applied sciences

MDPI

*Article*

# CitiusSynapse: A Deep Learning Framework for Embedded Systems

**Seungtae Hong *, Hyunwoo Cho and Jeong-Si Kim**

High Performance Embedded System SW Research Section, Artificial Intelligence Research Laboratory, Electronics and Telecommunications Research Institute (ETRI), Daejeon 34129, Korea; tenace@etri.re.kr (H.C.); sikim00@etri.re.kr (J.-S.K.)

* Correspondence: sthong@etri.re.kr; Tel.: +82-42-860-0941

**Abstract:** As embedded systems, such as smartphones with limited resources, have become increasingly popular, active research has recently been conducted on performing on-device deep learning in such systems. Therefore, in this study, we propose a deep learning framework that is specialized for embedded systems with limited resources, the operation processing structure of which differs from that of standard PCs. The proposed framework supports an OpenCL-based accelerator engine for accelerator deep learning operations in various embedded systems. Moreover, the parallel processing performance of OpenCL is maximized through an OpenCL kernel that is optimized for embedded GPUs, and the structural characteristics of embedded systems, such as unified memory. Furthermore, an on-device optimizer for optimizing the performance in on-device environments, and model converters for compatibility with conventional frameworks, are provided. The results of a performance evaluation show that the proposed on-device framework outperformed conventional methods.

**Keywords:** deep learning framework; embedded systems; on-device; OpenCL acceleration



**Citation:** Hong, S.; Cho, H.; Kim, J.-S. CitiusSynapse: A Deep Learning Framework for Embedded Systems. *Appl. Sci.* **2021**, *11*, 11570. https://doi.org/10.3390/app112311570

Academic Editor: Valentino Santucci

Received: 28 October 2021
Accepted: 2 December 2021
Published: 6 December 2021



## 1. Introduction

Deep neural networks (DNNs) have been widely adopted in various fields, such as in image and character recognition and object detection [1–10]. Driven by the increasing popularity of embedded systems, such as smartphones, active research is being conducted to explore on-device deep learning in embedded systems [11–15]. When compared with standard PC and server platforms, embedded systems have smaller amounts of memory, fixed-size storage devices, and low-power processors. Furthermore, embedded systems have a different computational operation processing architecture than PC/server platforms in terms of their central processing unit (CPU), graphics processing units (GPUs), and memory structure. In addition, various types of associated hardware (HW) platforms have been developed, according to diverse applications. However, most deep learning frameworks (PyTorch [16], TensorFlow [17], Caffe [18], etc.) are currently optimized for PC server platforms with abundant system resources. In particular, conventional deep learning frameworks use CUDA [19] or cuDNN [20] to accelerate deep learning operations, and are thus dependent on specific GPU hardware, such as Nvidia GPUs.

A deep learning framework that is specialized for embedded operation processing architectures with limited resources is required to perform on-device deep learning operations efficiently. Furthermore, to accelerate deep learning operations in various heterogeneous embedded systems, such as smartphones or embedded computers, a dedicated framework is required to support deep learning operation acceleration techniques that do not depend on specific HW.

Accordingly, in this study, we propose CitiusSynapse as a deep learning framework that is specialized for embedded systems. First, the proposed framework performs deep learning operations which are based on OpenCL [21] to accelerate deep learning operations

within various embedded systems. To do so, our framework provides a proprietary GPU-accelerated engine based on ARM OpenCL driver, supporting OpenCL 1.2/2.0 full profile. OpenCL is an open, universal, parallel computing framework managed by the Khronos Group, which supports parallel processing in diverse heterogeneous HW environments that include CPUs, GPUs, and other processors. Due to the fact that OpenCL is not dependent on specific HW, the same programs can be executed in a variety of embedded systems, as well as on standard PCs. Moreover, our framework maximizes parallel processing through an OpenCL kernel that is optimized for embedded GPUs and the structural characteristics of embedded systems, such as shared unified memory between CPUs and GPUs. Second, our framework provides an on-device inference performance optimizer for embedded systems. Third, our framework can import other framework models for compatibility. Finally, our framework was compared with conventional deep learning frameworks by a performance evaluation in an embedded board, equipped with an ARM Mali GPU, which is a popular embedded GPU. Figure 1 shows the overall architecture of the deep learning framework proposed in this study.

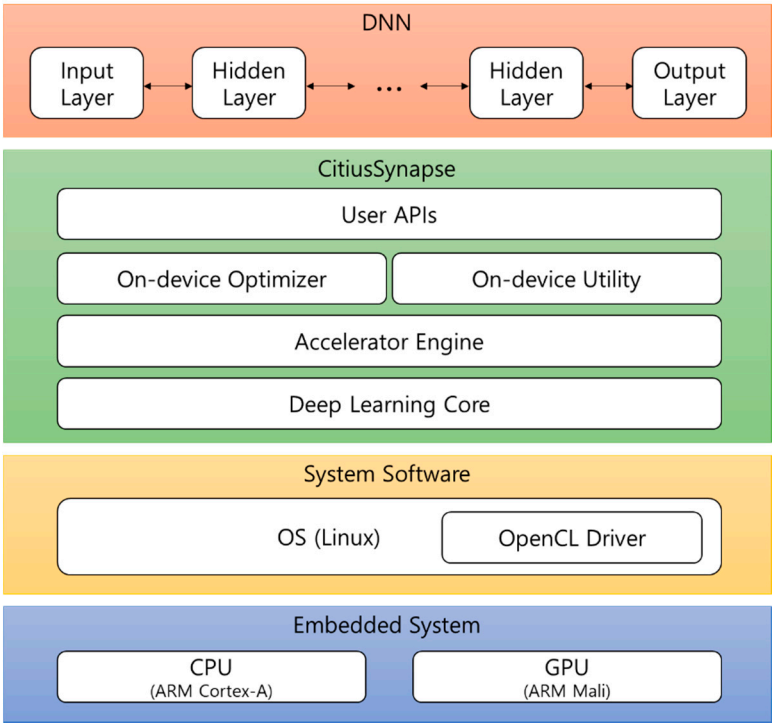


**Figure 1.** Overall architecture of our framework.

The major functions of each component are as follows.

- The deep learning core provides a data structure supporting a unified memory and a layer-oriented, implementation-based structure to minimize the deep learning operation cost.
- The accelerator engine provides a deep learning operation acceleration function that is optimized for embedded systems. The accelerator engine consists of a Basic Linear Algebra Subprograms (BLAS) library [22] that is optimized for an embedded GPU (CSblas), and a DNN-accelerated library, providing forward and backward operations of each layer constituting the DNN (clDNN).
- The on-device optimizer for inference optimizes the inference performance in an on-device environment. The on-device optimizer consists of a model optimizer for reducing operation costs through the combination of layers, an NDRange optimizer for searching the optimal parallel processing space of OpenCL, and a quantization optimizer for lightening the model through INT8 quantization.

- An on-device utility for compatibility provides a function for converting models that are trained in other frameworks, such as Caffe, PyTorch, or open neural network exchange (ONNX) [23] into our on-device framework. Moreover, it supports a function for automatically resetting the network by parsing the protocol buffer-based network configuration files.

Figure 2 shows the interaction flow of each component of our framework. First, the user sends network information to the user API of our framework (=CitiusSynapse) to perform inference (or training). Second, the user API initializes the network through the deep learning core. Third, the deep learning core constructs data and layer structures in conjunction with the accelerator engine. Fourth, the deep learning core executes deep learning operations that are necessary for inference or learning through an accelerator engine. At this time, the accelerator engine uses a CPU or GPU driver for operation. Finally, the deep learning core returns the final output to the user through user APIs.

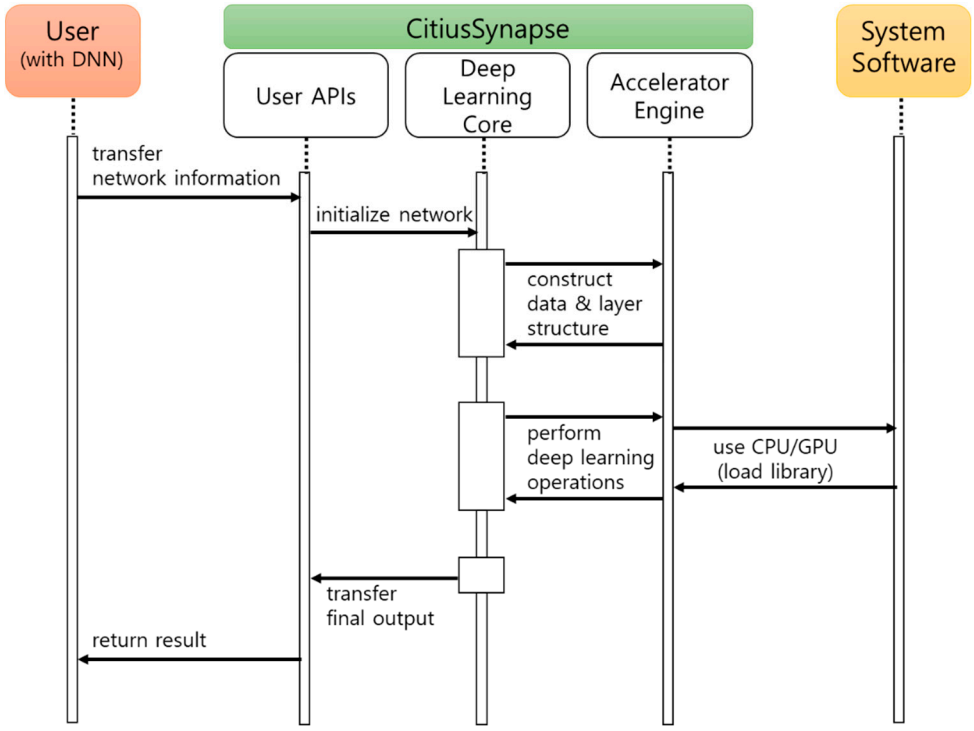


**Figure 2.** Interaction flow of framework components.

The remainder of this study is organized as follows. In Section 2, the background of this research and related works are reviewed. In Section 3, the deep learning core and accelerator engine of our framework are explained. Section 4 explains the on-device optimizer and model conversion utility. Section 5 shows the superiority of the proposed framework when compared to the conventional deep learning framework through a performance evaluation. Finally, conclusions and some discussion are presented in Section 6.

