White Paper

Philips Healthcare Uses the Intel® Distribution of OpenVINO™ Toolkit and the Intel® DevCloud for the Edge to Accelerate Compressed Sensing Image Reconstruction Algorithms for MRI

Authors

G Anthony Reina
Intel Corporation
Maurice Stassen
Philips Research
Nicola Pezzotti
Philips Research, TU Eindhoven
Dennis Moolenaar
Philips Research
Dmitry Kurtaev
Intel Corporation
Abhishek Khowala
Intel Corporation

Table of Contents

Abstract

Compressed sensing (CS) is a signal processing technique that enables faster scan times in medical imaging. Philips Healthcare integrated CS methods into their magnetic resonance imaging (MRI) scanners to reduce scan time by up to 50 percent for 2D and 3D sequences, compared to Philips scans without Compressed SENSE, with virtually equal image quality. Recently, deep learning methods have been explored for reconstructing MRI images, showing good results in terms of image quality and speed of reconstruction. Philips Healthcare and Intel report on two hybrid frequency-domain/image-domain encoder/decoder architectures that produce excellent results in MRI reconstruction. We show how these two neural networks can be accelerated on Intel® hardware through use of the Intel® Distribution of OpenVINO™ Toolkit. The toolkit allows Philips Healthcare to speed up their deep learning inference by as much as 54x over standard, unoptimized TensorFlow 1.15, as tested in Philips' proprietary Linux environment on Intel® Xeon® processors.[1] We further describe how to leverage the Intel® DevCloud for the Edge, which allowed Philips Healthcare to compare performance of their deep learning models on Intel Xeon and Intel Core™ processors, Intel Movidius Vision Processing Units (VPUs), FPGAs, and integrated GPU hardware in order to design deep learning products of various performance, price, power, and form factors.

Introduction

Compressed sensing (CS) is a signal processing technique that reconstructs a signal with far fewer data samples than would ordinarily be required by the Nyquist-Shannon sampling theorem (Candès and Tao, 2004; Donoho, 2006). CS has become an important element in medical imaging because it produces high-quality scans with fewer data points (Lustig et al., 2007; Chen et al., 2008; Graff and Sidky, 2015). This allows for shorter scanning times. In 2017, Philips Healthcare integrated CS methods into their magnetic resonance imaging (MRI) scanners, which reduced scan times by up to 50 percent for 2D and 3D sequences, compared to Philips scans without Compressed SENSE, with virtually equal image quality.

In its basic form, CS is a mathematical approach to solving an undetermined linear system (Figure 1). In general, such a problem is ill-posed—that is, there are an infinite number of solutions to the undetermined system. Compressed sensing adds two critical constraints to the problem: (1) the sampling must be incoherent (that is, random), and (2) the signal must be sparse in some domain (for example, the wavelet domain). Given these two constraints, there are several methods that can find a solution.

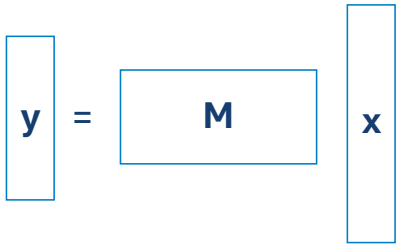

Figure 1. An undetermined linear system—the vector **x** is much "taller" than the matrix **M**. In general, an exact solution for the vector **x** cannot be found given the vector **y** and the measurement matrix **M** unless **x** is found to be sparse and **M** is incoherently sampled. Given those two conditions, compressed sensing can be used to find a solution.

In general, CS solutions involve the L1 minimization of a regularization term through various approaches including least squares methods, such as Lasso (Tibshirani, 1996), and iterative optimization methods, such as, Iterative Forward-Backward Pursuit (Wang *et al.*, 2016) and the Iterative Shrinkage-Thresholding Algorithm (Beck A, Teboulle M., 2009). More recently, deep learning methods provided an alternative approach to solving the sparsity problem and promises to provide reconstructions at lower computational cost. Among those, several different approaches are presented, including Generative Adversarial Methods models (Yang, 2018), Encoder-Decoder models (Jin, 2016) and unrolled iterative schemes (Pezzotti *et al.* 2020).

In this whitepaper, we highlight two CS deep learning methods for MR image reconstruction: the open-sourced W-Net (Souza and Frayne, 2018) and a proprietary model from Philips Healthcare based on ISTANet (referred to here as Adaptive-CS-Net). Philips Healthcare partnered with Intel to benchmark their custom CS topology and determine if Intel hardware could provide the mission-critical speed necessary to use this method in their future product designs. Because the Philips topology is proprietary, the open-sourced W-Net model has been used to demonstrate the changes needed to optimize both models. We demonstrate how the Intel Distribution of OpenVINO toolkit accelerated Philips' CS model across a wide range of Intel hardware, including CPUs, VPUs, FPGAs, , and integrated GPUs—all of which can be evaluated for free on the Intel DevCloud for the Edge.

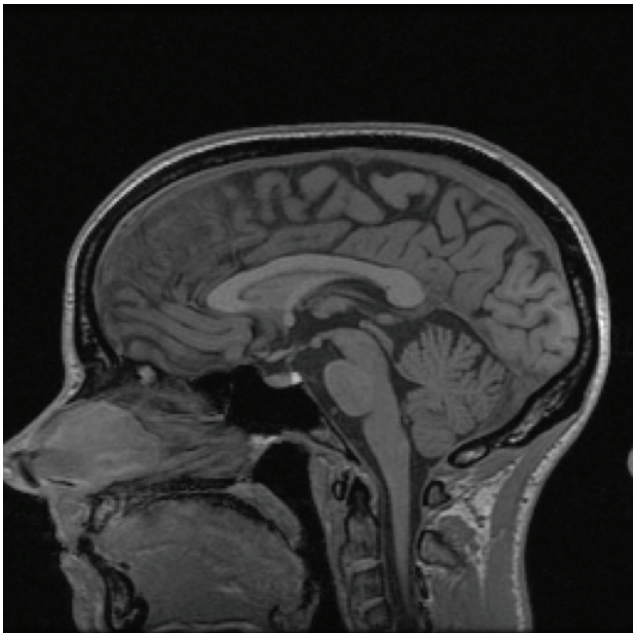

Figure 2. Visualization of a single slice from validation subset of Calgary-Campinas Public Dataset. Every scan has between 170 and 180 slices of 256x256 dimensions.

