al.*, "Pruning filters for efficient convnets," *arXiv preprint 1608.08710*, 2016.
[9] E. Park, J. Ahn, and S. Yoo, "Weighted-entropy-based quantization for deep neural networks," in *Proceedings of CVPR, IEEE*, 2017, pp. 5456–5464.
[10] Y. D. Kim *et al.*, "Compression of deep convolutional neural networks for fast and low power mobile applications," *arXiv preprint 1511.06530*, 2015.
[11] S. Ren *et al.*, "Faster r-cnn: Towards real-time object detection with region proposal networks," in *Proceedings of NIPS*, 2015, pp. 91–99.
[12] J. Dai *et al.*, "R-fcn: Object detection via region-based fully convolutional networks," in *Advances in neural information processing systems*, 2016, pp. 379–387.
[13] J. Redmon *et al.*, "You only look once: Unified, real-time object detection," in *Proceedings of CVPR, IEEE*, 2016, pp. 779–788.
[14] J. Redmon and A. Farhadi, "Yolo9000: better, faster, stronger," *arXiv preprint 1612.08242*, 2016.
[15] W. Liu *et al.*, "Ssd: Single shot multibox detector," in *Proceedings of ECCV*. Springer, 2016, pp. 21–37.
[16] J. Deng *et al.*, "Imagenet: A large-scale hierarchical image database," in *Proceedings of CVPR, IEEE*. IEEE, 2009, pp. 248–255.
[17] J. Redmon, "Darknet: Open source neural networks in c," http://pjreddie.com/darknet/, 2013–2016.
[18] Openmp website. [Online]. Available: https://www.openmp.org/