

Parallelizing the Interpolation between Latent Space of Autoencoder Networks

Tansin Jahan
School of Computer Science
Carleton University, Ottawa, Canada
tansinjahan@cmail.carleton.ca

Overview

- Related work
- Project overview
- Convolutional Neural Network
- Algorithm
- Parallelism
- CUDA parallel programming framework
- Introduction to Numba
- Experiment
- Analysis of results
- Limitation and Challenges
- Questions

Related Work

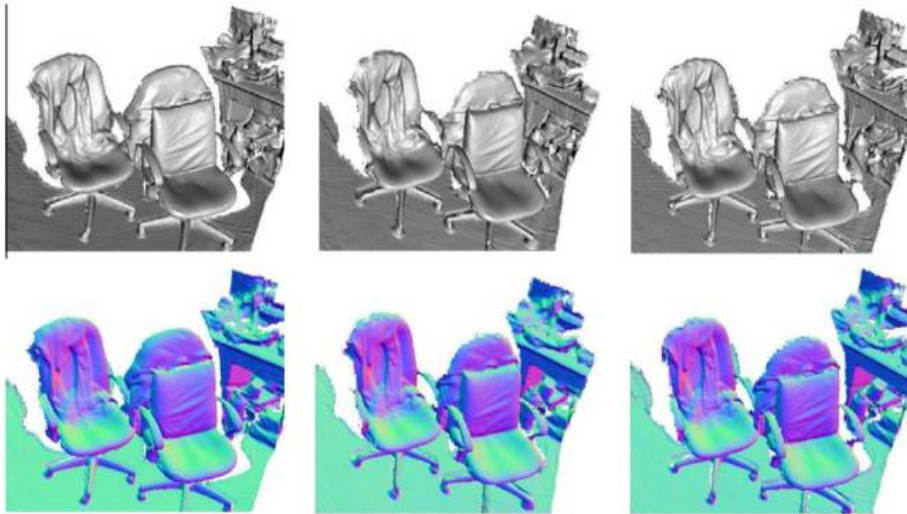
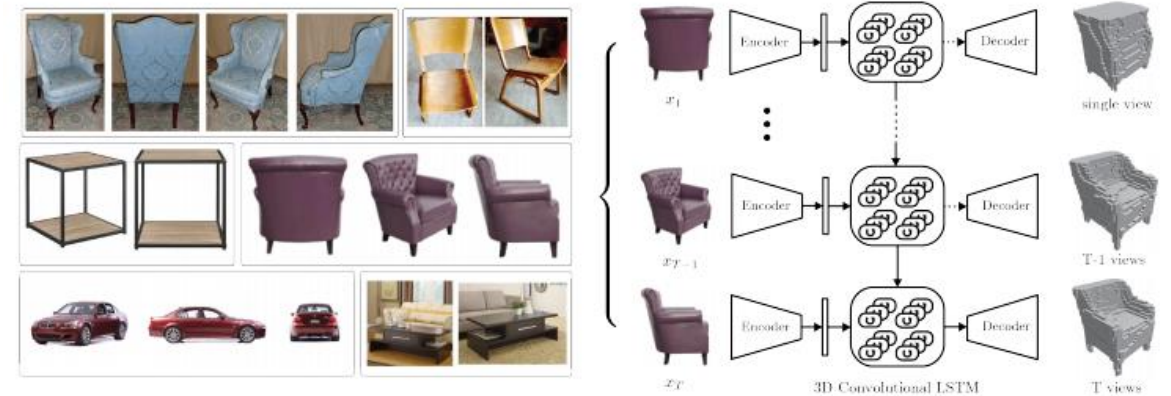


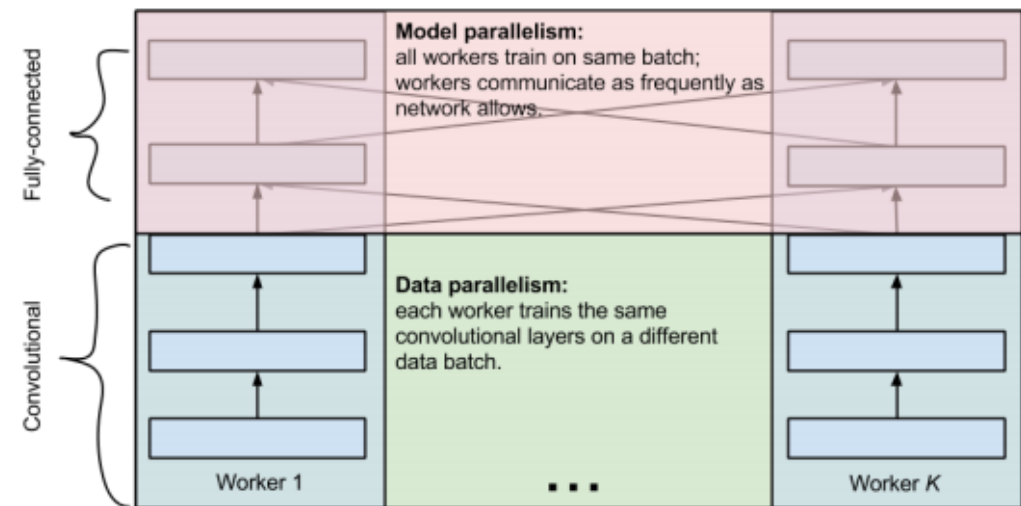
Fig. 6. Reconstruction of the "chairs" scene. Top row are the Phong shading results and bottom are normal maps. The first column are KF_{312} , the middle column are OF_9 and the third column are OF_{10} .

