# An Untimed SystemC Model of GoogLeNet

Emad Malekzadeh Arasteh and Rainer Dömer

Center for Embedded and Cyber-Physical Systems, University of California Irvine
{emalekza,doemer}@uci.edu

**Abstract.** Deep learning and convolutional neural network (CNN) have been shown to solve image classification problems fast and with high accuracy. However, these algorithms tend to be very computationally intensive and resource hungry, hence making them difficult to use on embedded devices. Towards this end, we need system-level models for analysis and simulation. In this report, we describe a newly designed untimed SystemC model of GoogLeNet, a state-of-the-art deep CNN using OpenCV library. The SystemC model is automatically created from a Caffe model using a generator tool. We successfully validate the functionality of the model using Accellera SystemC 2.3.1 simulator. Then, we use RISC (Recoding Infrastructure for SystemC) to speed up the simulation by exploiting thread-level parallelism and report extensive experimental results.

**Keywords:** System-level modeling · Parallel discrete event simulation · Deep learning · Convolutional neural network · SystemC

## 1 Introduction

Computer vision (CV) as a scientific field aims to gain understanding of images and video. CV covers a wide range of tasks, such as object recognition, scene understanding, human motion recognition, etc. One of the core problems in visual recognition is image classification. Image classification is the problem of assigning a descriptive label to an input image from a fixed set of categories. Deep learning and convolutional neural network (CNN) have been shown to solve this hard image classification problem fast and with acceptable precision.

Early work on CNN dates back to 1989 with the LeNet network for handwritten digit recognition [7]. However, the early 2010s started a new era for CNN applications by the introduction of AlexNet [5] for image classification. Growth of computing power, availability of huge datasets that can be used for training, and rapid innovation in deep learning architectures have paved the way for the success of deep learning techniques in recent years [11].

A CNN mainly consists of alternating convolution layers and pooling (subsampling) layers. Each convolution layer extracts features in the input by applying trainable filters to the input. Later, the convolved feature is fed to an activation function, for example a Rectifier Linear Unit (ReLU) to introduce nonlinearity and obtain activation maps. Each pooling layer downsamples the activation maps to reduce computation and memory usage in the network. Features extracted from previous convolution and pooling layers are fed to a fully connected layer to perform classification. Typically, a softmax activation function can be placed following the final fully connected layer to output the probability corresponding to each classification label. For example, LeNet-5, a CNN for digit recognition, as depicted in Figure 1, contains three convolution layers, two sub-sampling layers, and one fully connected layer [8].

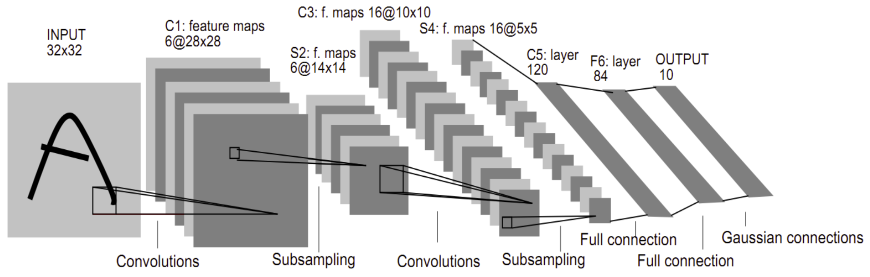

**Fig. 1.** Architecture of LeNet-5, a CNN for digits recognition [8]

In this paper, we develop an untimed SystemC model of GoogLeNet [12], a state-of-the-art deep CNN. Following the top-down specification approach for a classical system on chip design [1][2], our goal is to separate communication parts from computation parts. To achieve this, we exploit the fact that a neural network is a directed graph where the nodes are different layers in the network and edges connect neighboring layers.

Latest trends in cutting edge deep neural network architectures like ResNeXt (2016) [13], FractalNet (2016) [6], DenseNet [3] (2017), etc. show a substantial increase in the number of multiple parallel connections between layers in the network. This comes with a high level of thread-level parallelism, which parallel simulators can take advantage of for faster simulations.

The rest of this paper is organized as follows: Section 2 describes high level structure of GoogLeNet. Section 3 describes SystemC modeling details of each layer and the overall GoogLeNet model. Section 4 presents sequential and parallel simulation results with an analysis of valuable observations. At last, Section 5 concludes this case study.

