DNNMark: A Deep Neural Network Benchmark Suite for GPUs

Shi Dong
Northeastern University
shidong@ece.neu.edu

David Kaeli
Northeastern University
kaeli@ece.neu.edu

ABSTRACT

Deep learning algorithms have been growing in popularity in the machine learning community based on their ability to accurately perform clustering and classification in a number of domains. One commonly used class of deep learning techniques is deep neural networks (DNNs). They are composed of a massive number of artificial neurons and many hidden layers. As a complex scientific computing problem, deep neural networks encompass a rich set of computing-intensive and data-intensive workloads including convolution, pooling, and inner products. All of these workloads can be used as standalone programs to benchmark hardware performance. As the GPU develops into a popular platform used to run deep learning algorithms, hardware architects should be equipped with a representative set of benchmarks that can be used to explore design tradeoffs. This suite of workloads can be constructed from a number of primitive operations commonly found in deep neural networks.

In this paper, we present DNNMark, a GPU benchmark suite that consists of a collection of deep neural network primitives, covering a rich set of GPU computing patterns. This suite is designed to be a highly configurable, extensible, and flexible framework, in which benchmarks can run either individually or collectively. The goal is to provide hardware and software developers with a set of kernels that can be used to develop increasingly complex workload scenarios. We also evaluate selected benchmarks in the suite and showcase their execution behavior on a Nvidia K40 GPU.

CCS CONCEPTS

•General and reference → Evaluation; •Computing methodologies → Neural networks; •Computer systems organization → Neural networks; •Software and its engineering → Software design engineering;

KEYWORDS

Deep Neural Network, Benchmark Suite, GPU, cuDNN

ACM Reference format:
Shi Dong and David Kaeli. 2016. DNNMark: A Deep Neural Network Benchmark Suite for GPUs. In *Proceedings of GPGPU-10,*
, February 04-05, 2017, Austin, TX, USA, 10 pages.
DOI: http://dx.doi.org/10.1145/3038228.3038239

GPGPU-10,

DOI: http://dx.doi.org/10.1145/3038228.3038239

1 INTRODUCTION

Currently, a large number of researchers from both academia and industry are expending significant effort to develop new deep learning frameworks. This excitement about deep learning algorithms is based on their impressive performance and great potential to be applied to wide range of "smart" products. So far, there have been a number of emerging products that are empowered by advances in machine learning.

Deep learning algorithms and applications are quickly becoming the primary focus of attention for many leading research and industrial labs. Two such examples from Google include AlphaGo [26] and self-driving vehicles [12]. We continue to see industry and academia exploring deep learning to address a number of challenging problem domains.

The general architecture used in deep learning is a deep neural network (DNN), an artificial neural network that is composed of artificial neurons and many hidden layers (e.g., convolution, pooling, and local response normalization). In order to develop a more robust deep neural network model that has higher capacity, researchers keep increasing the number of in-network parameters. This increases the depth of the network by adding more layers, which results in computations that can take days or even weeks to finish.

Considering that the computations of each layer in the deep neural networks are matrix-based, they can be processed in a parallel fashion in most cases. Additionally, the computations within a layer have few dependencies. Based on these characteristics, deep neural network computations can significantly benefit from using computer platforms that can exploit data-level parallelism. We can leverage a Graphic Processing Unit (GPU), versus relying on a CPU, to accelerate DNN processing. Moreover, training a deep neural network model requires an enormous amount of training data, which has been a fundamental barrier to leveraging DNNs in a number of application domains. Traditional computer systems (i.e., CPUs) are incapable of providing enough computational power to tackle the training task. Thus, heterogeneous computer systems composed of both CPUs and GPUs are becoming the defacto standard platform for deep learning algorithms. In such systems, the computation associated with each layer in the DNN is offloaded to the GPU device.

However, if we want to tune performance of DNN computations on a GPU platform, there is a void of tools available. While there are a number of existing deep learning frameworks (e.g. Caffe[14]) that provide GPU support in order to shorten the training time, they focus on reducing the programming effort versus tuning performance. This prior work paid little attention to tuning the software as it is mapped to hardware. Existing frameworks cannot satisfy the needs of designers of next-generation hardware platforms, who desperately need a framework where they can evaluate design tradeoffs when executing deep learning algorithms. While some researchers

have considered extracting DNN primitives and embedding them in reconfigurable hardware, this approach has been shown to be too labor intensive. Furthermore, configurability will be compromised by a customized solution. We still need to perform model training, the overhead of which can vary significantly based on the platforms available.

In this paper, we present DNNMark, a benchmark suite that is designed to address many of the problems discussed above. First, each of DNN primitive workloads can be easily invoked separately, without any sacrifice on configurability. One can specify any type of scenarios for benchmarking on an algorithm-specific level, and configuring such parameters is no longer an issue in this suite. Second, the actual measurement can be done during the execution of any specific kernel. Our framework allows us to ignore the setup stages and focus on only the training steps.

Unlike other deep learning frameworks, attaching a real database for training purposes is not mandatory anymore. This capability will greatly aid the computer architecture community, which is more interested in designing/tuning hardware/software, and less interested in the details or configuration of the deep neural net.

Depending on the specific configuration, deep neural networks can involve combinations of DNN primitives. A model composed of two or more primitive functions may be more desirable in terms of performance evaluation. In such cases, a composed model rather than standalone primitives, are preferred. To provide this capability, DNNmark can be extended to more sophisticated DNN models, where layers are connected to, and dependent upon, each other.