Octree-based fusion for realtime 3D reconstruction (2012)
[Ming Zeng, Fukai Zhao, Jiayang Zheng, Xinguo Liu]



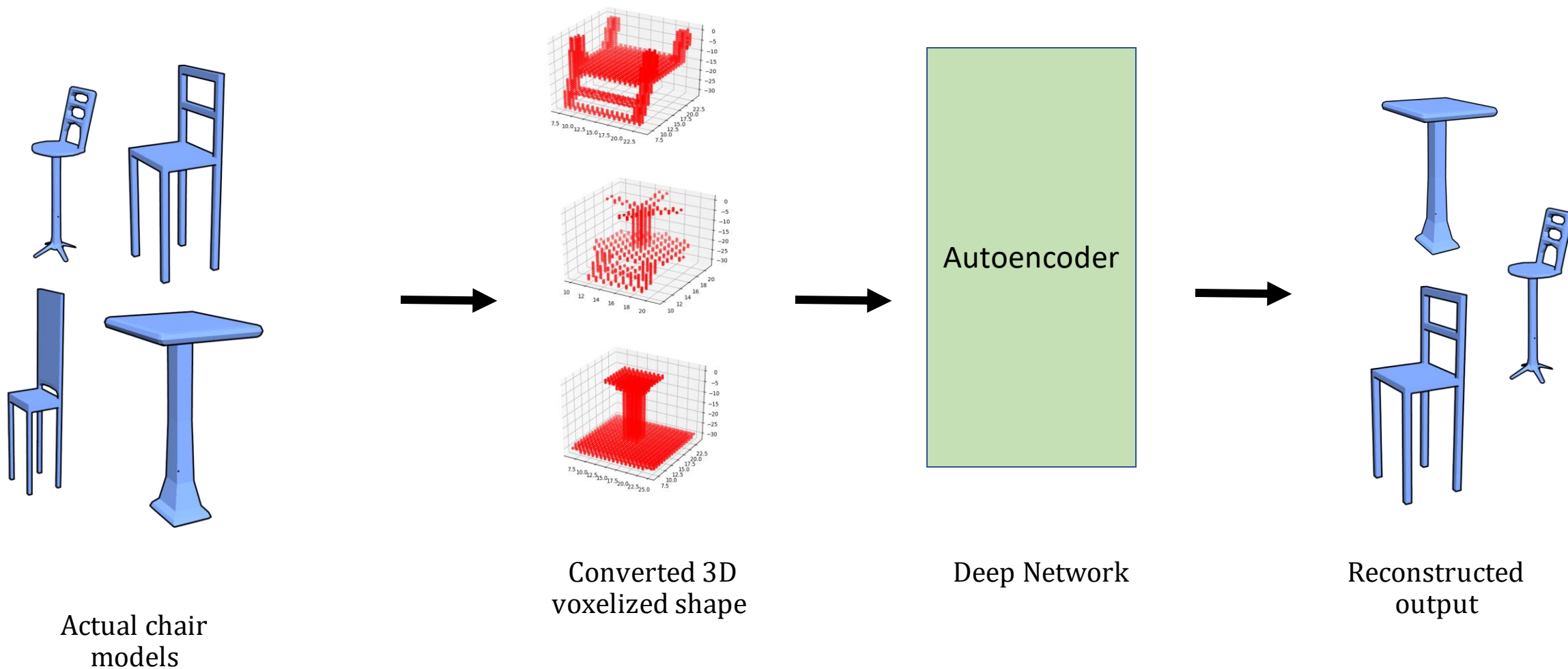
(a) Images of objects we wish to reconstruct (b) Overview of the network

3D-R2N2: A Unified Approach for Single and Multi-view 3D Object Reconstruction (2016)
[Christopher B. Choy, Danfei Xu, JunYoung Gwak, Kevin Chen, Silvio Savarese]