## 2. Background and Related Works

### *2.1. OpenCL*

OpenCL consists of a host program executed in a host, and a kernel executed in a computing device. The host program defines an index space for executing a kernel, and a work-item, or each instance, of the kernel is executed at each point of the index space. The work-group consists of multiple work-items, and all work-groups have the same size. For the size of the work-group, there is a global work-group size (GSZ) that indicates the total number of work-items, and a local work-group size (LSZ), which indicates the number of work-items per work-group.

An index space is divided into work-groups of the same size, in which each work-group is executed in parallel in one compute unit within a computing device. Therefore, the LSZ must be set to allow each work-group to be executed in parallel as much as possible in a compute unit, to maximize the performance during operations in OpenCL.

### 2.2. Accelerated Libraries for Deep Learning

The operation process of deep learning involves forward and backward operations, which are mostly vector or matrix operations. BLAS consists of a set of routines for performing general linear algebraic operations, such as vector or matrix operations. BLAS generally uses the specifications defined in Netlib [22], but there are various BLAS libraries optimized for each HW type, such as a CPU or GPU.

CPU-based BLAS libraries include the Linear Algebra Package (LAPACK) [24], Automatically Tuned Linear Algebra Software (ATLAS) [25], and OpenBLAS [26]. LAPACK and ATLAS only support single-core CPUs, whereas OpenBLAS supports multi-core CPUs.

CUDA-based BLAS libraries include cuBLAS [27], CUTLASS [28], and KBLAS [29], in which cuBLAS is a BLAS library officially provided by Nvidia. A CUDA-based BLAS library is only operable on a system equipped with an Nvidia GPU, and is inoperable in embedded systems, such as mobile devices.

In addition to the BLAS library, the Nvidia CUDA Deep Neural Network Library (cuDNN) can perform acceleration operations which are based on a GPU, for each layer of a DNN. Based on CUDA, cuDNN supports forward and backward operations in convolution, pooling, and softmax layers, which are essential layers of a DNN. Similar to CUDA, cuDNN is operable only in systems which are equipped with an Nvidia GPU.

OpenCL-based BLAS libraries are typically conducted using open-source code, because OpenCL is an open, universal parallel computing framework. Well-known OpenCL-based BLAS libraries include clBLAS [30], ViennaCL [31], and CLBlast [32]. The OpenCL-based BLAS library is currently optimized for the PC environment, thus failing to guarantee an optimized performance in embedded GPUs. In addition, insufficient research has been conducted on OpenCL-based convolutional neural network (CNN) accelerator libraries that support forward and backward operations of a layered label, as in cuDNN. The ARM compute library (ACL) [33] is a deep learning accelerated library for ARM CPUs and GPUs, managed by ARM Ltd. The ACL is only operable in ARM CPUs and GPUs; Caffe or TensorFlow models can be used when ArmNN [34] is used, but ACL alone cannot be linked with conventional deep learning frameworks.

### 2.3. Deep Learning Frameworks

Currently, the most popular deep learning frameworks are PyTorch, TensorFlow, Keras [35], Theano [36], Caffe, MXnet [37], Deeplearning4j [38], and Darknet [39]. The majority of deep learning frameworks are difficult to use in embedded systems because CUDA and cuDNN are used to accelerate deep learning operations. Therefore, an OpenCL-based GPU-accelerated library needs to be linked for deep learning frameworks to be efficiently executed in embedded systems. OpenCL Caffe [40], DeepCL [41], TensorFlow Lite [42], and Darknet on OpenCL [43] are deep learning frameworks that support OpenCL-based GPU-accelerated libraries at present.

Because OpenCL Caffe is managed in the GitHub repository of Caffe, the existing interface of Caffe can be used, while executing deep learning operations through linking with ViennalCL, clBLAS, and CLBlast. DeepCL can be linked with EasyCL [44] and clBLAS, providing the C++ and Python API. TensorFlow Lite is a lightened subset of the trained TensorFlow model for on-device inference execution. TensorFlow Lite accelerates deep learning operations through the Google neural network API (NNAPI) [45] and GPU delegates. Darknet on OpenCL is a revised version of the existing Darknet that uses CUDA and cuDNN to ensure that OpenCL can be used.

## 3. Deep Learning Core and Accelerator Engine

In order to perform efficient deep learning in embedded systems, fast inference speed is essential. In addition, because the embedded system has limited system resources, especially memory, it is very important not only to increase the inference speed, but also to reduce the memory usage. Therefore, our framework proposes a deep learning core and accelerator engine that can improve the parallel processing performance of OpenCL and minimize its memory usage. The accelerator engine maximizes the parallel processing performance of OpenCL, and the deep learning core provides data and layer structures to efficiently manage the accelerator engine.

Meanwhile, the embedded system has a different system structure, depending on the HW platform. For example, ARM Mali GPU does not have local memory, whereas Qualcomm Adreno GPU does. Also, even with the same ARM Mali GPU, the basic unit used for parallel processing is different (explained in detail in Section 3.2.1). Therefore, in order to improve the parallel processing performance of OpenCL in an embedded system, a structural analysis of the system and its optimization should be performed.

Currently, various studies for optimizing the inference in embedded systems are being conducted, but their utilization is limited as the optimization techniques of OpenCL are very different, according to the characteristics of HW and DNN. Therefore, our framework automatically analyzes the structural characteristics of embedded systems and optimizes them accordingly using our proprietary accelerator engine. Through this, our framework enables users to efficiently perform deep learning in embedded systems without any knowledge of OpenCL and embedded systems.

### *3.1. Deep Learning Core*

A deep learning core consists of a data structure for managing deep learning operation data, and a layer structure for each layer constituting a DNN.

#### 3.1.1. Data Structure with Unified Memory

As shown in Figure 3a, a standard PC has a separate memory for the CPU (host) and GPU, in which the OpenCL buffer that is generated in the CPU needs to be copied to the GPU in order to execute OpenCL in the GPU. Moreover, data also need to be copied from the GPU to the CPU in order to import the result of operations executed in the GPU. Notably, recent CPUs have a built-in GPU, and share memory through the built-in GPU structure. However, in most PCs, the CPU and GPU are configured independently for the efficiency of deep learning operations.

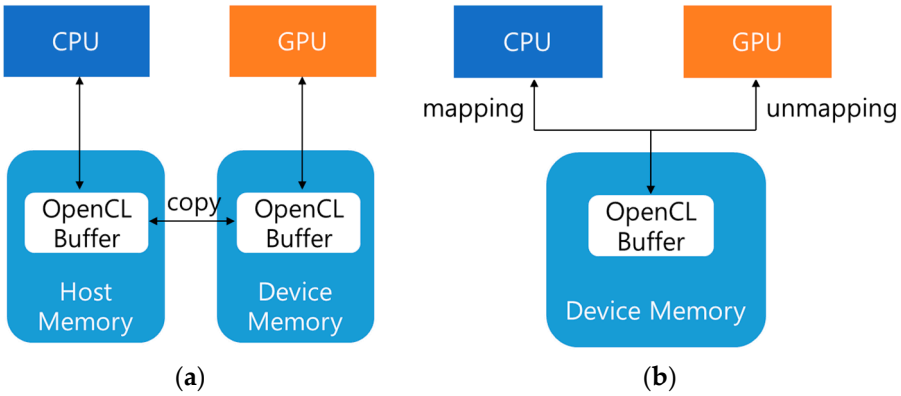


**Figure 3.** Comparison of memory architecture with and without unified memory. (**a**) Memory architecture without unified memory, (**b**) Memory architecture with unified memory.

In contrast, embedded systems have a unified memory structure, in which the CPU and GPU share a single memory space, as shown in Figure 3b; thus, data can be shared between the CPU and GPU without transfer. The memory required for deep learning

operations can be minimized, because the CPU and GPU do not need to create separate memory spaces. In particular, for an embedded system with limited bandwidth for memory size and data transfer, it is necessary to minimize the operational costs required for deep learning operations, using the characteristics of unified memory.

However, the data structure of conventional deep learning frameworks does not support the unified memory of an embedded system, and thus requires an unnecessarily costly data transfer between the CPU and GPU. To overcome this drawback, our framework, an OpenCL buffer, is generated in the device memory, as shown in Figure 3b, and the memory is shared using the memory mapping and unmapping functions of OpenCL. In other words, the generated buffer is mapped if the OpenCL buffer is to be used in the CPU, while the generated buffer is unmapped if the OpenCL buffer is to be used in the GPU.

When executing operations in the hidden layers of a DNN, only the OpenCL buffer for the GPU is generated, and used without mapping or unmapping. Due to the fact that the GPU result needs to be returned to the CPU when the final output is returned, it is unnecessary to return the GPU result to the CPU through mapping when the operations are executed in hidden layers. Moreover, the acceleration performance of OpenCL is reduced when the operation result is returned from the GPU to the CPU during operation execution in hidden layers, because the asynchronous operation processing structure of OpenCL cannot be applied. The asynchronous operation processing structure of OpenCL is explained in Section 3.2.2. Also, the performance improvement by unified memory is explained in detail in Appendix A.

#### 3.1.2. Layer Structure with Layer-Oriented Implementation

Our framework is implemented in the C++11 programming language to be operable in various embedded systems, and provides a layer-oriented implementation-based layer structure to effectively use the data structure. Figure 4 shows the architecture of the layer structure, which is linked to the data structure of our framework.

