# C-GOOD: C-code Generation Framework for Optimized On-device Deep Learning

Duseok Kang
Seoul National University
kangds0829@snu.ac.kr

Euiseok Kim
Nalbi Inc.
ikess0330@gmail.com

Inpyo Bae
Seoul National University
inpyo@csap.snu.ac.kr

Bernhard Egger
Seoul National University
bernhard@csap.snu.ac.kr

Soonhoi Ha
Seoul National University
sha@snu.ac.kr

## ABSTRACT

Executing deep learning algorithms on mobile embedded devices is challenging because embedded devices usually have tight constraints on the computational power, memory size, and energy consumption while the resource requirements of deep learning algorithms achieving high accuracy continue to increase. Thus it is typical to use an energy-efficient accelerator such as mobile GPU, DSP array, and customized neural processor chip. Moreover, new deep learning algorithms that aim to balance accuracy, speed, and resource requirements are developed on a deep learning framework such as Caffe[16] and Tensorflow[1] that is assumed to run directly on the target hardware. However, embedded devices may not be able to run those frameworks directly due to hardware limitations or missing OS support. To overcome this difficulty, we develop a deep learning software framework that generates a C code that can be run on any devices. The framework is facilitated with various options for software optimization that can be performed according to the optimization methodology proposed in this paper. Another benefit is that it can generate various styles of C code, tailored for a specific compiler or the accelerator architecture. Experiments on three platforms, NVIDIA Jetson TX2[23], Odroid XU4[10], and SRP (Samsung Reconfigurable Processor)[32], demonstrate the potential of the proposed approach.

## CCS CONCEPTS

• **Computer systems organization** → *Embedded systems*;

## KEYWORDS

Deep Learning Framework, Software Optimization, On-device learning

## 1 INTRODUCTION

Deep learning is making significant progress in almost all areas of machine learning, including image classification, object detection, and so on. Extensive research efforts are being made to improve the accuracy of deep learning, paying a huge cost of computational, memory, and energy requirements. To apply such artificial intelligence to our daily life, it is necessary to make edge devices intelligent. The current practice to make a device intelligent is to use a cloud service accessed via a mobile network. There are several concerns with this practice of cloud-based intelligence such as privacy, dependence on the network condition, difficulty of personalization, and so on. As a consequence, on-device learning or inference has recently been drawing keen research attention with the goal to run deep learning algorithms directly on the device to relieve those concerns.

Executing deep learning algorithms on mobile embedded devices is challenging because embedded devices usually have tight constraints on the computational power, memory size, and energy consumption while the resource requirements of deep learning algorithms achieving high accuracy continue to increase. To cope with increasing computation complexity, it is common to use an energy-efficient accelerator such as mobile GPU and DSP array or to develop a customized neural processor chip. In this paper, we focus on software solutions that can be applied to existing embedded devices to optimize deep learning networks.

One approach is to develop new deep learning algorithms that seek to balance accuracy, speed, and resource requirements. Squeezenet[15], Tiny-YOLO[27], and MobileNet[13] are such examples. Another approach is to develop software optimization techniques that take advantage of the statistical nature of deep learning algorithms. Approximate computing can achieve almost the same accuracy with significantly reduced resource requirements and is used in various approximation techniques such as pruning[22], quantization[24], and low-rank approximation[18].

New algorithms are being developed every day, and many software optimization techniques exist. It is thus necessary to develop a systematic methodology to explore the wide design space of algorithm selection and software optimizations for a given hardware platform. On the other hand, many deep learning algorithms are developed on a deep learning (DL) framework such as Caffe[16], Torch[5], or Tensorflow[1] that is assumed to run directly on the target hardware. However, many embedded devices may not be able to run those frameworks due to hardware limitations or missing OS support.

To meet the requirements of hardware independence and systematic optimization methodology, we propose a novel deep learning framework, called C-GOOD (C-code Generation Framework for Optimized On-device DL), in this paper. C-GOOD is not an executable framework. For hardware independence, however, C-GOOD generates a C-code that can be compiled for and run on any embedded device. A side benefit of this approach is to generate various styles of C code or an OpenCL code, tailored for a specific compiler or the


*ICCAD '18, November 5–8, 2018, San Diego, CA, USA*

ACM ISBN 978-1-4503-5950-4/18/11…$15.00
https://doi.org/10.1145/3240765.3240786

accelerator architecture. Since C-GOOD is based on an existent C-based DL framework, Darknet[25], the training of a deep learning algorithm is assumed to be performed with Darknet. For inference, however, instead of running Darknet directly on the device, the generated code from C-GOOD is running on the device.

C-GOOD is also facilitated with various options for software optimization that can be performed according to the optimization methodology proposed in this paper. We explain how C-GOOD applies optimization techniques to various deep learning networks such as Darknet[26][1], Darknet19[26], Tiny-YOLO[27] and YOLOv2[28] on three different hardware platforms: the Jetson TX2[23], the Odroid XU4[10], and the Samsung Reconfigurable Processor (SRP)[32]. Experiments confirm that the proposed methodology is independent of the platforms and software optimizations can be performed easily with C-GOOD.

## 2 RELATED WORK

To reduce computational and memory requirements for inferencing on embedded devices, one approach is to develop a new neural network that requires less computational power and a smaller memory size while preserving comparable accuracy with the existing networks. The model size of SqueezeNet[15], for example, is less than 1 MB while its accuracy is comparable to Alexnet[20] for image classification. DenseNet[14] provides comparable accuracy to ResNet[11] with about a third of the parameters. For object detection, Tiny-YOLO[27] is a popular network for embedded devices: the model size is 61 MB at a performance of 22.15 FPS (frames per second) on Jetson TX2[23] for VOC object detection.

Several optimization techniques exist that reduce the resource requirements for a given neural network by applying approximate computing techniques. Pruning[22] is one of these techniques that removes connection links between the input and the output of a layer or filters composing convolution layers by zeroing out weight values that have a negligible effect on the accuracy. Pruning reduces the amount of computation and the memory space. Low-rank approximation[18] and quantization[24] are another two popular techniques in this category. While there are several optimization techniques available, no research has been performed on how to apply those techniques systematically to the best of our knowledge.