Datasets

Open-source Dataset—Calgary-Campinas

The Calgary-Campinas Public Brain MR dataset consists of T1-weighted MR scans from 359 older adult subjects on scanners from three different vendors at 1.5 and 3.0 T magnetic field strengths (Souza *et al.*, 2018) (Figure 2). The dataset is a collaborative effort between the University of Calgary and the University of Campinas with the goal of developing innovative deep learning models to reconstruct, process, and analyze MR scans.

Open Dataset: fastMRI

Philips Healthcare provided the knee subset of the fastMRI dataset for its model (Zbontar *et al.*, 2018). The data was acquired by NYU Langone Health with a 2D protocol with a 15-channel knee coil array using Siemens MR machines at two different field strengths: 1.5T and 3T. The single coil data, which is used in this study, was emulated starting from the multi-coil channel acquisition. The dataset contains two different acquisition protocols, Proton Density and Proton Density with Fat Suppression. It comprised of 973 volumes for the training set (34,742 slices), and 199 volumes for the validation set (7,135 slices). For the undersampling, a randomized 1D mask with fully sampled center is used.

Model Topology

W-Net

W-Net is a neural network topology that was developed by Souza and Frayne (2018). Because the Philips topology is proprietary, the W-Net model was chosen to demonstrate the changes needed to optimize CS topologies in general. Briefly, the model consists of two U-Net topologies (Ronneberger *et al.*, 2015) connected by a 2D inverse Fast Fourier Transform (IFFT2D) that converts the sampled MR k-space into an image domain (Figure 3). W-Net effectively upsamples and de-noises the scan in both the frequency and spatial domains using 2D convolutional filters. This allows it to provide a low-loss, compressed reconstruction of the MRI despite the large undersampling of the k-space input (Figure 4). A pre-trained TensorFlow/Keras version of the W-Net model was published by Souza and Frayne (2018). The source code and model can be found on GitHub under the MIT license.

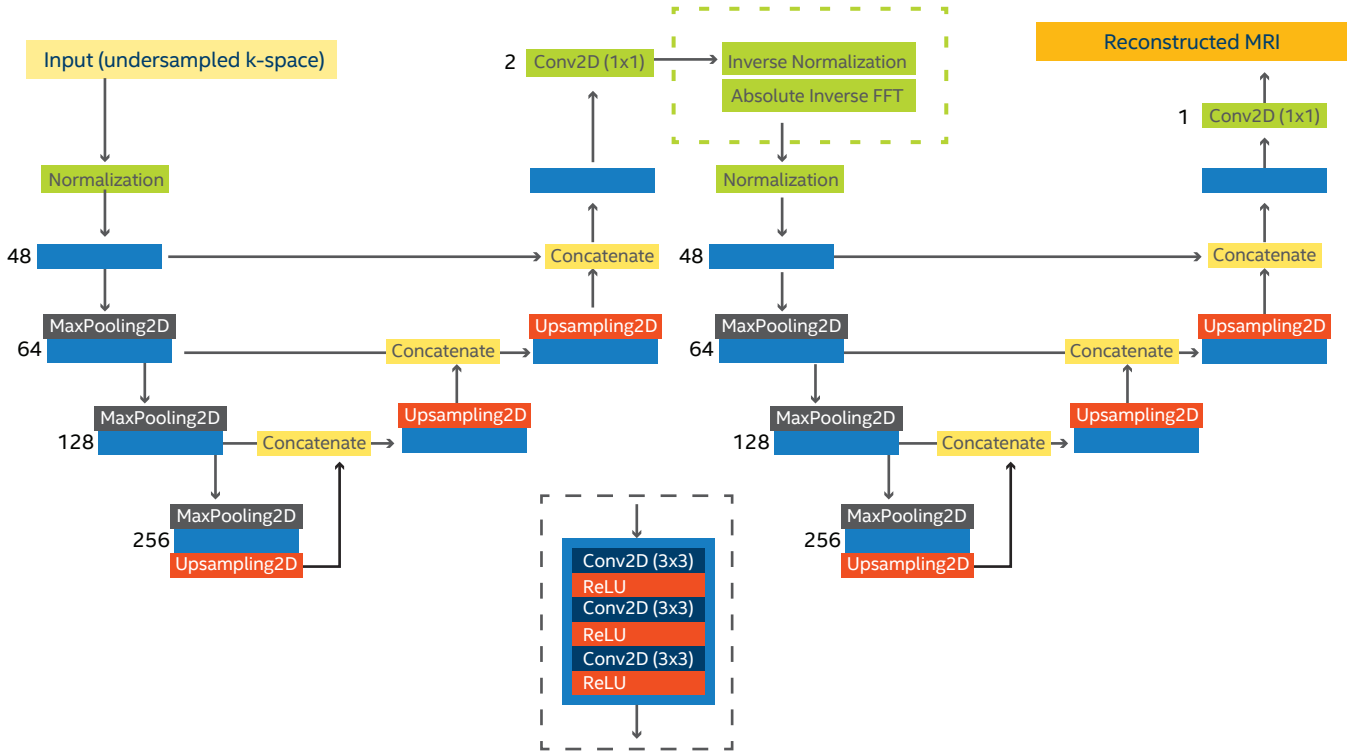

Figure 3. W-Net topology (Souza and Frayne, 2018). It consists of two U-Net encoder/decoder networks (Ronneberger *et al.*, 2015). The first U-Net (left) reconstructs the k-space representation from the undersampled k-space. The second U-Net (right) reconstructs the image from the k-space transformation. An inverse Fast Fourier Transform (IFFT2D) connects the frequency and image domains. Custom OpenVINO extensions were created for the TensorFlow/Keras subgraph within the green dotted box.