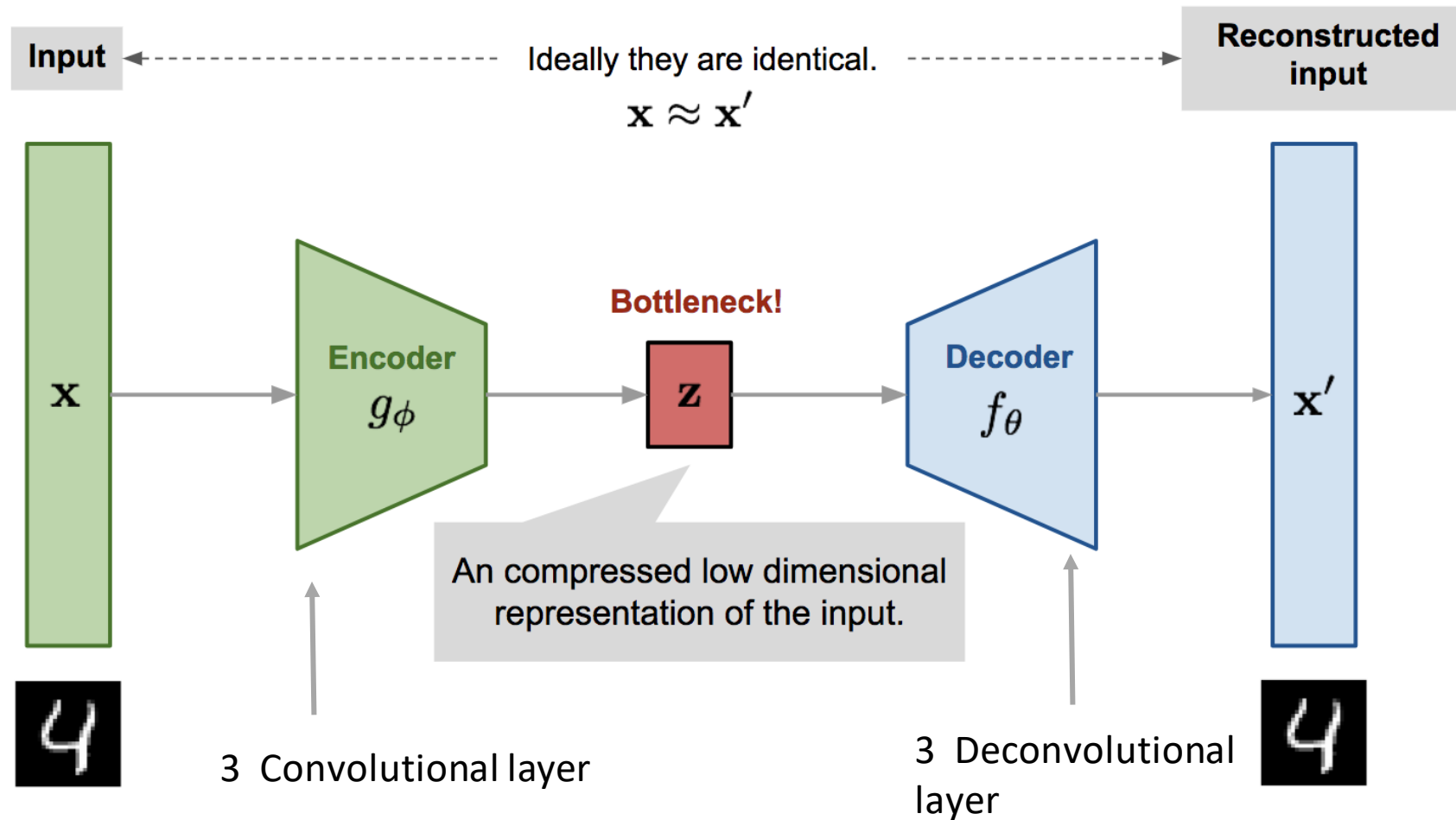


One weird trick for parallelizing convolutional neural networks (2014)
[Alex Krizhevsky]

