Singapore Management University
Institutional Knowledge at Singapore Management University

Research Collection School Of Information Systems

School of Information Systems

6-2016

DeepSense: A GPU-based deep convolutional neural network framework on commodity mobile devices

HUYNH NGUYEN LOC
Singapore Management University, nlhuynh.2014@phdis.smu.edu.sg

Rajesh Krishna BALAN
Singapore Management University, rajesh@smu.edu.sg

Youngki LEE
Singapore Management University, YOUNGKILEE@smu.edu.sg

DOI: https://doi.org/10.1145/2935643.2935650

Citation

HUYNH NGUYEN LOC; BALAN, Rajesh Krishna; and LEE, Youngki. DeepSense: A GPU-based deep convolutional neural network framework on commodity mobile devices. (2016). *WearSys'16: Proceedings of the 2016 Workshop on Wearable Systems and Applications: June 30, 2016, Singapore*. 25-30. Research Collection School Of Information Systems.
Available at: https://ink.library.smu.edu.sg/sis_research/3276

DeepSense: A GPU-based Deep Convolutional Neural Network Framework on Commodity Mobile Devices

Loc N. Huynh, Rajesh K. Balan, Youngki Lee
School of Information Systems
Singapore Management University
{nlhuynh.2014, rajesh, youngkilee}@smu.edu.sg

ABSTRACT

Recently, a branch of machine learning algorithms called deep learning gained huge attention to boost up accuracy of a variety of sensing applications. However, execution of deep learning algorithm such as convolutional neural network on mobile processor is non-trivial due to intensive computational requirements. In this paper, we present our early design of *DeepSense* - a mobile GPU-based deep convolutional neural network (CNN) framework. For its design, we first explored the differences between server-class and mobile-class GPUs, and studied effectiveness of various optimization strategies such as *branch divergence elimination* and *memory vectorization*. Our results show that *DeepSense* is able to execute a variety of CNN models for image recognition, object detection and face recognition in soft real time with no or marginal accuracy tradeoffs. Experiments also show that our framework is scalable across multiple devices with different GPU architectures (e.g. Adreno and Mali).

Keywords

Mobile GPU; Mobile sensing application; Deep learning

1. INTRODUCTION

A variety of smart glasses are continuously emerging, opening up new opportunities for *continuous vision sensing applications*. For example, *WhoIsThis* application reminds user of names of nearby people in a large conference by recognizing faces from first-person-view video streams. The conventional processing pipeline in these applications is to continuously capture videos or images, extract a set of distinguishing features, and run inference algorithms. Nowadays, various deep learning algorithms such as deep neural network (DNN) or convolutional neural network (CNN) are getting huge attention, as they are known to achieve higher inference accuracy for various vision-based applications [4, 9, 8].

Deep learning algorithms, however, incur heavy computational overhead and power consumption when executing on wearable or mobile devices. A conventional approach to overcome these challenges is offloading computation onto powerful clouds. However, this approach has a few fundamental limitations. First, it has potential threats to expose private data of users. Captured first-person-view images often contain sensitive information such as where they are located, who they are with, which activities they are doing. This may prevent users from offloading data to the clouds, invalidating the use of cloud resources. Second, continuously sending video streams to clouds consumes huge bandwidth which is a big concern when users are connected via cellular networks. Moreover, offloading is no longer effective in scenarios where network connectivity is poor or unavailable.

In this paper, we propose and explore an alternative approach, a *DeepSense* framework, to execute deep learning algorithms on mobile devices without cloud offloading. By leveraging mobile graphical processing unit (GPU) recently integrated into smartphones, we aim to support developers for 1) adopting a wide range of existing DNN, CNN models trained to run on server-class machines with minimal programming effort, 2) achieving real-time or soft real-time latency for continuous sensing and intervention, 3) minimizing energy consumption on the computing mobile devices. Our *DeepSense* framework is built up on OpenCL [10], which is now officially supported by a number of mobile GPUs such as Adreno and Mali.