The LPIRC (Low-Power Image Recognition Challenge)[8] has recently been started to promote researches on on-device object detection. By setting score function related to power consumption and accuracy, it encourages researchers to explore all design spaces including hardware and software optimization. The winners of LPIRC 2016 [34] and LPIRC 2017 [17] applied several SW optimization techniques to a selected network on the state-of-the-art hardware platform, Fast-RCNN[9] on Jetson TX1 and Tiny-YOLO[27] on Jetson TX2 respectively. The former applied pipelining, changing the structure of the network, finding the best batch size, lowering the input resolution, using 16-bit quantization, and applying SVD (Singular Vector Decomposition). The latter applied pipelining, Tucker decomposition, 16-bit quantization and CPU multithreading. Without the aid of the DNN framework for software optimization, however, they performed design space exploration manually and selected a network first and applied several optimization techniques next.

[1]Darknet here is the name of the network developed by the developers who have also developed the Darknet framework.

DeepX[21] is a software accelerator for deep learning to compress fully-connected layers at runtime, which incurs run-time overhead. It is in contrast to the proposed technique where compression is performed statically during the design space exploration process. In addition, DeepX does not compress convolution layers.

There exist several DNN frameworks proposed for embedded platforms. Caffe2[31] is a deep learning framework that supports a variety of embedded platforms including iOS, Android, Raspbian, and Tegra. Another deep learning framework, Tensorflow lite[33], supports several embedded platforms, such as iOS and Android. Those frameworks allow users to implement and run deep learning networks using python and C++. Even though Caffe2 and Tensorflow lite is a lightweight and scalable deep learning framework that supports multiple platforms, it is not a general framework that can be applied to any embedded device. For example, those frameworks have an assumption that an OS (Operating System) is available on supported hardware platforms. However, there are embedded systems, which support no OS and only execute a binary file received. Also, using many libraries makes it difficult to use those platform for several embedded systems, because of the difficulty of porting those libraries to embedded systems.

There is an approach to generate C code for each embedded device to solve problems including supporting embedded platforms that do not support OS and porting libraries to embedded devices. Generating code for a specific embedded device and a specific deep learning network not only makes the codes easier to use for any embedded device but also improves performance by optimizing the network for the given device and network. Caffepresso[12] has chosen this approach. It is a Caffe-compatible framework that generates C-codes tailored for various accelerators such as FPGAs, DSPs, GPUs, and RISC multicores. In this respect, CaffePresso is similar to the proposed framework.

There are three differences between our proposed framework and Caffepresso. First, unlike CaffePresso, which uses a single type of a computing unit on a platform, C-GOOD can use any computing unit on a platform with heterogeneous computing units. Second, C-GOOD supports a variety of deep learning network optimization techniques as well as operation optimizations for a specific hardware platform, and makes it easy to apply those optimization techniques by leaving them as an option for users to select. For example, it supports low-rank approximation, quantization, merging batch normalization, and input size reduction. And it may generate a loop unrolled code to help the compiler to produce more efficient code on a specific platform. However, Caffepresso optimizes mappings and deep learning operations only for the given hardware, not providing options for network optimization of the deep learning algorithms. Lastly, C-GOOD provides a profile function at each layer. Based on the profiled information, C-GOOD evaluates various possible implementation options and selects the best one at the layer-level to optimize the deep learning network.

Darknet[25] is another neural network framework written in C with two optional dependencies, OpenCV to support various functions for image processing and CUDA to support NVIDIA GPU computation. To improve the efficiency and portability, three major extensions are made in C-GOOD. One is to make new function definitions or provide a different set of libraries if a library is not supported in the target platform. For the Odroid platform that does not support CUDA, for instance, C-GOOD provides and generates OpenCL kernels for GPU computation. The second extension is to generate target-specific C code. For the SRP platform, loop unrolling

techniques are applied in the code generation step to help the SRP compiler optimize the deep learning operations efficiently. The last extension is to optimize some operations for inferencing. When batch size is 1, for example, a reshape layer, which changes the shape of the feature map while merging two or more previous feature maps into one feature map, can be skipped by allocating the previous layers' output memory continuously. It also reduces memory allocation.

# 3 BACKGROUND

## 3.1 Deep Learning Networks

The proposed methodology is applied to two machine learning problems: image classification and object detection. In image classification, the probabilities of a single image frame belonging to a predetermined number of classes are computed. The problem is usually associated with a dataset such as EMNIST[4] for human handwriting recognition, CIFAR[19] or ImageNet[6] for image recognition. For this problem, deep learning has shown excellent performance compared to other machine learning algorithms with popular networks such as VGG[30], ResNet[11], and DenseNet[14]. Object detection is the problem of finding instances of real-world objects in an image, identifying the most relevant class of an object and additionally finding a bounding box that correctly contains the corresponding object. This problem is usually defined with a dataset such as VOC[7], or ImageNet[6]. The difference between the datasets lies in the types and number of objects: 20 classes for VOC and 200 for ImageNet. A typical objective function for image classification and object detection on embedded platforms is accuracy per watt. A popular measure of accuracy is *mAP* (mean Average Precision)[7] for object detection.

Deep neural networks consist of a cascade of multiple layers that perform feature extraction and transformation. Each layer uses the output from the previous layer as its input. Even though there are various types of deep neural networks, we focus on convolutional neural networks (CNN) in this paper since it is known that CNNs require less memory while giving good performance for image classification and object detection applications.