Project Overview



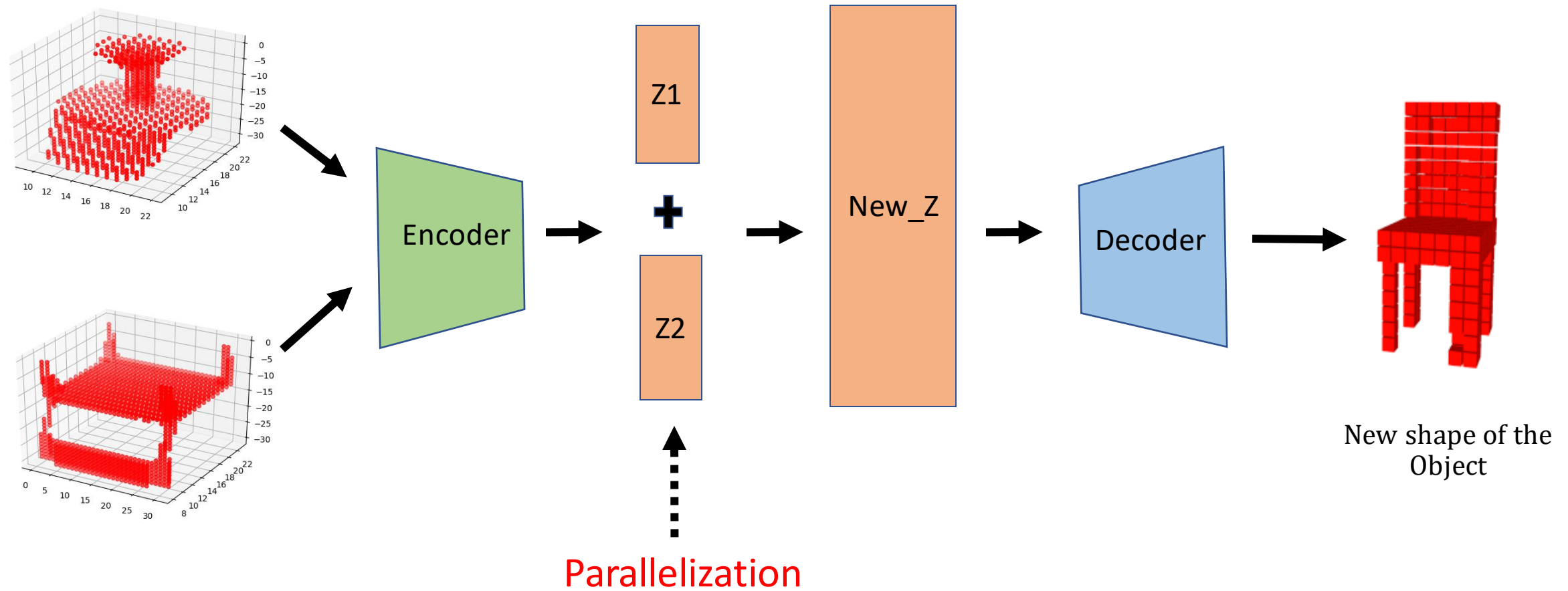
Autoencoder



Autoencoder

```
def autoencoder(inputs):  
    # encoder  
    # 32 x 32 x 32 x 1 -> 16 x 16 x 16 x 32  
    # 16 x 16 x 16 x 32 -> 8 x 8 x 8 x 16  
    # 8 x 8 x 8 x 16 -> 2 x 2 x 2 x 8  
    net = layers.conv3d(inputs, 32, [5, 5, 5], stride=2, padding='SAME')  
    net = layers.conv3d(net, 16, [5, 5, 5], stride=2, padding='SAME')  
    net = layers.conv3d(net, 8, [5, 5, 5], stride=4, padding='SAME')  
  
    latent_space = net  
  
    # decoder  
    # 2 x 2 x 2 x 8 -> 8 x 8 x 8 x 16  
    # 8 x 8 x 8 x 16 -> 16 x 16 x 16 x 32  
    # 16 x 16 x 16 x 32 -> 32 x 32 x 32 x 1  
    net = layers.conv3d_transpose(net, 16, [5, 5, 5], stride=4, padding='SAME')  
    net = layers.conv3d_transpose(net, 32, [5, 5, 5], stride=2, padding='SAME')  
    net = layers.conv3d_transpose(net, 1, [5, 5, 5], stride=2,  
padding='SAME', activation_fn=tf.nn.tanh)  
  
    return latent_space, net
```