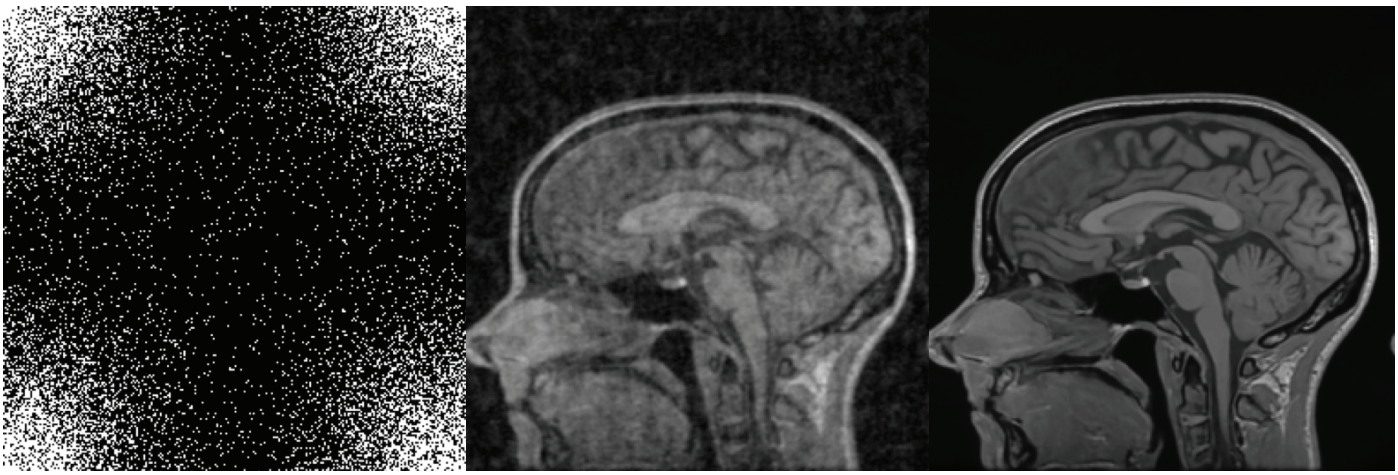

Figure 4. A random sampling mask that uses only about 20 percent of the MR k-space (left). Visualization of origin slice with a mask gives a poor reconstruction with a PSNR of 21.5 (middle). The reconstructed slice with W-Net gives a much better reconstruction with a PSNR of 34.8 (right).

Adaptive-CS-Net

Philips Healthcare, Research and the Leiden University Medical Center, created a custom architecture (Pezzotti et al, 2020) based on the ISTA-Net model (Zhang and Ghanem, 2018). It uses successive blocks of multiscale transforms to perform the CS reconstruction along with several MR priors. Figure 5 shows the resulting architecture, named Adaptive-CS-Net, which is trained to produce high resolution MR reconstructions of the knee with a significant reduction in the sampling rate from the MRI k-space.

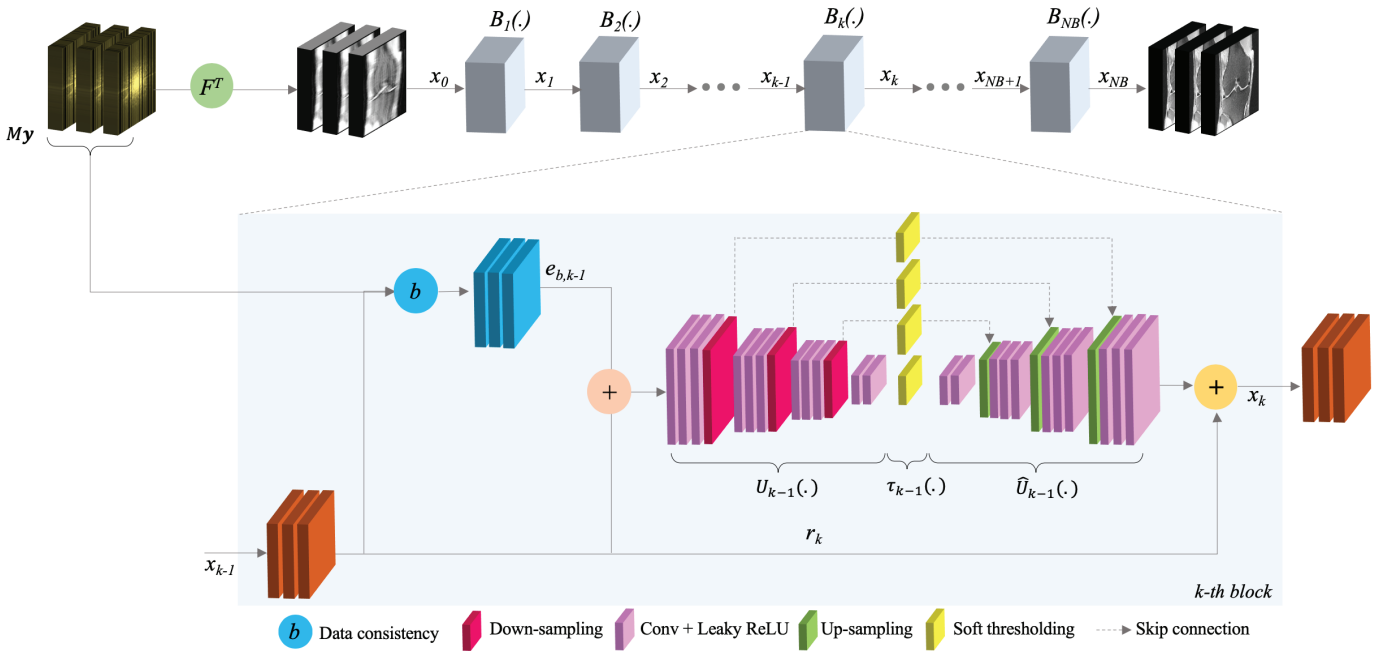

Figure 5. The Adaptive-CS-Net architecture. Successive encoding (U_k (.)) and decoding (Uˆ_k (.) filter blocks are trained to provide the optimal image reconstruction.

Optimization Using the Intel Distribution of OpenVINO Toolkit

Pre-trained TensorFlow 1.15 models for W-Net and Adaptive-CS-Net were used as the baseline benchmark for inference tests. Standard, unoptimized TensorFlow 1.15 was used for benchmarking TensorFlow as prescribed by Philips proprietary Linux environment (via `pip install tensorflow=1.15`). We used the Intel Distribution of OpenVINO toolkit version 2020.2 to accelerate the inference latency of these models. This toolkit allows developers to optimize neural network inference on Intel® CPU, FPGA, VPU, and integrated GPU hardware.

To enable the toolkit, we first converted the TensorFlow/Keras model to an Intermediate Representation (IR) format using the OpenVINO Toolkit Model Optimizer command line tool. The following examples demonstrate how we optimized the W-Net model. The same procedure was used to optimize the Adaptive-CS-Net model.