As a first step towards this direction, our work is focused on supporting CNN that is widely adopted by various vision sensing applications. We first investigated several existing CNN models (such as AlexNet [4], Vgg-F [1], Vgg-M [1], Vgg-verydeep-16 [9], Vgg-Face [8], etc.), and found out that over 90% of computation occurred within convolutional layers, increasing the processing latency significantly. To reduce the latency, *DeepSense* offloads the convolutional layers to mobile GPUs considering unique characteristics of mobile GPUs as well as the data representation within the CNN structure. Moreover, it adopts various optimization strategies such as *branch divergence elimination* and *memory vectorization* to further reduce latency. Finally, *DeepSense* provides developers the ability to trade off accuracy and latency with the use of half floating points in computation.

We conducted extensive experiments on 3 commodity smartphones (Samsung Galaxy S5, Note 4 and S7) with three 3 CNN models (Vgg-F, Vgg-M and Vgg-16). Our results show that *DeepSense* can achieve soft real-time latency (less than 1.5 second) for various CNN models. With the use of half floating points, *DeepSense* can further reduce latency; for instance, running Vgg-F takes 403ms, 259ms and 155.2ms with only 4.62% accuracy drop on Samsung Galaxy S5, Note

WearSys '16, June 30, 2016, Singapore.
© 2016 ACM. ISBN 978-1-4503-4326-8/16/06. . . $15.00
DOI: http://dx.doi.org/10.1145/2935643.2935650

4 and S7 respectively. We believe that more carefully devised optimization techniques and adoption of more powerful GPUs on smartphones would make it feasible to execute large-scale models on mobile devices in real time.

The contribution of our paper can be summarized as follows:

- We proposed *DeepSense* , an OpenCL-based framework to run various deep learning inference algorithms on mobile GPUs; it now supports various CNN models with low latency and power consumption. Note that OpenCL has highly advantageous in that it supports a wide range of commodity mobile GPUs (e.g., Adreno and Mali) comparing to CUDA-based devices (e.g., Nvidia Jetson [5]).
- We explored a variety of design choices and optimization techniques to efficiently execute CNN on mobile devices (such as memory vectorization, data representation, usage of half floating points).
- We conducted experiments using various models (AlexNet, Vgg-F, Vgg-M, Vgg-16, Vgg-Face, etc.) on variety of mobile GPU (Adreno 330, 410 and Mali T880). Our preliminary results show that we are able to execute Vgg-F in real-time (803ms on S5, 480ms on Note 4 and 361ms on S7) without any accuracy drop. In addition, with the calculation of half floating point enabled, the execution time of Vgg-F on S5 is reduced to 450ms by sacrificing only 4.62% accuracy.

2. BACKGROUND

We begin with a brief introduction of the two underlying techniques of our system: OpenCL and CNN.

2.1 OpenCL

OpenCL [10] is a framework to support parallel programming across heterogeneous platforms such as central processing units (CPUs), graphical processing units (GPUs) or even digital signal processors (DSPs). Recently, OpenCL has been widely supported on both popular smartphone processors (e.g., Snapdragon and Exynos) and popular mobile GPUs(e.g., Adreno and Mali).

In order to use OpenCL for parallel programming, developers first need to divide their problem into a number of small identical sub-problems, then implement each subproblem as OpenCL kernel code. The OpenCL run-time will spawn multiple parallel processing units (i.e., *work-items*), each runs independent compiled kernel program and is scheduled to be executed on multi-core CPU, GPU or both depending on the charateristics of application requirements.

Its flexible parallel programming model and applicability on a wide range of mobile processors serve the goal and functionality of *DeepSense*, and thus we adopt OpenCL as our underlying programming and execution framework.

2.2 Convolutional Neural Network

Convolutional neural network (CNN) is a type of feed-forward neural network that is widely adopted for image and video recognition [4, 8].