Interpolation in latent space



Convolutional Neural Network

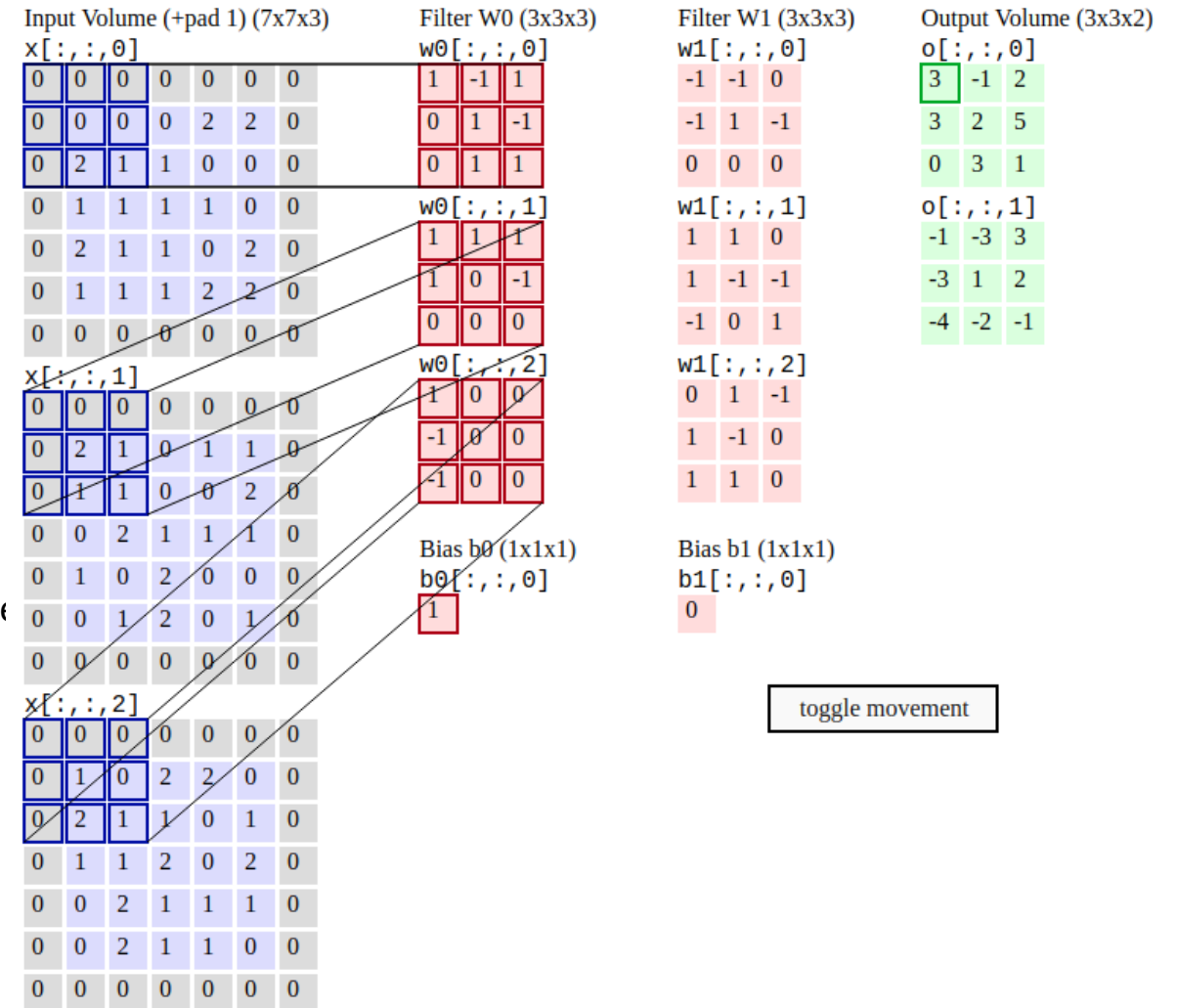
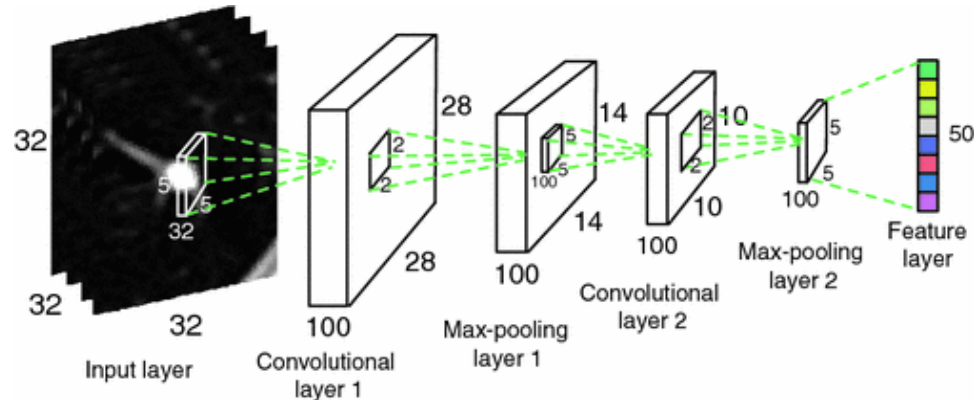
A simple Convolutional Network is a sequence of layers: **Convolutional Layer**, **Pooling Layer**, **Fully-Connected Layer**

for example -

[INPUT - CONV - RELU - POOL - FC]

Convolution Layer -

- Takes input with dimension [wxhxd]
- Requires number of filters, F (e.g. [3x3x3])
- Define stride, S and other hyperparameters
- Produce output volume [wxhxd] using F, S and other hyperparameters



Algorithm

1. for i = first input volume
2. for j = second input volume
3. $Z1$ = z vector for i
4. $Z2$ = z vector for j
5. for t = 0 to 1
6. $\text{new_z} = (1-t) \times Z1 + t \times Z2$
7. run decoder network
8. $t = t + 0.1$

i = all 50 rows from the input file

j = all 50 rows from the input file except i

$$Z1 = [2 \times 2 \times 2 \times 8] = 64$$

$$Z2 = [2 \times 2 \times 2 \times 8] = 64$$

Why parallelism

- GPU utilization for heavy computation of data
- Increase performance (linear speedup)