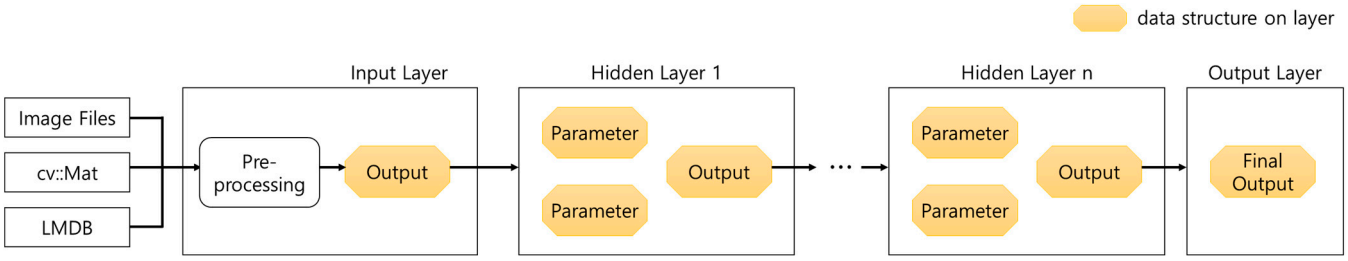


**Figure 4.** Structure diagram of the layer structure linked to the data structure.

The layer structure consists of an input layer, a hidden layer, and an output layer that is appropriate for the DNN configuration. There is one input layer and one output layer, and there may be multiple hidden layers, depending on the DNN configuration. Each layer has an independent data structure that exists in a parameter or output format, depending on the layer characteristics. Here, the output of each layer is delivered as the input to the following layer.

The input layer supports the Mat object of OpenCV [46], in addition to image files in JPEG or PNG formats being appropriate for an on-device inference environment. Processing the data captured in the actual environment by converting them to image files is inefficient in on-device environments. Furthermore, the Lightning Memory-Mapped Database (LMDB) [47] is supported as an input to use multiple image files efficiently during training. In addition, the input layer supports pre-processing, such as mean subtraction and normalization, and can be easily used through C++11-based user APIs.

Hidden layers receive the output of an input layer, or previous hidden layers, as their input. According to the layer characteristics, hidden layers may receive multiple inputs or the output of a specific step through user APIs, instead of the output of a previous layer.

Finally, the output layer has the final output that is computed, based on the output of the hidden layers.

Table 1 presents the list of hidden and output layers supported by our framework, based on the layer name used in the Caffe framework. The convolution layer of our framework uses symmetric padding as the default, which may result in an operation difference from the learning model of TensorFlow, which uses asymmetric padding as the default. Hence, our framework uses a padding layer [48] to compensate for the difference in the padding algorithm used in the TensorFlow learning model.

**Table 1.** List of hidden and output layers of our framework.

| Layer Type | Layer Name |
| --- | --- |
| Vision layer | Convolution, Depthwise convolution, Deconvolution, Pooling (Max/Average/Global Average), Upsample, Focus [1] |
| Recurrent layer | LSTM (Long Short-Term Memory) |
| Normalization | Batch Normalization, Scale |
| Activation | ReLU, ReLU6, Sigmoid, H-Swish, SiLU [1] |
| Utility | Eltwise (Sum/Product), Concat, Reshape, Power, Permute, padding |
| Common | Full-Connected (=Inner-Product) |
| Output | Softmax, Yolo v2/v3/v5, EAST [2], kNN, PoseNet [3] |
| Updater | SGD, Adam |

[1] Used only in Yolo v5. Available online: https://github.com/ultralytics/yolov5 (accessed on 23 August 2021). [2] Available online: https://github.com/SURFZJY/EAST-caffe (accessed on 23 August 2021). [3] Available online: https://github.com/microsoft/human-pose-estimation.pytorch (accessed on 23 August 2021).

### 3.2. Accelerator Engine

The accelerator engine consists of OpenCL-based BLAS (CSblas), which is optimized for embedded GPUs, and a DNN-accelerated library (clDNN). CSblas and clDNN execute parallel processing using OpenCL, for which a separate OpenCL kernel is used. An OpenCL kernel is generally built during runtime, based on the characteristics of OpenCL, which allows the same OpenCL kernel to be operable in various HW platforms. However, building OpenCL kernels during runtime is inefficient. Therefore, our framework uses raw C++11 string literals to ensure that the code of OpenCL kernels is embedded in the accelerator engine when the framework is built. OpenCL kernels that are embedded in the accelerator engine are converted to an OpenCL execution binary during inference (or learning) in the on-device environment. Kernels are constructed only during the first inference, and the generated binaries are cached for reuse, starting from the second inference. Here, only the kernels which were used for inference were built, to prevent the generation of unnecessary kernel binaries.

#### 3.2.1. CSblas

BLAS is divided into vector–scalar operation routines (LEVEL 1), matrix–vector operation routines (LEVEL 2), and matrix–matrix operation routines (LEVEL 3) according to the operation data type. Each routine of BLAS is categorized as single (float), double, complex, or double complex according to the operation precision, and supports single and half (fp16) precision, which are frequently used in the latest DNNs.

Table 2 presents the common routines used in the latest DNNs, among the BLAS operation routines that are provided by our framework. AXPY is often used in the scale layer and Eltwise layer, SCAL is used in the scale layer, GEMV is used in the batch normalization layer, and GEMM is used in the convolution layer and fully connected layer. The operations of GEMV and GEMM are distinguished depending on whether the matrix is transposed. Specifically, two GEMV operations may be distinguished (GEMV_T and GEMV_N) depending on whether matrix A is transposed (true or false), whereas GEMM

categorizes four operations (GEMM_NN, GEMM_NT, GEMM_TN, GEMM_TT) depending on whether matrices A and B are transposed.

**Table 2.** List of representative BLAS routines provided by our framework.

| Routine Level | Routine Name | Operation |
|---|---|---|
| Level 1 | AXPY | Y = αX + Y (X, Y are vectors, α is a scalar) |
| | SCAL | X = αX (X is a vector, α is a scalar) |
| Level 2 | GEMV | Y = αAX + βY (X, Y are vectors, α, β are scalars, A is a matrix) |
| Level 3 | GEMM | C = αAB + βC (A, B, C are matrices, α, β are scalars) |

In general, an OpenCL device supports single instruction multiple data (SIMD) commands, and simultaneously loads/stores multiple nearby data using the data type of an OpenCL vector, or executes the same operation for multiple data. Our framework drastically reduces the overhead in operations and memory access using the characteristics of OpenCL.

The maximum amount of data that can be simultaneously processed in OpenCL is determined based on the vector register size of the device. For instance, an ARM Mali-T860 has a 128-bit vector register and can simultaneously load/store up to four 32-bit data values. In contrast, an ARM Mali-G52 has a 256-bit vector register, and thus can simultaneously load/store up to eight 32-bit data values. Therefore, the vectorization size must be determined by considering the size of the vector register, in order to maximize the parallelism of OpenCL. However, analyzing the vector register size of an OpenCL device is inconvenient for both developers and users. Therefore, our framework automatically analyzes the device information through the OpenCL parameter and sets the vectorization value that is optimized for the device.

Our framework uses CL_KERNEL_PREFERRED_WORK_GROUP_SIZE_MULTIPLE to set the vectorization value. The respective parameter value represents a multiple LSZ value that is optimized for the current device. For example, when the CL_KERNEL_PREFERRED_WORK_GROUP_SIZE_MULTIPLE value is four, the LSZ value must be set to a multiple of four in order to optimize the performance. This value generally corresponds to the vector register size of an OpenCL device. The proposed framework sets the vectorization value using this parameter information because the vector register size of an OpenCL device cannot be directly confirmed using the parameter information.

Figure 5 illustrates the schematic diagram of the parallel processing of GEMM_NN, which is the LEVEL 3 routine of BLAS using OpenCL in our framework. GEMM_NN is the case in which matrices A and B are not transposed in GEMM. GEMM_NN executes the multiplication of matrices A and B, in which the value of one row in matrix A is read and multiplied by all the values read in matrix B. Next, *n* data values are continuously loaded from columns in matrix B using *vloadn*. Furthermore, *n* data values are simultaneously processed using *vloadn* and *vstoren* in matrix C, because the column position for loading data in matrix B corresponds to the column position for storing the result values in matrix C.

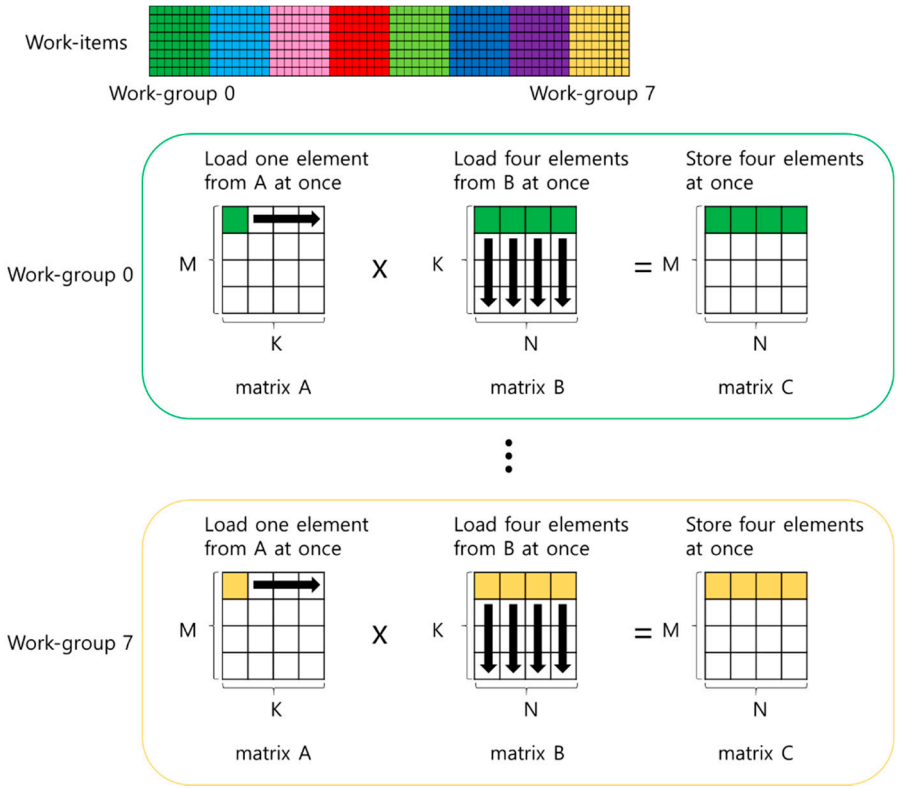


**Figure 5.** Parallel processing example of GEMM_NN using OpenCL.

3.2.2. clDNN

The layer proposed in the latest DNNs has limitations that hinder the utilization of BLAS. For example, when an operation is executed using GEMM for depth-wise convolution, which is added in MobileNet-v1 [49], GEMM needs to be repeatedly executed sequentially by the number of filters. Furthermore, there are layers that cannot be executed using BLAS, such as a pooling layer. In this case, our framework provides a DNN-accelerated library (clDNN), which executes parallel processing based on OpenCL for each layer consistency of the DNN. The essential functions of clDNN are in-place kernel optimization and asynchronous queue execution.