The rest of this paper is organized as follows. In section 2, we provide an overview of the basic elements and structure of a deep neural network, and we describe a typical GPU architecture. In section 3, we provide a list of all the individual DNN primitives supported in the current benchmark suite. In section 4, we discuss the design process and workflow for DNNMark. In section 5, we showcase DNNMark by evaluating selected benchmarks on one Nvidia GPU. In section 6 we discuss the benefits of this new suite of workloads, and in section 7, we discuss related work. Finally, in section 8 we conclude the paper.

2 BACKGROUND

2.1 Deep Neural Network

The basic unit in a deep neural network is an artificial neuron, which is a simple mathematical model, as presented in Figure 1. The neuron is composed of operations such as multiplications and summations, as shown in Equation 1.

$$Y = \sum_{i=1}^{n} W_i X_i + b \tag{1}$$

Besides simple linear data transformations, the neuron also involves non-linear activation functions in order to handle scenarios where problems are not linearly separable. The commonly used nonlinear activation functions are Rectified Linear Unit (ReLU)[22], sigmoid[22], and hyperbolic tangent (tanh)[22]. In the architecture of a deep neural network, the non-linear activiation functions should be used together with the linear transforms.

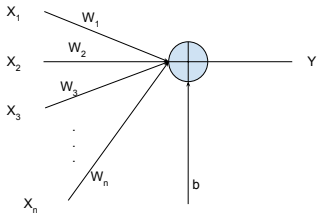


Figure 1: Model of an artificial neuron.

Multiple neurons and activation functions can be grouped together to apply a series of the transformations on the input data. Several groups of data transformations can be concatenated to form a fully-connected feed forward network. Figure 2 shows one instance of a fully-connected feed forward network model. As shown, the network is composed of several layers, including an input layer that takes multi-dimensional data as input, and an output layer that delivers the inference produced by the classifier, and several hidden layers constructed by many nodes of artificial neurons and activation functions. All nodes are fully connected through edges associated with a set of learnable weights and biases.

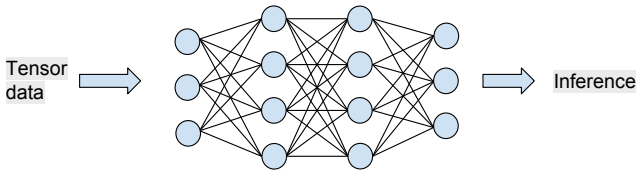


Figure 2: Model of a fully-connected feed forward network.

In other variants of a DNN (e.g., a convolutional neural network), 2D convolution and down-sample pooling are involved as additional forms of linear transformations [16]. More details on DNNs will be introduced in section 3.

2.1.1 Training and Inference. To describe the entire training process mathematically, we can treat the whole network model as a loss function, specifying inputs, outputs and model parameters as arguments. The objective is try to optimize the in-network parameters so that the overall loss can be minimized, as follows:

$$W = \underset{W}{\operatorname{argmin}} L(X, Y, W) \tag{2}$$

where L is the loss function, X is a set of inputs, Y is a set of outputs, and W is set of parameters.

Training deep neural networks is an iterative and time consuming task. It usually requires more than a thousand iterations before the network parameters are properly trained. More specifically, during each iteration a loss value that represents the training quality will be computed and parameters are updated for the next iterations. This process will repeat until the loss reduces to a desirable value. The network's inference abilities are highly dependent on the quality of the training. Normally, only when the training loss converges and the prediction accuracy reaches an acceptable level, can training finally stop. One of the most effective algorithms used

to update the parameters is *Stochastic Gradient Descent*. It is an iterative algorithm that processes a mini-batch of the training data [16], as expressed in equation 3.

$$W_{i+1} \leftarrow W_i - \alpha \sum_{n=1}^{m} \frac{\partial L}{\partial W_i} \tag{3}$$

where m is the number of mini-batches, W_i are the current parameters, W_{i+1} are the updated parameters, α is the learning rate, and $\frac{\partial L}{\partial W_i}$ is the derivative of the loss function with respect to the parameters.

The detailed mechanism for training can be further divided into a forward and a backward propagation. The purpose for performing forward propagation during training is to calculate the loss of the overall network based on the current parameters. The ultimate goal of the backward propagation is to obtain $\frac{\partial L}{\partial W_i}$ for the SGD algorithm. When computing the loss, we are able to calculate the derivatives with respect to the inputs, outputs, and parameters of each layer by applying the derivative chain rule in a backward-cascaded fashion.

After training is complete, the inference can be computed through another forward propagation, but this time the calculation of the loss can be skipped. The results of the inference are also referred to as a *prediction*, which should be a vector containing probabilities for each label from the predefined classes. We can also scale the outcome to allow prediction accuracy to seem more intuitive. In practice, a well-trained deep neural network is able to recognize objects with incredible accuracy. In some cases, the performance is very close to the capabilities of the human brain [27].

2.2 Graphic Processing Unit

For the past decade, Graphic Processing Units (GPU) have been growing in popularity as a general purpose computing platform. Originally designed for rendering graphics efficiently, a GPU was designed with many fixed functional units specific to graphics rendering, and only supported very limited programmability. Since some scientific computing problems map nicely to existing GPU architectures, researchers developed workarounds that allowed GPUs to be used for general computation. More recently, GPU vendors (Nvidia, AMD and Intel) have added programmability to their hardware to meet the growing demands for highly-parallel general purpose computing.