Parallel Languages 

Cilk
OpenCL
MPI
CUDA

```
Untitled1.ipynb
File Edit View Insert Runtime Tools Help
CODE TEXT CELL CELL
import numpy as np
import timeit

config = tf.ConfigProto()
config.gpu_options.allow_growth = True

X = np.random.rand(10000,3)
Y = np.random.rand(3,10000)

with tf.device('/cpu:0'):
    def multiply_two_array(X,Y):
        return np.dot(X,Y)

    result = multiply_two_array(X,Y)
    cpu_result = tf.convert_to_tensor(result, np.float32)
    print("this is the CPU result", cpu_result)

with tf.device('/gpu:0'):
    def multiply_two_array(X,Y):
        return np.dot(X,Y)

    result = multiply_two_array(X,Y)
    gpu_result = tf.convert_to_tensor(result, np.float32)
    print("this is the GPU result", gpu_result)

sess = tf.Session(config=config)
# Test execution once to detect errors early.
try:
    sess.run(tf.global_variables_initializer())
except tf.errors.InvalidArgumentError:
    print(
        '\n\nThis error most likely means that this notebook is not '
        'configured to use a GPU. Change this in Notebook Settings via the '
        'command palette (cmd/ctrl-shift-P) or the Edit menu.\n\n')
    raise

def cpu():
    sess.run(cpu_result)

def gpu():
    sess.run(gpu_result)

print('CPU (s):')
cpu_time = timeit.timeit('cpu()', number=10, setup="from __main__ import cpu")
print(cpu_time)
print('GPU (s):')
gpu_time = timeit.timeit('gpu()', number=10, setup="from __main__ import gpu")
print(gpu_time)
print('GPU speedup over CPU: {}'.format(int(cpu_time/gpu_time)))

this is the CPU result Tensor("Const_4:0", shape=(10000, 10000), dtype=float32, device=/device:CPU:0)
this is the GPU result Tensor("Const_5:0", shape=(10000, 10000), dtype=float32, device=/device:GPU:0)
CPU (s):
1.4686352919998171
GPU (s):
0.927858432999983
GPU speedup over CPU: 1x
```

```
Untitled1.ipynb
File Edit View Insert Runtime Tools Help
CODE TEXT CELL CELL
import numpy as np
import timeit

config = tf.ConfigProto()
config.gpu_options.allow_growth = True

X = np.random.rand(100,3)
Y = np.random.rand(3,100)

with tf.device('/cpu:0'):
    def multiply_two_array(X,Y):
        return np.dot(X,Y)

    result = multiply_two_array(X,Y)
    cpu_result = tf.convert_to_tensor(result, np.float32)
    print("this is the CPU result", cpu_result)

with tf.device('/gpu:0'):
    def multiply_two_array(X,Y):
        return np.dot(X,Y)

    result = multiply_two_array(X,Y)
    gpu_result = tf.convert_to_tensor(result, np.float32)
    print("this is the GPU result", gpu_result)

sess = tf.Session(config=config)
# Test execution once to detect errors early.
try:
    sess.run(tf.global_variables_initializer())
except tf.errors.InvalidArgumentError:
    print(
        '\n\nThis error most likely means that this notebook is not '
        'configured to use a GPU. Change this in Notebook Settings via the '
        'command palette (cmd/ctrl-shift-P) or the Edit menu.\n\n')
    raise

def cpu():
    sess.run(cpu_result)

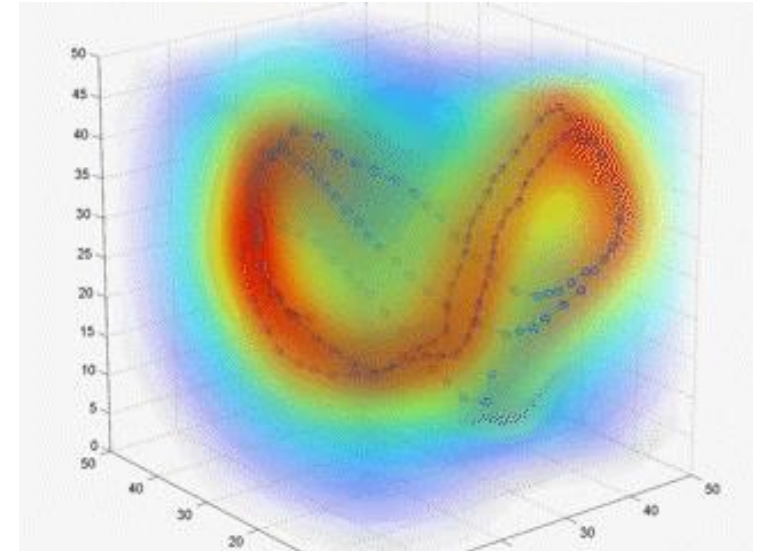
def gpu():
    sess.run(gpu_result)

print('CPU (s):')
cpu_time = timeit.timeit('cpu()', number=10, setup="from __main__ import cpu")
print(cpu_time)
print('GPU (s):')
gpu_time = timeit.timeit('gpu()', number=10, setup="from __main__ import gpu")
print(gpu_time)
print('GPU speedup over CPU: {}'.format(int(cpu_time/gpu_time)))

this is the CPU result Tensor("Const_2:0", shape=(100, 100), dtype=float32, device=/device:CPU:0)
this is the GPU result Tensor("Const_3:0", shape=(100, 100), dtype=float32, device=/device:GPU:0)
CPU (s):
0.004676589999917269
GPU (s):
0.00869353999996747
GPU speedup over CPU: 0x
```