## 2 GoogLeNet Structure

GoogLeNet is a deep CNN for image classification and detection that was the winner of the ImageNet Large Scale Recognition Competition (ILSVRC) in 2014 with only 6.67% top-5 error [12]. GoogLeNet was proposed and designed with computational efficiency and deployability in mind. The two main features of

GoogLeNet are (1) using 1x1 convolution layer for dimension reduction and (2) applying network-in-network architecture to increase representational power of the neural network [12].

GoogLeNet is 22 layers deep when only layers with learned parameters are considered. The total number of layers (independent building blocks) is 142 distinct layers. The main constituent layer types are convolution, pooling, concatenation, and classifier. GoogLeNet includes two auxiliary classifiers that are used during training to combat the so-called vanishing gradient problem. The detailed types of layers inside GoogLeNet and the number of each type of layers are summarized in Table 1.

**Table 1.** GoogLeNet layer summary

| Layer type | Count |
|---|---|
| Convolution | 57 |
| ReLU | 57 |
| Pooling | 14 |
| LRN | 2 |
| Concat | 9 |
| Dropout | 1 |
| InnerProduct | 1 |
| Softmax | 1 |
| Total | 142 |

Our focus for now is on inference by using the proposed neural network architecture, and not the training for fine-tuning network parameters or suggesting improved network architecture. Therefore, our model does not include the two auxiliary classifier layers.

A schematic view of GoogLeNet is depicted in Figure 2. An image is fed in on the left, and processed by all layers. Then, a vector with probabilities for the set of categories comes out on the right. The index of a class with a maximum probability is looked up in a table of synonym words that outputs the class of the object in the image, i.e. “space shuttle”.

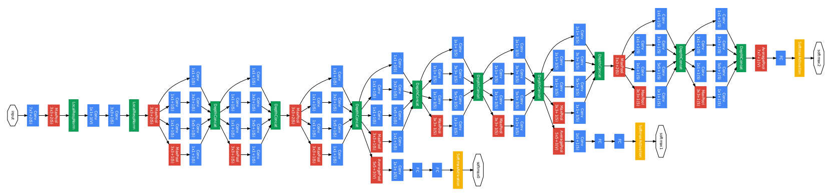

**Fig. 2.** GoogLeNet network with all the bells and whistles [12]

To get pre-trained network parameters, we have used the Caffe (Convolutional Architecture for Fast Feature Embedding) model zoo. Caffe is a deep learning framework originally developed at University of California, Berkeley, and is available under BSD license [4]. The GoogLeNet Caffe model comes with (1) a binary file `.caffemodel` that contains network parameters, and (2) a text file `.prototxt` that specifies network architecture. Including weights and bias values, there are a total of 5.97 million learned parameters in GoogLeNet.

We also use another text file listing 1000 labels used in ILSVRC 2012 challenge that includes a synonym ring or synset of those labels.

## 3 SystemC Modeling of GoogLeNet

We now describe how we design a SystemC model of GoogLeNet.

### 3.1 Reference Model using OpenCV

Our SystemC model of GoogLeNet is implemented based on an original model using OpenCV 3.4.1, a library of computer vision functions mainly aimed for real-time applications written in C/C++ [10]. The OpenCV library was originally developed by Intel and is now free for use under the open-source BSD license. OpenCV uses an internal data structure to represent an n-dimensional dense numerical single-channel or multi-channel array, a so called `Mat` class. Therefore, our model uses the `Mat` data type to store images, weight matrices, bias vectors, feature maps, and class scores. This becomes practical while interacting with various OpenCV APIs.

Furthermore, OpenCV provides an interface class, `Layer`, that allows for construction of constituent layers of neural networks. A `Layer` instance is constructed by passing layer parameters and is initialized by storing its learned parameters. A `Layer` instance computes an output `Mat` given an input `Mat` by calling its `forward` method. We refer to this class as OpenCV `layer` for the rest of this paper. OpenCV also provides utility functions to load an image and read a Caffe model from `.prototxt` and `.caffemodel` files.