Compared to traditional architectures that are used in CPUs, a GPU has a simpler pipeline and only supports in-order execution. Nevertheless, GPUs still outperform CPUs in many cases where instruction throughput matters. GPUs are well-suited to execute programs that run well on parallel platforms. A number of research studies have reported 2 to 3 orders of speedup when moving their application from a CPU to a GPU [9]. This amount of speedup is achieved for multiple reasons. First, a GPU has many more cores than a CPU (thousands versus 4-8), meaning that the GPU has significantly more computing power and resources. Second, a GPU supports a massive number of concurrently executing threads. So a GPU does not only improve the degree of parallelism by providing thousands of cores, but is also able to hide latency through fast context switching between thousands of threads. A GPU also has high memory bandwidth, as well as other features that support memory throughput that allows us to provide high utilization.

Today's GPU is a highly programmable architecture. CUDA[6], which was introduced in 2006 by Nvidia, is one of the most commonly used general purpose computing languages for GPUs. When using the CUDA programming model, GPU-specific functions are referred to as kernels. In order to run kernels on a GPU, the programmer has to specify the number of CUDA threads block and number of CUDA threads per block. By doing so, the defined kernel functions can be scheduled on the GPU through a runtime library. As a computing platform, CUDA provides a rich set of libraries, with plenty of API functions to utilize GPU resources efficiently.

2.3 cuDNN and cuBLAS

CuDNN [3] is a highly optimized CUDA library that provides primitives for computing deep neural networks. Given that this library can produce high performance and has been widely used in several deep learning frameworks, e.g. Caffe, we consider elements of cuDNN in our proposed benchmark suite. In addition to that, cuDNN provides a selection of algorithms and modes for certain primitives such as convolution, pooling, local response normalization, and etc. Consequently, the programmer has a lot of flexibility when configuring the parameters of the primitives. The usage of cuDNN can be organized as three parts: i.) the tensor and descriptor API functions, ii.) the utility functions for special purposes, and iii.) the forward and backward functions of different layer types [8]. The tensor and descriptor functions are mainly responsible of creating data and layer specific descriptors. The utility functions are used to create a handle or return a recommended algorithm for a layer, for example, convolution. The forward and backward functions are where the actual deep learning computation takes place.

cuBLAS [7] is an implementation of BLAS (Basic Linear Algebra Subprograms) that run on top of the CUDA runtime. Similar to cuDNN, it is a finely-tuned library that provides API functions for matrix or vector operations. But compared to cuDNN, the usage of cuBLAS is more straightforward. The programmer does not need extra utility functions to configure the model or create descriptors. All the programmer needs to do is to invoke APIs with proper arguments. Our benchmark suite leverages this library to implement computation of the fully-connected layer, as the computation can be easily represented as a matrix-matrix multiplication.

3 SUPPORTED DNN PRIMITIVES

Next, we describe each of the benchmarks included in DNNMark.

3.1 Convolution

The convolution algorithm in DNN is a 2-D image convolution [24]. It is effective in producing proper feature maps through multiple convolution kernels shared by all the data. Figure 3 shows a 2-D image convolution. In this figure, a convolution kernel of 3X3 is applied to an area of the image with the same size. The input data is X_i and the kernel weights are W_i. Through the computation shown in equation 4, we can calculate a single result for Y based on the data covered by the kernel as follows:

$$Y = \sum_{i=1}^{k} W_i X_i \tag{4}$$

In the equation, k is the kernel size. Next, the kernel moves to the next data sample, and we repeat the same process until all the feature maps are calculated.

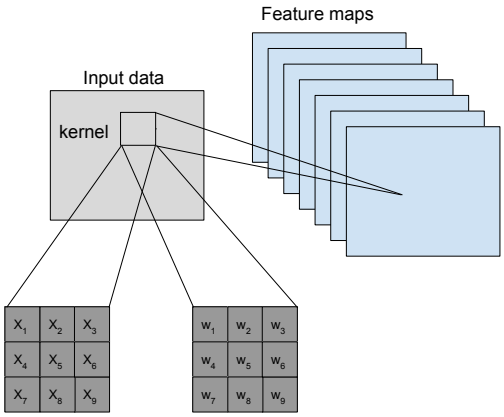


Figure 3: 2-D image convolution

It should be noted that the computation of the 2-D image convolution looks similar to that of the fully-connected layer. However, the two operations are in fact totally distinct in two respects. One is that, during image convolution, the weights are shared by all the data samples (or pixels, if image data is used). On the contrary, each data sample is assigned a unique weight in the fully-connected layer. Another difference is that the output of the convolution is derived from a selected set of data determined by kernel size, while in the fully-connected layer computation, all data has to be considered.

In terms of running the 2-D convolution on a GPU, cuDNN provides multiple algorithmic options, such as Direct, GEMM, FFT [23], and Winograd [17]. The Direct algorithm expresses the convolution as a direct convolution [3]; The GEMM method transforms the entire process into a matrix-matrix multiplication; The FFT and Winograd algorithms are fast implementations that are widely used [17], the former requires significant memory space, while the latter is memory-efficient[8]. cuDNN also allows the programmer to select the execution parameters in terms of speed and memory usage, and cuDNN will correspondingly recommend either a performance-oriented or a memory-efficient algorithm, depending on specific settings. In order to capture the various behaviors of the different algorithms in a flexible manner, DNNMark supports a configuration flow, where the algorithm-specific parameters are configurable.

3.2 Pooling

Next, we consider pooling operations that are performed as part of a DNN. The goal of down-sampling pooling is to reduce the amount of computation for the following layers. In addition, pooling is also an effective approach in practice to improve robustness, selecting the most representative value from a sub-group of the feature map. By doing so, interference of neighboring pixels can be reduced significantly. As described in Alexnet [16], the pooling layer can reduce the error rate by around 0.4%. Normally, pooling groups are not overlapped, so the pooling layer can be viewed as a grid of pooling groups spaced k data samples apart [16], where k is the size of the pooling group. Figure 4 shows an example of the feature map data being divided into 16 pooling groups.

