OpenCL caffe: Accelerating and enabling a cross platform machine learning framework

Junli Gu
gujunli@gmail.com

Yibing Liu
liuyb11@mails.tsinghua.edu.cn

Yuan Gao
yuan.gao@noplz.name

Maohua Zhu
zhumaohua@gmail.com

ABSTRACT

Deep neural networks (DNN) achieved significant breakthrough in vision recognition in 2012 and quickly became the leading machine learning algorithm in Big Data based large scale object recognition applications. The successful deployment of DNN based applications pose challenges for a cross platform software framework that enable multiple user scenarios, including offline model training on HPC clusters and online recognition in embedded environments. Existing DNN frameworks are mostly focused on a closed format CUDA implementations, which is limiting of deploy breadth of DNN hardware systems.

This paper presents OpenCL™ caffe, which targets in transforming the popular CUDA based framework caffe [1] into open standard OpenCL backend. The goal is to enable a heterogeneous platform compatible DNN framework and achieve competitive performance based on OpenCL tool chain. Due to DNN models' high complexity, we use a two-phase strategy. First we introduce the OpenCL porting strategies that guarantee algorithm convergence; then we analyze OpenCL's performance bottlenecks in DNN domain and propose a few optimization techniques including batched manner data layout and multiple command queues to better map the problem size into existing BLAS library, improve hardware resources utilization and boost OpenCL runtime efficiency.

We verify OpenCL caffe's successful offline training and online recognition on both server-end and consumer-end GPUs. Experimental results show that the phase-two's optimized OpenCL caffe achieved a 4.5x speedup without modifying BLAS library. The user can directly run mainstream DNN models and achieves the best performance for a specific processors by choosing the optimal batch number depending on H/W properties and input data size.

[†]OpenCL caffe code is released under BSD-2 Clause license at: `https://github.com/amd/OpenCL-caffe`

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

IWOCL '16 April 19-21, 2016, Vienna, Austria

© 2016 Copyright held by the owner/author(s).

ACM ISBN 978-1-4503-4338-1/16/04.

DOI: http://dx.doi.org/10.1145/2909437.2909443

Keywords

OpenCL; Deep neural networks (DNN); Deep learning frameworks

1. INTRODUCTION

DNN algorithms are high complex layered structured nonlinear models that extract patterns automatically from Big Data input. DNN's success was largely due to the accumulated Big Data and the massive computing capabilities provided by GPU processors. Today's main stream DNN model has millions to billions parameters with its layer depth increasing from 8-layer Alexnet [8] in 2012 to 152-layer [7] in 2015. The high complexity of the DNN model requires a software framework that provides well defined algorithmic APIs for domain experts, meanwhile deploys high optimized machine learning kernels to release the hardware processors' computing capabilities. In the past few years, a couple of DNN software frameworks have shown up including ConvNet [3], caffe, MXNet [4], and Tensor Flow [5] etc. Among these frameworks, caffe tends to become one of the most popular thanks to its good performance and clear interfaces for users. DNN has been successfully deployed in large scale of image and voice recognition, face recognition, searching engine, advertisements, financial fraud detection etc. Depending on the specific user scenarios, the underlying system requirement has been evolving from super computers, HPC clusters to commercial cloud and embedded systems. The application trend has driven towards a cross platform framework that enables the same DNN model being deployed at different scenarios with minimum developing efforts. Unfortunately, most of the stated above frameworks are integrated with CUDA libraries, which results in the limitation of hardware platforms for DNN deployment.

Open Computing Language(OpenCL[‡])is an open standard parallel programming language developed to execute on cross heterogeneous platforms including CPUs, GPUs, FPGAs, DSPs and application specific accelerators. OpenCL was originally developed by Apple, Inc and later submitted to Khronos Group. By today OpenCL becomes the only well accepted standard and supported by a variety of commercial chip manufacturers[§] . Unlike other programming languages, OpenCL exposes more hardware interfaces for programmers

[‡]https://en.wikipedia.org/wiki/OpenCL

[§]OpenCL is being supported by the following processors manufactures till today, including Altera, AMD, Apple, ARM Holdings, Creative Technology, IBM, Imagination Technologies, Intel, Nvidia, Qualcomm, Samsung, Vivante, Xilinx, ZiiLABS and etc.

and provides the flexibility to support different manufacture's computing components. In order to enable compatibility between different platforms, OpenCL program detects the specific devices and compiles at runtime.