Figure 1 shows an example of CNN architecture which consists three fundamental layers: *convolutional*, *pooling* and *fully connected*. To briefly explain, each convolutional layer applies multiple filters to convert lower-level features from the previous layer into higher-level features. A pooling layer is used to capture invariants that do not change even when an image output by a convolutional layer is translated, rotated or scaled. Finally, a fully connected layer aggregates extracted high-level features for further classification task.

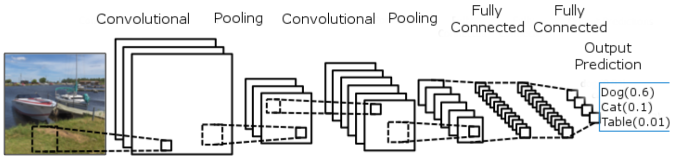


Figure 1: Convolutional Neural Network [5]

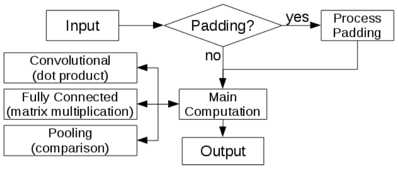


Figure 2: Processing Flow of Single Layer

As shown in Figure 2, a CNN layer consists of two main processing steps: *Input Padding* and *Main Computation*. The input padding step is required to match the output of previous layer as an input of current layer. For example, borders of input images can be zero-padded to match the input size of the current layer. Once padding is done, each layer conducts the core computational operations; for convolutional layers, dot products are the key operations. For pooling and fully-connected layers, comparison and matrix multiplications are the core operations, respectively.

3. CNN PERFORMANCE BREAKDOWN

In this section, we breakdown the performance of CNN in order to identify its bottleneck for optimization. To study the performance of CNN, we use five existing models including AlexNet, VGG models (Vgg-f, Vgg-m, Vgg-verydeep-16, Vgg-Face). Table 1 shows the important properties (such as application, accuracy, number of parameters and architecture) of the models. Vgg-Face is trained to recognize human faces (out of 2,622 candidates) within an image while the other models are trained to classify images into one of 1,000 categories. It is noticeable that the accuracy is affected by two factors: (1) the number of convolutional layers, and (2) the size of model (which implies the size of filters and the stride to apply the filter on the input).

To understand the bottleneck, we measure the running time of different CNN layers on Samsung Galaxy Note 4. We implemented a CPU version of a CNN executor in C/C++ using Android NDK. For best CPU performance, we com-

	App	Size (M)	Top-1 Acc.	Top-5 Acc.	Arch.
AlexNet	IR	60.8	58.2%	80.8%	5c,3p,3fc
Vgg-f	IR	60.8	58.6%	80.9%	5c,3p,3fc
Vgg-m	IR	102.9	63.1%	84.5%	5c,3p,3fc
Vgg-16	IR	138.4	71.7%	90.5%	13c,5p,3fc
Vgg-face	FR	145	98.95%	-	13c,5p,3fc

Application(IR: image recognition, FR: face recognition),
Size: number of parameters
Architecture(c: convolutional layers, p: pooling layers, fc: fully connected layers)

Table 1: CNN Models

	Conv.(ms)	FC.(ms)	Pooling(ms)	Total(ms)
Vgg-F	8072	1079	26	9177
Vgg-M	19521	2122	156	21800
Vgg-16	213371	2408	882	216662

Table 2: CNN Latency Breakdown

piled the program with *armeabi-v7a ABI(Application Binary Interface)* to enable external floating point processing unit.

Table 2 shows the excution time per types of layers. Most importantly, computation bottleneck is occurred within convolutional layers for all three inspected models. For instances, over 87% of the processing time in Vgg-F is occupied by the convolutional layer followed by 11% and 0.2% for fully-connected and pooling layers, respectively. For a large model such as Vgg-16, over 98% of computation time is taken in convolutional layers. We also figured out that the total number of addition-multiplication operations within convolutional layers is much higher than operations within fully connected layers (e.g. Vgg-16 requires 15346M addition-multiplication operations for convolutional layers comparing to only 123M for fully connected layers).

