CSDS 451: High-Performance Systems in AI Project

Project Report

Academic Term: Fall 2023
Project Title: Type 1: SYCL-based parallelization on CNN based AI models
Supervisor: Sanmukh kupanagari (sxk1942@case.edu)
Student Name: Sanjit Madhavan, Veerendra Varma Mavulate
Team No: 2
Student Email: sxm1828@case.edu, vxm262@case.edu

Objective: Type #1 (sycl-based):

• Take a CNN-based Machine Learning Model
• Calculate the inference time of the model using PyTorch
• Implement each convolution layer using one of the convolution algorithms discussed in class using one API
• Convert the layer to the correct matrix dimensions
• Use your PA2 implementation of blocked matrix multiplication and Intel MKL gemm based matrix Multiplication
• Compare the inference times of pytorch, blocked mm with the best parallelism parameter, and mkl gemm-based MM

Motivation: To compare the inference times of the CNN model given to find the best parallelism parameters using Block matrix multiplication and GEMM-based models available in one API environment.

1) CNN model being used – Inception-v4 - a convolutional neural network architecture that builds on previous iterations of the Inception family by simplifying the architecture and using more inception modules than Inception-v3.
2) The algorithm to be used is the Kn2Row algorithm, used on the convolutional layer before parallelizing the operation using Matrix Multiplication data parallelizing techniques. For block matrix multiplication we can change the no workers as a measure of parameter for comparing the performance.

Platforms Used : - intel oneAPI DPC++

Design summary:

1) **Model: Inception-V4**

Using PyTorch, a well-liked deep learning framework, the Inception v4 neural network model is implemented in full by the script. The script starts by loading the necessary libraries for deep learning tasks, such as the time measurement and basic mathematics libraries, as well as the PyTorch modules for neural network layers and functions. The custom autograd function defined in Kn2row2dFunction, which suggests an optimisation or specialised operation in 2D space, maybe for performance enhancement, is a fundamental component of this solution.

The script defines the Inception v4 network's architecture using a number of classes. These consist of InceptionStem, which is the network's starting section, BasicConv2d for basic convolutional layers, and InceptionA, B, and C, which are distinct Inception modules with parallel convolutional paths of varied kernel sizes. Reduction blocks are probably implemented by the ReductionA and ReductionB classes in order to decrease the spatial dimensions of the network. The InceptionV4 class, which stands for the complete Inception v4 architecture, is the result of these parts coming together. This thorough structure implies that the script may be used for more than just network definition; it may also be used to train or apply the network to certain tasks by utilising the special features of the Inception architecture, which is renowned for its effectiveness and precision in image recognition tasks.