1. In-place kernel optimization

A convolution layer generally proceeds in the order of *im2col* for the input matrix, GEMM for the weight parameter, and GEMM for the bias parameter (if the bias parameter is present). GEMM for the weight and bias parameters share the operation spaces of each matrix. Hence, unnecessary operation costs are added, as repetitive accesses occur for each matrix. To resolve this issue, an in-place kernel is used to simultaneously execute operations for the weight and bias parameters.

The bias of a convolution layer is determined by the number of weight filters, where the number of weight filters is identical to the number of output channels of the convolution layer. Based on these characteristics, the GEMM operation is executed for the weight parameter in a convolution layer and adds bias corresponding to the output channel when a bias parameter of a convolution layer is present. Moreover, activation is added to the final output of a kernel when an activation function, such as a rectified linear unit (ReLU), is present after a convolution layer. Accordingly, clDNN maximizes the OpenCL acceleration performance by minimizing the repeated costs of accessing the data. Table 3 shows the operation list of layers for which the in-place kernel optimization is provided in the proposed framework.

**Table 3.** List of operations in layers that provide in-place kernel optimization.

| Layer | In-Place Operation |
| --- | --- |
| Convolution layer | GEMM (weight) + GEMM (bias) + Activation (ReLU/ReLU6/Sigmoid/SiLU) |
| Depthwise convolution layer | Depthwise convolution algorithm (weight + bias) + Activation (ReLU/ReLU6) |
| Deconvolution layer | Col2im + Activation (ReLU) |
| Eltwise | Layer sum/production + Activation (ReLU) |
| Scale | Layer scale (weight + bias) + Activation (ReLU) |
| Full-Connected layer | GEMM (weight) + GEMM (bias) + Activation (ReLU/ReLU6/Sigmoid) |

2. Asynchronous queue execution

OpenCL has one command queue for each computing device, while the host enqueues commands, such as data transfer or kernel execution, to the command queue. When running an OpenCL program, the OpenCL runtime executes commands by sequentially dequeuing the commands that are enqueued in the command queue. Here, OpenCL uses synchronous queue execution, in which the program waits until the executed command ends. Synchronous queue execution has the limitation of delaying the processing speed of a host, because the right to control the command that is executed in a computing device does not return to the host until the command ends, and this induces kernel launch overhead [50]. Therefore, our framework processes each OpenCL command in an asynchronous manner, thus maximizing GPU utilization.

In contrast, when commands in the command queue are executed in an asynchronous queue execution method, the operation result of each layer must be delivered sequentially. Thus, our framework calls clFlush() when the final operation of each layer ends. clFlush() issues all commands in the command queue to the device, thus guaranteeing the execution order of the command input in the command queue. However, clFinish() blocks the host until all commands are executed. clFinish() produces a greater amount of overhead when compared to clFlush(), which simply issues commands to the device [51]. Therefore, GPU utilization is maximized, while guaranteeing the accuracy of the layer operation results, through use of clFlush().

Figure 6 shows the differences between the synchronous and asynchronous queue executions. In the synchronous queue execution method, each kernel is sequentially executed, regardless of the processing capacity of the computing device. In contrast, the asynchronous queue execution method can simultaneously process commands if the processing capacity of the computing device is sufficient, and the executed kernels are independent. Kernel2 and kernel3, in the example, use the output of kernel1 as input, but the operation results do not affect each other. Moreover, OpenCL can process independent tasks simultaneously because it supports task parallelism. Owing to these characteristics, our framework applies asynchronous queue execution to allow independent kernels to be processed simultaneously.

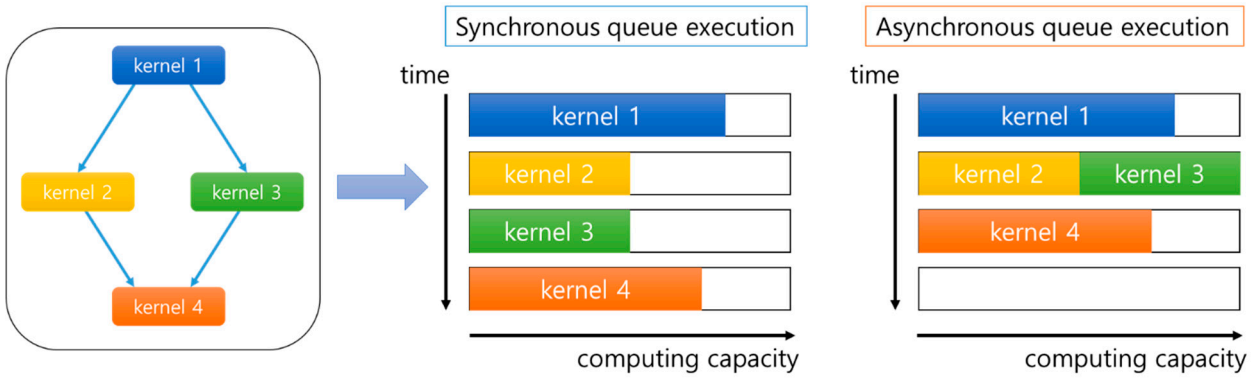


**Figure 6.** Differences in kernel execution between synchronous and asynchronous queue execution.

## 4. On-Device Optimizer and Utility

### *4.1. On-Device Optimizer for Inference*

An on-device optimizer is included in the proposed framework, which consists of a model optimizer for reducing the operating costs of a model through a combination of layers, an NDRange optimizer for searching the optimal parallel processing space of OpenCL, and a quantization optimizer for lightening the model through INT8 quantization.

#### 4.1.1. Model Optimizer

Recent DNNs, such as ResNet and MobileNet, have been applied with batch normalization [52] to improve training and inference performance. Batch normalization is typically used after a layer with parameters, such as a convolution layer. Batch normalization involves re-centering and re-scaling a layer's input to ensure that the DNN is faster and more stable. Therefore, batch normalization involves five parameters: mean, variance, moving average, scale weight, and scale bias. Each parameter is updated as the iterations proceed in the training phase, and is fixed in the inference phase. The parameters of batch normalization can be combined with an existing convolution layer during inference based on such characteristics. When the parameters of batch normalization are combined with a convolution layer, the GPU acceleration performance can be maximized by minimizing the repeated operations of OpenCL through in-place kernel optimization, as explained in Section 3.2.2.

Batch normalization has different layer compositions depending on the framework. For example, Caffe consists of two layers, including batch normalization and a scale layer, while TensorFlow and PyTorch have a single layer. Our framework can be divided into batch normalization and scale layers, as in Caffe. An open-source method is available for Caffe and PyTorch [53,54] to combine the batch normalization layers, but its use involves certain limitations, such as regularity in layer names. In contrast, the combination method of our framework analyzes the layer configuration information of the trained model and automatically identifies the possibility of combining it with batch normalization. The parameters of the model are updated, and a new model with combined layers is generated if batch normalization can be combined. Moreover, our framework provides model compatibility with other frameworks, such as Caffe or PyTorch, which enables it to support model optimization through layer combination in the models of other frameworks. This model compatibility is explained in detail in Section 4.2.

Algorithm 1 is the model optimization algorithm, based on the layer combination of the proposed framework. Model optimization through layer combination requires the parameters of the model to be updated and, therefore, the process is carried out offline in advance. Convolution, depth-wise convolution, and deconvolution layers can be combined with batch normalization through the model optimization algorithm.

The model optimization algorithm combines the parameters of batch normalization into layers with weight parameters. This uses a certain feature: that batch normalization

parameters are used as constants during inference. Machine epsilon for the probability value occurs during the division operation of the model optimization algorithm. However, because it is quite small, this does not affect the accuracy.

**Algorithm 1** Model Optimization Algorithm.

**Input**: *model*–a pre-trained model, *epsilon*–constant used in BatchNorm

```
n ← number of hidden layers in model
for i = 0 → n do
  if type(hidden_layers[i]) == Conv/DepthwiseConv/Deconv then
      if (type(hidden_layers[i + 1]) == BatchNorm) and
        (type(hidden_layers[i + 2]) == Scale) and (i <= n − 2) then
          weight, bias ← GetParam(hidden_layers[i])
          mean, variance, moving_average ← GetParam(hidden_layers[i + 1])
          scale_weight, scale_bias ← GetParam(hidden_layers[i + 2])
          scalef ← 1/moving_average
          mean ← mean * scalef
          variance ← variance * scalef
          rstd ← 1/sqrt(variance + epsilon)
          updated_weight ← weight * rstd * scale_weight
          updated_bias ← (bias–mean) * rstd * scale_weight + scale_bias
          Store parameters(updated_weight, updated_bias) to optimized model
          i ← i + 2
  else
      Store parameters in hidden_layers[i] to optimized model
  return optimized model
```

#### 4.1.2. NDRange Optimizer

As mentioned in Section 2.1, OpenCL is processed in parallel by dividing it into a work-group of the same size. To maximize the parallel processing performance of OpenCL, an optimal LSZ needs to be set, to efficiently divide and process the GSZ: the size of the entire data. However, it is difficult to determine the LSZ value because various factors need to be considered, such as the computational number of kernels or the processing performance of the GPU. Therefore, our framework provides an NDRange optimizer to search for the optimal LSZ in an on-device environment. The NDRange optimizer applies a brute-force methodology for the kernel being executed to search for the optimal value corresponding to the on-device environment. To minimize the search range, the space of a multiple of two or a multiple of four was searched. As explained in Section 3.2, in general, an optimal speed is observed when the search space is a multiple of two or four, because vectorization has been applied to an OpenCL kernel. Moreover, the kernel parameters are used as search parameters because different execution results may be produced depending on the input parameters, even if the kernels are the same. For example, the GEMM kernel uses M, N, and K parameters, which represent the matrix size as the search parameters. The value searched by the NDRange optimizer can be saved in a file format or embedded within the framework, and the optimal LSZ value is automatically chosen without re-searching, using the saved data obtained during the actual inference process. In addition, the NDRange optimizer calculates the GSZ based on the searched LSZ, to ensure the accuracy of the execution result. Therefore, the NDRange optimizer does not affect the accuracy.