4. DeepSense SYSTEM OVERVIEW

Figure 3 shows the overall architecture of *DeepSense* which consists of four main components including *model converter, model loader, inference scheduler, executor*.

Model converter: This module translates existing pretrained models from multiple representations into our predefined format due to different model's representation of different DNN frameworks. At present, *DeepSense* supports 3 formats of DNN trained by Caffe, MatConvNet and Yolo for different types of applications.

Model loader: Application triggers this module to load converted models into memory. It allocates appropriate host(CPU) and device(GPU) memory for individual layer's data structure to store both CNN/DNN's meta-data and weights. Our current implementation of *DeepSense* stores model's meta-data in host memory while all weights of convolutional and fully connected layers are stored in device memory. Other configurations such as enabling half floating point optimization is also processed by this module.

Inference Scheduler: Inference requests are submitted into this module's queue to be scheduled for execution. Since GPU is known to be good at executing single task, submitting multiple requests to mobile GPU might interfere each other tasks and increase the latency. In order to prevent interference, this scheduler is built to guarantee that only a single request is executed at a time.

Executor: Execution of inference request is taken place in this module. *Executor* takes allocated model's memory from *model loader*, input data from inference request and compute the output of CNN/DNN. During execution pipeline, only parts of operations such as padding, intermediate memory allocation are executed by CPU while the other heavy computation parts (e.g. convolutional, pooling and fully connected operations) are done by mobile GPU.

5. DESIGN CONSIDERATIONS

In this section, first, we investigate behaviours of *branch divergence* and *memory coalescing* on mobile GPU. Second, based on our observations, we propose a memory layout to represent input and parameters in effective manner to achieve low latency on mobile GPU. Finally, we study the impact of half floating point approximation on both the latency and accuracy for different CNN models.

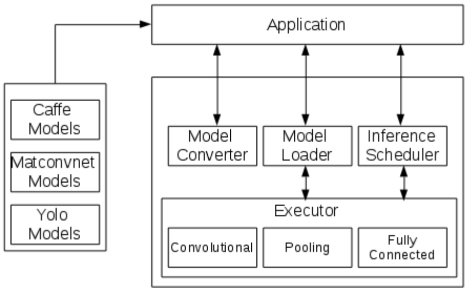


Figure 3: *DeepSense* System Overview

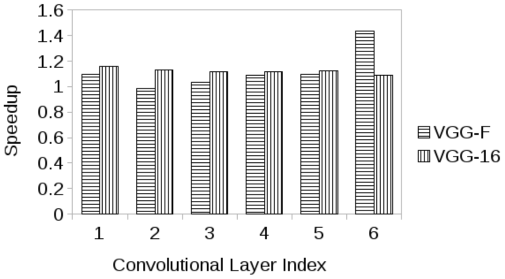


Figure 4: Explicit and Implicit padding

We perform evaluations on three different devices including Samsung Galaxy S5, Note 4 and S7 to make our design choices. These devices are powered by two different mobile GPU architectures, Adreno and Mali. Our version of Galaxy S7 integrates Mali T880 GPU while S5 and Note 4 are powered by Adreno 330 and 420 respectively.

5.1 Branch Divergence

One important issue to improve the latency of CNN execution on GPU is handling *padding* operation efficiently. This operation takes the input and pads data into in order to get desired size. Most of existing CNN models requires padding operations in many layers. Conventional CPU approach to solve this problem is to ignore padded input values when processing. However, this approach imposes condition branches which are inefficient to run on GPU (branch divergence problem). Since *DeepSense* is proposed to execute existing models, this problem should be studied carefully.

Within GPU program, branch divergence is a common problem which causes the GPU to process both conditional blocks of code. This problem increases the execution time of every work item running openCL kernel. However, behaviour of branch divergence when executing CNN on mobile GPU is still not fully evaluated. To address this problem, we consider two types of padding including implicit and explicit padding when executing CNN. The former one processes padding (e.g. ignore padded input using conditional branch) within GPU kernel code and possibly leads to *branch divergence*. On the other hand, the latter approach tries to avoid *branch divergence* by allocating new memory block and migrating corresponding input data into new location before executing GPU kernel. However, overhead occurred by multiple memory copying operations may significantly overwhelm the running time of GPU code.