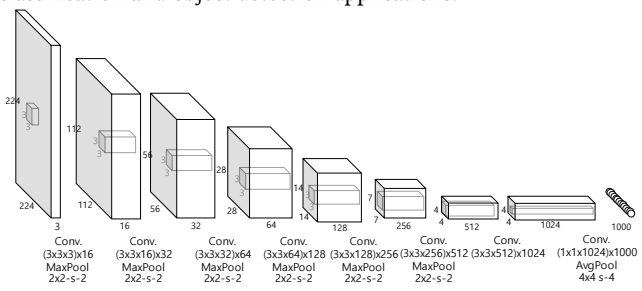


**Figure 1: Darknet Network**

Figure 1 illustrates the network structure of a CNN called *Darknet* that is selected as the DNN algorithm for image classification in this paper. Each layer consists of two main computation modules, convolution, and pooling. The convolution layers perform 3-D filtering of the input image group using 3-D filters represented by dashed boxes. The filter sizes and the number of filters are shown below each layer: $(3\times3\times3)\times16$ indicates *(filter size)* × *(the number of filters)*. The pooling layer uses sub-sampling to reduce the feature map size by taking the maximum value of the sub-sampled region (*max pooling*) or replacing the region with an average value (*average pooling*).

## 3.2 Software Optimization Techniques

In this section, we review two main software optimization techniques, low-rank approximation and quantization.

*3.2.1 Low-rank Approximation[18].* In a software implementation of a CNN, a 3-D convolution is computed by a matrix multiplication. Low-rank approximation aims to minimize a cost function by approximating a given matrix with another matrix with a lower rank. By reducing the rank of a matrix, we can reduce the required computational and memory resources. While this technique has been widely used for mathematical modeling and data compression, it has only recently been adopted for deep learning as an effective approximated computing method, often coupled with matrix decomposition techniques [18].

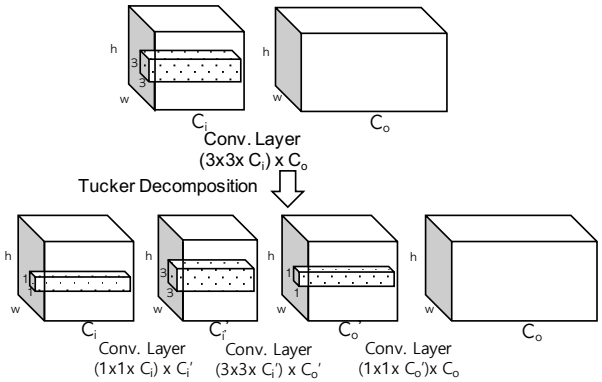


**Figure 2: Tucker Decomposition**

This work adopts the Tucker decomposition technique [18] illustrated in Figure 2. The top figure represents the original convolution that convolves the 3-D input matrix ($h\times w\times C_i$) and $C_o$ filter matrices of size $3\times3\times C_i$ to produce an output matrix ($h\times w\times C_o$). The number of multiplications involved in this convolution is $3\times3\times C_i\times C_o$ per pixel. As shown in the bottom figure, the weight matrix is decomposed into three[2] matrices with lower ranks and the original convolution is replaced with three smaller convolutions that are cascaded. The matrix size and the number of filters are depicted in the figure where $C_i'$ and $C_o'$ are smaller than $C_i$ and $C_o$ respectively. This decomposition reduces the total number of multiplications to $C_i\times C_i'+3\times3\times C_i'\times C_o'+C_o'\times C_o$. The memory requirements to save the filter matrices are also reduced by the same ratio. The key issue in this technique is to determine two variables, $C_i'$ and $C_o'$. Smaller values result in less time and space complexity at the expense of an increased computational error. In other words, the two parameters $C_i'$ and $C_o'$ have to be chosen judiciously, considering the tradeoff between the complexity and loss of accuracy.

*3.2.2 Quantization[24].* Quantization is a technique that reduces the bit width of data representation as much as possible while preserving the accuracy within a given bound. Quantization has been extensively researched for the design of hardware accelerators for DNNs since smaller bit widths lead to simpler computations as well as smaller memory requirements[24]. The key technique is to train the bit-reduced weights carefully in the training stage. A too large reduction can lead to significant degradation of accuracy; it

[2]If $C_i$ or $C_o$ is small, the weight matrix is decomposed into two matrices by merging the first or the last with the middle convolution, respectively.

remains an open problem what level of reduction can guarantee a certain accuracy for a given network.

While quantization is a very effective method to reduce the hardware complexity, its effectiveness is rather limited in software implementations. If the processor has a 32-bit multiplier, representing a variable with fewer bits does not lead to reduced computation complexity. However, if the processor supports simultaneous multiplication of two 16-bit values, the computation time can be halved by reducing the bit width of all variables to 16 bits. Additional benefits are to reduce the memory required to store the variables and to reduce the number of cache accesses and misses.

# 4 C-GOOD FRAMEWORK

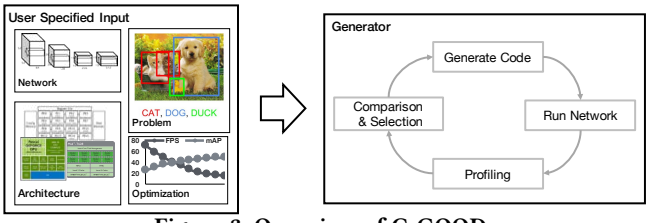


**Figure 3: Overview of C-GOOD**

Figure 3 shows the process of optimizing a deep learning network on a given embedded device in the C-GOOD framework. First, a user specifies a network configuration, a problem description, a platform description, and an optional optimization description. Based on this input information, C-GOOD generates a code, compiles the generated code, and performs layer-wise profiling during execution. Then the user determines which optimization technique will be used by comparing the execution times before and after a specific optimization technique is applied. Unless the user specifies the order of optimization techniques to be applied, a default order is followed in the proposed methodology. Finally, it generates an optimized C code for the given network and hardware platform.