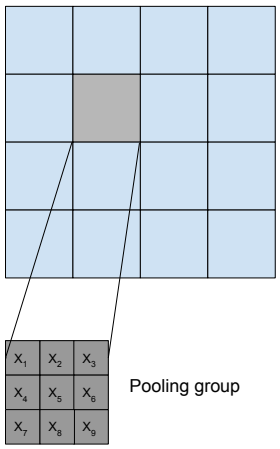


Figure 4: Feature map data divided into multiple pooling groups.

The down sampling within one pooling group can be completed either by selecting the maximum value (i.e., Max Pooling) or computing the average of the group (i.e., Average Pooling). cuDNN supports both modes, as does DNNMark.

3.3 Local Response Normalization

Local Response Normalization (LRN) [16] is performed by applying equation 5:

$$y_i = \frac{x_i}{k + \alpha(\sum_{j=\max(0,i-n/2)}^{\min(N-1,i+n/2)} (x_j)^2)^\beta} \quad (5)$$

where k, α, β are all the configurable parameters of LRN, N is the number of kernel maps, n is the window size for normalization, x_i is the input data, and y_i is the output with the same spatial location.

In general, LRN is basically doing normalization on one input with data from multiple adjacent kernel maps, but at the same relative position. This algorithm is inspired by a biological activity named *lateral inhibition* [5]. If we apply it, the prediction error rate can drop by 2%[16]. This computation is fully supported by cuDNN. After the related parameters are properly configured in the descriptor, the LRN API functions can be easily invoked.

3.4 Activation

As mentioned in section 2, the activation function is the non-linearity introduced in deep neural networks to handle linear inseparable problems. The commonly used activation functions include: ReLU (equation 6), sigmoid (equation 7), and tanh (equation 8). The equations below provide mathematical descriptions of these functions.

$$y_i = \max(0, x_i) \quad (6)$$

$$y_i = \frac{1}{1 + e^{-x_i}} \tag{7}$$

$$y_i = \frac{e^{x_i} - e^{-x_i}}{e^{x_i} + e^{-x_i}} \tag{8}$$

In cuDNN, the activation functions are configured with just one descriptor, but can be distinguished by different modes, indicating the activation function type. In order to be compatible, DNNMark also adopts a similar scheme where the activation functions are indicated by the configured mode rather than implemented as different activation layers separately.

3.5 Fully Connected

Based on our mathematical model of the fully-connected layer, the computation can be transformed into matrix-based operations. For example, as shown in figure 5, we have one fully-connected layer with only two artificial neurons (as in Figure 1). Given this model, the output of this layer can be represented as

$$\begin{aligned} Y_1 &= W_{11}X_1 + W_{12}X_2 + b_1 \\ Y_2 &= W_{11}X_1 + W_{12}X_2 + b_2 \end{aligned} \tag{9}$$

Equation 9 can be further transformed into matrix-vector operations as follows:

$$\begin{bmatrix} Y_1 \\ Y_2 \end{bmatrix} = \begin{bmatrix} W_{11} & W_{12} \\ W_{21} & W_{22} \end{bmatrix} \begin{bmatrix} X_1 \\ X_2 \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \end{bmatrix} \tag{10}$$

This example has only one set of data inputs. If there are multiple sets of data inputs, then the computation becomes a multiplication and summation of matrices.

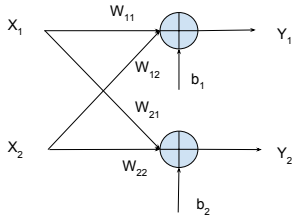


Figure 5: A simple example of a fully-connected layer.

CuDNN does not provide primitives for computations that only involve linear algebra operations. Due to this reason, DNNMark uses API call to cuBLAS to support computation of the fully-connected layer on a GPU.

3.6 Softmax

Softmax [22] is the core functions in the output layer. It interprets the output data from the previous hidden layer, and generates a set of probability-like values in the range of 0 to 1 (note, the sum of all of these values equals 1). For each value that Softmax computes, it represents the extent to which the input data should be classified into one of the predefined classes. The Softmax function is defined in equation 11:

$$Y_i = \frac{e^{x_i}}{\sum_{j=1}^{N} e^{x_j}} \tag{11}$$

where N is both the number of outputs from the previous layer and the number of classes.

The way to invoke this function in cuDNN is exactly the same way it is involved in the LRN layer, except Softmax has some additional configurable options for softmax-specfic algorithms and modes[8]. DNNMark considers them in the framework as one part of the configurability of the algorithm-specific parameters.

4 SOFTWARE DESIGN

DNNMark is developed in C++, and uses CUDA libraries such as cuDNN and cuBLAS, which were discussed in previous sections. We utilize the CMake tools to enable effortless compilation and build of the framework, and link Google libraries such as gflags [10] and glog [11] for ease of configuration and improved debugging. Unlike other GPU benchmark suites, e.g. Rodinia[2], where each benchmark is developed and built individually, DNNMark includes benchmarks that can be combined into a compound workload, with dependencies indicated between different benchmarks. This allows us to construct an actual DNN model. So that, beside running benchmarks separately, more complex scenarios that combine multiple primitives can be evaluated.

To support this benchmark architecture, we use a different strategy to generate benchmarks in this suite. Instead of developing each benchmark individually, we first develop a centralized DNNMark library, which encompasses all required functionalities of the DNN primitives. The library also provides a general easy-to-use interface for various tasks (e.g., configurating the system, initializing the system and running computations). Figure 6 shows the overall benchmark flow using our DNNMark library.