The pre-trained W-Net TensorFlow model was first converted to a single, frozen TensorFlow protobuf file (Figure 6, model filename "wnet_20.pb"). Figure 6 shows the Python script used to freeze a TensorFlow model. This script converts the model variables to constants and saves both the weights and graph definition into a single protobuf file. It should be noted that the OpenVINO toolkit's Model Optimizer tool can also work from TensorFlow checkpoints and TensorFlow SavedModel formats.

```
# Tested with Keras 2.2.4 and TensorFlow 1.15.0
# export PYTHONPATH=Hybrid-CS-Model-MRI/Modules/:$PYTHONPATH
import numpy as np
import tensorflow as tf
import frequency_spatial_network as fsnet

stats = np.load("Hybrid-CS-Model-MRI/Data/stats_fs_unet_norm_20.npy")
model = fsnet.wnet(stats[0], stats[1], stats[2], stats[3],
                   kshape = (5,5), kshape2=(3,3))

model_name = "Hybrid-CS-Model-MRI/Models/wnet_20.hdf5"
model.load_weights(model_name)

import keras as K
sess = K.backend.get_session()
sess.as_default()

graph_def = sess.graph.as_graph_def()
graph_def = tf.graph_util.convert_variables_to_constants(sess, graph_def,
['conv2d_44/BiasAdd'])

with tf.gfile.FastGFile('wnet_20.pb', 'wb') as f:
    f.write(graph_def.SerializeToString())
```

Figure 6. This Python script demonstrates how to freeze a TensorFlow model (the model filename is "wnet_20.pb"). Freezing a TensorFlow model converts the variables on the graph to constants and combines the weights and graph description into a single file.

Model Optimizer Extensions

The Intel Distribution of OpenVINO toolkit version 2020.2 does not have out-of-the-box support for the frequency domain operations that are necessary to execute these CS deep learning models (See Figure 3, subgraph within the green dotted box). Although these operations will be added to a future release, Philips Healthcare needed them to be supported immediately to meet their design timeline. Fortunately, the toolkit can be easily extended by developers to expand functionality and implement new layers. For W-Net, three new layers were created: the TensorFlow operation tf.dtypes.complex, which is the complex datatype (Complex); the TensorFlow operation tf.signal.ifft2d, which is the inverse 2D Fast Fourier Transform (IFFT2D); and, the TensorFlow operation tf.math.abs, which is the absolute value of a complex datatype (ComplexAbs). For the Adaptive-CS-Net the TensorFlow operation tf.signal.fft2d, which is the 2D Fast Fourier Transform (FFT2D), was also created in a similar fashion as the IFFT2D. Table 1 shows the subgraph section of the TensorFlow model for these operations/layers (Figure 3, subgraph within the green dotted box).

Table 1: Original subgraph in the TensorFlow topology.

Op	Inputs	Inputs Data Type	Output	Output Data Type
Add	1x256x256x2	DT_FLOAT	1x256x256x2	DT_FLOAT
StridedSlice	1x256x256x2	DT_FLOAT	1x256x256	DT_FLOAT
Complex	1x256x256 1x256x256	DT_FLOAT	1x256x256	DT_COMPLEX64
IFFT2D	1x256x256	DT_COMPLEX64	1x256x256	DT_COMPLEX64
ComplexAbs	1x256x256	DT_COMPLEX64	1x256x256	DT_FLOAT

Although we created custom CS layers from scratch, we simplified the design by continuing to work with floating point tensors and keeping the same input and output shapes of the subgraph. Although the Intel Distribution of OpenVINO toolkit did not have a complex-value datatype, we were able to emulate that datatype tensor using two-channel, float-value tensors with the real part in the first channel and the imaginary part in the second channel. Table 2 shows the optimized conversion for the TensorFlow subgraph from Table 1.

Table 2: The OpenVINO-optimized version of the TensorFlow subgraph in Table 1.

Op	Inputs	Inputs Data Type	Output	Output Data Type
Add	1x256x256x2	DT_FLOAT	1x256x256x2	DT_FLOAT
IFFT2D	1x256x256x2	DT_FLOAT	1x256x256x2	DT_FLOAT
Pow (2.0)	1x256x256x2	DT_FLOAT	1x256x256x2	DT_FLOAT
ReduceSum	1x256x256x2	DT_FLOAT	1x256x256	DT_FLOAT
Pow (0.5)	1x256x256	DT_FLOAT	1x256x256	DT_FLOAT

By maintaining the real and imaginary parts as separate channels, we did not need to split the input tensor into real and imaginary parts and could instead process it directly within the IFFT2D custom layer. Also, rather than explicitly adding a new ComplexAbs operation, we modified the Model Optimizer to convert the ComplexAbs operation into three existing operations: Pow (2.0), which performs an elementwise squaring of the tensor elements; ReduceSum, which sums the elements across the channels; and Pow (0.5), which performs an elementwise square root.

To modify the Model Optimizer, we created an extensions folder (`mo_extensions`) for the custom operations with the directory structure, as shown in Figure 7. Note that the `complex.py` and `complex_abs.py` operations do not require new operations to be defined in the Intel Distribution of OpenVINO toolkit, but are simply implemented using a combination of existing toolkit functions. Therefore, these Python scripts are placed under the `front->tf` subdirectory to indicate that they are just front-end scripts to the TensorFlow graph. On the other hand, `ifft2d.py` is a new operation that is not based on an existing toolkit function. For this reason, it is placed under the `ops` subdirectory to alert the Model Optimizer that the operation is to be defined by the developer using custom C++ code.

```
mo_extensions
  |-- ops
      |-- ifft2d.py
  |-- front
      |-- tf
          |-- complex.py
          |-- complex_abs.py
```

Figure 7. Directory structure of the Model Optimizer extensions.

Figures 8 to 10 show the Python code that allows the Model Optimizer to create the custom operations to convert the TensorFlow subgraph (Figure 3, green-dotted box) to a toolkit IR. Figure 8 shows the graph replacement for the complex datatype. This operation in the subgraph receives 1x256x256x2 tensor as input and produces 1x256x256x1 complex datatype tensor. In the case of ComplexAbs we chose a different approach as shown in Figure 9. Instead of subgraph pattern matching, we replace every entry of ComplexAbs operation to a chain of three operations that give the equivalent mathematical result: elementwise square, reduced summation across the two channels, and an elementwise square root. Finally, Figure 10 shows how to register the IFFT2D operation; the actual implementation of this custom operation will be defined elsewhere. The "Infer" option in the new class specifies the shape propagation method. This allows the custom IFFT2D node to define its shape based on the input shape.