## 4.1 Input Information

```
[net]

[convolutional]
name=conv1
batch_normalize = 1
filters = 16
size = 3
stride = 1
padding = 1
activation = leaky
bottom=input

[pool]
name=pool1
pool_type = MAX
size = 2
stride = 2
bottom=conv1
…
```

**Network Configuration**

```
[pipeline]

[tucker_decomposition]
Network=darknet-tucker.cfg
Weight=darknet-tucker.weights
Layerwise=1

[quantization]
Network=darknet-half.cfg
Weight=darknet.weight
Layerwise=1

[merge_batch_normalization]

[input_resolution]
Size=(256, 256) (224, 224) … (128,128)
```

**Optimization Description**

```
platform=x86
OpenCV=1
Thread=0
…
```

**Platform Description**

```
classes=200
test = validation.txt
problem = detector
dataset = imagenet
…
```

**Problem Description**

**Figure 4: An example of input information**

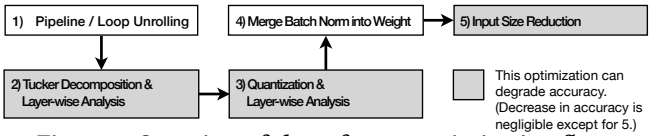


**Figure 5: Overview of the software optimization flow**

The input information of C-GOOD consists of one optional input file, an optimization description file, and three necessary input files for network configuration, problem description, and platform description. An example is shown in Fig. 4. Since it is assumed to use the Darknet framework for training a deep neural network, C-GOOD uses almost the same input file format as Darknet for network configuration and problem. However, the platform description file and the optimization description file are additional input files that Darknet does not use.

*4.1.1 Network Configuration.* The network configuration file represents how layers are composed in a deep learning network, and specifies the parameters of each layer such as kernel size, stride, padding, and a flag indicating whether or not batch normalization technique is used.

*4.1.2 Problem Description.* The problem description file contains information about the problem. It includes the name of a dataset such as ImageNet, VOC or Cifar, the type of the dataset such as Detection or Classification, and the path of a text file that contains the path of test images.

*4.1.3 Platform Description.* The platform description file describes the HW platform that the user wants to use. This file includes the architecture of the device, whether it supports thread programming and a specific library such as OpenCV and CUDA, and so on.

*4.1.4 Optimization Description.* The optimization description file is an optional file that selects the optimization techniques and specifies the order of application. The optimization techniques supported in the current implementation include loop unrolling, pipelining, Tucker composition, quantization, merging batch normalization into weights, and input size reduction. If the user does not provide this file, C-GOOD applies optimizations in the default order proposed in the optimization methodology depicted in Figure 5. In the example of Fig. 4, C-GOOD proceeds pipelining first and Tucker decomposition, quantization, merge batch normalization, and input resolution reduction in series.

## 4.2 Code Generation & Optimization Methodology

Figure 5 shows the default optimization procedure in the proposed software optimization methodology. For a given network, a baseline code without any optimization can be generated as shown in Fig. 6. After allocating a workspace, or memory space, for the network in *set_workspace()*, *inference()* is called. After inference for all images is completed, *wrap_up()* frees dynamically allocated data structures. In *inference()*, we first read an image in *frontend()*, perform inference through multiple layers in *forward_layers()*, and produce results in *backend()*. Now we explain how the code is modified after each optimization technique is applied.

*4.2.1 Pipelining.* For embedded devices with heterogeneous processing elements, pipelining can improve the throughput performance of the network. The network can be partitioned into

```
1 void main()
2     load_weight();
3     init_layers();
4     set_workspace();
5     inference();
6     wrap_up();
```
(a)

```
void inference()
    for each test image
        frontend(image);
        forward_layers();
        backend();
```
(b)

```
void forward_layers()
    conv1_forward();
    pool1_forward();
        ...
    conv8_forward();
    pool4_forward();
```
(c)

**Figure 6: Baseline C-code (pseudo code) that is platform-independent**

three stages: input preprocessing, network inference, and post-processing[3] of the last layer's output feature map. By making the hardware accelerator or GPU take the network inference stage and letting the CPU cores perform the other two stages, we can increase the throughput performance.

```
1  void inference()
2      front_thread_init();
3      for each test image
4          while(front_flag);
5          forward_layers();
6          front_flag= 1;
7          backend();
```
(a)

```
void frontend_thread()
    n = 0;
    while(n < NUM_IMAGE)
        while(!front_flag);
        read_inputs(input_buf);
        front_flag=0;
        n++;
```
(b)

```
1  void inference()
2      front_thread_init();
3      back_thread_init();
4      for each test image
5          while(front_flag);
6          forward_layers();
7          while(back_flag);
8          front_flag = 1;
9          back_flag=1;
```
(c)

```
void backend_thread()
    n = 0;
    while(n < NUM_IMAGE)
        while(!back_flag);
        processing_outputs();
        back_flag=0;
        n++;
```
(d)

**Figure 7: Code modification by applying pipeline**

To apply pipeline technique to the front-end, we make *frontend_thread* and some changes in *inference*. Figure 7 (a) shows how the code is changed in *inference()* to apply pipelining at the front end, which is highlighted by red texts. A separate thread, called *fronend_thread*, is initialized and run in *frontend_thread_init()*. Two concurrent threads are synchronized by an array of shared flags. A similar modification is made for back-end pipelining as illustrated in Figure 7 (c) and (d).

*4.2.2 Loop Unrolling[2].* GEMM (General Matrix Multiplication), which is a popular function in convolution computation, consists of a 3-nested loop inside. In some architectures, loop unrolling is an effective way to increase the utilization of processing elements for effective parallel processing. Figure 8 shows how the loop body of *gemm_nn()* can be unrolled when the unrolling factor of each loop is set to 2 and the total unrolling factor is 8. When unrolling factors are set, unrolling code is generated through three processes: making as many temporary variables as the total unrolling factor to store the partial result, generating the unrolled loop body with temporary variables, and adding extra code if the number of iterations is not a multiple of unrolling factor.