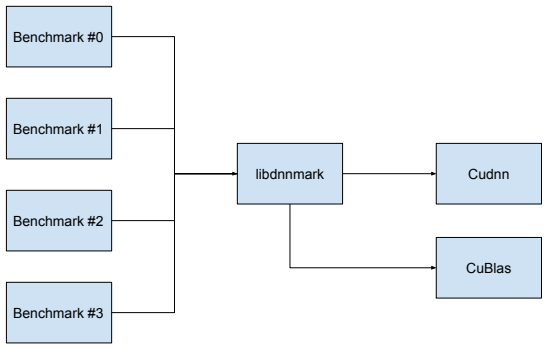


Figure 6: Linkage graph of DNNMark.

As indicated in Figure 6, we first build up the DNNMark library *libdnnmark* with external CUDA libraries: cuDNN and cuBLAS. When this is complete, several benchmarks can be built through linking to this library. Following these steps, benchmark development can be simplified and DNN-specific development can be managed much easier.

The centralized DNNMark library has a well established architecture, where the software workflow is well defined. As shown in Figure 7, the framework has several software modules that handle different tasks and can also work cooperatively as a whole. The *Config Parser* is in charge of interpreting information from the external configuration files and adding layers based on the configuration parameters to the layer pool, and a data structure that holds all of

the configured layers supporting various DNN primitive types. The major role of the *Initializer* is to set up the tensors and the descriptors based on specified parameters through several encapsulated utility classes in DNNMark. And the Runner module is the one that launches computations of the DNN primitives. As presented in the figure, there is also a Data Manager module that interacts with the layer pool. It is a functional software class that maintains and manages data chunks in GPU memory. Details of this mechanism will be explained later.

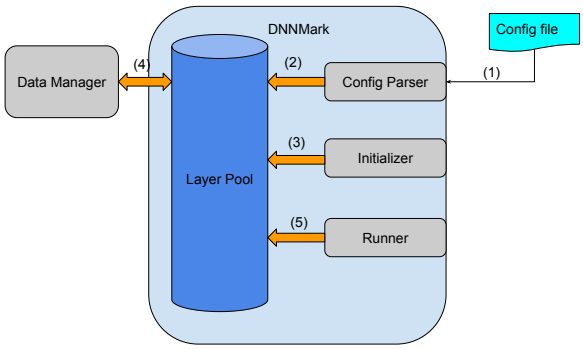


Figure 7: The software architecture of DNNMark.

From Figure 7, we can see the specific execution order, marked with numbers, in the workflow. More specifically, once the DNNMark framework is started, we first access the configuration file and locate the target sections in order to extract the parameters. Next, one or more layers are added to the layer pool and initialization starts. During the initialization phase, configured layers request data chunks from the Data Manager and keep track of them. Finally, with model construction complete, the actual computations can be started for benchmarking.

As mentioned previously, one of the most critical functions in DNNMark is data management. The main job of the Data Manager is to maintain and manage data chunks in GPU memory, as well as interact with layers in the layer pool. In order to obtain memory space for input/output or parameters, the layers can request GPU data chunks of a specific size and then the Data Manager will return the Data IDs corresponding to the requested GPU data chunks. With these IDs, layers are able to assign the addresses of the allocated memory spaces. This design not only liberates layers from performing data allocation, but also provides a clean data management mechanism to create composed models, where some data chunks are shared between layers. Figure 7 provides an example of how GPU data is managed by the Data Manager module. As shown in Figure 8, the three layers are assigned with unique Data IDs, which can be further used to locate the actual memory space in GPU memory. Given this design and implementation, DNNMark is highly configurable, easy to use, and highly extensible.

Configurability: Following the workflow presented earlier, the first step of initiating a benchmark is to read a self-defined DNNMark-specific configuration file. The file contains all the configurable parameters. In this step, both DNNMark and the layer specific parameters will be extracted. List 1 shows an example of a configuration file.

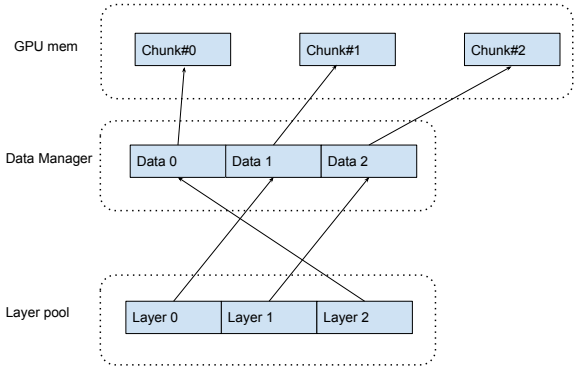


Figure 8: Data management mechanism of DNNMark.

Listing 1: Example of a configuration file.

```
[DNNMark]
run_mode=standalone

[Convolution]
name=conv1
n=100
c=3
h=256
w=256
previous_layer=null
conv_mode=cross_correlation
num_output=32
kernel_size=5
pad=2
stride=1
conv_fwd_pref=fastest
conv_bwd_filter_pref=fastest
conv_bwd_data_pref=fastest
```

As shown in the configuration file, there is one general parameter which defines the framework running mode. The next field specifies one convolution layer, and includes both general and algorithm-specific parameters. The general parameters include those regarding layer names, data and kernel dimensions, while the algorithm-specific ones specify convolution mode and algorithmic preferences for running forward and backward propagation. As we can see, allowing for configurable algorithm-specific parameters in this framework enables us to explore a wide variety of GPU execution patterns in a more flexible way. In addition, another aspect of this flexibility is that when explicitly indicated, a benchmark is able to run a subset of the DNN primitives without the need to specify a new configuration file.