With these three Python scripts in the `mo_extensions` folder, we were able to use Model Optimizer to convert the frozen TensorFlow model to an IR with the following command:

```
python3 /opt/intel/openvino/deployment_tools/model_optimizer/mo_tf.py \
    --input_model wnet_20.pb \
    --input_shape "[1, 256, 256, 2]" \
    --extensions /path/to/mo_extensions
```

where:

`--input_model` path to a frozen graph

`--input_shape` TensorFlow model's input shape (note that this assumes the standard TensorFlow NHWC input tensor format)

`--extensions` path to our custom extensions

```
# mo_extensions/front/tf/complex.py
import logging as log
from mo.front.common.replacement import FrontReplacementSubgraph
from mo.graph.graph import Graph

class Complex(FrontReplacementSubgraph):
    enabled = True

    def pattern(self):
        return dict(
            nodes=[
                ('strided_slice_real', dict(op='StridedSlice')),
                ('strided_slice_imag', dict(op='StridedSlice')),
                ('complex', dict(op='Complex')),
            ],
            edges=[
                ('strided_slice_real', 'complex', {'in': 0}),
                ('strided_slice_real', 'complex', {'in': 1}),
            ])

    @staticmethod
    def replace_sub_graph(graph: Graph, match: dict):
        inp0 = match['strided_slice_real'].in_port(0).get_source().node
        inp1 = match['strided_slice_imag'].in_port(0).get_source().node
        complexNode = match['complex']


        if inp0.id != inp1.id:
            log.debug('The pattern does not correspond to Complex subgraph')
            return

        complexNode.out_port(0).get_connection().set_source(inp0.out_port(0))
```

Figure 8. Python code for `complex.py` custom operation to store a complex datatype.

```
# mo_extensions/front/tf/complex_abs.py
import numpy as np
from mo.front.common.replacement import FrontReplacementOp
from mo.graph.graph import Graph, Node
from mo.ops.const import Const
from extensions.ops.elementwise import Pow
from extensions.ops.ReduceOps import ReduceSum

class ComplexAbs(FrontReplacementOp):
    op = "ComplexAbs"
    enabled = True

    def replace_op(self, graph: Graph, node: Node):
        pow_2 = Const(graph, {'value': np.float32(2.0)}).create_node()
        reduce_axis = Const(graph, {'value': np.int32(-1)}).create_node()
        pow_0_5 = Const(graph, {'value': np.float32(0.5)}).create_node()

        sq = Pow(graph, dict(name=node.in_node(0).name + '/sq', power=2.0)) \
             .create_node([node.in_node(0), pow_2])

        sum = ReduceSum(graph, dict(name=sq.name + '/sum')) \
              .create_node([sq, reduce_axis])

        sqrt = Pow(graph, dict(name=sum.name + '/sqrt', power=0.5)) \
               .create_node([sum, pow_0_5])

        return [sqrt.id]
```

Figure 9. Python code for `complex_abs.py` custom operation to store a complex absolute value operation. Note the TensorFlow operation is split into three parts: an elementwise square, a reduce sum, and an elementwise square root.

```
# mo_extensions/ops/ifft2d.py
from mo.front.common.partial_infer.elemental import copy_shape_infer
from mo.graph.graph import Graph
from mo.ops.op import Op

class IFFT2D(Op):
    op = 'IFFT2D'
    enabled = True

    def __init__(self, graph: Graph, attrs: dict):
        super().__init__(graph, {
            'type': __class__.op,
            'op': __class__.op,
            'infer': copy_shape_infer
        }, attrs)
```

Figure 10. Python code for ifft2d.py custom operation.

CPU Extensions Library

For the Intel Distribution of OpenVINO toolkit IR model we created in the previous steps, the IFFT2D (and FFT2D) operations are not supported out-of-the-box by the toolkit version 2020.2. The previous steps only registered them with the Model Optimizer as custom CPU extensions. They did not actually implement those custom functions. To complete the custom operations, we need to provide the custom C++ code so that the toolkit's Inference Engine knows how to execute the custom operation in runtime.

Fortunately, in addition to its deep learning libraries, the Intel Distribution of OpenVINO Toolkit also contains OpenCV, a de facto standard library for computer vision. OpenCV has both FFT and IFFT implementation that have been optimized to run on Intel hardware. Therefore, all our custom C++ code needs to do is make a library call to the OpenCV optimized functions to complete the IFFT2D and FFT2D operations for the Inference Engine.

The toolkit uses a command line tool called Extension Generator (`extgen.py`) to generate a CPU extensions library template. Following is the log of the Extension Generator command line session used to create the custom C++ extension code for the IFFT2D operation:

```
$ python3 /opt/intel/openvino/deployment_tools/tools/extension_generator/extgen.py \
    new \
    --ie-cpu-ext \
    --output_dir=/path/to/fft_extensions

Generating:
  Model Optimizer:
    Extractor for Caffe Custom Layer: No
      Extractor for MxNet Custom Layer: No
    Extractor for TensorFlow Custom Layer: No
      Framework-agnostic operation extension: No
    Inference Engine:
      CPU extension: Yes
      GPU extension: No

Enter operation name:    IFFT2D

Enter type for parameters that will be read from IR in format
  <param1> <type>
  <param2> <type>
  ...
Example:
  length int

  Supported cpu types: int, float, bool, string, listfloat, listint

Enter 'q' when finished:    q

********************************************************************************************
Check your answers for the Inference Engine extension generation:

1.  Operation name:                                                          IFFT2D
2.  Parameters types in format <param> <type>:                               []

********************************************************************************************

Do you want to change any answer (y/n) ? Default 'no'
no

The following folders and files were created:

Stub files for the Inference Engine CPU extension are in /path/to/fft_extensions/user_ie_extensions/cpu folder
```