*4.2.3 Tucker Decomposition.* As a low-rank approximation technique, we use Tucker decomposition as explained in Section 3.2.1. Since the Tucker decomposition transforms a convolution layer into a cascade of two or three convolutional layers, it is necessary to change the network configuration input file but without code change. Since it is reported that Tucker decomposition is not beneficial at all times [17], C-GOOD provides a performance profiling

[3]In the final output feature map, each proposed answer is encoded as a tensor. The post-processing includes decoding of such data and selecting the final answers.

```
1 void gemm_nn (float *A, float *B, float *C)
2   for (i=0; i<I; i++)
3       for (j=0; j<J; j++)
4           for (k=0; k<K; k++)
5               C[J*i + j] += A[K*i + k] * B[J*k + j];
```

```
1  void unrolled_gemm_nn (float *A, float *B, float *C)
2    float c0, …, c7;
3    for (i=0; i<I-1; i+=2)
4        for (j=0; j<J-1; j+=2)
5            for (k=0; k<K-1; k+=2)
6                c0 = A[K*(i+0) + (k+0)] * B[J*(k+0) + (j+0)];
7                c1 = A[K*(i+0) + (k+1)] * B[J*(k+1) + (j+0)];
8                c2 = A[K*(i+0) + (k+0)] * B[J*(k+0) + (j+1)];
9                c3 = A[K*(i+0) + (k+1)] * B[J*(k+1) + (j+1)];
10               c4 = A[K*(i+1) + (k+0)] * B[J*(k+0) + (j+0)];
11               c5 = A[K*(i+1) + (k+1)] * B[J*(k+1) + (j+0)];
12               c6 = A[K*(i+1) + (k+0)] * B[J*(k+0) + (j+1)];
13               c7 = A[K*(i+1) + (k+1)] * B[J*(k+1) + (j+1)];
14               c0 += c1; c2 += c3; c4 += c5; c6 += c7;
15               C[J*(i+0) + (j+0)] += c0;
16               C[J*(i+0) + (j+1)] += c2;
17               C[J*(i+1) + (j+0)] += c4;
18               C[J*(i+1) + (j+1)] += c6;
```

**Figure 8: Unrolled GEMM Code**

function at each layer. In the current implementation, the user is supposed to apply Tucker decomposition repeatedly with a set of candidate dimension values of decomposed matrices ($C_i'$ and $C_o'$ in Figure 2) and find the best dimension values among the candidates for each layer. It will be a future work to automate this process to find the best values of $C_i'$ and $C_o'$.

*4.2.4 Quantization.* C-GOOD gives users the option to use 16-bit data instead of 32-bit data, which is denoted as quantization optimization. Since quantization does not improve performance in all cases like Tucker decomposition case, C-GOOD performs layer-wise profiling before and after quantization is applied to decide in which layers quantization is beneficial.

```
1  void
2  forward_layers()
3      conv1_forward();
4      pool1_forward();
5          ...
6      conv8_forward();
7      pool4_forward();
8
```
(a)

```
void forward_layers()
    f2h(input_data);
    conv1_forward();
    pool1_forward();
        ...
    conv8_forward();
    pool4_forward();
    h2f(pool4_data);
```
(b)

```
void forward_layers()
    conv1_forward();
    pool1_forward();
    f2h(pool1_data);
        ...
    conv8_forward();
    pool4_forward();
    h2f(pool4_data);
```
(c)

Half Type

**Figure 9: Code changes for applying quantization and layer-wise quantization (f2h is a function that converts float data to half and h2f do vise versa.)**

Figure 9 (b) and (c) shows two examples that apply quantization to different regions of layers. For 16-bit quantization, we need to convert the input data to 16-bit before the first convolution layer of the region and convert the result data to 32-bit data after the last convolution layer of the region. Note that we use different libraries or function definitions for 16-bit data at the lower level, which is not shown in the figure.

*4.2.5 Merge Batch normalization.* Batch normalization is a technique to normalize the input data to have unit variance and zero mean. It is known that normalization stabilizes the learning process and accelerates learning. Since batch normalization is a linear transformation, it can be merged with the previous convolution layer by adjusting the filter weights at compile-time. It reduces the overall network execution time.

*4.2.6 Input Resolution Reduction.* Since the computational complexity and the memory space for activation data are proportional to the image size, it is advantageous to reduce the input image size as much as possible while accuracy loss is tolerable. If the objective function is defined as accuracy per power consumption, reducing the image size is advantageous. If the user specifies the input size, the input image size is adjusted accordingly by a front-end thread that performs image resizing.

## 5 EXPERIMENTS

We evaluate the effectiveness and viability of C-GOOD with three widely different hardware platforms: the Jetson TX2[23], the Odroid XU4[10], and the SRP platform[32]. Jetson TX2 is an embedded AI computing device made by NVIDIA. It consists of a quad-core ARM A57 CPU, a dual NVIDIA Denver core, and a Pascal GPU that includes 256 CUDA cores. It is currently the most powerful embedded device supported by the cuDNN libraries. Odroid XU4 consists of a quad-core ARM A15, a quad-core ARM A7, and a Mali-T628 MP6 GPU. Since the platform does not support CUDA, the proposed framework includes an OpenCL-backend to use the Mali GPU.

The SRP is a coarse-grained reconfigurable array processor that has 16 heterogeneous processing elements and supports 32bit floating point calculations with a peak performance of 4GFlops. It contains a 320KB on-chip data SRAM. Since the array processor can be configured as a single VLIW processor for processing of sequential programs, it can run the full DNN algorithm. However, the SRP does not run an operating system, and there is no support for libraries. Thus we need to generate a standalone C-code that can be compiled for the array processor by the SRP compiler. Note that this platform cannot run any existent DNN platform such as Caffe2[31], Tensorflow lite[33], and even Darknet[25].

### 5.1 Performance Comparison

Since the Darknet framework can be run directly on Jetson TX2, we first compare the performance between the Darknet framework and the code generated by C-GOOD. For comparison, we use three neural networks supported by the Darknet framework: YOLOv2[28] for object detection and the Darknet19[26] and DenseNet[14] for image classification, both with the ImageNet dataset. Table 1 displays the speed comparison result and Table 2 compares the memory usage of two frameworks.