### 3.2 Modeling Goals

Given the OpenCV primitives, we set three design goals in the early stage of model development as follows:

1. *Generic layers*: Since GoogLeNet is composed of only a handful of layer types, the layers shall be parameterized by their attributes using a custom constructor. For example, a pooling layer shall be parameterized by its type (max-pooling or average pooling), its kernel size, its stride, and the number of padding pixels.

2. *Self-contained layers*: Each layer shall implement the functionality it requires without the need of an external scheduler to load its input or in case load its parameters. For example, a convolution layer shall have a dedicated method to load its parameters (weight matrix and bias vector) used only at the time of construction.
3. *Reuseable and modular code*: Since most CNNs share a common set of layers, the code shall be structured in a way to enable the feeding of any kind of CNN with minimum effort. For example, the layer implementation shall be organized as code template blocks and the SystemC model shall be autogenerated using only the network model defined by Caffe model files.

Note that these goals will allow us to easily generate a SystemC model also for other Caffe CNNs. At the same time, the models generated will have a well-organized structure that enables static analysis. Specifically, this allows us to perform parallel simulation with RISC [9], as described in Section 3.6 below.

## 3.3 Layer Implementation

Each layer in the CNN is defined as a `sc_module` with one input port and one output port. Ports are defined as `sc_port` and are parameterized by our own defined interface classes, `mat_in_if` and `mat_out_if`. These user-defined interfaces are derived from `sc_interface` and declare `read` and `write` access methods with a granularity of `Mat`. The choice of `Mat` for a granularity of port parameterization simplifies the design by focusing on the proper level of abstraction at this stage of modeling. As an example, the module definition of the first convolution layer `conv1_7x7_s2` is shown in Listing 1.1.

As shown in lines 41-53 of Listing 1.1, each module has several attributes that are all defined as data members inside the class definition. For example, a convolution module is defined by its name, number of outputs, number of pixels for padding, kernel size, and number of pixels for stride. If a layer also has learned parameters, two Mat objects are defined as member variables to store the weight matrix and the bias vector. In that case, their values are initialized at the time of module construction. For example, a convolution module has a designated load method that reads pre-trained Caffe model files and stores weight and bias values in the `weights` and `bias` member variables.

```
1  class conv1_7x7_s2_t : sc_core::sc_module
2  {
3
4  public:
5    sc_core::sc_port<mat_in_if>  blob_in;
6    sc_core::sc_port<mat_out_if> blob_out;
7
8    SC_HAS_PROCESS(conv1_7x7_s2_t);
9
10   conv1_7x7_s2_t(sc_core::sc_module_name n_,
11       String name_,
12       unsigned int num_output_,
      unsigned int pad_,
      unsigned int kernel_size_,
      unsigned int stride_,
      unsigned int dilation_,
      unsigned int group_) :
    sc_core::sc_module(n_),
    name(name_),
    num_output(num_output_),
    pad(pad_),
    kernel_size(kernel_size_),
    stride(stride_),
    dilation(dilation_),
    group(group_),
    weights(4, weight_sz, CV_32F, weight_data),
    bias(4, bias_sz, CV_32F, bias_data)
  {
    load();
    SC_THREAD(main)
  }

  void load();
  void main();;
  void run(std::vector<Mat> &inpVec,
      std::vector<Mat> &outVec);

private:

  String              name;
  unsigned int        num_output;
  unsigned int        pad;
  unsigned int        kernel_size;
  unsigned int        stride;
  unsigned int        dilation;
  unsigned int        group;
  static const int    weight_sz[4];
  unsigned int        weight_data[64*3*7*7];
  static const int    bias_sz[4];
  unsigned int        bias_data[64];
  Mat                 weights;
  Mat                 bias;

};
```

**Listing 1.1.** Conv1_7x7_s2 module definition

Each module has also a main thread that continuously reads its input port, computes results, and writes those to its output port. Data processing is handled by the `run` method. Here, we rely on OpenCV to perform the computations. The `run` method creates an instance of OpenCV `layer` and calls its `forward` method by passing references to input `Mat` and output `Mat` objects.