As indicated in the log above, the OpenVINO Extension Generator command line tool creates a C++ template (`ext_ifft2d.cpp`) for the custom operation. We then added the implementation of IFFT2D over 4-dimensional input tensors using the Intel Distribution of OpenCV, as shown in Figure 11. Similarly, the FFT2D function from the Adaptive-CS-Net model was created using the same command line tool.

```
// fft_extensions/user_ie_extensions/cpu/ext_ifft2d.cpp
#include "ext_list.hpp"
#include "ext_base.hpp"
#include <cmath>
#include <vector>
#include <string>
#include <algorithm>

#include <opencv2/opencv.hpp>
#include <inference_engine.hpp>

namespace InferenceEngine { namespace Extensions { namespace Cpu {

static cv::Mat infEngineBlobToMat(const InferenceEngine::Blob::Ptr& blob) {
    // NOTE: Inference Engine sizes are reversed.
    std::vector<size_t> dims = blob->getTensorDesc().getDims();
    std::vector<int> size(dims.begin(), dims.end());
    auto precision = blob->getTensorDesc().getPrecision();
    CV_Assert(precision == InferenceEngine::Precision::FP32);
    return cv::Mat(size, CV_32F, (void*)blob->buffer());
}

class IFFT2DImpl: public ExtLayerBase {
public:
    explicit IFFT2DImpl(const CNNLayer* layer) {
        addConfig(layer, { { ConfLayout::PLN, false, 0 } },
                         { { ConfLayout::PLN, false, 0 } });
    }

    StatusCode execute(std::vector<Blob::Ptr>& inputs,
                       std::vector<Blob::Ptr>& outputs,
                       ResponseDesc *resp) noexcept override {
        cv::Mat inp = infEngineBlobToMat(inputs[0]);
        cv::Mat inp = infEngineBlobToMat(outputs[0]);

        const int n = inp.size[0];
        const int h = inp.size[2];
        const int w = inp.size[3];
        cv::Mat complex(h, w, CV_32FC2), interleavedOut(h, w, CV_32FC2);
        for (int i = 0; i < n; ++i) {
            std::vector<cv::Mat> components = {
                cv::Mat(h, w, CV_32F, inp.ptr<float>(i, 0)),
                cv::Mat(h, w, CV_32F, inp.ptr<float>(i, 1))
            };
            cv::merge(components, complex);

            cv::idft(complex, interleavedOut, cv::DFT_SCALE);

            components = {
                cv::Mat(h, w, CV_32F, out.ptr<float>(i, 0)),
                cv::Mat(h, w, CV_32F, out.ptr<float>(i, 1))
            };
            cv::split(interleavedOut, components);
        }
        return OK;
    }
};

REG_FACTORY_FOR(ImplFactory<IFFT2DImpl>, IFFT2D);

}}}
```

Figure 11. C++ code for custom IFFT2D operation. Note that the OpenVINO™ extension generator command line tool created this C++ template. We added custom OpenCV calls to the cv::idft function within the “Execute” function. This allows OpenVINO to leverage the OpenCV optimized IFFT2D implementation during inference.

Lastly, we need to link the Intel Distribution of the OpenCV runtime library by adding the following lines to the generated CMakeLists.txt:

```
find_package(OpenCV REQUIRED)
include_directories(PRIVATE ${OpenCV_INCLUDE_DIRS})
target_link_libraries(${TARGET_NAME} ${OpenCV_LIBS})
```

After compilation, a shared library called **libuser_cpu_extension.so** should be created. It can be loaded into the CPU plugin when the Inference Engine is first loaded:

```
from openvino.inference_engine import IENetwork, IECore

net = IENetwork("wnet_20.xml", "wnet_20.bin")
ie = IECore()
ie.add_extension("libuser_cpu_extension.so", "CPU")
exec_net = ie.load_network(net, "CPU")
```

Benchmarking

All benchmarks were performed on the Intel DevCloud for the Edge, which was chosen by Philips Healthcare because it allowed them to easily compare how their specific deep learning models performed on a variety of Intel hardware without the need for purchasing and setting up the hardware. The DevCloud also allowed Philips Healthcare to consider price, power, and form factor requirements for their specific product needs, although these requirements and comparisons are outside the scope of the current paper.

All Intel Distribution of OpenVINO toolkit benchmarks were performed using the published benchmarking tool. A grid search of various batch sizes, number of inference streams, and number of inference requests were performed and the best results were reported.

All TensorFlow benchmarks were performed using a custom Python script and standard, unoptimized TensorFlow 1.15 (using `pip install tensorflow=1.15`). After a warm up of 10 inference requests, the elapsed times were recorded for 100 successive inference requests using a grid search of batch sizes and threads. The best results from the grid search were reported.

Performance

The Intel DevCloud for the Edge was used to ensure repeatable benchmarks. The Intel DevCloud provides access to different Intel hardware targets and has a pre-installed version of the Intel Distribution of OpenVINO toolkit for fast prototyping and experimentation. Figure 12 shows the relative speedups of toolkit versus TensorFlow on the W-Net model. The toolkit improved inference speed by 19 percent and 20 percent on the Intel Core i7 and Intel Xeon E3 processors, respectively. On the Intel Xeon Gold 6138 processor, the toolkit's optimizations gave more than a 3x speedup in inference.[1] It should be noted that the W-Net topology is a much smaller model than the Adaptive-CS-Net model, which means that there are fewer places to optimize the model.