This paper presents OpenCL caffe, an industry effort in porting one of the most popular DNN frameworks caffe into OpenCL backend, examines its performance bottlenecks and proposes optimization techniques to achieve performance boost. This paper also provides the detailed performance evaluation based on OpenCL caffe and further shares OpenCL's compatibility analysis in machine learning domain.

2. PHASE ONE: OPENCL BACKEND PORTING STRATEGIES AND ANALYSIS

Typically, a DNN toolkit is a hierarchical software framework that integrates both software and hardware layers. Software level includes machine learning SDK and APIs, providing different algorithm layers. So far caffe supports 40+ different layers. Hardware level abstraction needs to handle hardware resources allocation and utilizations to guarantee algorithm correctness and the best performance, e.g. CPU-GPU task assignment, memory management, data transfer, and computing resources utilization.

2.1 OpenCL porting strategies

Caffe framework is originally written in C++ and CUDA. CUDA and OpenCL are very different in hardware device abstraction, memory buffer management, synchronization, data transfers etc. The OpenCL backbend porting is not a straightforward engeerning process, as it includes efforts of re-designing the related software interfaces. We break the task into two phases. Phase one targets at a layerwise porting that maintains the original convergence. After all layers are ported to OpenCL, phase two focuses on performance optimization.

Now we discribe more details in phase one. Original caffe provides a layered C++ design in which each layer has both CPU and GPU implementation and underlying memory structure maintains the data coherence. In order to maintain the algorithm convergence, when we port a certain layer to OpenCL, we keep all other layers in CPU implementations. We verify the convergence by training a real model and by comparing the convergence curves to CPU implementations. Next we replace all CUDA functions with OpenCL. For convolutional layers, we use the original scheme of im2col to convert convolutional operations to linear algebra functions in BLAS library. Original caffe uses cuBLAS in both convolutional layers and fully connected classifier layers, which we all replace with corresponding clBLAS[¶] functions. For other small layers like pooling layer, dropout, non-linear functions, normalization layers, and softmax loss layers, we replace CUDA kernels with new OpenCL kernels. DNN training relies on random number generators to dropout some specific layers' neuron activation values. An efficient GPU-end random number generator will avoid frequent data copies from CPU-end random number generator. We use open sourced threeFry random generator [6]. OpenCL provides the flexibility to select various existing devices at runtime, we uses different keywords to differentiate in device initialization process. To give an example, we use *clGetDeviceInfo* with *Device_type_GPU* to select all underlying GPUs and then use keyword *unified memory* to differentiate whether underlying device is an integrated GPU or a discrete GPU.

[¶]https://github.com/clMathLibraries/clBLAS

2.2 OpenCL backend bottleneck analysis

After the whole caffe project ported to OpenCL backend, we conduct profiling analysis to study the performance bottlenecks. We use AMD profiling tool CodeXL, assisted with OpenCL event and printf for timing and kernel analysis. OpenCL profiling demonstrates a few bottlenecks as the following. First OpenCL's online compilation ended up in frequently calling *clBuildProgram* to create each GPU kernel. To give a specific example, for 100 iterations of Cifar [2] training, there were 63 *clBuildProgram* calls that took about 68% of the time. Second, convolutional layers take up most of the computation time. BLAS' performance suffered from irregular tall and skinny matrix sizes from different layers. Apparently DNN convolution layers bring in a new set of matrix sizes that fall in the unoptimized region of BLAS library. The bottleneck exists for all BLAS implementations, while clBLAS is even slower on average due to less optimizations. Third, there are load balance challenges because convolution filter channels, filter size, input and output feature map sizes are different for all layers. Typically feature map sizes shrink as the depth increases which results in less data parallelism. The GPU can not be fully utilized for small images and small convolution filters. Forth, straightforward data transfers can take up the critical path for large ImageNet. An ideal case would be overlapping data transfers with DNN computation.

3. PHASE TWO: OPENCL CAFFE PERFORMANCE OPTIMIZATION TECHNIQUES

In the next chapters, we are going to introduce the optimization techniques to boost OpenCL caffe performance. Based on stated above bottlenecks, we are going to focus on avoiding OpenCL online compiling overhead, increasing data parallelism by batched manner data layout unrolling and increasing task parallelism by multiple command queues.