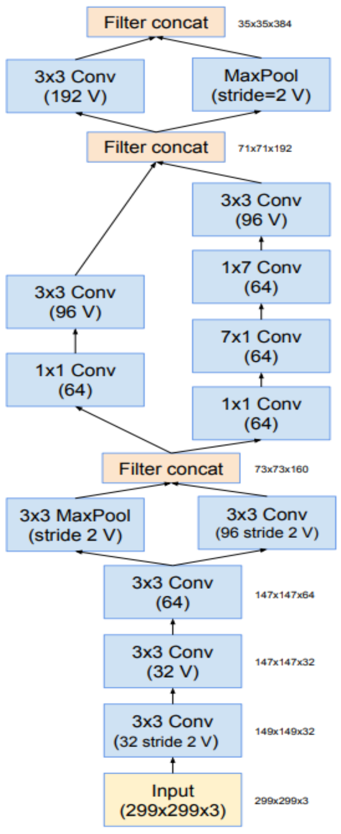


Figure. Schema for the Inception-v4 and Inception-ResNet-v2 networks

```python
class InceptionStem(nn.Module):
    def __init__(self, in_channels):
        super(InceptionStem, self).__init__()
        self.conv1 = BasicConv2d(in_channels, 32, kernel_size=3, stride=2)
        self.conv2 = BasicConv2d(32, 32, kernel_size=3)
        self.conv3 = BasicConv2d(32, 64, kernel_size=3, padding=1)
        self.branch1 = nn.MaxPool2d(3, stride=2)
        self.branch2 = nn.Sequential(
            BasicConv2d(64, 96, kernel_size=3, stride=2),
            #BasicConv2d(96, 96, kernel_size=3),
        )
        self.branch3 = nn.Sequential(
            BasicConv2d(160, 64, kernel_size=1, ),
            BasicConv2d(64, 96, kernel_size=3),
        )
        self.branch4 = nn.Sequential(
            #nn.AvgPool2d(3, stride=2),
            BasicConv2d(160, 64, kernel_size=1),
            BasicConv2d(64, 96, kernel_size=3),
        )

        self.branch5 = nn.MaxPool2d(2, stride=2)
        self.branch6 = nn.Sequential(
            BasicConv2d(192, 192, kernel_size=3, stride=2),
        )


    def forward(self, x):
        x = self.conv1(x)
        x = self.conv2(x)
        x = self.conv3(x)
        x1 = self.branch1(x)
        x2 = self.branch2(x)
        x3 = self.branch3(torch.cat((x1, x2), 1))
        x4 = self.branch4(torch.cat((x1, x2), 1))
        x5 = self.branch5(torch.cat((x3, x4), 1))
        x6 = self.branch6(torch.cat((x3, x4), 1))

        #print(torch.cat((x5, x6), 1).shape)
        return torch.cat((x5, x6), 1)
```

Performing complex feature extraction on the input tensor, the InceptionStem class encapsulates the first convolutional stem of the inception module. A parallel processing pathway is provided by branches with MaxPool2d and Sequential convolutions, while the input's channel depth is expanded by sequential BasicConv2d layers with different kernel sizes and strides. The feature map is diversified by concatenations of intermediate outputs before to the succeeding layers. The final concatenation combines processed streams into a multi-scaled feature-rich tensor, ready for the next sets of inception layers.

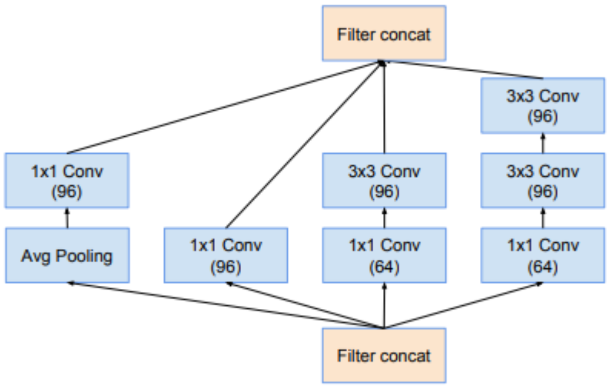


Figure. InceptionA Block

```
class InceptionA(nn.Module):
    def __init__(self, in_channels):
        super(InceptionA, self).__init__()
        self.branch1 = BasicConv2d(in_channels, 96, kernel_size=1)
        self.branch2 = nn.Sequential(
            BasicConv2d(in_channels, 64, kernel_size=1),
            BasicConv2d(64, 96, kernel_size=3, padding=1),
        )
        self.branch3 = nn.Sequential(
            BasicConv2d(in_channels, 64, kernel_size=1),
            BasicConv2d(64, 96, kernel_size=3, padding=1),
            BasicConv2d(96, 96, kernel_size=3, padding=1),
        )
        self.branch4 = nn.Sequential(
            nn.AvgPool2d(3, stride=1, padding=1),
            BasicConv2d(in_channels, 96, kernel_size=1),
        )


    def forward(self, x):
        x1 = self.branch1(x)
        x2 = self.branch2(x)
        x3 = self.branch3(x)
        x4 = self.branch4(x)
        return torch.cat((x1, x2, x3, x4), 1)
```

A straight 1x1 BasicConv2d, two-layered convolutions augmenting to 3x3 with padding for feature complexity, and an AvgPool2d branch followed by a 1x1 convolution for spatial feature smoothing are the four parallel branches that make up the InceptionA class's inception module. After that, in the forward pass, these branches are concatenated along the channel dimension to produce a composite feature map that incorporates depth and multi-scale spatial data from the input tensor.