#### 4.1.3. Quantization Optimizer

The INT8 quantization technique has recently gained popularity for reducing the computation required of the latest DNNs. INT8 quantization is a technique for compressing a 32-bit floating-point (FP32) number into an 8-bit integer value. A scale factor for converting an FP32 value into an INT8 value must be calculated in advance to apply INT8 quantization. The scale factor is divided into an activation scale factor, for changing the layer input value, and a weight scale factor, for changing the weight parameters. A single

activation scale factor is used in each layer, and the number of weight scale factors matches the number of filters in each layer.

A suitable scale factor must be calculated based on the calibration dataset, to minimize the difference in the distributions of the FP32 and INT8 values. Accordingly, our framework uses Kullback–Leibler divergence (KLD) [55] as the activation scale factor, and min-max normalization as the weight scale factor to calculate each scale factor [56].

Quantization of the weight parameters using the weight scale factor is typically conducted offline to reduce the quantization overhead during runtime. In contrast, quantization of input values is performed during runtime, because these values cannot be specified in advance. Furthermore, the operation result that is obtained through INT8 quantization must be dequantized to FP32 to be delivered to the next layer. More specifically, INT8 quantization entails quantization of input values and dequantization of the final output, in which quantization and dequantization overhead causes a reduction in the performance gain from INT8 quantization. Therefore, optimization of an INT8 operation kernel based on OpenCL, that offsets the quantization and dequantization overhead, is absolutely necessary to maximize the performance gain through INT8 quantization. For this purpose, the proposed framework utilizes a Winograd convolution algorithm [57,58] by applying INT8 quantization. The Winograd convolution algorithm drastically reduces the overall operation complexity by increasing the number of addition operations, to reduce the number of multiplication operations in a convolution layer, which is a common method used. The operation complexity and accuracy vary depending on the output and filter size in the Winograd convolution algorithm. Our framework uses a Winograd convolution algorithm with a $2 \times 2$ output and a $3 \times 3$ filter, to guarantee accuracy.

Figure 7 illustrates the schematic diagram of the Winograd convolution algorithm in which INT8 quantization is applied. Input and filter values must be converted because of the nature of the Winograd convolution algorithm, which may result in multiplying 4 and 16 values, respectively, in the worst-case scenario. Therefore, after converting the input value to 6 bits and the weight parameter to 4 bits, the converted input and filter values are then used to execute the INT8 GEMM operation, in order to guarantee that the conversion operation result is 8 bit. If an 8 bit dot product instruction that supports GPU acceleration is available, the respective instructions must be used to improve the INT8 operation performance. ARM and Qualcomm GPUs are well-known platforms that support the GPU acceleration of 8 bit dot product instructions. Our framework supports the arm_dot instruction [59] of the ARM GPU. When the arm_dot instruction is used, 8 bit dot products, used in INT8 GEMM, can be processed as one instruction, maximizing the performance gain due to INT8 quantization.

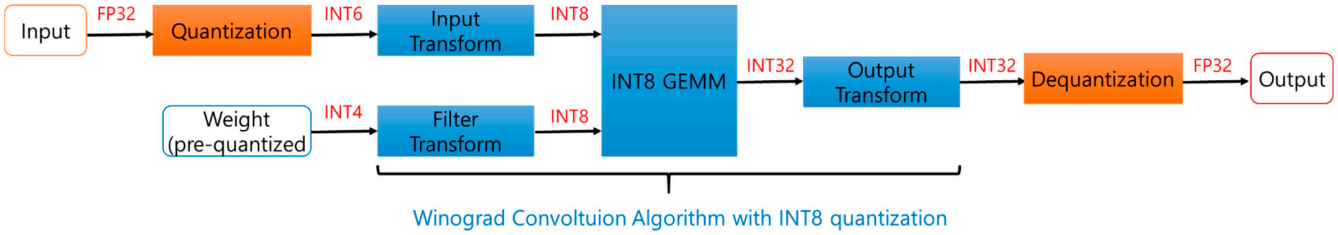


**Figure 7.** Schematic diagram of Winograd convolution algorithm applying INT8 quantization.

### *4.2. Model Converter for Compatibility*

Our framework provides a utility for importing a model that is trained with other deep learning frameworks for compatibility. Our framework provides the best compatibility with frameworks which are based on C/C++11. In particular, most Caffe-based learning models can be directly converted to the proposed framework because they are based on a static network configuration description. As depicted in Figure 8, the process of converting the model is as follows.

- First, a neural network visualization tool, such as Netron [60], needs to be used to identify the overall network structure.
- Second, a neural network is configured in the same order and connection using the user APIs of our framework.
- Finally, the model is converted to the model format for our framework layer-by-layer according to the layer order of a neural network that has been reset in our framework by loading the Caffe-based learning model (.caffemodel).

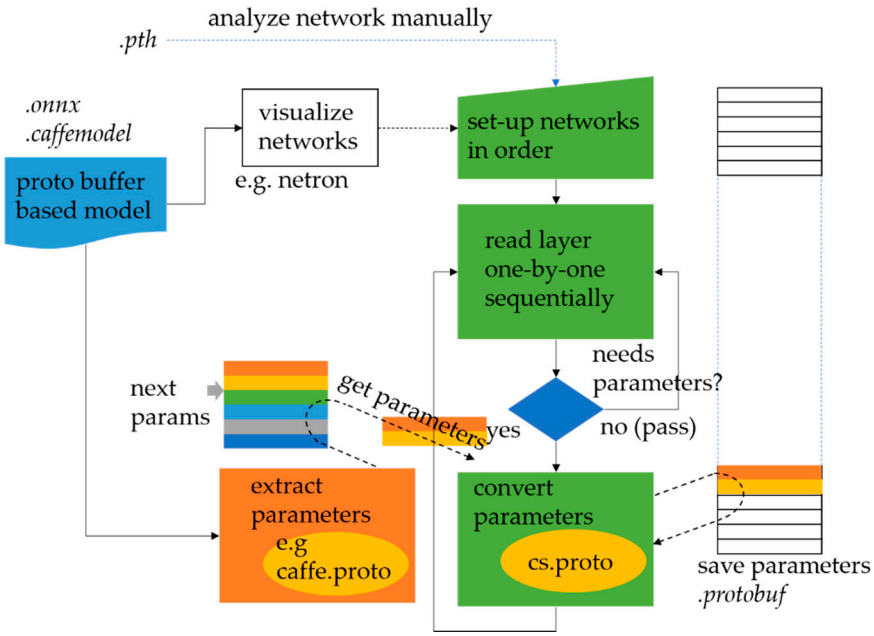


**Figure 8.** Model conversion process of our framework.

In Figure 8, basically, it is a fully manual process for the user to write the initialization codes of layers using the user APIs of our framework. However, if network configuration can be provided with a network configuration file, the set-up process will be done automatically in our framework.

It is notable that, when the network is set up in order, this order does not necessarily mean the actual connecting sequences, but rather indicates the order that saves the parameters in the corresponding framework. Because not all the layers hold the parameters, we can select and convert parameters from other formats to that of our framework by checking the type, layer-by-layer, if the network is set up in the correct sequence.

The proposed framework also supports ONNX, which is applied as a de facto standard for converting models between frameworks. Similar to Caffe, ONNX also statically describes the network based on a protocol buffer [61]; thus, visualization of a neural network through Netron and model conversion is relatively straightforward. Hence, on a framework such as TensorFlow, our framework provides a two-step model conversion function through ONNX.

PyTorch provides a C/C++ based Torch script [62], but the basic language is Python, and a large number of learning models are actually saved in a state dictionary format. Hence, a Python-based utility is provided for converting the state dictionary model of PyTorch into the protocol buffer format which is defined in our framework.

Most deep learning frameworks, including our framework, provide user APIs to allow users to directly program neural networks. In contrast, they may have an internal module that parses external text files for the network configuration, and establishes a neural network on its own. This has the advantage that the neural network can be easily changed and tested at the level of the application program without any modifications to the source code. Many developers use such a method when they wish to obtain comparison

results quickly; certain frameworks, such as Darknet and Caffe, explicitly require such network configuration files.

To externally describe network configuration as text-based on protocol buffers, our framework also optionally supports parsing and conversion into our framework for the convenience of development. In other words, a network written by our user APIs can be automatically generated internally by parsing the network configuration file (.prototxt) of Caffe.

## 5. Experiments

### 5.1. Experimental Setup

To demonstrate the usefulness of our framework, we conducted performance evaluations on its inference time and memory usage. Table 4 shows the performance evaluation environment. The RK3399 [63] and Odroid-N2+ [64] computing devices are equipped with ARM CPUs and GPUs, and are used in embedded systems. However, the currently popular platform Jetson Nano [65] does not support OpenCL, and only supports CUDA and cuDNN. Therefore, RK3399 and Odroid N2+ are largely used for performance evaluation, and Jetson Nano is used to relatively compare the performance of our framework and CUDA-based deep learning frameworks, such as PyTorch.