3.1 Avoid OpenCL online compilation overheads

Online compilation enables OpenCL to support multiple platforms, at the expense of time consumed on OpenCL program initialization when there are many kernels. OpenCL caffe includes two categories of kernels, including DNN specific kernels for convolution, pooling, dropout and non-linear functions etc, and kernels generated by clBLAS. We introduced several techniques to reduce the compilation time.

For DNN specific kernels, we build all the kernels into *clProgram* object when the GPU device is initialized. During execution, we create the specific kernel when it is called for the first time, and cache them to accelerate the future calls. clBLAS takes more time on kernel generation and compilation. This is because clBLAS generates the source code for required kernels online according to the input sizes of matrices to get the optimal performance. In DNN application the dimension of matrix is multivariate, hence various copies of source code are generated and compiled one by one. We figured out to store the generated binary kernels onto the hard disk, so that they can be directly loaded without compilation next time. Fortunately, subsequent versions of clBLAS library have already solved this problem through

offline compilation on some specific devices, and we suggest users to use it directly.

3.2 Boost data parallelism: batched manner data layout transformation

As analyzed in section 2, the major performance bottleneck of caffe is BLAS computation converted from convolution, as DNN problem sizes are new to math libraries which fall into the less optimized corners. In the literature, BLAS optimizations are usually done by breaking down big problem size into smaller kernels and map them on to different GPU computing units. The best performance can be achieved by balancing the workloads between different computing units and by keeping all hardware threads busy while fully utilizing local memory resources. BLAS has its favored matrix sizes as GPU hardware threads are grouped into a tiled manner, while each tile usually supports 256 concurrent threads. To be specific, clBLAS achieves the peak performance when matrix sizes are square and divisible by 4096 or 1024. The left part of table 1 lists all the matrix sizes for classical AlexNet for ImageNet[9] input. As we can see, the matrix sizes are not only not square but also the shapes are irregular and sizes vary for different layers. The matrix sizes M, N, K are decided by the number of convolutional layers, channels, filter sizes and feature map sizes, which are part of the model parameters determined by the algorithm experts. Figure 1 shows the data layout for original caffe's convolutional layer. Each image is represented as a 3D tensor in (channels, width, height). Caffe implementation first uses im2col function to convert all the input feature maps from one image into the matrix in the middle. Each column in the middle matrix represents one channel for the given image, with each all sub-columns in this column represent the sliding window for the convolutional filters. Then we can call BLAS function to multiply the input matrix by weight matrix and generates the output feature maps.

Since BLAS performance are very sensitive to M, N, K, while N and K are strictly limited by the model hyper-parameters to guarantee correctness. We propose batched manner data unrolling scheme to increase the size of M, with the goal of increasing the data parallelism and moving the matrix sizes from ill-performed region to favored region. In order to do so, batched manner unrolling use a 4D tensor (n, channels, width, height) for im2col to generate a much wider matrix for the input matrix, while n represents image level parallelism. Figure 2 shows how it works. Compared to figure 1, the differences include the following. First, im2col now takes multiple images' all input channels and convert them into the middle matrix, where the columns in neighboring images are laid out continuously shoulder by shoulder in the bigger matrix. Second, the output features maps are also mixed in the similar shoulder by shoulder manner which destroys the original tensor structure that requires each images' output channels be continuous in the memory. In order to keep tensor structure consistent with original caffe, we need to change the entire data layout for convolutional layers for both backward and forward prorogation. The number of images n is a design tradeoff that can be chosen by exploring the optimal match among certain convolutional layer's hyper-parameters, input image sizes, GPU hardware architecture, and BLAS design space. Experiments of clBLAS 2.4 or 2.6 on AMD GPUs demonstrate that best performance can be achieved when n is 16 or 32, though the optimal number can change with different GPUs and different clBLAS version. The right half of table 1 shows the new matrix sizes M', N', K' after batched manner unrolling and the evaluated performance boost for AlexNet training on AMD FirePro™ W9100 GPU. As we can see, data layout unrolling increases the performance significantly, which is 11x, 12x, 10x, 9x, 16x respectively the 5 convolution layer's. Section 4 provides a detailed performance evaluation of batched manner optimization vs original clBLAS. For the best performance on different GPUs, an auto-select online heuristic can help select the optimal batch number for different DNN models and even different layers of the same model.