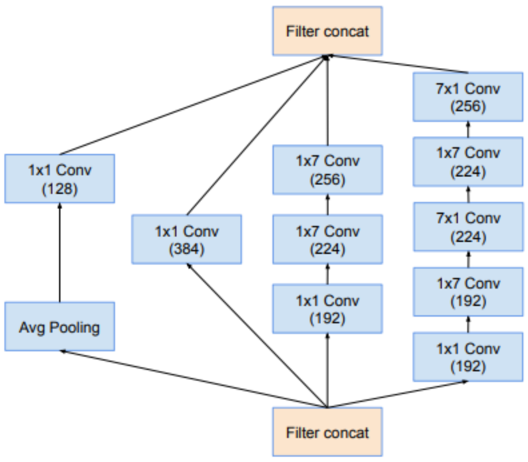


Figure. InceptionB Block

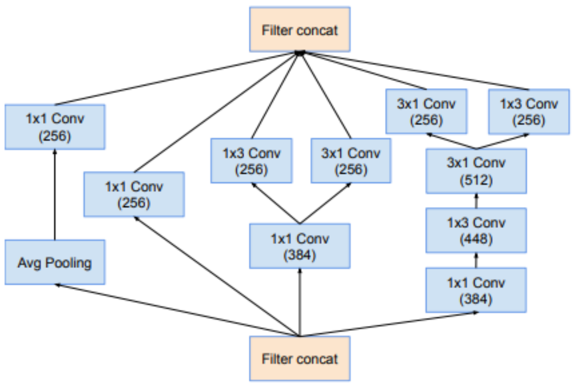


Figure. InceptionC Block

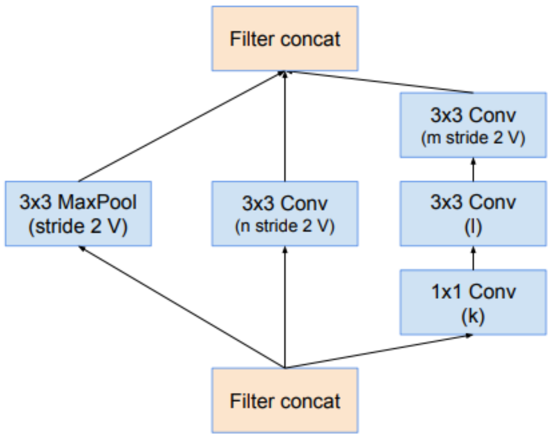


Figure. ReductionA Block

```
class ReductionA(nn.Module):
    def __init__(self, in_channels):
        super(ReductionA, self).__init__()
        self.branch1 = BasicConv2d(in_channels, 384, kernel_size=3, stride=2)
        self.branch2 = nn.Sequential(
            BasicConv2d(in_channels, 192, kernel_size=1),
            BasicConv2d(192, 224, kernel_size=3, padding=1),
            BasicConv2d(224, 256, kernel_size=3, stride=2),
        )
        self.branch3 = nn.MaxPool2d(3, stride=2)
    def forward(self, x):
        x1 = self.branch1(x)
        x2 = self.branch2(x)
        x3 = self.branch3(x)

        #print("Reduction A")
        #print(torch.cat((x1, x2, x3), 1).shape)
        return torch.cat((x1, x2, x3), 1)
```

The ReductionA class implements a reduction block with three branches: a BasicConv2d with a 3x3 kernel and stride of 2 for dimensionality reduction, a Sequential block with 1x1 and 3x3 convolutions (the latter with stride 2 and padding to maintain spatial dimensions), and a 3x3 MaxPool2d with stride 2 for pooling. In the forward pass, these branches work in tandem to compress spatial dimensions and expand feature depth, with their outputs concatenated along the channel dimension.