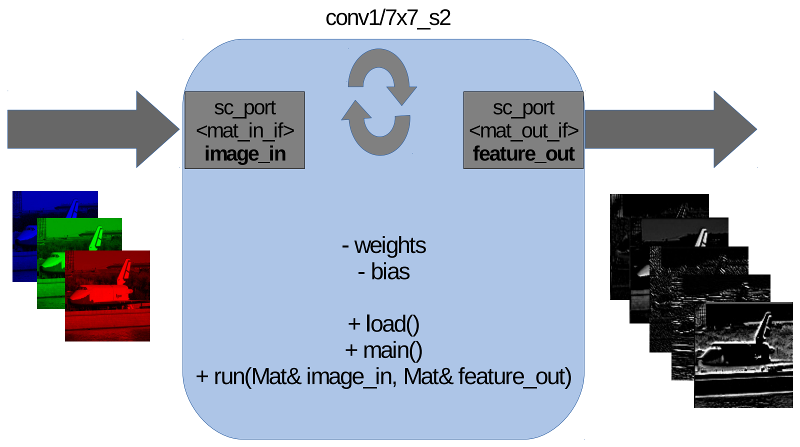

**Fig. 3.** Convolution layer

As an example, Figure 3 illustrates the module defining the first convolution layer in GoogLeNet. The input to the module is a `Mat` object containing 3 color channels of 224x224 pixels of the input "space shuttle" image and the output is another `Mat` object containing 64 feature maps with the size of 112x112 pixels.

## 3.4 Netspec Generator

Since each convolution layer consists of different parameters, writing module declarations by hand is an error-prone and tedious task. Moreover, declaring all modules and queues in the top level GoogLeNet module, instantiating them with the correct parameters, and binding queues to neighboring modules is also a laborious task. Therefore, we develop a generator tool to automatically extract the network architecture from a textual protocol buffer `.prototxt` and the network learned parameters from binary protocol buffer `.caffemodel`. The generator, called `netspec`, is written in Python and uses Python interface to Caffe library, `pyCaffe`, in order to read `.caffemodel` and `.prototxt` files to construct its internal data representation of the neural network. `Netspec` then uses this data structure to generate SystemC code for all convolution modules and the top level GoogLeNet module with all interconnecting FIFO channels.

## 3.5 Validation by Simulation

A top level test bench validates our GoogLeNet SystemC model against the reference OpenCV implementation. The test bench instantiates our SystemC GoogLeNet module which contains all modules inside the network with all the interconnecting queues as Design under Test (DUT). It also instantiates a stimulus module to feed the design with images of size 224x224 with three color channels, and a monitor module to read the final class scores and output the label with the maximum probability (Figure 4). To measure the performance of the model, our test bench can also be configured to continuously feed in a stream of images. In that case, a checker module is plugged inside the monitor to check the correct classification and its probability against the reference model.

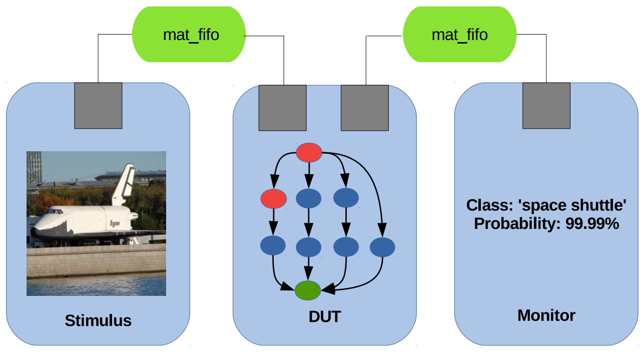

**Fig. 4.** Top-level test bench

### 3.6 Modular Source File Structure and Build Flow

Following good practices of SystemC coding, we place each module definition in a header file `.hpp` and the corresponding module implementation in a `.cpp` file. Also, to explore parallelism existing in the GoogLeNet system level model using RISC, we decide to split the implementation into two separate `.cpp` files. One `.cpp` file contains only methods that directly call OpenCV APIs $\langle module\_name\_cv.cpp\rangle$ and the other only contains the `main` method implementation that does not directly interact with OpenCV APIs $\langle module\_name.cpp\rangle$. This prevents RISC from unnecessarily analyzing and instrumenting the code inside the OpenCV library, by only feeding object files generated from CV parts and not including OpenCV library source code.