**Table 4.** Performance evaluation environment.

| System | RK3399 | Odroid N2+ | Jetson Nano |
|---|---|---|---|
| CPU | ARM Cortex-A72 (2 cores, 2.0 Ghz) ARM Cortex-A53 (4 cores, 1.5 Ghz) | ARM Cortex-A73 (4 cores, 2.4 Ghz) ARM Cortex-A53 (2 cores, 2.0 Ghz) | ARM Cortex-A57 (4 cores, 1.43 Ghz) |
| GPU | ARM Mali-T860 MP4 | ARM Mali-G52 | NVIDIA Maxwell GPU (128 cores) |
| Memory | 4 GB | 4 GB | 4 GB |
| Storage | 64 GB eMMC | 64 GB eMMC | 128 GB |
| OS | Ubuntu 18.04.7 | Ubuntu 18.04.5 | Ubuntu 18.04.5 (JetPack 4.5.1) |
| OpenCL support | OpenCL 1.2 | OpenCL 2.0 | Not supported (CUDA 10.2, cuDNN 8.0) |

In addition, TensorFlow Lite only supports OpenCL-based DNN acceleration in Android. Hence, we compared the inference speed relative to NNAPI, which is a typical Android-based GPU-accelerated engine, based on the performance evaluation results of the AI benchmark [66]. In the AI benchmark, the inference performance was measured for various networks, such as for MobileNet-v2 and Inception-v3, in the GPUs of diverse mobile devices, in addition to the GPU of a general PC.

A shared library between our framework and OpenCL Caffe was generated to perform a performance evaluation in an environment that was similar to the on-device operational application environment. The inference time and memory consumption were then compared through a test application, implemented with the generated shared library. In contrast, OpenCL Caffe uses ViennaCL as the default for a GPU-based accelerated engine, and additionally supports clBLAS and CLBlast. CLBlast is known to have the best performance among OpenCL-based BLAS libraries that are currently disclosed [15]; therefore, CLBlast was used as the GPU-based accelerated engine for OpenCL Caffe. Moreover, OpenBLAS, which uses a multi-core, was used as a CPU-based BLAS library in our framework, as well as OpenCL Caffe.

Table 5 presents a list of the DNNs used in the performance evaluation. The model, trained in OpenCL Caffe, was converted using the model converter of our framework to conduct a performance evaluation with DNNs having the same layer configuration and parameters. OpenCL Caffe does not support YOLOv3 [67], EAST [68], or CRNN [69], and it

is separately managed in a Git repository that supports CUDA. Therefore, the performances of Yolo v3, EAST, and CRNN were only evaluated in our framework.

**Table 5.** List of DNNs used in performance evaluation.

| DNN | Number of Class (Dataset) | Size of Trained-Model (MB) | Size of Input Images (Width × Height × Channel) |
|---|---|---|---|
| VGG-16 [70] | 1000 (ImageNet [71]) | 527 | 224 × 224 × 3 |
| ResNet-18 [72] | 1000 (ImageNet) | 44.6 | 224 × 224 × 3 |
| ResNet-101 [73] | 1000 (ImageNet) | 170 | 224 × 224 × 3 |
| MobileNet-v2 [74] | 1000 (ImageNet) | 13.5 | 224 × 224 × 3 |
| Yolo v3 | 20 (VOC [75]) | 14.4 | 320 × 320 × 3 |
| EAST | 1 (Text Region) | 15.9 | 320 × 320 × 3 |
| CRNN | 10 (Digit) | 23.9 | 128 × 32 × 3 |

### *5.2. Comparison of Inference Times*

Table 6 presents a comparison of the inference times between OpenCL Caffe and our framework. For the performance evaluation, inference was continuously performed on 100 images in each DNN, and then the average inference time was measured. The images which were used for inference were randomly selected from the dataset used in each DNN. In addition, the first inference time was excluded, because the build time of OpenCL kernels was added during the first inference, when the DNN inference was accelerated based on OpenCL. For our framework, the inference time, to which the model and NDRange optimization were applied, was also measured. The performance of VGG-16, to which the model and NDRange optimization were applied, was not evaluated, because batch normalization was not used for VGG-16.

**Table 6.** Comparison of inference times (ms) in Odroid N2+/RK3399.

| System | DNN | OpenCL Caffe | | Our Framework | | | | Speed-Up Ratio (%) |
|---|---|---|---|---|---|---|---|---|
| | | CPU | GPU | CPU | GPU | GPU (+MO [1]) | GPU (+MO&RO [2]) | |
| Odroid N2+ | VGG-16 | 1618 | 12,005 | 1606 | 1207 | - | - | 134 [3] |
| | ResNet-18 | 236 | 1792 | 227 | 203 | 141 | 129 | 183 [4] |
| | ResNet-101 | 877 | 7522 | 814 | 908 | 536 | 515 | 170 [4] |
| | MobileNet-v2 | 308 | 3082 | 280 | 232 | 62 | 40 | 770 [4] |
| | Yolo v3 | Not supported | | 841 | 367 | 118 | 80 | 1051 [5] |
| | EAST | Not supported | | 795 | 459 | 176 | 153 | 520 [5] |
| | CRNN | Not supported | | 80 | 58 | 42 | 40 | 200 [5] |
| RK3399 | VGG-16 | 2562 | 10,164 | 2855 | 2218 | - | - | 116 [3] |
| | ResNet-18 | 366 | 1656 | 354 | 363 | 255 | 246 | 149 [4] |
| | ResNet-101 | 1930 | 7242 | 1299 | 1467 | 963 | 928 | 208 [4] |
| | MobileNet-v2 | 444 | N/A (error) | 930 | 354 | 94 | 64 | 694 [4] |
| | Yolo v3 | Not supported | | 1603 | 529 | 193 | 146 | 1098 [5] |
| | EAST | Not supported | | 1457 | 652 | 254 | 224 | 650 [5] |
| | CRNN | Not supported | | 99 | 81 | 63 | 61 | 162 [5] |

[1] Model optimization. [2] NDRange optimization. [3] CPU in OpenCL Caffe/GPU in our framework. [4] CPU in OpenCL Caffe/GPU (+MO/RO) in our framework. [5] CPU in our framework/GPU in our framework.

The evaluation results show that the GPU-based inference performance in OpenCL Caffe was considerably reduced when compared with that of the CPU. Because OpenCL Caffe is optimized for a general PC environment, it does not guarantee that the performance is optimized for an embedded system. The GPU-based inference performance using OpenCL had improved further than the CPU in most cases in our framework. In particular, the performance improved even further when model optimization (MO) and NDRange optimization (RO), which were proposed in our framework, were applied.

When the inference time was compared between OpenCL Caffe and our framework, the results show that performance was improved by 134% for heavy DNNs, such as VGG-16, and by 770% for light DNNs, such as MobileNet-v2. VGG-16 has a large number of parameters, and the operation is executed sequentially in each layer; thus, an optimization method applicable to an embedded system is limited, because operational resources are also limited. In contrast, the GPU acceleration performance significantly improved through the methods proposed in our framework for lightened DNNs, such as ResNet or MobileNet. In particular, in-place kernel optimization that simultaneously executes operations for weight and bias parameters can be applied if model optimization, which combines the batch normalization layer, is applied, which minimizes unnecessary operating costs. The evaluation also shows that the OpenCL-based acceleration performance can be maximized by setting the optimal parallel processing space for each OpenCL kernel when the NDRange optimizer is applied.

Table 7 presents a comparison of the inference time between our framework and AI benchmark. The inference time of the GPU, similar to the ARM Mali-G52 equipped in Odroid-N2+ used in the performance evaluation of this study, is summarized for comparison. In the AI benchmark, AI Benchmark 3.0.0 [76], operated in Android, was used for the performance evaluation. AI Benchmark 3.0.0 provides the benchmark functions to compare CPU and NNAPI-based execution results. NNAPI was used in the AI benchmark to evaluate the performance of mobile devices. The inference time of MobileNet-v2 was 51 ms when the GPU (Mali-G52 MP2), which was most similar to the performance evaluation of this study among various performance evaluations of the AI benchmark, was used. The inference time of MobileNet-v2 was 40 ms when the model and NDRange optimization of our framework were applied. Accordingly, these results verify that our framework is competitive with popular DNN acceleration techniques, such as NNAPI.

**Table 7.** Comparison of inference times (ms) of our framework and AI Benchmark.

| DNN | Our FrameWork | AI Benchmark AI Benchmark 3.0.0 with NNAPI | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | Odroid N2+ (Mali-G52) | MediaTek Helio G90T (Mali-G76 MP4) | Exynos 9810 Octa (Mali-G72 MP18) | Exynos 8895 Octa (Mali-G71 MP20) | MediaTek Helio P70 (Mali-G72 MP3) | MediaTek Helio P65 (Mali-G52 MP2) | MediaTek Helio P60 (Mali-G72 MP3) | Exynos 9609 (Mali-G72 MP3) | Exynos 9610 (Mali-G72 MP3) |
| Mobile Net v2 | 40 [1] | 37 | 72 | 63 | 66 | 51 | 68 | 61 | 77 |

[1] It applies model and NDRange optimization at our framework.

Table 8 presents a comparison of the inference time between PyTorch and our framework. Our framework performed inference on an Odroid N2+, and PyTorch performed inference on a Jetson Nano. This is because, as mentioned earlier, Jetson Nano does not support OpenCL. The Jetson Nano has 128 CUDA cores and shows a performance of up to 472 GFLOPS (GPU FLoating point Operations Per Second) [65]. In comparison, the Odroid N2+ has six execution engines and shows a performance of approximately 163 GFLOPS [64,77]. In the case of ResNet-18, the difference in inference speed occurs as much as the difference in GFLOPS between Jetson Nano and Odroid N2+. In contrast, in the case of MobileNet-v2, our framework showed a faster inference speed than PyTorch. This is because, when the parameters of the model are small, the efficiency of in-place kernel optimization and asynchronous queue execution is relatively high, as well that as the parallel processing performance of OpenCL is maximized through the on-device optimizer.

**Table 8.** Comparison of inference times (ms) of PyTorch and our framework.

| DNN | PyTorch Jetson Nano with CUDA/cuDNN (472 GFLOPS) | Our Framework Odroid N2+ with OpenCL (163 GFLOPS) | | |
|---|---|---|---|---|
| | GPU | GPU | GPU (+MO) | GPU (+MO&RO) |
| ResNet-18 | 43 [1] | 203 | 141 | 129 |
| MobileNet-v2 | 51 [1] | 58 | 42 | 40 |

[1] It uses TorchVision pre-trained model: pytorch/vision:v0.10.0.