We carefully evaluate both approaches with two differ-

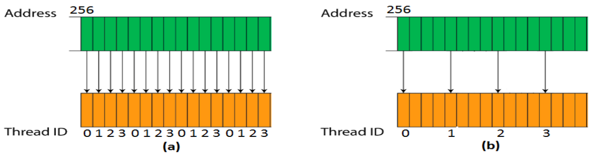
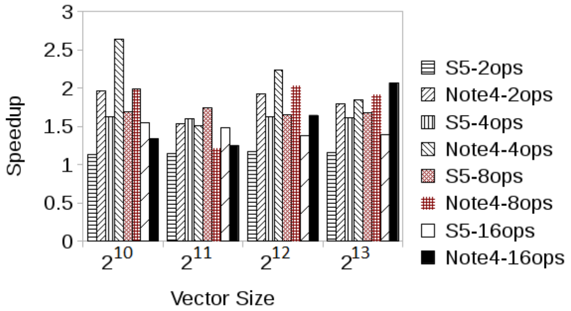


Figure 5: Memory Coalescing vs Memory Vectorization



Each work item computes a fraction of output values (2, 4, 8, 16 and 32 values)

Figure 6: Memory Coalescing and Memory Vectorization.

ent models (Vgg-16 and Vgg-f) on Samsung Galaxy Note 4 with Adreno 420 GPU. For easy comparison, we use speedup which is defined as a latency fraction between using implicit and explicit approaches.

$$speedup = \frac{runtime_{implicit}}{runtime_{explicit}}$$

Figure 4 shows speedup over the first six layers of two models. In most of cases, running explicit padding is faster than executing implicit padding within GPU kernel. We observe that the sixth layer of model Vgg-F has high speedup due to two reasons. First, the input size of that layer is small so there is little overhead of padding operations. Second, the amount of addition and multiplication operations that needs to be processed is large so the processing latency is overwhelming the padding overhead. Finally, as Vgg-16 consists of more layers than Vgg-F, we also notice the similar characteristic as the processing reaches later layers.

5.2 Memory Coalescing vs Memory Vectorization

We observe that correctly reading data into GPU's work items can significantly reduce latency. We explore two approaches including *memory coalescing* and *memory vectorization*. *memory coalescing* makes multiple work items access memory within single transaction by leveraging underlying memory bank. Whenever a work item reads a single item, memory bank caches consecutive items so that other work items can access data items faster. In contrast, *memory vectorization* utilizes high memory bandwidth by optimistically reading a block of contiguous items into work item's private memory. Memory access patterns are shown in Figure 5.

We use vector addition program to evaluate two proposed techniques. To compute each value within output vector, it requires only a single addition operation but accesses to three memory locations (two input and one output locations). This application is best fit for us to measure the memory throughput and latency of two approaches. To evaluate, we define speedup as a latency fraction between memory coalescing and memory vectorization for comparison.

Mem Repre.	# blks to access	Max blk size
[c x d x d]	c x d blocks of d values	11
[d x d x c]	d x d blocks of c values	512

Table 3: Memory Representation and Maximum Number of Memory Accesses per Work Item

$$speedup = \frac{runtime_{coalescing}}{runtime_{vectorization}}$$

Figure 6 shows the speedup between two techniques. First, we observe that memory vectorization outperforms memory coalescing in all cases. Second, using block size of 4 values results in speedup around 1.7 on S5 and 2.0 on Note 4.

As the result, we organize our data in a way to be loaded as a block of contiguous data into each work item using memory vectorization.

5.3 Memory Representation

Representation of data in memory also affect latency of executing CNN. Within OpenCL kernel, parameters are represented as 1D array or 3D image of data. The input and parameters of convolutional layer is 3D and 4D array respectively which have to be reshaped into 1D array or 3D image. However, maximum size of 3D image is also limited by OpenCL framework and the running GPU hardware. To address arbitrary size of parameters and input, all data is reshaped into 1D array. The question is how to represent it in order to achieve best performance.