First `.caffemodel` and `.protoxt` files are fed to the `netspec` tool to generate code for convolution modules and the overall GoogLeNet module. Once these modules are generated, all $\langle module\_name.cpp\rangle$ and $\langle module\_name\_cv.cpp\rangle$ files are passed to the GNU compiler to generate the object files. Then, the object files are passed all together to the GNU linker with OpenCV and SystemC libraries to obtain the final executable. Running the executable requires the Caffe model files to load convolution modules with weights and bias values and also a synset file to read the class names.

The build flow specifically for RISC requires minimum effort due to our early decision to split the OpenCV source code from the model source code. Since RISC prefers all the source code in a single file, all header files and implementation files are merged into one file. This flattened source code, with object files generated from the OpenCV part of the modules, is then fed to RISC which then generates a multithreaded parallel executable.

## 4 Experimental Results

Our untimed SystemC model of GoogLeNet compiles and simulates successfully with Accellera SystemC 2.3.1. For parallel simulation, we also compile and simulate the model using RISC V0.5.1 to speed up simulator run time. Both simulation results match the OpenCV reference model output.

### 4.1 Performance Setup

We use two different computer platforms to benchmark the simulations. The specifications of each platform are shown in Table 2. We name platforms based on the number of logical cores visible to the operating system. The number of logical cores is double the number of physical cores when hyper-threading technology (HTT) is enabled.

To have reproducible experiments, the Linux CPU scaling governor is set to 'performance' to run all cores at the maximum frequency, and file I/O operations i.e. *cout* are minimized. SystemC 2.3.1 and OpenCV 3.4.1 are built with debugging information [1].

Moreover, the OpenCV library can be built with support for several parallel frameworks, such as POSIX threads (pthreads), Threading Building Blocks (TBB), and Open Multi-Processing (openMP), etc. We build OpenCV with the support for pthread to run in multithreaded mode and also without support for pthread to run only on a single-thread. Lastly, the stimulus module is configured to feed 500 images with size of 224x224 pixels to the model.

**Table 2.** Platform specification

| Platform name | 4-core host | 8-core host | 16-core host | 32-core host |
|---|---|---|---|---|
| OS | CentOS 7.6 | CentOS 7.6 | CentOS 6.10 | CentOS 6.10 |
| CPU Model name | Intel E3-1240 | Intel E3-1240 | Intel E5-2680 | Intel E5-2680 |
| CPU frequency | 3.4 GHz | 3.4 GHz | 2.7 GHz | 2.7 GHz |
| #cores | 4 | 4 | 8 | 8 |
| #processors | 1 | 1 | 2 | 2 |
| #threads per core | 1 | 2 | 1 | 2 |

[1] OpenCV has built with -O0 flag meaning (almost) no compiler optimizations.

### 4.2 Simulation Results

For benchmarking, we measure simulation time using Linux */usr/bin/time* under CentOS. This time function provides information regarding the system time, the user time, and the elapsed time. Measurements are reported for sequential SystemC simulation using Accellera SystemC compiled with POSIX threads. Parallel simulation is performed using RISC simulator V0.5.1 in non-prediction (NPD) mode. Tables 3 to 6 show the measurements for each simulation mode on the four different platforms using the single-thread and multithreaded OpenCV. In case of parallel simulations, we set the maximum number of concurrent threads allowed by the RISC simulator to the number of available logical cores on each platform.

**Table 3.** Measurement results on 4-core host (HTT off)

| | Single-thread | | Multithreaded | |
|---|---|---|---|---|
| Time (sec) | Accellera | RISC | Accellera | RISC |
| User time | 627.19 | 651.59 | 680.01 | 664.02 |
| System time | 1.55 | 1.11 | 34.26 | 18.26 |
| Elapsed time | 629.49 | 253.29 | 199.44 | 234.36 |
| CPU utilization | 99% | 257% | 358% | 291% |
| Speedup | 1x | 2.48x | 3.15x | 2.68x |

**Table 4.** Measurement results on 16-core host (HTT off)

| | Single-thread | | Multithreaded | |
|---|---|---|---|---|
| Time (sec) | Accellera | RISC | Accellera | RISC |
| User time | 912.79 | 921.95 | 1164.69 | 960.48 |
| System time | 34.76 | 42.19 | 705.22 | 134.25 |
| Elapsed time | 947.93 | 275.29 | 154.45 | 260.7 |
| CPU utilization | 99% | 350% | 1210% | 419% |
| Speedup | 1x | 3.44x | 6.13x | 3.63x |

### 4.3 Analysis

Table 3 allows the following observations:

1. **RISC introduces thread-level parallelism**

   RISC is faster than single-thread OpenCV with Accellera and it speeds up simulator run time up to 2.48x on the 4-core machine.