### 5.3. Comparison of Memory Usage for Inference

Table 9 shows a comparison of the memory usage during inference between OpenCL Caffe and the proposed framework. Memory usage was compared only in Odroid-N2+, in which the average memory usage and the maximum memory usage during inference were measured. In all networks, the memory usage during GPU-based inference was lower than that of OpenCL Caffe. For ResNet-101, in particular, the average memory usage decreased by approximately 52%, while the maximum memory usage decreased by approximately 24%. The memory object maintenance costs of OpenCL were minimized, as our framework supports the unified memory architecture of embedded systems. While OpenCL Caffe was designed to utilize an open-source-based GPU-accelerated engine, such as CLBlast, our framework provides a proprietary GPU-accelerated engine for embedded systems that minimizes the auxiliary memory that is required for OpenCL operations.

**Table 9.** Comparison of memory usage (MB) in Odroid N2+.

| System | DNN | Usage Type | OpenCL Caffe | | Our Framework | | Reduction Ratio (%) [1] |
|---|---|---|---|---|---|---|---|
| | | | CPU | GPU | CPU | GPU | |
| Odroid N2+ | VGG-16 | Average | 1164 | 1724 | 944 | 1288 | 25 |
| | | Max | 2334 | 2620 | 1842 | 2236 | 15 |
| | ResNet-18 | Average | 395 | 688 | 496 | 500 | 27 |
| | | Max | 395 | 707 | 602 | 657 | 7 |
| | ResNet-101 | Average | 807 | 1551 | 809 | 744 | 52 |
| | | Max | 807 | 1710 | 1116 | 1302 | 24 |
| | MobileNet-v2 | Average | 550 | 1007 | 468 | 831 | 17 |
| | | Max | 550 | 1226 | 685 | 939 | 23 |

[1] 1 − (GPU in our framework)/(GPU in OpenCL Caffe).

## 6. Conclusions and Discussion

In this study, we have proposed a deep learning framework that is specialized for embedded systems. The proposed framework provides a deep learning core, including data and layer structures, an OpenCL-based accelerator engine, and an on-device optimizer to improve the learning and inference performance on real devices, and a model converter for compatibility.

The deep learning core of our framework was designed for unified memory, thus minimizing the operation costs required for OpenCL data management by preventing unnecessary data transfer between the CPU and GPU. In addition, the accelerator engine provides the BLAS library, which is optimized for the embedded GPU, in-place kernel optimization that minimizes the repeated access cost of data, and asynchronous queue execution for maximizing the GPU utilization.

The on-device optimizer for inference provides a model optimizer for improving the inference performance by combining layers, an NDRange optimizer for searching for the optimal parallel processing space, and a quantization optimizer, which reduces the computation amount through INT8 quantization.

Finally, our framework provides a model converter for compatibility, enabling it to use models trained in conventional deep learning frameworks.

The performance evaluation shows that our framework compares favorably in terms of inference time and memory usage with OpenCL Caffe, which is an existing deep learning

framework. Furthermore, our framework was confirmed as being competitive against NNAPI, according to the performance evaluation results of the AI benchmark.

Currently, research on on-device learning, based on field data in embedded systems, is challenging. Our framework focuses on optimizing the inference performance of embedded systems, and provides a limited functionality for on-device learning. Therefore, we plan to study an acceleration engine and an on-device optimizer for on-device learning.

Also, our framework is optimized for the ARM CortexA CPU and Mali GPU in a Linux environment. Therefore, the framework will be expanded in the future to enable its use in other HW environments and operating systems that are used in various common mobile devices, such as Android or Qualcomm.

Finally, we plan to study a tuning algorithm that automatically analyzes the problems and potential of parallelism, based on the characteristics of the embedded system, and explore the optimization mechanism for this.

**Author Contributions:** Conceptualization, S.H.; methodology, S.H. and H.C.; software, S.H. and H.C.; validation, S.H.; formal analysis, S.H.; resources, S.H. and H.C.; writing—original draft preparation, S.H. and H.C.; writing—review and editing, S.H. and H.C.; supervision, J.-S.K.; project administration, J.-S.K.; All authors have read and agreed to the published version of the manuscript.

**Funding:** This work was supported by the Institute for Information & communications Technology Planning & Evaluation(IITP) grant funded by the Korean government(MSIT) (No.2017-0-00142, Development of Acceleration SW Platform Technology for On-device Intelligent Information Processing in Smart Devices).

**Institutional Review Board Statement:** Not applicable.

**Informed Consent Statement:** Not applicable.

**Data Availability Statement:** Not applicable.

**Conflicts of Interest:** The author declare no conflict of interest.

## Appendix A

This appendix describes the performance analysis and discussion of each technique of our framework to identify which block contributes the most towards the performance improvement. In DNN, inference time usually consists of the sum of the execution times of input, hidden, and output layers. The execution time of the input layer can be expressed as the sum of the preprocessing time ($T_{Pre}$) and the data transfer time to the GPU ($T_{CopyToGPU}$). The $T_{Pre}$ is a fixed value, but the $T_{CopyToGPU}$ can be minimized by utilizing data sharing, which is one of the features of unified memory (see Section 3.1.1).

The execution time of the hidden layer is calculated as the sum of the operation execution times of each hidden layer. Because the execution time of the hidden layer occupies most of the inference time, our framework focused on minimizing the execution time of the hidden layers. For this, we proposed an OpenCL-based accelerator engine. The accelerator engine can utilize existing CPU-based BLAS libraries, as well as use proprietary OpenCL-based BLAS and DNN acceleration libraries proposed by our framework. In addition, in order to maximize the efficiency of DNN acceleration in the hidden layer, the accelerator engine performs calculations only on the GPU. This is because the acceleration performance of OpenCL decreases if operation results are returned between devices when performing operations in the hidden layer. In addition, when accelerating the DNN operation based on OpenCL, not only the execution time of OpenCL kernels, but also the build and launch time of the kernels, are additionally required. Therefore, the accelerator engine has focused on minimizing the build and launch times as well as the execution time of the kernels (see Sections 3.1.1 and 3.2).

Table A1 shows the performance comparison with and without unified memory. For performance comparison, inferences were continuously performed on 100 images of each DNN. Overall latency includes the overall execution time, data preprocessing, and data

transfer time. Memory usage represents the average memory usage for inference over 100 images.

In terms of the inference time per one image, the acceleration performance improved by the unified memory is quite small. This is because, as described above, most of the inference time is occupied by the operation time of the hidden layers, and the data transmission time can be reduced only in the input/output layer. However, it is confirmed that performance can be improved in terms of overall latency when inference is continuously performed using changed input data. Therefore, when using an inference service in on-device, an accelerator engine that supports unified memory can be a great help in reducing overall latency.

In terms of memory usage, it was confirmed that performance gains can be obtained when unified memory is used. In particular, the larger the number of parameters and the greater the depth, the greater the memory usage when unified memory is not used. Therefore, it was confirmed that memory usage can be minimized through unified memory in the case of embedded systems with limited system resources, especially memory.

**Table A1.** Comparison of performance with and without unified memory.

| System | DNN | With Unified Memory | | Without Unified Memory | |
|---|---|---|---|---|---|
| | | Overall Latency (ms) | Memory Usage (MB) | Overall Latency (ms) | Memory Usage (MB) |
| Odroid N2+ | ResNet-18 | 13,273 | 500 | 14,081 | 507 |
| | ResNet-101 | 52,031 | 744 | 53,158 | 858 |
| | MobileNet-v2 | 4356 | 831 | 5199 | 855 |

## References


1. Redmon, J.; Divvala, S.; Girshick, R.; Farhadi, A. You only look once: Unified, real-time object detection. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR), Las Vegas, NV, USA, 27–30 June 2016; pp. 779–788.
2. Min, H.; Beanbonyka, R.; Hongchang, L.; Hyeonung, J.; Joonho, O.; Seongjun, C. Multi-class classification of lung diseases using CNN models. *Appl. Sci.* **2021**, *11*, 9289. [CrossRef]
3. Gihwi, K.; Ilyoung, C.; Qinglong, L.; Jaekyeong, K. A CNN-based advertisement recommendation through real-time user face recognition. *Appl. Sci.* **2021**, *11*, 9705.
4. Geunwoo, B.; Joonhyuk, Y. RT-SPeeDet: Real-time IP–CNN-based small pit defect detection for automatic film manufacturing inspection. *Appl. Sci.* **2021**, *11*, 9632.
5. Oh, Y.R.; Park, K.; Jeon, H.B.; Park, J.G. Designing acoustic scene classification models with CNN variants. *ETRI J.* **2020**, *42*, 761–772. [CrossRef]
6. Song, J.; Kim, J.; Choi, S.; Kim, J.; Kim, I. Evaluations of AI-based malicious PowerShell detection with feature optimizations. *ETRI J.* **2021**, *43*, 549–560. [CrossRef]
7. Yun, K.; Kwon, Y.; Oh, S.; Moon, J.; Park, J. Vision-based garbage dumping action detection for real-world surveillance. *ETRI J.* **2019**, *41*, 494–505. [CrossRef]
8. Yoon, Y.C.; Park, S.Y.; Park, S.M.; Lim, H. Image classification and captioning model considering a CAM-based disagreement loss. *ETRI J.* **2019**, *42*, 67–77. [CrossRef]
9. Yoo, S.B.; Han, M. Temporal matching prior network for vehicle license plate detection and recognition in videos. *ETRI J.* **2020**, *42*, 411–419. [CrossRef]
10. Kim, M.; Kim, S. Robust appearance feature learning using pixel-wise discrimination for visual tracking. *ETRI J.* **2019**, *41*, 483–493. [CrossRef]
11. Jeonghyun, L.; Sangkyun, L. Robust CNN compression framework for security-sensitive embedded systems. *Appl. Sci.* **2021**, *11*, 1093. [CrossRef]
12. Jinsoo, K.; Jeongho, C. Low-cost embedded system using convolutional neural networks-based spatiotemporal feature map for real-time human action recognition. *Appl. Sci.* **2021**, *11*, 4940. [CrossRef]
13. Sebastian, C.; Andrzej, C. Vehicle detection with self-training for adaptative video processing embedded platform. *Appl. Sci.* **2020**, *10*, 5763. [CrossRef]
14. Khan, W.; Daud, A.; Alotaibi, F.; Aljohani, N.; Arafat, S. Deep recurrent neural networks with word embeddings for Urdu named entity recognition. *ETRI J.* **2019**, *42*, 90–100. [CrossRef]