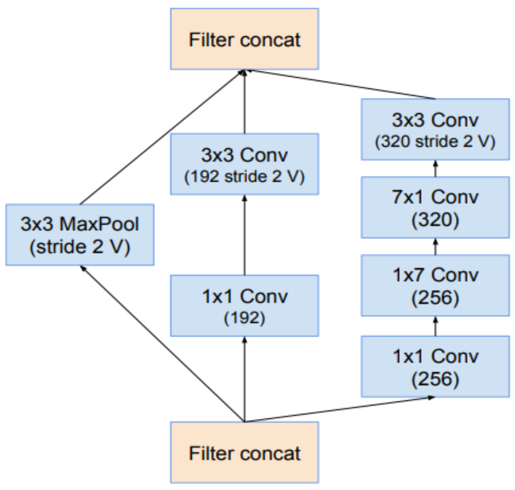
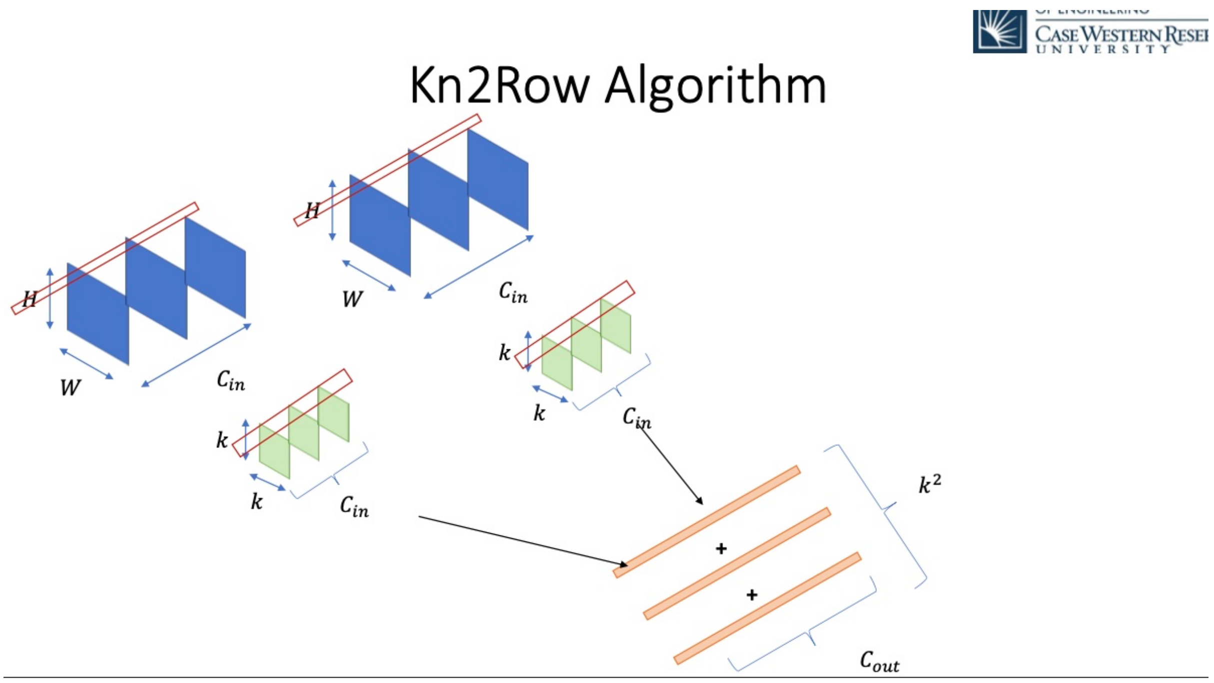


Figure. ReductionB Block

```python
class ReductionB(nn.Module):
    def __init__(self, in_channels):
        super(ReductionB, self).__init__()
        self.branch1 = nn.Sequential(
            BasicConv2d(in_channels, 192, kernel_size=1),
            BasicConv2d(192, 192, kernel_size=3, stride=2),
        )
        self.branch2 = nn.Sequential(
            BasicConv2d(in_channels, 256, kernel_size=1),
            BasicConv2d(256, 256, kernel_size=(1, 7), padding=(0, 3)),
            BasicConv2d(256, 320, kernel_size=(7, 1), padding=(3, 0)),
            BasicConv2d(320, 320, kernel_size=3, stride=2),
        )
        self.branch3 = nn.MaxPool2d(3, stride=2)
    def forward(self, x):
        x1 = self.branch1(x)
        x2 = self.branch2(x)
        x3 = self.branch3(x)

        #print("Reduction B")
        #print(torch.cat((x1, x2, x3), 1).shape)
        return torch.cat((x1, x2, x3), 1)
```