**Table 1: Speed comparison with Darknet (Unit : FPS)**

| Problem | Networks | Darknet | Ours | Piped[4] |
|---|---|---|---|---|
| ImageNet DET | YOLOv2[28] | 5.81 | 6.39 | 7.11 |
| ImageNet CLS | Darknet19[26] | 21.26 | 23.37 | 32.44 |
| ImageNet CLS | DenseNet201[14] | 12.39 | 17.45 | 21.03 |

Darknet reads multiple images at once by multithreading for object detection, which is similar to pipelining in its effect on the performance while no such technique is used for image classification. The table shows both the FPS measured without pipelining and the FPS with front-end pipelining only in C-GOOD. For object detection, the **Piped** code generated by C-GOOD should be compared with Darknet for a fair comparison. It is observed that C-GOOD shows 22.4% higher performance. For image classification, C-GOOD is 9.9% faster than Darknet without pipelining for Darknet19 and 40.8% faster for DenseNet201. If front-end pipelining is applied, the performance improvement is increased to 52.6% and 69.7%, respectively. Because both approaches use cuDNN for convolution layers, the gain is mainly attributed to network-specific code generation and skipping operations for the reshape layer which changes the shape of the feature map. When concatenating feature maps from two or more previous layers, the code generated by C-GOOD allocates memory more efficiently.

[4]For a fair comparison, we applied pipelining only to the front-end part that reads and resizes images.

**Table 2: Memory usage comparison with Darknet (Unit : MB)**

| Problem | Networks | Darknet | Ours | Piped[3] |
|---|---|---|---|---|
| ImageNet DET | YOLOv2[28] | 861 | 761 | 777 |
| ImageNet CLS | Darknet19[26] | 655 | 649 | 675 |
| ImageNet CLS | DenseNet201[14] | 734 | 657 | 687 |

Memory footprint is an important metric for an embedded device that has a limited size of memory. Table 2 displays the maximum RSS(Resident Set Size) value of the process, measured by using *ps* command. For object detection, the **Piped** code generated by C-GOOD uses 84MB less memory than Darknet. This gain comes from two reasons. First, C-GOOD optimizes the memory usage for the reshape layer. Second, there is a difference in pipelining implementation between C-GOOD and Darknet: darknet uses four threads to read input, but C-GOOD uses two threads. It is observed that Darknet19 for image classification shows a similar size of memory for both C-GOOD and Darknet since Darknet19 does not include a reshape layer nor does it contain multithreading for reading input. In case of DenseNet201 that include 98 reshape layers, C-GOOD uses 77MB less memory than Darknet. Even with pipelining, it uses 47MB less memory.

**Table 3: Memory usage and speed when using Caffe2**

| Problem | Networks | Speed (Unit:FPS) | Memory (Unit:MB) |
|---|---|---|---|
| ImageNet CLS | DenseNet121[14] | 13.89 | 1061 |
| ImageNet CLS | DenseNet201[14] | 7.69 | 1459 |

Caffe2[31] was recently introduced as a deep learning framework for embedded systems and is becoming popular. Table 3 shows the memory usage and speed for the Caffe2 framework with two networks, DesneNet201 and DenseNet121, on the same Jetson TX2 board. DenseNet121 model is obtained from the Caffe2 official github[3] and DenseNet201 is manually converted from the Caffe model[29]. For DenseNet201, it is observed from Table 1 that C-GOOD performs more than twice better than Caffe2 in terms of speed. Because YOLOv2 and Darknet19 have a layer not supported by Caffe2, comparison with those networks could not be made.

### 5.2 Jetson TX2

**Table 4: Applied optimization techniques**

| No. | Technique | No. | Technique |
|---|---|---|---|
| (1) | Frontend Pipelining | (5) | Quantization |
| (2) | Backend Pipelining | (6) | Layer-wise Quantization |
| (3) | Tucker Decomposition | (7) | Merge BatchNorm |
| (4) | Layer-wise Tucker | (8) | Loop Unrolling |

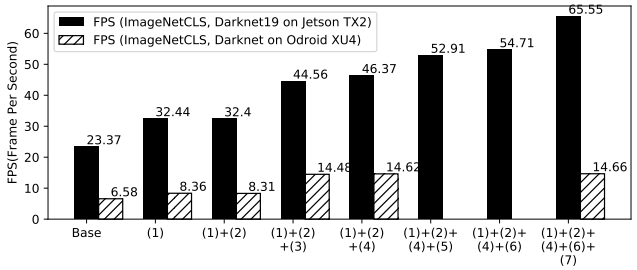


**Figure 10: Performance change with application of optimization methods for image classification**

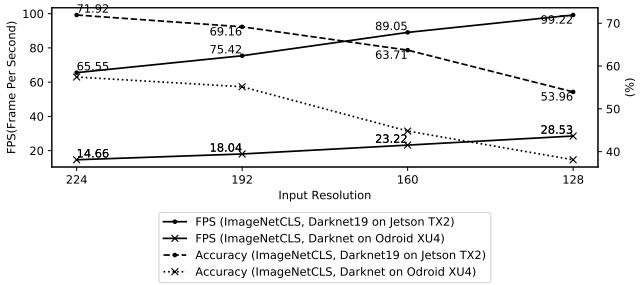


**Figure 11: Performance and accuracy change with decreasing input's resolution for image classification**

The next set of experiments is conducted by applying the proposed software optimization procedure, using C-GOOD, to a Jetson TX2 platform. Figure 10 shows how the computation speed improves as the optimization process is applied to the Darknet19 network for image classification. In the figure, the horizontal axis indicates the optimization methods added from the origin and the vertical axis indicates the FPS performance. Each number represents an optimization technique shown in Table 4. As an optimization method is added, the performance is improved, and after all optimizations are applied, the performance reaches a 2.8 × improvement compared to the baseline implementation that is identical to that shown in Table 1.

It can be observed that the Tucker decomposition yields the largest improvement and layer-wise exploration produces additional gains for both Tucker decomposition and quantization. There is no gain with backend pipelining because the workload of post-processing is negligible. Figure 11 shows the trade-off between accuracy and speed as the input image size varies. The speed increases almost linearly with the input size while the accuracy drops rather abruptly when the input size decreases too much.