Suppose we have a convolutional layer with these characteristics:

- Input: size of [h x w] and c channels
- Weight parameters: n filters, each filter has size of [d x d] and c channels
- Output: size of [h' x w'] and n channels

To compute single output value, CNN does a dot product between a single filter and portion of input data with identical size to filter. This operation requires to read filter's parameters and portion of input into work item. In the end, each work item will trigger a lot of memory reading operations. Reducing number of memory reading operations per work item may result in improving latency. Fortunately, OpenCL provides vload$n$/vstore$n$ to allow reading/writing a block of contiguous memory up to 16 float values at a time. Reducing total number of memory reading operations is now corresponding to maximizing the size of contiguous memory block. We try to organize data in CNN in the way that we can maximize the size of single block that each work item has to read into its memory space.

The filter size and input which is used in dot product operation can be represented in 2 ways: [c x d x d] and [d x d x c]. However, since the input to this operation is only a portion of layer's input, its memory is not contiguous. In this case, the former approach can access a block size of maximum d contiguous values while the latter approach can access to a block size of maximum c contiguous values.

Table 3 shows the total number of accesses to contiguous memory block for each representation. We investigate several models including AlexNet, Vgg-verydeep-16, Vgg-f, Vgg-M and observe that the maximum size of d is much smaller than the maximum size of c. As a result, using [d x

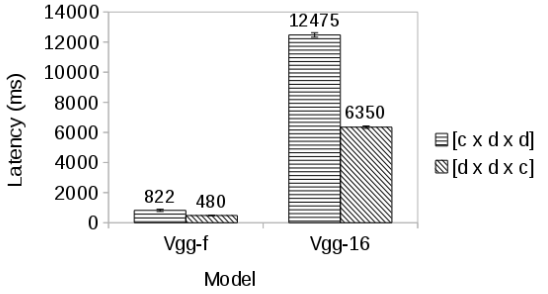


Figure 7: Latency of Memory Representations

d x c] as representation of filter, we can maximize the size of contiguous memory block as well as reduce the number of memory reading operations that needed to be called.

Figure 7 shows the latency comparison between using two representation approaches for different CNN models on Samsung Galaxy Note 4. Important observation is that using [d x d x c] approach is more efficient than other approach. For instances, Vgg-16 reduces latency 1.96 times when migrating from using [c x d x d] to [d x d x c] representation.

Finally, input and filters are represented as [h x w x c] and [n x h x w x c] corresponding to our design choice.

5.4 Half Floating Point

Another optimization technique to improve executing latency on GPU is to minimize transferring data between host memory and GPU. In this section, we propose an algorithm to convert parts of CNN into half floating point to reduce memory bandwidth usage for data transferring in order to improve executing time.

During CNN inference using GPU, a large amount of memory bandwidth and latency is consumed for reading/storing floating point values from/to main memory. To address this problem, we convert parts of CNN's parameter into floating point 16 bits instead of 32 bits. Cutting half of data to load into single work item reduces latency significantly. However, this approach may suffer accuracy drop. To address this problem, we develop a greedy algorithm as shown in algorithm 1 to choose the most suitable layers to convert parameters into half floating point as follow:

```
Algorithm 1: Half floating point approximation
Input: (1)desired accuracy_loss, (2)CNN model, (3)list
     T of convolutional layers of CNN, (4) validation
     set
Output: list L of convolutional layers
Algorithm:
current_accuracy_loss = 0
while current_accuracy_loss < accuracy_loss AND size(L)
     != size(T):
   tmp_list = {}
   for all layer l in T:
      if l is not in L
         temporarily convert l to floating point 16 bit
         compute tmp_accuracy_loss from validation set
         add l, tmp_accuracy_loss> to tmp_list
      endif
   endfor
   pick l from tmp_list with lowest tmp_accuracy_loss
   current_accuracy_loss = tmp_accuracy_loss
   if tmp_accuracy_loss < accuracy_loss
      add l into L
      permanently convert l into floating point 16 bit
   endif
endwhile
```