Table 1: Matrix unrolling techniques for convolution layers (AlexNet, groups=2)

Layers	Original M,N,K	Unrolled M',N',K'	speedup
\conv1	3025,96,363	48400,96,363	11
\conv2	729,128,1200	11664,128,1200	12
\conv3	169,384,2034	2704,384,2034	10
\conv4	169,192,1728	2704,192,1728	9
\conv5	169,128,1728	2704,128,1728	16

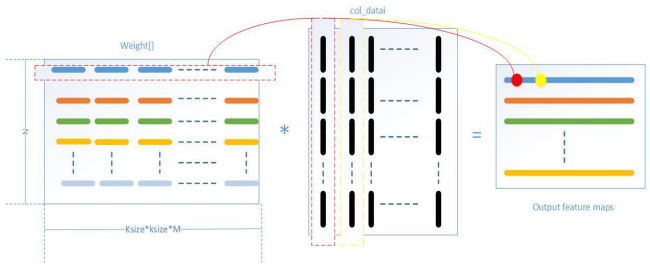

Figure 1: Original BLAS based convolution data layout

3.3 Boost task parallelism: multiple command queues

In parallel computing, data parallelism and task (instruction) parallelism are two dimensions to increase performance improvement by parallel machines. Section 3.2 has focused on increasing data parallelism by inter-image data unrolling. In OpenCL, a command queue is a task sharing center between CPU and GPU where CPU enqueues tasks (kernels)

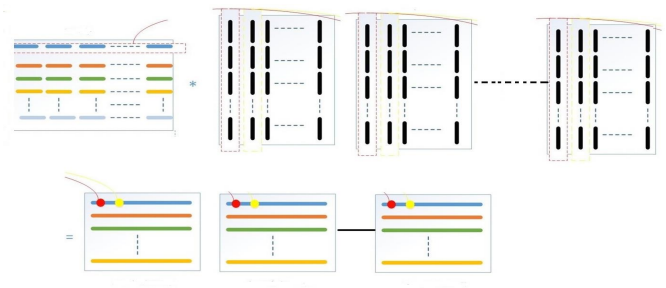

Figure 2: Unrolled convolution data layout to increase data parallelism

Table 2: Offline training speed(AlexNet, mini-batch=128)

Platforms (GPU&CPU)	Speed (images/s)
AMD W9100 & A10-7850k	255
AMD R9 Fury & A10-7850k	261
AMD R290X @1000MHz & A10-7850k	268
AMD S9150 @900MHz & Xeon E5-2640	227

Table 3: Online recognition speed(AlexNet, mini-batch=128)

Platforms (GPU&CPU)	Speed (images/s)
AMD W9100 & A10-7850k	590
AMD R9 Fury & A10-7850k	699
AMD R290X @1000MHz & A10-7850k	606
AMD S9150 @900MHz & Xeon E5-2640	452

and GPU consumes tasks, one at a time. In order to increase the task parallelism, we use multiple command queues to enable concurrent tasks on one GPU. This technique is very useful to increase GPU utilization when one GPU kernel is too small to occupy all the hardware threads. Though, multiple command queues require the concurrent tasks have no data or instruction dependency on each other. Caffe has offered independent tasks at certain layer, for example, convolutional layers with independent groups. OpenCL caffe has adopted multiple command queues to explore the performance potential. And we have observed the performance effects have close relation to image size, minibatch size, layer complexity and hardware resources. To be more specific, for small Cifar images training, multiple command queues improve offline training speed by 10%. While for large ImageNet training, this technique only sees a speed-up for small minibatch size, 12% for minibatch size of 10; for large minibatch sizes, we sometimes observe a slow down due to interfering between two queues and extra synchronization overheads. We believe with efficient task queues and hardware scheduling mechanism, the potential performance gain can further improve.