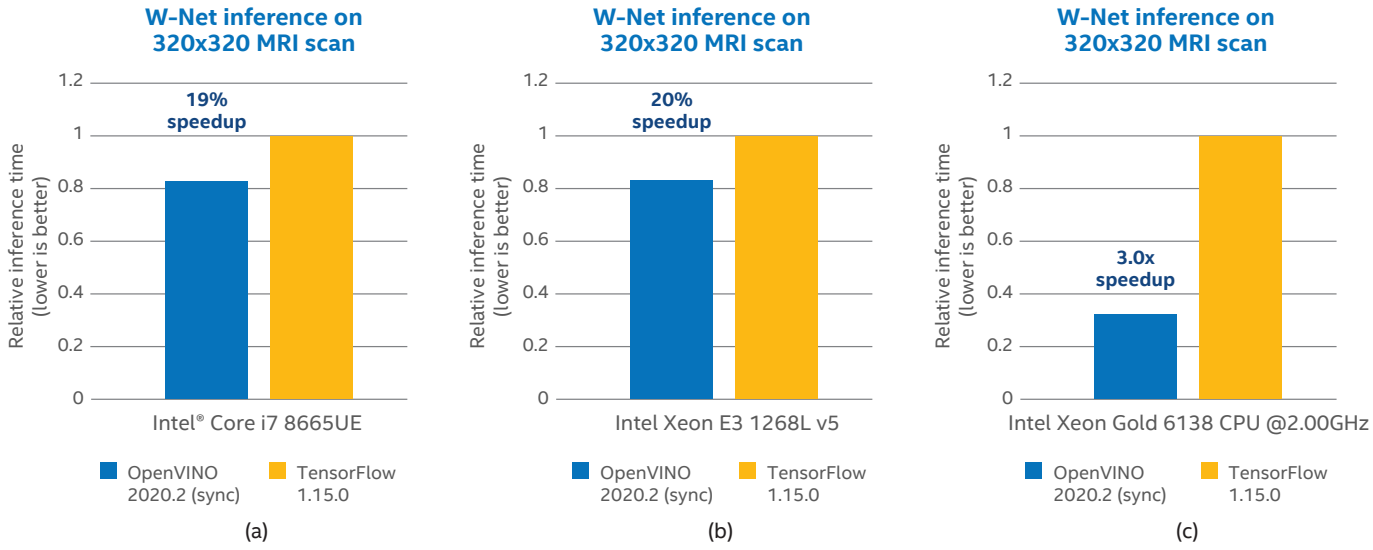

Figure 12. Comparing the unoptimized TensorFlow 1.15.0 (orange) versus OpenVINO™ toolkit 2020.2 (blue) time to process W-Net with a 170x256x256 MRI scan by running the network 170 times for each slice on (a) Intel® Core™ i7-8665UE, (b) Intel® Xeon® E3 1268L v5, and (c) Intel® Xeon® Gold 6138 processors. The toolkit improved inference speed by 19% and 20% on the Intel Core i7 and Intel Xeon E3 processors, respectively. On the Intel Xeon Gold 6138 processor, the toolkit optimizations gave a 3x speedup in inference.[1]

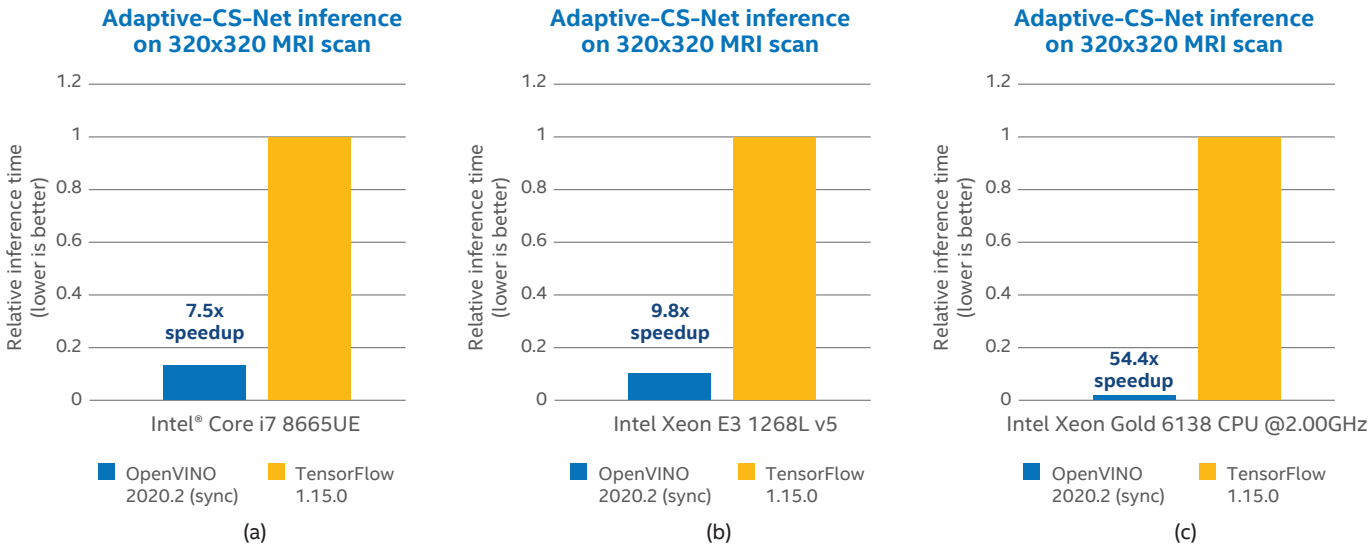

Figure 13. Comparing the unoptimized TensorFlow 1.15.0 (orange) versus OpenVINO™ toolkit 2020.2 (blue) time to process the Adaptive-CS-Net model with a 320x320 input image on (a) Intel® Core™ i7-8665UE, (b) Intel® Xeon® E3 1268L v5, and (c) Intel® Xeon® Gold 6138 processors. The OpenVINO toolkit optimizations sped up inference by 7.5x, 9.8x, and 54.4x on this hardware, respectively. It should be noted that the Adaptive-CS-Net model was a larger, more complex model than the open-sourced W-Net. This allowed the toolkit to find more optimizations than could be obtained from the W-Net topology.[1]

Figure 13 compares the relative inference latency for the Adaptive-CS-Net model between unoptimized TensorFlow version 1.15 and Intel Distribution of OpenVINO toolkit version 2020.2 on three different platforms: (a) Intel® Core™ i7-8665UE processor, (b) Intel® Xeon® E3 1268L v5 processor, and (c) Intel® Xeon® Gold 6138 processor. The toolkit's optimizations sped up inference by 7.5x and 9.8x on the Intel Core i7 and Intel Xeon E3 processors. The Intel Xeon Gold 6138 processor showed an impressive 54.4x speedup in inference via the toolkit optimizations.[1] The higher complexity and greater size of the proprietary Phillips Healthcare model gave the OpenVINO toolkit more places where it could obtain this significant optimization in speed.

Conclusion

The Intel Distribution of OpenVINO toolkit allows developers to deploy their deep learning models with improved inference on a variety of Intel hardware. Using the toolkit's custom extensions feature, Intel was able to speed up the compressed sensing workloads for Philips Healthcare by as much as 54x on an Intel Xeon Gold 6138 processor compared with unoptimized TensorFlow. Philips Healthcare was also able to leverage the Intel DevCloud for the Edge to quickly benchmark their CS models on Intel CPU, VPU, FPGA, and integrated GPU hardware. This platform helps Philips Healthcare when they plan and develop new products by allowing them to easily assess various deep learning pipeline parameters such as performance, price, power, and form factors for their designs.