Convenience: We provide a straightforward method to construct DNN-related benchmarks thanks to the centralized design scheme of DNNMark. Our framework significantly shortens benchmark development time and simplifies the procedure of adding additional DNN primitives. For example, the data management and utility functions can be reused so that programmers only need

to concentrate on functional elements when adding a new primitive. Currently, we support six different DNN primitives that are essential parts to compose AlexNet [16]. But in the future, more primitives can be added easily.

Extensibility: As shown in list 1, DNNMark allows the user to specify the run mode as a framework-specific configuration parameter. Presently, we support two modes: standalone and composed. In standalone mode, benchmarks run in a totally isolated manner, meaning that only one DNN primitive is invoked at a time. However, under certain circumstances, running a composed model, where multiple DNN primitives are connected to represent a full network model, could be more interesting. In a composed mode, the user can add a single DNN primitive to new benchmark. The procedure is as simple as editing a configuration file. List 2 shows another example of a configuration file with the running mode set to "composed".

Listing 2: Example of a configuration file using composed mode.

```
[DNNMark]
run_mode=composed

[Convolution]
name=conv1
n=1
c=3
h=32
w=32
previous_layer=null
conv_mode=cross_correlation
num_output=32
kernel_size=5
pad=2
stride=1
conv_fwd_pref=fastest
conv_bwd_filter_pref=fastest
conv_bwd_data_pref=fastest

[Pooling]
name=pool1
previous_layer=conv1
pool_mode=max
kernel_size=3
pad=0
stride=2

[FullyConnected]
name=relu1
previous_layer=pool1
num_output=32
```

In this example, there are three layers, Convolution, Pooling, and Fully Connected, linked together to form a new model. Other than changing the run mode, the "previous_layer" parameter needs to be defined correctly in order to run properly.

5 EVALUATION

5.1 Environment Setup

We select the Nvidia K40c [21] micro-architecture from the Nvidia Kepler family of GPUs [19] to demonstrate the utility of our benchmarks. Table 1 describes the hardware details. The library details can be found in table 2.

Table 1: Nvidia K40c Configurations

Type	k40
Number of processor cores	2880
Warp size	32 threads
SIMD lane width	8
Maximum threads per block	1024
Number of 32-bit registers	65536
Maximum registers per threads	255
Configurable shared memory/L1 cache	64KB
Read-only data cache	48KB
L2 cache	1536KB
Memory interface	384-bit
Memory bandwidth	208 GB/sec
Memory size	12GB

Table 2: Library Details

Libraries	Version
CUDA	8.0
CuBlas	8.0
CuDNN	5.0

5.2 Selected Benchmarks

In our evaluation, we chose to evaluate individual DNN primitive, which include benchmarks that are based on some of the layers in AlexNet. Table 3 lists all the selected benchmarks and the dimensions of their inputs. Note that for each benchmark, both forward and backward propagation are evaluated.

For some of benchmarks, there is more than one kernel in the corresponding API function. In order to simplify the evaluation and comparison between benchmarks, we selected the most representative kernel, as shown in table 4.

5.3 Results

We use the nvprof[20] profiling tool and present results based on several selected metrics provided by the tool.

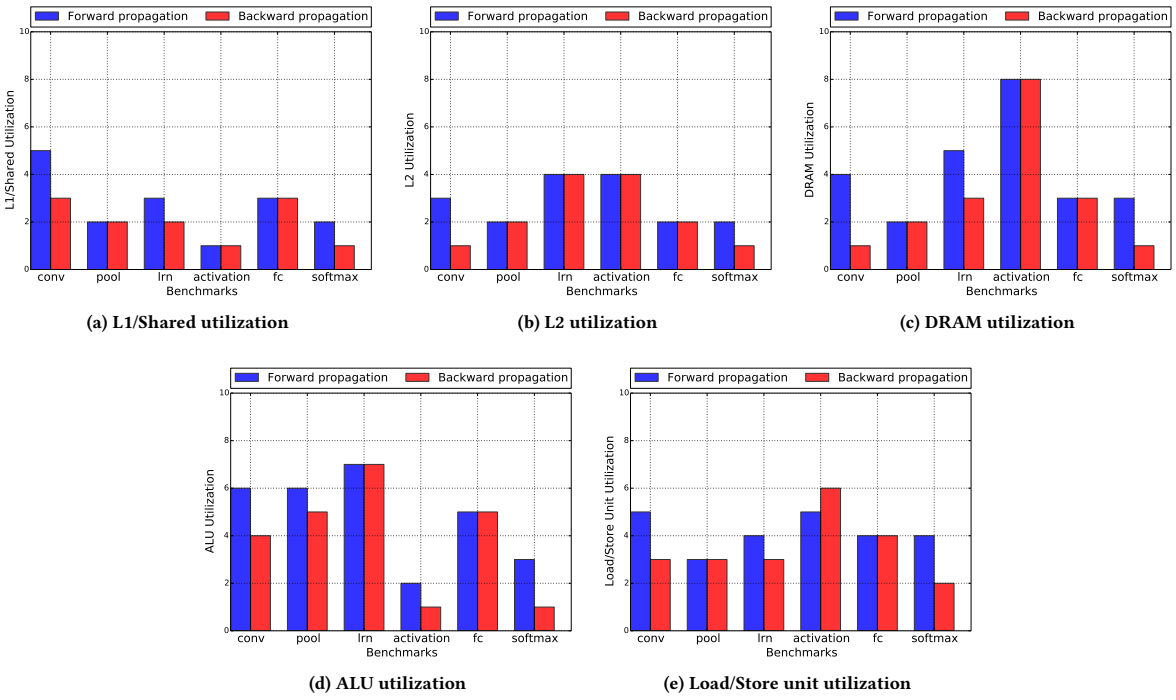
Figure 9-11 present the execution Instruction Per Cycle (IPC), occupancy, and eligible warps per cycle for six different DNN primitives, with both forward and backward propagation. IPC show instruction throughput and performance. Achieved occupancy reflects, to some extent, the current thread activity. The number of eligible warps per cycle reflects the effectiveness of the warp scheduler. Additionally, Figure 12 shows the GPU utilization of different hardware resources, including: L1/shared memory, L2 cache, DRAM, the ALU and the Load/Store Unit.