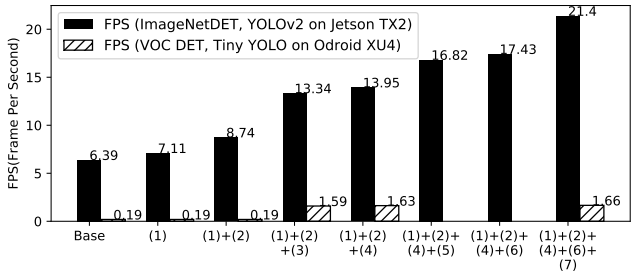


**Figure 12: Performance change with application of optimization methods for object detection**

A similar experiment is conducted with YOLOv2 for object detection. As shown in Figure 12, we were able to achieve significant

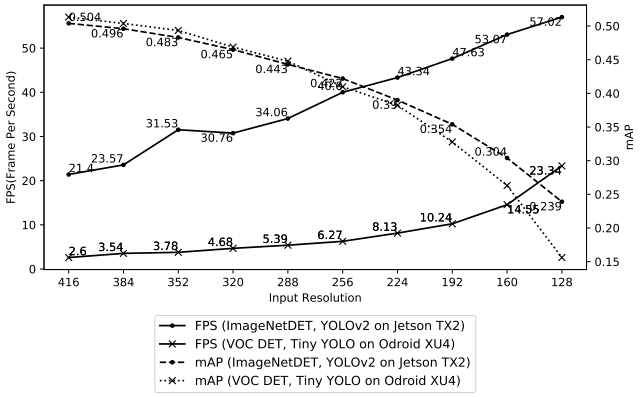


**Figure 13: Performance and accuracy change with decreasing input's resolution for object detection**

performance improvements by 3.35 times compared with the baseline implementation. Figure 13 shows the trade-off between speed and accuracy as the input size is changed. We observe an outlier consistently when the input size decreases from 384 to 352, which needs further investigation in the future.

## 5.3 Odroid XU4

For Odroid XU4 that does not support CUDA, C-GOOD generates an OpenCL code, based on our own OpenCL kernels for the DNN layers. [5] Since the Odroid XU4 platform has much lower computation power than Jetson TX2, experiments are conducted with smaller networks: *Darknet* with the ImageNet dataset for image classification and *Tiny-YOLO* with the VOC dataset for object detection. The optimization flow presented in Section 4.2 is applied except quantization since the MALI-T628 GPU does not support 16-bit half-precision computations. Experimental results are displayed in the same figures as those of the Jetson TX2 experiments from Figure 10 to Figure 13. Software optimization results in a 2.2× performance improvement for Darknet and a 8.7× improvement for Tiny-YOLO. Unlike the Jetson TX2 case, there is no performance gain achieved with pipelining since the CPU workload is negligible compared with the GPU workload for inferencing. The steep accuracy drop is observed at a larger input size than the Jetson TX2 case for image classification, but no difference is observed for object detection.

## 5.4 The Samsung Reconfigurable Processor

Since the SRP processor has the limited computing power and on-chip SRAM size, we applied our methodology to Darknet[26] only for image classification. From the original Darknet network, three max-pooling layers are removed since the dataset is changed to CIFAR that assumes a smaller image size than ImageNet. Experiments are conducted on a cycle-accurate simulator instead of a real device since access to a development board was not available. Note that there exists no DNN framework that supports the SRP processor except the proposed framework.

Figure 14 shows performance improvements on the SRP when our methodology is applied. Because SRP does not support thread and 16bit data type, we cannot applied the optimization techniques using threads and quantization technique. The hardware-specific optimization step is applied as the first step to generate

[5] Since the OpenCL kernels are not fully optimized, there is still room for improvement.

loop-unrolled C-code to ease compiler-optimization. These loop-unrolling compiler optimizations achieve a 10.93× performance improvement. Similarly to the other platforms, the speed of inference is improved by more than a factor of two by the Tucker decomposition in all layers, while the Merge-BN optimization does not raise performance noticeably. Experimental results for varying input sizes are shown in Figure 15. Again, we have to find a good compromise between accuracy and performance (or power) for the given objective function.

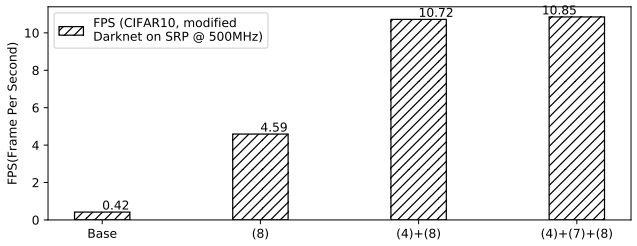


**Figure 14: Performance change with application of optimization methods for DarkNet (SRP)**

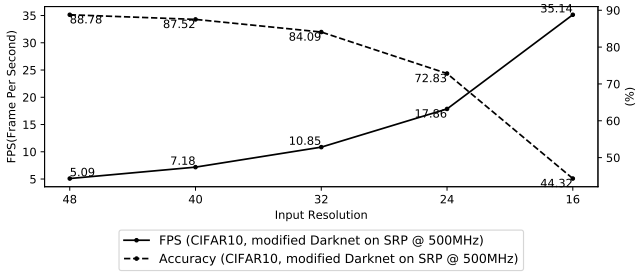


**Figure 15: Performance and accuracy change with decreasing input's resolution (SRP)**

## 6 CONCLUSION

In this paper, we propose a novel DNN framework, called C-GOOD, that generates a C code that can be run on any embedded platforms. It is an optimization-aware framework in which a user can specify which optimization techniques will be applied to a given network. Then the optimized C-code is automatically generated, which helps the user to explore the design space of network selection and software optimization. A software optimization methodology is also introduced to apply optimization techniques systematically: pipelining, layer-wise Tucker decomposition, layer-wise quantization, merging batch normalization into weights, and input size reduction.

The proposed methodology has been applied to three different hardware platforms: the Jetson TX2, the Odroid XU4, and the Samsung Reconfigurable Processor. Experimental results show that the baseline C code generated from C-GOOD is better than the Darknet framework in terms of fps performance. In addition, the proposed optimization methodology improves the performance by 2.2 to 25.83 times over the baseline unoptimized code for a given hardware platform.