15. Ha, D.; Kim, M.; Moon, K.; Jeong, C.Y. Accelerating on-device learning with layer-wise processor selection method on unified memory. *Sensors* **2021**, *21*, 2364. [CrossRef] [PubMed]
16. PyTorch. Available online: https://pytorch.org/ (accessed on 23 August 2021).
17. TensorFlow. Available online: https://www.tensorflow.org/ (accessed on 23 August 2021).
18. Caffe. Available online: https://caffe.berkeleyvision.org/ (accessed on 23 August 2021).
19. NVIDIA Corporation. CUDA, Release: 11.4.1. 2021. Available online: https://developer.nvidia.com/cuda-toolkit (accessed on 23 August 2021).
20. NVIDIA Corporation. cuDNN, Release: 8.2.2. 2021. Available online: https://developer.nvidia.com/cudnn (accessed on 23 August 2021).
21. Khronos Group. OpenCL (Open Computing Language). Available online: https://www.khronos.org/opencl/ (accessed on 23 August 2021).
22. BLAS (Basic Linear Algebra Subprograms). Available online: http://www.netlib.org/blas/ (accessed on 23 August 2021).
23. Open Neural Network Exchange (ONNX). Available online: https://onnx.ai (accessed on 25 August 2021).
24. LAPACK (Linear Algebra PACKage). Available online: http://www.netlib.org/lapack/ (accessed on 23 August 2021).
25. Automatically Tuned Linear Algebra Software (ATLAS). Available online: http://math-atlas.sourceforge.net/ (accessed on 23 August 2021).
26. Xianyi, Z.; Kroeker, M. OpenBLAS: An Optimized BLAS Library. Available online: https://www.openblas.net/ (accessed on 23 August 2021).
27. NVIDIA Corporation. cuBLAS, Release: 11.4.1. 2021. Available online: https://developer.nvidia.com/cublas (accessed on 23 August 2021).
28. NVIDIA Corporation. CUTLASS: A Collection of CUDA C++ Template Abstractions for Implementing High-Performance Matrix-Multiplication (GEMM) at All Levels and Scales within CUDA. Available online: https://github.com/NVIDIA/cutlass (accessed on 23 August 2021).
29. Abdelfattah, A.; Keyes, D.; Ltaief, H. Kblas: An optimized library for dense matrix-vector multiplication on gpu accelerators. *ACM Trans. Math. Softw.* **2016**, *42*, 1–31. [CrossRef]
30. clBLAS. Available online: https://github.com/clMathLibraries/clBLAS (accessed on 23 August 2021).
31. ViennaCL. Available online: http://viennacl.sourceforge.net/ (accessed on 23 August 2021).
32. Nugteren, C. CLBlast: A Tuned OpenCL BLAS Library. In Proceedings of the International Workshop on OpenCL (IWOCL), Oxford, UK, 14–16 May 2018; pp. 1–10.
33. ARM Compute Library. Available online: https://github.com/arm-software/ComputeLibrary (accessed on 23 August 2021).
34. ARM NN. Available online: https://github.com/ARM-software/armnn (accessed on 23 August 2021).
35. Keras. Available online: https://keras.io/ (accessed on 23 August 2021).
36. Theano. Available online: https://pypi.org/project/Theano/ (accessed on 23 August 2021).
37. MXNet. Available online: https://mxnet.apache.org/versions/1.8.0/ (accessed on 23 August 2021).
38. Deeplearning4j. Available online: https://deeplearning4j.org/ (accessed on 23 August 2021).
39. Darknet: Open Source Neural Networks in C. Available online: https://pjreddie.com/darknet/ (accessed on 23 August 2021).
40. OpenCL Caffe. Available online: https://github.com/BVLC/caffe/tree/opencl (accessed on 23 August 2021).
41. DeepCL: Deep convolutional neural networks in OpenCL. Available online: http://deepcl.hughperkins.com/ (accessed on 23 August 2021).
42. TensorFlow Lite: ML for Mobile and Edge Devices. Available online: https://www.tensorflow.org/lite (accessed on 23 August 2021).
43. Sowa, P.; Izydorczyk, J. Darknet on OpenCL: A Multi-Platform Tool for Object Detection and Classification. Preprints 202007.0506.v1. 2020. Available online: https://www.preprints.org/manuscript/202007.0506/v1 (accessed on 23 August 2021).
44. EasyCL. Available online: https://github.com/hughperkins/EasyCL (accessed on 23 August 2021).
45. Android Neural Networks API. Available online: https://developer.android.google.cn/ndk/guides/neuralnetworks (accessed on 23 August 2021).
46. OpenCV. Available online: https://opencv.org/ (accessed on 23 August 2021).
47. LMDB (Lightning Memory-Mapped Database). Available online: https://symas.com/lmdb/ (accessed on 23 August 2021).
48. Wetzler, A. Add a Padding Layer to Caffe. Available online: https://github.com/twerdster/caffe (accessed on 23 August 2021).
49. Howard, A.G.; Zhu, M.; Chen, B.; Kalenichenko, D.; Wang, W.; Weyand, T.; Adam, H. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv* **2017**, arXiv:1704.04861.
50. Gondhalekar, A.; Feng, W.C. Exploring FPGA Optimizations in OpenCL for Breadth-First Search on Sparse Graph Datasets. In Proceedings of the 30th International Conference on Field-Programmable Logic and Applications (FPL), Virtual Conference, 31 August–4 September 2020; pp. 133–137.
51. Kim, S.; Oh, S.; Yi, Y. Minimizing GPU Kernel Launch Overhead in Deep Learning Inference on Mobile GPUs. In Proceedings of the 22nd International Workshop on Mobile Computing Systems and Applications (HotMobile), Virtual Conference, 24–26 February 2021; pp. 57–63.

52. Ioffe, S.; Szegedy, C. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In Proceedings of the International Conference on Machine Learning (ICML), Lille, France, 6–11 July 2015; pp. 448–456.
53. Merge Batch Normalization in Caffe. Available online: https://github.com/NHZlX/Merge_bn_Caffe (accessed on 23 August 2021).
54. Merge Convolution and Batchnorm Layers in both Caffe and PyTorch. Available online: https://github.com/zym1119/Merge_BN (accessed on 23 August 2021).
55. Kullback, S.; Leibler, R.A. On information and sufficiency. *Ann. Math. Stat.* **1951**, *22*, 79–86. [CrossRef]
56. Oh, C.; Park, G.; Kim, S.; Kim, D.; Yi, Y. Towards Real-time CNN Inference from a Video Stream on a Mobile GPU (WiP Paper). In Proceedings of the 21st ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems, London, UK, 15–20 June 2020; pp. 136–140.
57. Winograd, S. *Arithmetic Complexity of Computations*; Siam: Philadelphia, PA, USA, 1980; Volume 33.
58. Lavin, A.; Gray, S. Fast algorithms for convolutional neural networks. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR), Las Vegas, NV, USA, 27–30 June 2016; pp. 4013–4021.
59. Exploring the Arm Dot Product Instructions. Available online: https://community.arm.com/developer/tools-software/tools/b/tools-software-ides-blog/posts/exploring-the-arm-dot-product-instructions (accessed on 23 August 2021).
60. Lutz, R. Visualizer for Neural Networks: Netron. Available online: https://netron.app and https://github.com/lutzroeder/netron (accessed on 25 August 2021).
61. Google Inc. Protocol Buffer. Available online: https://developers.google.com/protocol-buffers (accessed on 25 August 2021).
62. Facebook Inc. Torch-script. Available online: https://pytorch.org/docs/stable/jit.html (accessed on 25 August 2021).
63. iEM-RK3399. Available online: http://www.falinux.com/product_SOM_1.html (accessed on 23 August 2021).
64. HARDKERNEL Corporation. ODROID-N2+ with 4GByte RAM. Available online: https://www.hardkernel.com/ko/shop/odroid-n2-with-4gbyte-ram-2/ (accessed on 23 August 2021).
65. NVIDIA Corporation. Jetson Nano. Available online: https://developer.nvidia.com/embedded/jetson-nano-developer-kit (accessed on 12 October 2021).
66. Ignatov, A.; Timofte, R.; Kulik, A.; Yang, S.; Wang, K.; Baum, F.; Van Gool, L. AI benchmark: All about deep learning on smartphones in 2019. *arXiv* **2019**, arXiv:1910.06663.
67. MobileNet-YOLO Caffe. Available online: https://github.com/eric612/MobileNet-YOLO (accessed on 23 August 2021).
68. A Caffe Implementation of EAST Text Detector. Available online: https://github.com/SURFZJY/EAST-caffe (accessed on 23 August 2021).
69. Convolutional Recurrent Neural Network (CRNN) in Caffe. Available online: https://github.com/yalecyu/crnn.caffe (accessed on 23 August 2021).
70. VGG Caffe. Available online: https://github.com/davidgengenbach/vgg-caffe (accessed on 23 August 2021).
71. Deng, J.; Dong, W.; Socher, R.; Li, L.J.; Li, K.; Fei-Fei, L. Imagenet: A large-scale hierarchical image database. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR), Miami, FL, USA, 20–25 June 2009; pp. 248–255.
72. ResNet-18 Caffemodel on ImageNet. Available online: https://github.com/HolmesShuan/ResNet-18-Caffemodel-on-ImageNet (accessed on 23 August 2021).
73. Deep Residual Networks. Available online: https://github.com/KaimingHe/deep-residual-networks (accessed on 23 August 2021).
74. MobileNet-Caffe. Available online: https://github.com/shicai/MobileNet-Caffe (accessed on 23 August 2021).
75. Everingham, M.; Van Gool, L.; Williams, C.K.; Winn, J.; Zisserman, A.L. The pascal visual object classes (voc) challenge. *Int. J. Comput. Vis.* **2010**, *88*, 303–338. [CrossRef]
76. AI Benchmark. Available online: https://ai-benchmark.com (accessed on 23 November 2021).
77. Wikipedia. Mali (GPU). Available online: https://en.wikipedia.org/wiki/Mali_(GPU) (accessed on 12 October 2021).