**Table 5.** Measurement results on 8-core host (HTT on)

| | Single-thread | | Multithreaded | |
|---|---|---|---|---|
| Time (sec) | Accellera | RISC | Accellera | RISC |
| User time | 621.49 | 961.44 | 1164.13 | 1046.39 |
| System time | 1.52 | 1.28 | 84.06 | 34.85 |
| Elapsed time | 622.68 | 254.07 | 184.09 | 232.57 |
| CPU utilization | 100% | 378% | 678% | 464% |
| Speedup | 1x | 2.45x | 3.38x | 2.67x |

**Table 6.** Measurement results on 32-core host (HTT on)

| | Single-thread | | Multithreaded | |
|---|---|---|---|---|
| Time (sec) | Accellera | RISC | Accellera | RISC |
| User time | 911.98 | 1177.02 | 2124.87 | 1299.86 |
| System time | 35.31 | 52.76 | 1838.35 | 224.84 |
| Elapsed time | 947.7 | 273.27 | 155.72 | 274.29 |
| CPU utilization | 99% | 450% | 2544% | 555% |
| Speedup | 1x | 3.46x | 6.08x | 3.45x |

2. **OpenCV parallelism is even faster than RISC**

   We observe that multithreaded OpenCV speeds up simulator run time using Accellera up to 3.15x on the 4-core machine. Therefore, thread-level parallelism in OpenCV primitives is more efficient than thread-level parallelism at SystemC level.

3. **Combining OpenCV and RISC parallelism does not deliver the best speedup**

   Since RISC and OpenCV threads unknowingly from each other compete for resources, exploiting parallelism in RISC and OpenCV at the same time does not increase the speedup. For example, multithreaded OpenCV using RISC (2.68x) performs worse than multithreaded OpenCV using Accellera (3.15x) on the 4-core machine.

4. **RISC performance improves slightly with OpenCV parallelism**

   RISC gains small speedup by also using parallelism in OpenCV. For example, RISC speeds up multithreaded OpenCV (2.68) in comparison with single-thread OpenCV (2.48x).

Table 4 supports observations 1 through 4 as well. It also allows for the following observation:

5. **Performance does not scale by the number of cores**

Quadratic increase in the number of cores only leads to double increase in performance. Relative good speed up to 3.15x on the 4-core machine does not scale to 16-core machines and only gets 6.13x speedup compared to sequential single-thread simulator run time.

Table 5 and 6 use hyper-threading technology (HTT) and allow for the following observations:

6. **HTT is ineffective for this application**
   Enabling HTT slightly improves speedup from 3.15x on the 4-core machine without HTT to 3.38 on the 4-cores with HTT (8-cores). In case of 16-cores to 32-cores, performance has not improved at all.
7. **HTT substantially increases user and system time**
   We observe that the user and system times increase significantly with HTT turned on. At this point, the origin of this time increase is unclear for us. We will investigate this further in more detailed future research.

In summary, Figure 5 shows the speedups for different sources of parallelism: single-threaded OpenCV using **RISC**, multithreaded **OpenCV** using Accellera and multithreaded **OpenCV** using **RISC**. The illustration shows a significant speedup using parallelism introduced by RISC and multithreaded OpenCV. It also demonstrates that combining OpenCV and RISC parallelism does not provide a remarkable speedup.