4. PERFORMANCE EVALUATION

We evaluate OpenCL caffe's performance on mainstream AMD server-end and consumer-end graphic cards, as shown in the Table 2 and the Table 3 for offline training and online recognition performance respectively. The evaluation OS is Ubuntu 14.04 with clBLAS 2.6. The DNN model we evaluate is AlexNet for ImageNet with mini-batch size 128. AMD Hawaii series of GPUs including W9100, R290x and S9150 achieve offline training speed of upto 268 images per second and online recognition speed of upto 606 images per second. AMD latest Fiji GPU R9 Fury has achieved offline training speed of 261 images per second and online recognition of 699 images per second.

As shown in Figure 3, we also compare the performance of caffe with CUDA (including both cuBLAS and cuDNN v2) vs OpenCL (including both original clBLAS and batched manner optimization) on TitanX and R9 Fury respectively, Alexnet model with mini-batch size 100. The goal of this comparison is to compare an apple to apple performance. As expected the OpenCL caffe batched manner dramatically improve the performance comparing to the OpenCL caffe with original clBLAS and has achieved similar performance to the caffe with cuBLAS. This comparison has reflected the apple to apple performance based on math libraries. Compared to the highly optimized machine learning library cuDNN library, which has adopted a lot of low level optimizations, OpenCL caffe still has a performance gap of 2x. However, this also demonstrates that with the same level of optimizations, OpenCL caffe also has the potential to achieve the competitive performance with cuDNN. Given the current performance, the OpenCL caffe is still competitive in terms of performance per dollar, considering the market price difference of R9 Fury $560 and the TitanX $1000.

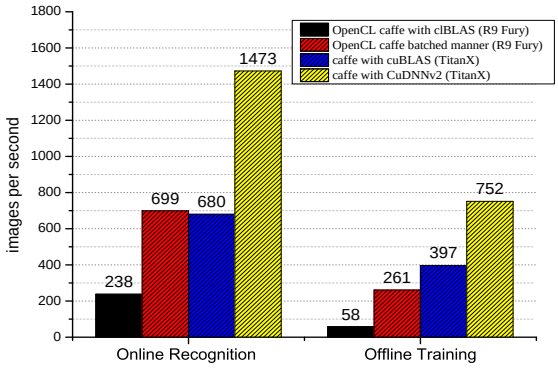

Figure 3: Performance comparison caffe with CUDA vs OpenCL

5. CROSS PLATFORM CAPABILITY ANALYSIS

Cross platform and portability is one of the major goals of OpenCL caffe. In principle, OpenCL caffe can run on all the devices compatible with OpenCL 1.2 and above. But in practice we discover the following engineering difficulties. First, a lot of chip manufacturers still only support lower version OpenCL, like OpenCL 1.1, some even with a subset of the whole functions. Second, there are some differences in specific manufactures' extension and keywords. For example, caffe uses a lot of template in GPU kernels to support different floating point precision. But it turns out the template keywords for different manufactures are different, which adds more difficulty for the same code to run on different platform without modifications. We believe that the practice of OpenCL caffe is very important and timely in examining the capability of OpenCL's cross platform and calling for support from the community.

6. CONCLUSIONS

This paper presents OpenCL caffe, an effort to use open standard to enable a cross platform DNN framework and achieve competitive performance. We describe the OpenCL porting strategies that guarantee algorithm convergence, examine the performance bottlenecks and propose three key optimization techniques including kernel caching to avoid

OpenCL online compilation overheads, a batched manner data layout scheme to boost data parallelism and multiple command queues to boost task parallelism. The optimization techniques effectively map the DNN problem size into existing OpenCL math libraries, improve hardware resources utilization and boost performance by 4.5x. The cross platform capability of OpenCL caffe is an on-going engineering effort to conquer the differences of various OpenCL versions and extensions supported by different manufacturers and will require the community's support to make happen.

7. REFERENCES

[1] caffe. http://caffe.berkeleyvision.org.

[2] Cifar. http://www.cs.toronto.edu/~kriz/cifar.html.

[3] Convnet. https://github.com/sdemyanov/ConvNet.

[4] Mxnet. https://github.com/dmlc/mxnet.

[5] Tensor flow. http://www.tensorflow.org/tutorials.

[6] Threefry random generator. https://www.deshawresearch.com/resources_random123.html.

[7] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. page arXiv:1512.03385, 2015.

[8] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. page 1097Ű1105, December 2012.

[9] e. Olga Russakovsky, Jia Deng*. Imagenet large scale visual recognition challenge. pages arXiv:1409.0575, 2014, 2014.