## ACKNOWLEDGMENTS

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

## REFERENCES

[1] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, et al. 2016. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467* (2016).
[2] I. Bae, Barend Harris, H. Min, and Bernhard Egger. 2018. Auto-Tuning CNNs for Coarse-Grained Reconfigurable Array-based Accelerators. In *CASES, 2018 International Conference on*. IEEE.
[3] caffe2. 2017-2018. models. https://github.com/caffe2/models. (2017-2018).
[4] G. Cohen, S. Afshar, J. Tapson, and A. van Schaik. 2017. EMNIST: Extending MNIST to handwritten letters. In *Proceedings of IJCNN, IEEE*. 2921–2926.
[5] R. Collobert, K. Kavukcuoglu, and C. Farabet. 2011. Torch7: A Matlab-like Environment for Machine Learning. In *Proceedings of NIPS Workshop*.
[6] J. Deng, W. Dong, R. Socher, L. J. Li, K. Li, and L. Fei-Fei. 2009. Imagenet: A large-scale hierarchical image database. In *Proceedings of CVPR, IEEE*. 248–255.
[7] M. Everingham, L. Van Gool, C. K. Williams, J. Winn, and A. Zisserman. 2010. The pascal visual object classes (voc) challenge. *IJCV* 88, 2 (2010), 303–338.
[8] K. Gauen, R. Rangan, A. Mohan, Y. H. Lu, W. Liu, and A. C. Berg. 2017. Low-power image recognition challenge. In *Proceedings of ASP-DAC, IEEE*. 99–104.
[9] R. Girshick. 2015. Fast r-cnn. In *Proceedings of ICCV, IEEE*. 1440–1448.
[10] hardkernel. 2017. ODROUD-XU4. http://www.hardkernel.com/main/products/prdt_info.php. (2017).
[11] K. He, X. Zhang, S. Ren, and J. Sun. 2016. Deep residual learning for image recognition. In *Proceedings of CVPR, IEEE*. 770–778.
[12] G. Hegde, N. Ramasamy, N. Kapre, et al. 2016. CaffePresso: an optimized library for deep learning on embedded accelerator-based platforms. In *CASES, 2016 International Conference on*. IEEE, 1–10.
[13] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam. 2017. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861* (2017).
[14] G. Huang, Z. Liu, K. Q. Weinberger, and L. van der Maaten. 2016. Densely connected convolutional networks. *arXiv preprint arXiv:1608.06993* (2016).
[15] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer. 2016. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and< 0.5 MB model size. *arXiv preprint arXiv:1602.07360* (2016).
[16] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell. 2014. Caffe: Convolutional Architecture for Fast Feature Embedding. *arXiv preprint arXiv:1408.5093* (2014).
[17] D. Kang, J. Kang, D. Kang, S. Yoo, and S. Ha. 2018. Joint Optimization of Speed, Accuracy, and Energy for Embedded Image Recognition Systems. In *Proceedings of DATE, IEEE*.
[18] Y. D. Kim, E. Park, S. Yoo, T. Choi, L. Yang, and D. Shin. 2015. Compression of deep convolutional neural networks for fast and low power mobile applications. *arXiv preprint arXiv:1511.06530* (2015).
[19] A. Krizhevsky. 2012. Learning Multiple Layers of Features from Tiny Images. (05 2012).
[20] A. Krizhevsky, I. Sutskever, and G. E. Hinton. 2012. Imagenet classification with deep convolutional neural networks. In *Proceedings of NIPS*. 1097–1105.
[21] N. D. Lane, S. Bhattacharya, P. Georgiev, C. Forlivesi, L. Jiao, L. Qendro, and F. Kawsar. 2016. DeepX: A Software Accelerator for Low-Power Deep Learning Inference on Mobile Devices. In *Proceedings of IPSN, ACM/IEEE*. 1–12. https://doi.org/10.1109/IPSN.2016.7460664
[22] H. Li, A. Kadav, I. Durdanovic, H. Samet, and H. P. Graf. 2016. Pruning filters for efficient convnets. *arXiv preprint arXiv:1608.08710* (2016).
[23] Nvidia. 2017. NVIDIA JETSON. http://www.nvidia.com/object/embedded-systems-dev-kits-modules.html. (2017).
[24] E. Park, J. Ahn, and S. Yoo. 2017. Weighted-Entropy-Based Quantization for Deep Neural Networks. In *Proceedings of CVPR, IEEE*. 5456–5464.
[25] J. Redmon. 2013–2016. Darknet: Open Source Neural Networks in C. http://pjreddie.com/darknet/. (2013–2016).
[26] J. Redmon. 2013–2016. ImageNet Classification. https://pjreddie.com/darknet/imagenet/. (2013–2016).
[27] J. Redmon. 2013–2016. YOLO: Real-Time Object Detection. http://pjreddie.com/darknet/yolo/. (2013–2016).
[28] J. Redmon and A. Farhadi. 2016. YOLO9000: Better, Faster, Stronger. *arXiv preprint arXiv:1612.08242* (2016).
[29] shicai. 2017. DenseNet-Caffe. https://github.com/shicai/DenseNet-Caffe. (2017).
[30] K. Simonyan and A. Zisserman. 2014. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556* (2014).
[31] Facebook Open Source. 2017–2018. Caffe2. https://caffe2.ai. (2017–2018).
[32] D. Suh, K. Kwon, S. Kim, S. Ryu, and J. Kim. 2012. Design space exploration and implementation of a high performance and low area coarse grained reconfigurable processor. In *Proceedings of ICFPFT*. IEEE, 67–70.
[33] Tensorflow. 2017–2018. Tensorflow lite. https://www.tensorflow.org/mobile/tflite/. (2017–2018).
[34] C. Wang, Y. Wang, Y. Han, L. Song, Z. Quan, J. Li, and X. Li. 2017. CNN-based object detection solutions for embedded heterogeneous multicore SoCs. In *Proceedings of ASP-DAC, IEEE*. 105–110.