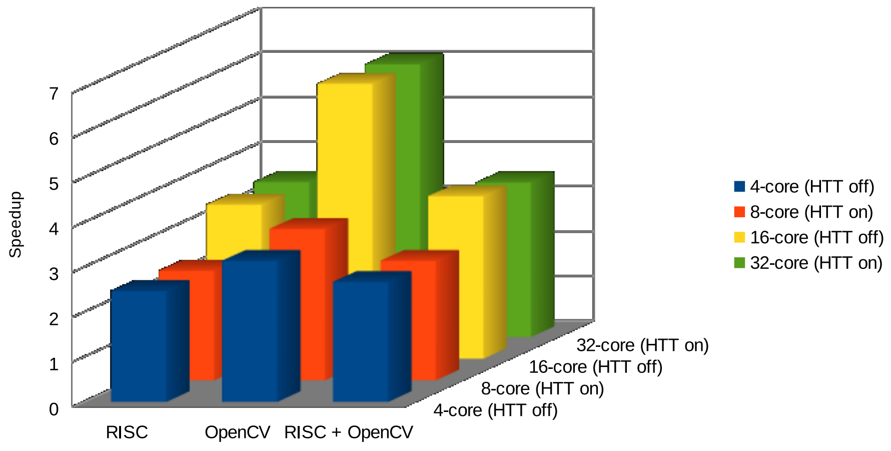

**Fig. 5.** Speedup comparison on different platforms based on the source of parallelism

## 5 Conclusion

In this report, we have described an untimed SystemC model of GoogLeNet using OpenCV 3.4.1 library. We also developed a tool to automatically generate

SystemC code from Caffe model files. We successfully simulated the generated model using Accellera SystemC 2.3.1 and RISC V0.5.1.

Experimental results show significant simulation speedups using RISC, as well as using multithreaded OpenCV. Results also show that combining OpenCV and RISC parallelism did not deliver significant speedup.

## References

1. Gerstlauer, A., Dömer, R., Peng, J., Gajski, D.D.: System Design: A Practical Guide with SpecC. Kluwer (2001)
2. Grötker, T., Liao, S., Martin, G., Swan, S.: System Design with SystemC. Kluwer (2002)
3. Huang, G., Liu, Z., van der Maaten, L., Weinberger, K.Q.: Densely connected convolutional networks. In: 2017 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2017. pp. 2261–2269. IEEE Computer Society (2017)
4. Jia, Y., Shelhamer, E., Donahue, J., Karayev, S., Long, J., Girshick, R., Guadarrama, S., Darrell, T.: Caffe: Convolutional architecture for fast feature embedding. arXiv preprint arXiv:1408.5093 (2014)
5. Krizhevsky, A., Sutskever, I., Hinton, G.E.: Imagenet classification with deep convolutional neural networks. In: NIPS (2012)
6. Larsson, G., Maire, M., Shakhnarovich, G.: Fractalnet: Ultra-deep neural networks without residuals. CoRR **abs/1605.07648** (2016)
7. Le Cun, Y., Jackel, L.D., Boser, B., Denker, J.S., Graf, H.P., Guyon, I., Henderson, D., Howard, R.E., Hubbard, W.: Handwritten digit recognition: Applications of neural network chips and automatic learning. Comm. Mag. **27**(11), 41–46 (Nov 1989)
8. LeCun, Y., Haffner, P., Bottou, L., Bengio, Y.: Object recognition with gradient-based learning. In: Shape, Contour and Grouping in Computer Vision. p. 319 (1999)
9. Liu, G., Schmidt, T., Cheng, Z., Mendoza, D., Dömer, R.: RISC Compiler and Simulator, Release V0.5.0: Out-of-Order Parallel Simulatable SystemC Subset. Tech. Rep. CECS-TR-18-03, CECpS, UCI (Sep 2018)
10. OpenCV Tutorials, Load Caffe framework models. https://docs.opencv.org/3.4/d5/de7/tutorial_dnn_googlenet.html, accessed: 2019-05-11
11. Sze, V., Chen, Y., Yang, T., Emer, J.S.: Efficient processing of deep neural networks: A tutorial and survey. Proceedings of the IEEE **105**(12), 2295–2329 (2017)
12. Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S.E., Anguelov, D., Erhan, D., Vanhoucke, V., Rabinovich, A.: Going deeper with convolutions. In: IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2015. pp. 1–9 (2015)
13. Xie, S., Girshick, R.B., Dollár, P., Tu, Z., He, K.: Aggregated residual transformations for deep neural networks. CoRR **abs/1611.05431** (2016)