Table 3: Selected benchmarks

Benchmark	Batch size	Number of Channel	Height	Width
Convolution	100	3	256	256
Max Pooling	100	3	256	256
LRN	100	32	256	256
Activation(ReLU)	100	32	256	256
Fully connected	100	9162	1	1
Softmax	100	1000	1	1

Table 4: Selected kernels of benchmarks

Benchmark	Forward Kernel	Backward Kernel
Convolution	implicit_convolve_sgemm	wgrad_alg0_engine
Max Pooling	pooling_fw_4d_kernel	pooling_bw_kernel_max
LRN	lrn_fw_Nd_kernel	lrn_bw_Nd_kernel
Activation(ReLU)	activation_fw_4d_kernel	activation_bw_4d_kernel
Fully connected(FC)	sgemm_sm35_ldg_nt_128x16x64x16x16	sgemm_sm35_ldg_nt_128x16x64x16x16
Softmax	softmax_fw_kernel	softmax_fw_kernel



(a) L1/Shared utilization **(b) L2 utilization** **(c) DRAM utilization**

(d) ALU utilization **(e) Load/Store unit utilization**

Figure 12: A breakdown of GPU utilization.

From these profiling results, we can see the detailed program behavior of each benchmark. For example, from the IPC results, we find that activation has relatively poor IPC for both forward and backward executions. From Figure 12, we see that activation has low ALU utilization, but high utilization of DRAM and the Load/Store Unit. The occupancy also captures this behavior, because lower occupancy implies lower thread activity. From these observations, we can conclude that the execution of activation is memory-bound.

From the IPC results in Figure 9, we see that convolution, pooling, LRN and the fully-connected layer all have high IPC. They have high ALU utilization and so we can conclude they are compute dominated. But, one interesting observation is that even though backward pooling has significantly high occupancy, the IPC is in fact not as high as forward pooling.

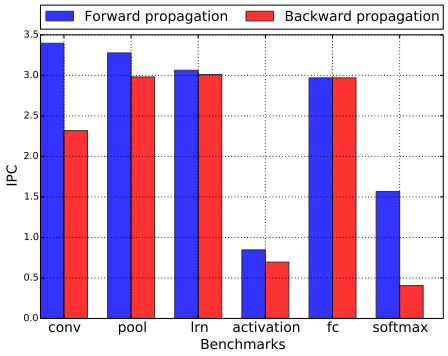


Figure 9: Instructions per cycle achieved.

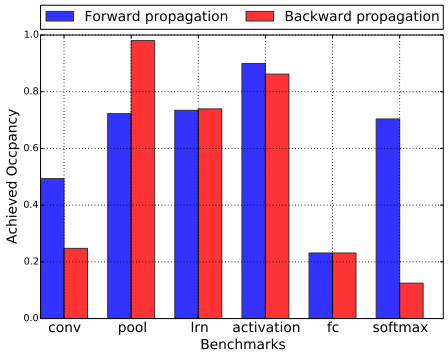


Figure 10: The occupancy achieved.

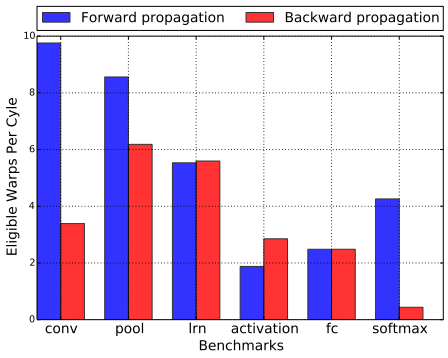


Figure 11: The number of eligible warps per cycle.

6 DISCUSSION

Deep neural network is not only a standalone application but also a compound of many other applications. Given this particular structure, optimizing a single computation problem is no longer a solution for deep learning algorithms. the entire problem has to be considered as a whole. Nevertheless, treating deep neural networks as a complete black-box system also hinder the possibility to explore potential technique to leverage current hardware. The recent solution to accelerate training of DNN is still limited to adding more expensive hardware, building large-scale distributed systems and computing clusters[4].

HPC solutions to deep learning are effective. Weeks of training reduces to days or even hours. But the price is also high. The question is whether learning applications are making efficient use of current hardware. In order to answer this question, a complete profiling of each primitive from DNN needs to be done. Only with the detailed hardware execution information can computer architects begin to propose new architectural features to support this workload. Instead of throwing more hardware at deep learning applications, we feel it is a good time to take a step back and explore the potential for new architectural features.

Since a GPU has impressive computing power and high-bandwidth memory to run the training phase of deep learning, it has become an essential hardware platform for many deep learning systems. Given that the GPU has a significantly larger number of compute resources than traditional computer architectures, we need to make efficient use those massive resources. Equipped with DNNMark, architects can now carry out a wide range of studies to improve performance and power. We plan to explore newer features available being delivered on recent GPUs, including concurrent kernel execution [19] and dynamic parallelism [19].