The ReductionB class in PyTorch defines a reduction block that channels input through three branches: the first performs a dimensionality reduction using a 1x1 convolution followed by a 3x3 BasicConv2d with stride 2; the second branch sequences through a 1x1 convolution, a pair of asymmetric convolutions with 1x7 and 7x1 kernels applying cross-channel and spatial feature extraction, and concludes with a 3x3 stride-2 convolution; the third executes a 3x3 MaxPool2d with stride 2. These branches process and downsample the input tensor, which are then concatenated to construct a compacted feature map in the forward pass.

2) Algorithm – CNN Kn2row algorithm



for the convolution operation, we will be doing the Kn2row variant with the inception v4 architecture.

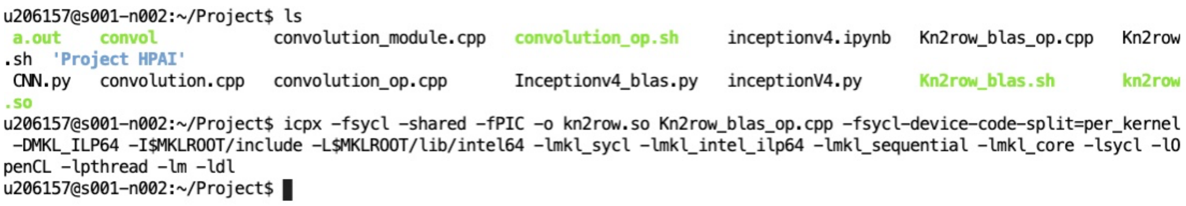
Module used : - Sycl/ c++ implementation

```
#include<iostream>
#include<cstring>
#include<chrono>
#include <sycl/sycl.hpp>
#include "oneapi/mkl/blas.hpp"
```

Note :- please use this compile code so that the .so file can be generated.

icpx -fsycl -shared -fPIC -o kn2row.so Kn2row_blas_op.cpp -fsycl-device-code-split=per_kernel -DMKL_ILP64 -I$MKLROOT/include -L$MKLROOT/lib/intel64 -lmkl_sycl -lmkl_intel_ilp64 -lmkl_sequential -lmkl_core -lsycl -lOpenCL -lpthread -lm -ldl

we will be creating a shared library (.so file) so that we can call the sycl implementation into the python Inception V4 code. The .so file can be found in the dir of the compiled file.



```
u206157@s001-n002:~/Project$ ls
 a.out     convol            convolution_module.cpp    convolution_op.sh     inceptionv4.ipynb    Kn2row_blas_op.cpp    Kn2row
.sh  'Project HPAI'
 CNN.py    convolution.cpp   convolution_op.cpp        Inceptionv4_blas.py   inceptionV4.py       Kn2row_blas.sh        kn2row
.so
u206157@s001-n002:~/Project$ icpx -fsycl -shared -fPIC -o kn2row.so Kn2row_blas_op.cpp -fsycl-device-code-split=per_kernel
 -DMKL_ILP64 -I$MKLROOT/include -L$MKLROOT/lib/intel64 -lmkl_sycl -lmkl_intel_ilp64 -lmkl_sequential -lmkl_core -lsycl -lO
penCL -lpthread -lm -ldl
u206157@s001-n002:~/Project$ █
```