Why Parallelization inside **Z** vector

- Lower dimension but maximum features
- Can be reshape easily
- Linearly grows with input data
- Involve significant amount of time while comparing with different combination of dataset

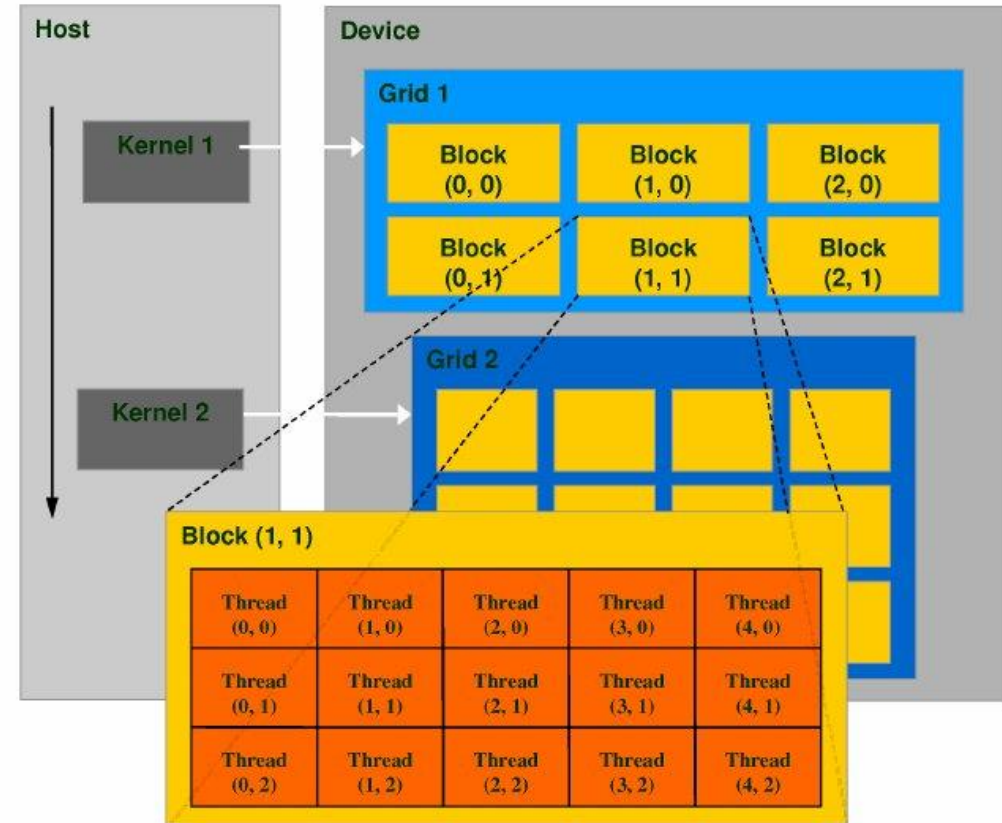


CUDA Parallel Programming Framework

- Host
- Device
- Kernel
- Threads
- Block
- Grid

CUDA supported languages

- C
- C++
- fortran
- Python



CUDA programming

(Introduction to Numba)

- Numba supports CUDA GPU programming
- Compiles Python code into CUDA kernels and device functions following the CUDA execution model.
- Numba makes it appear that the kernel has direct access to NumPy arrays

```
import numpy as np
from timeit import default_timer as timer
from numba import vectorize
```

Python decorators



```
@vectorize(["float32(float32,float32)", target='cuda'])
```

```
def vectorAdd(a,b):
    return a + b
```

```
def main():
    N = 3200000
```

```
A = np.ones(N, dtype = np.float32)
B = np.ones(N, dtype = np.float32)
C = np.zeros(N, dtype = np.float32)
```

```
start = timer()
C = vectorAdd(A,B)
vectoradd_time = timer() - start
```

```
print("C[:5] = " + str(C[:5]))
print("C[-5:] = " + str(C[-5:]))
```

```
print("Vector Add took %f seconds:" , vectoradd_time)
```

```
if __name__ == "__main__":
    main()
```