Model	CPU-FP32(ms)	GPU-FP32(ms)	GPU-FP16(ms)
Vgg-F	9177	480	259
Vgg-M	21800	1166	558
Vgg-16	216662	6315	2922

Table 4: Full and Half Floating Point Latency on Note 4

Model	Top-1 Acc. Drop	Top-5 Acc Drop
Vgg-F	5.82%	4.62%
Vgg-M	3.96%	3%
Vgg-16	2.62%	1.66%

Table 5: Half Floating Point Accuracy Drop

We set desired accuracy drop at 5% and run algorithm 1 on three image recognition models (Vgg-f, Vgg-m and Vgg-16). We use the first 5000 images from ILSVRC2012 validation set [4] to measure accuracy drop since it is also validation set used to evaluate original models.

First, it is surprising that we can convert all convolutional layers into using half floating point for less than 5% of top-5 accuracy drop. Table 5 also points out that low accurate model suffers accuracy drop more than high accurate models even though the number of layers and parameters to be converted into half floating point are less than other models.

Second, inference time reduces significantly in our experiments on Samsung Galaxy Note 4 as shown in Table 4. Converting to half floating point, latency reduces 1.85, 2.08, 2.16 times when executing Vgg-F, Vgg-M, Vgg-16 respectively. That means within convolutional layers, memory bandwidth is highly utilized and needed to be taken into consideration for further improvement.

5.5 Performance Overview

We combine several proposed techniques to design *DeepSense* framework. As shown in Table 4, *DeepSense* significantly reduces inference time up to 74 times comparing to conventional CPU implementation. For small and medium models such as Vgg-F and Vgg-M, *DeepSense* executes one inference within 600ms. For a large model such as Vgg-16, *DeepSense* is still able to provide reasonable latency within 3 seconds. Furthermore, energy consumption for single inference request is also shown in Table 6. From our calculation, continuously executing *DeepSense* for vision sensing with Vgg-F model can last up to 2.5 hours on commodity devices with only modest battery capacity at 2000mAh.

6. FUTURE DIRECTIONS

Enabling CNN inference on commodity mobile devices is just the beginning. There are still plenty of works to optimize our *DeepSense* framework.

GPU local memory: Latency can be mitigated by reducing memory bandwidth usage. Another technique to solve memory bandwidth problem is to leverage GPU's local memory which is shared across multiple work items. A work group of multiple work items only needs to load a large input

Model	FP-32(mJ)	FP-16(mJ)
Vgg-F	1135	665
Vgg-M	2584	1487
Vgg-16	14491	8767

Table 6: Consumed Energy on Galaxy Note 4

data once and share between each other. However, capacity of local memory is different across multiple devices and platforms. Hence, this approach should be considered carefully to work efficiently on wide range of devices. For example, Galaxy S5 has 8KB of local memory while S7 has 32KB.

Convolutional layer approximation: Another approach is to reduce the number of operations we have to process by re-training large convolutional layer into multiple smaller ones [3]. However, this approach should be deeply explored to find out the best manner to trade-off between accuracy drop and latency reduction.

Number of work items optimization: Finding optimal number of work items for OpenCL program also reduces latency significantly. Since amount of ALUs (Arithmetic Logic Unit) is different across devices, optimal number of work items might be different. Our next step is to focus on developing heuristic algorithm to estimate how many work items should be used for each CNN's layer across multiple devices with different GPU architectures.

7. RELATED WORK

Deep learning inference optimization: There has been a number of prior work to reduce training time of CNN and DNN [7]. However, little work has focused on optimizing inference time as most prior works used powerful servers and desktop machines for inferences. A few works aim at optimizing inference time. For instance, Vanhoucke et al. develops a suite of low-level optimization techniques to reduce the inference latency (e.g., using fixed point arithmetic and SSSE3/SSE4 instructions on x86 machines)[11]. Also, approximation techniques are developed to reduce latency with trade-offs in accuracy [3]. However, these studies were focused on powerful desktop or server machines.