We are using the terminal for compilation if an .sh file is being used please include this compile code

```
    //Multiplying the kernel and input matrices
    queue q(property::queue::enable_profiling{});

    float *matrixA = sycl::malloc_shared<float>(k_row * k_col * channels_out * channels, q);
    float *matrixB = sycl::malloc_shared<float>(channels * input_W, q);
    float *matrixC = sycl::malloc_shared<float>(k_row * k_col * channels_out * input_W, q);

    for(int i=0; i<k_row * k_col * channels_out; i++){
        for(int j=0; j<channels; j++){
            matrixA[i * channels + j] = kn2row_kernel_array[i][j];
        }
    }

    for(int i=0; i<channels; i++){
        for(int j=0; j<input_W; j++){
            matrixB[i * input_W + j] = kn2row_input_array[i][j];
        }
    }
```

```python
class Kn2row2d(nn.Module):
    def __init__(self, in_channels, out_channels, kernel_size, stride=1, padding=0, **kwargs):
        super(Kn2row2d, self).__init__()

        # check for kernel sizes for inceptionv4 architecture
        if type(kernel_size) is tuple:
            k_row = kernel_size[0]
            k_col = kernel_size[1]
        else:
            k_row = kernel_size
            k_col = kernel_size

        self.weight = nn.Parameter(torch.randn(out_channels, in_channels, k_row, k_col))
        self.bias = nn.Parameter(torch.randn(out_channels))
        self.stride = stride
        self.padding = padding
        self.out_channels = out_channels
```

Because Inception V4 has kernels with different dimensions we will be passing the kernels as row and col.

3) Parallelization techniques – GEMM

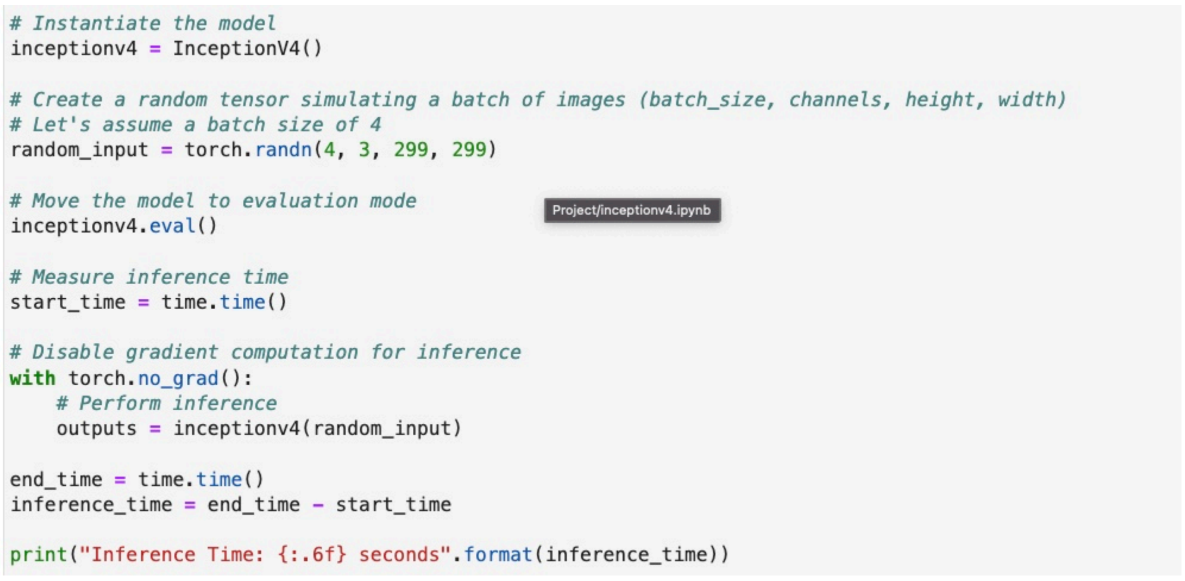
GEMM is a crucial operation in many numerical algorithms and applications, including linear algebra libraries, machine learning, and scientific computing. Efficient implementations of GEMM are essential for optimizing the performance of these applications, and various techniques, such as parallelization and optimization for specific hardware architectures, are often employed to accelerate GEMM operations.

```cpp
        float alpha = 1.0, beta = 1.0;

        mkl::transpose transA = mkl::transpose::nontrans;
        mkl::transpose transB = mkl::transpose::nontrans;
        int ldA = channels, ldB = input_W, ldC = input_W;

        // Define a DPC++ kernel for matrix multiplication
        gemm_done = mkl::blas::row_major::gemm(q, transA, transB, k_row * k_col * channels_out, input_W,
channels, alpha, matrixA, ldA, matrixB, ldB, beta, matrixC, ldC, gemm_dependencies);

        float *productMatrix[channels_out * k_row * k_col];
        for (int i=0; i <(k_row * k_col * channels_out); i++ ){
            productMatrix[i] = new float[input_W];
            for(int j=0; j<input_W; j++ ){
                productMatrix[i][j] = matrixC[i * (input_W) + j];
            }
        }
```