7 RELATED WORK

Deep learning frameworks such as Caffe[14], TensorFlow[1], and Thano[28] all provide frameworks that utilize deep neural networks, supporting GPU execution. Caffe has a full set of DNN primitives as both normal CUDA kernels and as part of CuDNN. It can be utilized, to some extent, as a workload provider, but has limited configurability. Furthermore, modifying Caffe to support all the desired features would be challenging.

TensorFlow is designed for deploying deep learning on heterogeneous distributed systems[1]. We plan to consider this workload in our future work.

Theano is another popular framework built in Python[28]. While Theano is similar to Caffe, turning it into a configurable GPU benchmark suite requires extra efforts. Other GPU benchmark suites, such as Rodinia[2], include many linear algebra and machine learning applications. But the goal of Rodinia is to cover general-purpose GPU computing patterns versus those found in deep neural networks.

8 CONCLUSION

DNNMark is a deep neural network benchmark suite designed to provide a configurable, extensible, and flexible method for profiling computation of individual DNN primitive or composed DNN models. It now includes six commonly used DNN primitive functions and provides a framework with easy and targeted benchmarking. Moreover, it supports both general and algorithm specific configurability and opens the possibility of extending current DNN primitives to complex models as new benchmarks. From the evaluation results, the metrics are able to show different computing patterns of various benchmarks from aspects of GPU specific information which is an invaluable guide for future research. So it can be concluded that DNNMark can serve as a great benchmark suites to obtain insights of deep neural networks on Nvidia GPUs.

Besides that, we want to include following future work as next steps.

- We plan to enrich this benchmark suite by supporting more DNN primitives such as batch normalization[13] and RNN related functions[25].
- We plan to include optimization algorithms (such as SGD) and provide a composed model configuration of real DNN models.
- We plan to add support of real database(e.g. CIFAR10[15] and Mnist[18]) instead of generating random number as input data of workloads.
- We plan to expand our characterization and analysis on this benchmark suite, which could lead to new insights of deep neural networks with respect to GPU-specific behavior.

REFERENCES

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. (2015). http://tensorflow.org/

[2] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. 2009. Rodinia: A Benchmark Suite for Heterogeneous Computing. *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC)* (2009), 44–54.

[3] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. 2014. cuDNN: Efficient Primitives for Deep Learning. *arXiv preprint arXiv:1410.0759* (2014).

[4] Adam Coates, Brody Huval, Tao Wang, David Wu, Bryan Catanzaro, and Ng Andrew. 2013. Deep learning with COTS HPC systems. *Proceedings of the 30th International Conference on Machine Learning (ICML-13)* 28 (2013), 1337–1345.

[5] Ronald A. Cohen. 2011. *Lateral Inhibition.* Springer New York.

[6] NVIDIA Corporation. 2014. CUDA C Programming Guide. (2014).

[7] NVIDIA Corporation. 2015. CuBlas library v7.5. (2015).

[8] NVIDIA Corporation. 2016. CuDNN library v5.0. (2016).

[9] Randima Fernando. 2004. *GPU Gems: Programming Techniques, Tips and Tricks for Real-Time Graphics.* Pearson Higher Education.

[10] Google. 2016. How To Use gflags. (2016).

[11] Google. 2016. How To Use Google Logging Library. (2016).

[12] Erico Guizzo. 2016. How Google's Self-Driving Car Works. (2016).

[13] Sergey Ioffe and Christian Szegedy. 2015. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. *Proceedings of the 32nd International Conference on Machine Learning (ICML-15)* (2015), 448–456.

[14] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. 2014. Caffe: Convolutional Architecture for Fast Feature Embedding. *arXiv preprint arXiv:1408.5093* (2014).

[15] Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton. 2009. CIFAR-10 (Canadian Institute for Advanced Research). (2009).

[16] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. ImageNet Classification with Deep Convolutional Neural Networks. *Advances in Neural Information Processing Systems 25* (2012), 1097–1105.

[17] Andrew Lavin and Scott Gray. 2015. Fast Algorithms for Convolutional Neural Networks. *arXiv preprint arXiv:1509.09308* (2015).

[18] Yann LeCun and Corinna Cortes. 2010. MNIST handwritten digit database. (2010).

[19] NVIDIA. 2012. NVIDIA's Next Generation CUDA^{TM} Compute Architecture, Kepler^{TM} GK110. (2012).

[20] NVIDIA. 2016. CUDA Toolkit Documentation. (2016).

[21] NVIDIA. 2016. TESLA GPU ACCELERATORS FOR SERVERS. (2016).

[22] Genevieve B. Orr and Klaus-Robert Mueller (Eds.). 1998. *Neural Networks : Tricks of the Trade.* Lecture Notes in Computer Science, Vol. 1524. Springer.

[23] Victor Podlozhnyuk. 2007. FFT-based 2D convolution. (2007).

[24] Victor Podlozhnyuk. 2013. Image Convolution with CUDA. (2013).

[25] Haim Sak, Andrew Senior, and Franoise Beaufays. 2014. Long Short-Term Memory Based Recurrent Neural Network Architectures for Large Vocabulary Speech Recognition. *arXiv preprint arXiv:1402.1128* (2014).

[26] David Silver and Google DeepMind Demis Hassabis. 2016. AlphaGo: Mastering the ancient game of Go with Machine Learning. (2016).

[27] Yaniv Taigman, Ming Yang, Marc'Aurelio Ranzato, and Lior Wolf. 2014. DeepFace: Closing the Gap to Human-Level Performance in Face Verification. *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (June 2014).

[28] Theano Development Team. 2016. Theano: A Python framework for fast computation of mathematical expressions. *arXiv e-prints* abs/1605.02688 (May 2016).