[1] See backup for configuration details. For more complete information about performance and benchmark results, visit www.intel.com/benchmarks. Refer to https://software.intel.com/articles/optimization-notice for more information regarding performance and optimization choices in Intel software products.

References

- Beck A and Teboulle M, "A Fast, Iterative Shrinkage-Thresholding Algorithm for Linear Inverse Problems" SIAM J. Imaging Sciences, 2(1), pp. 183-202. 2009.
- Candès EJ and Tao T, "Near-optimal signal recovery from random projections: Universal encoding strategies," Applied and Computational Mathematics, Calif. Inst. Technol., Tech. Rep., 2004.
- Chen GH, Tang J, and Leng S, "Prior image constrained compressed sensing (PICCS): A method to accurately reconstruct dynamic CT images from highly undersampled projection data sets" Medical Physics Letters 35(2):660-663, Feb 2008.
- Donoho D, "Compressed Sensing" IEEE Transactions on Information Theory, 52(4):1289-1306, Apr 2006.
- Geerts-Ossevoort L, de Weerdt E, Duijndam A, van IJperen G, Peeters H, Doneva M, Nijenhuis M and Huang A, "Compressed SENSE", https://philipsproductcontent.blob.core.windows.net/assets/20180109/619119731f2a42c4acd4a863008a46c7.pdf, accessed 12 APR 2020.
- Graff CG and Sidky EY, "Compressive sensing in medical imaging" Appl Opt. 54(8):C23-C44, Mar 2015.
- Jin KH, McCann MT, Froustey E, and Unser M, "Deep convolutional neural network for inverse problems in imaging," IEEE Transactions on Image Processing, vol. 26, no. 9, pp. 4509–4522, 2017

- Lustig M, Donoho D, Pauly JM, "Sparse MRI: The Application of Compressed Sensing for Rapid MR Imaging" Magnetic Resonance in Medicine 58:1182-1195, 2007.
- Pezzotti N, Yousefi S, Elmahdy M, Gemert J, Schülke C, Doneva M, Nielsen T, Kastryulin S, Lelieveldt B, van Osch M, Weerdt E, Staring M. "An Adaptive Intelligence Algorithm for Undersampled Knee MRI Reconstruction: Application to the 2019 fastMRI Challenge". https://arxiv.org/abs/2004.07339, 2020, last accessed: 1 MAY 2020
- Philips Healthcare, "Philips receives U.S. FDA 510(k) clearance for its Ingenia Elition MR solution", https://www.philips.com/a-w/about/news/archive/standard/news/press/2018/20180605-philips-receives-us-fda-510k-clearance-for-its-ingenia-elition-mr-solution.html, June 5, 2018. Accessed online Mar 19, 2020
- Ronneberger O, Fischer P, and Brox T, "U-net: Convolutional networks for biomedical image segmentation," in International Conference on Medical image computing and computer-assisted intervention. Springer, 2015, pp. 234–241.
- Souza R, Lucena O, Garrafa J, Gobbi D, Saluzzi M, Appenzeller S, Rittner L, Frayne R, and Lotufo R, "An open, multi-vendor, multi-field-strength brain MR dataset and analysis of publicly available skull stripping methods agreement."NeuroImage 170 (2018): 482-494.
- Souza R and Frayne R, "A Hybrid Frequency-domain/Image-domain Deep Network for Magnetic Resonance Image Reconstruction", arXiv preprint, 20 October 2018 https://arxiv.org/abs/1810.12473
- Tibshirani R, Regression shrinkage and selection via the lasso. Journal of the Royal Statistical Society. Series B (Methodological), pp. 267–288, 1996.
- Wang F, Zhang J, Sun G, and Geng T, "Iterative Forward-Backward Pursuit Algorithm for Compressed Sensing" J. Electrical and Computer Engineering, May 2016.
- Yang G, Yu S, Dong H, Slabaugh G, Dragotti PL, Ye X, Liu F, Arridge S, Keegan J, Guo Y, "DAGAN: Deep de-aliasing generative adversarial networks for fast compressed sensing MRI reconstruction," IEEE transactions on medical imaging, vol. 37, no. 6, pp. 1310–1321, 2018.
- Zbontar J, Knoll F, Sriram A, Muckley MJ, Bruno M, Defazio A, Parente M, Geras KJ, Katsnelson J, Chandarana H, Zhang Z, Drozdzal M, Romero A, Rabbat MG, Vincent P, Pinkerton J, Wang D, Yakubova N, Owens E, Zitnick CL, Recht MP, Sodickson DK, and Lui YW. "fastMRI: An Open Dataset and Benchmarks for Accelerated MRI." ArXiv, abs/1811.08839, 2019, last accessed 1 MAY 2020.
- Zhang J and Ghanem B, "ISTA-Net: Interpretable Optimization-Inspired Deep Network for Image Compressive Sensing" https://arxiv.org/pdf/1706.07929.pdf 18 JUN 2018

Backup: System Configurations

Testing completed April 2020 by Intel using the Intel DevCloud for Edge.

System	Intel® Core™ i7-8665UE Processor	Intel® Xeon® Gold 6138 Processor	Intel Xeon E3 1268L v5 Processor
CPU	Intel Core i7-8665UE processor	Intel Xeon Gold 6138 processor	Intel Xeon E3-1268Lv5 processor
Sockets / Physical cores	1 / 4	2 / 20	1 / 4
Intel Hyper-Threading Technology / Turbo Setting	ON / ON	ON / n/a	ON / ON
Memory	16 GB	192 GB	32 GB
OS	Ubuntu 18.04.2 LTS	Ubuntu 18.04.2 LTS	Ubuntu 18.04.2 LTS
Kernel	Linux 4.15.0-96	Linux 4.15.0-88	Linux 4.15.0-96
Software	Intel® Distribution of OpenVINO™ toolkit 2020.2	Intel Distribution of OpenVINO toolkit 2020.2	Intel Distribution of OpenVINO toolkit 2020.2
Test Date	April 28, 2020	April 28, 2020	April 28, 2020
Microcode	0xca	0x2000069	0xd6
Precision and Batch Size	FP32, Batch: 1	FP32, Batch: 1	FP32, Batch: 1
Spectre/Meltdown Mitigation	True	True	True

0620/EH/MESH/PDF ♻ Please Recycle 338355-001US