4) Inference time comparison

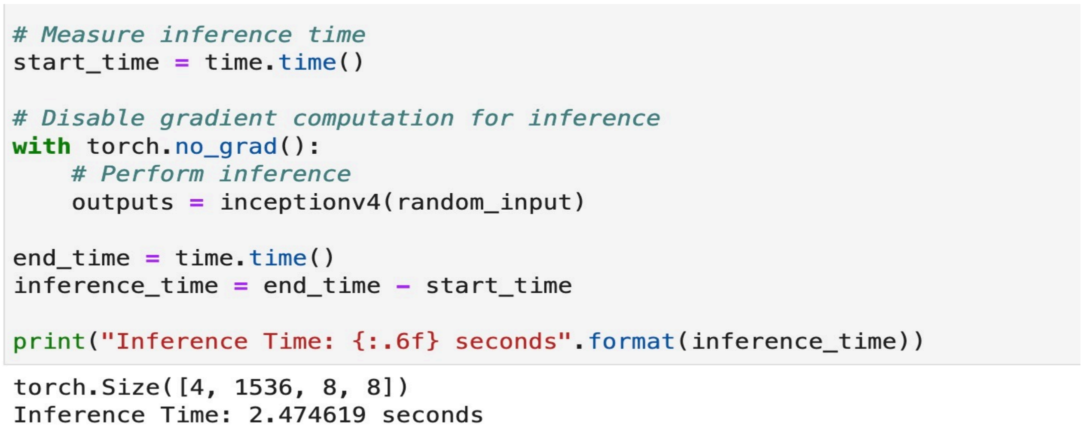
For inference time we will be timing the time taken for the forward pass of a model

Input used (Channels = 3, H = 299, W = 299)



```python
# Instantiate the model
inceptionv4 = InceptionV4()

# Create a random tensor simulating a batch of images (batch_size, channels, height, width)
# Let's assume a batch size of 4
random_input = torch.randn(4, 3, 299, 299)

# Move the model to evaluation mode
inceptionv4.eval()

# Measure inference time
start_time = time.time()

# Disable gradient computation for inference
with torch.no_grad():
    # Perform inference
    outputs = inceptionv4(random_input)

end_time = time.time()
inference_time = end_time - start_time

print("Inference Time: {:.6f} seconds".format(inference_time))
```

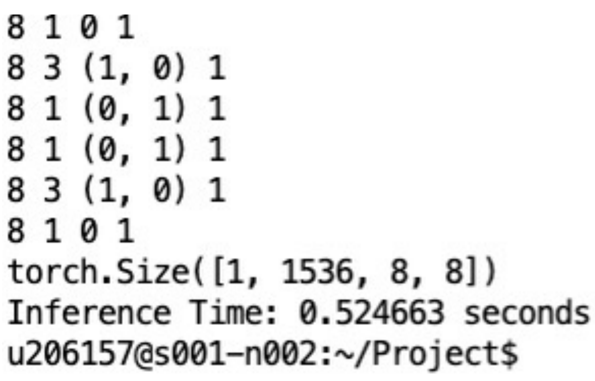
For naïve Inception V4 it took 2.474619 sec



```python
# Measure inference time
start_time = time.time()

# Disable gradient computation for inference
with torch.no_grad():
    # Perform inference
    outputs = inceptionv4(random_input)

end_time = time.time()
inference_time = end_time - start_time

print("Inference Time: {:.6f} seconds".format(inference_time))
```

```
torch.Size([4, 1536, 8, 8])
Inference Time: 2.474619 seconds
```

For Inception v4 Kn2row blas implementation it took 0.524663 sec



```
8 1 0 1
8 3 (1, 0) 1
8 1 (0, 1) 1
8 1 (0, 1) 1
8 3 (1, 0) 1
8 1 0 1
torch.Size([1, 1536, 8, 8])
Inference Time: 0.524663 seconds
u206157@s001-n002:~/Project$
```