Cloud offloading: Ha at el. proposed the Gabriel framework [2] to support cognitive assistance applications using cloudlet to minimize occurred latency. Different from these work, we focused on scenarios where cloud offloading is not feasible, and explored the opportunities to use mobile GPUs to enable real-time deep learning inferences.

Deep learning on mobile devices: Lane at el. have taken crucial first steps towards real-time execution of DNN and CNN on mobile devices [5, 6]. DeepX framework enables the execution of DNN and CNN on mobile devices [5] by splitting computations across multiple co-processors. In addition, the authors showed that it is feasible to run entire DNN for audio sensing applications on low-power mobile DSPs [6]. We believe that our work can complement DeepX in the following ways. First, DeepX is designed with a ML principal-driven approach where our works takes a system-driven optimization approach, giving the potential opprotunities to use both approaches together for further latency reduction. Second, DeepX is effective in reducing the latency of fully-connected layers while our framework focused on reducing latency of convolutional layers. Finally, we believe *DeepSense* is the first to run on OpenCL framework, and presents experiments with GPUs on commercially available mobile devices in the market.

8. CONCLUSION

In this paper, we propose an early design of *DeepSense*, a GPU-based deep convolutional neural network framework based on OpenCL for commodity mobile devices. We investigate padding and memory representation problems to make CNN work efficiently on mobile GPU. Our results show that *DeepSense* is able to execute several CNN models in real-time within a second. Furthermore, *DeepSense* enables user to leverage half floating point processing to significantly reduce latency with acceptable accuracy drop (e.g., Vgg-F inference time takes only 160ms on Samsung Galaxy S7 with only 4.62% accuracy drop).

9. ACKNOWLEDGMENT

This research is supported by the National Research Foundation, Prime Minister's Office, Singapore under its IDM Futures Funding Initiative.

10. REFERENCES

[1] K. Chatfield, K. Simonyan, A. Vedaldi, and A. Zisserman. Return of the devil in the details: Delving deep into convolutional nets. *arXiv preprint arXiv:1405.3531*, 2014.

[2] K. Ha, Z. Chen, W. Hu, W. Richter, P. Pillai, and M. Satyanarayanan. Towards wearable cognitive assistance. In *Proceedings of the 12th annual international conference on Mobile systems, applications, and services*, pages 68–81. ACM, 2014.

[3] M. Jaderberg, A. Vedaldi, and A. Zisserman. Speeding up convolutional neural networks with low rank expansions. *arXiv preprint arXiv:1405.3866*, 2014.

[4] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012.

[5] N. D. Lane, S. Bhattacharya, P. Georgiev, C. Forlivesi, L. Jiao, L. Qendro, and F. Kawsar. Deepx: A software accelerator for low-power deep learning inference on mobile devices.

[6] N. D. Lane, P. Georgiev, and L. Qendro. Deepear: robust smartphone audio sensing in unconstrained acoustic environments using deep learning. In *Proceedings of the 2015 ACM International Joint Conference on Pervasive and Ubiquitous Computing*, pages 283–294. ACM, 2015.

[7] M. Mathieu, M. Henaff, and Y. LeCun. Fast training of convolutional networks through ffts. *arXiv preprint arXiv:1312.5851*, 2013.

[8] O. M. Parkhi, A. Vedaldi, and A. Zisserman. Deep face recognition. *Proceedings of the British Machine Vision*, 1(3):6, 2015.

[9] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[10] J. E. Stone, D. Gohara, and G. Shi. Opencl: A parallel programming standard for heterogeneous computing systems. *Computing in science & engineering*, 12(1-3):66–73, 2010.

[11] V. Vanhoucke, A. Senior, and M. Z. Mao. Improving the speed of neural networks on cpus. In *Proc. Deep Learning and Unsupervised Feature Learning NIPS Workshop*, volume 1, 2011.