Numba Kernels

Kernel Declaration

```
from numba import cuda

@cuda.jit
def my_kernel(io_array):
    """
    Code for kernel.
    """
    # code here
```

Kernel Invocation

```
import numpy

# Create the data array - usually initialized some other way
data = numpy.ones(256)

# Set the number of threads in a block
threadsperblock = 32

# Calculate the number of thread blocks in the grid
blockspergrid = (data.size + (threadsperblock - 1)) // threadsperblock

# Now start the kernel
my_kernel[blockspergrid, threadsperblock](data)

# Print the result
print(data)
```

Project Implementation

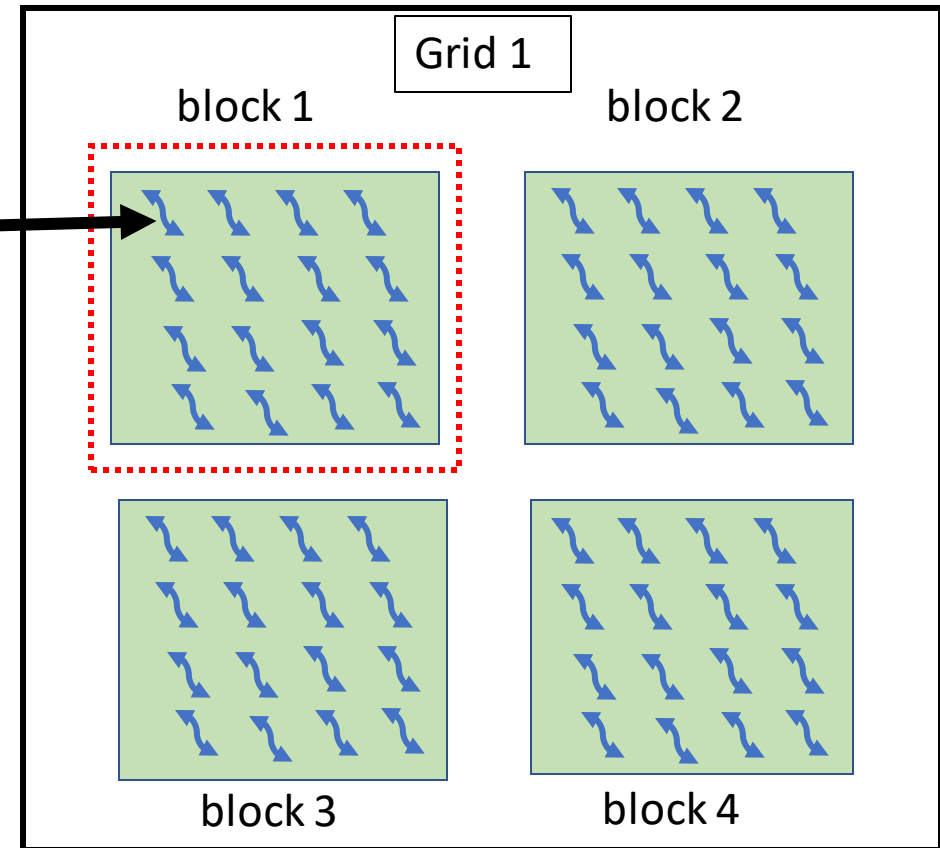
- Dataset -- > 50 shapes of chair (each with dimension [32 x 32 x 32])
- Epoch = 5
- Z vector = [2x2x2x8] -> reshape into [8, 8]
- Thread = [4 , 4]
- Blocks = [2, 2]

```
TPB = 4
l_space1 = np.reshape(l_space1, (8,8))
l_space2 = np.reshape(l_space2, (8,8))
print("lspace 1", l_space1)
print("lspace 2", l_space2)
dA = cuda.to_device(l_space1)
dB = cuda.to_device(l_space2)
c = np.empty_like(l_space2)
dC = cuda.to_device(c)

start = timer()
threadsperblock = (TPB, TPB)

blockspergrid_x = int(math.ceil(l_space1.shape[0] / threadsperblock[0]))
blockspergrid_y = int(math.ceil(l_space2.shape[1] / threadsperblock[1]))
blockspergrid = (blockspergrid_x, blockspergrid_y)
for t in range(0, 100):
    interpolationBetnLatentSpace[blockspergrid, threadsperblock](dA, dB,
dC, t)
```

threads



Implemented CUDA architecture for the project

Threads Index Calculation

@numba.cuda.jit

```
def interpolationBetnLatentSpace(A,B,C,t):  
    tx = cuda.threadIdx.x  
    ty = cuda.threadIdx.y  
    bx = cuda.blockIdx.x  
    by = cuda.blockIdx.y  
    bw = cuda.blockDim.x  
    bh = cuda.blockDim.y  
    row = tx + bx * bw  
    col = ty + by * bh  
  
    if row < C.shape[0] and col < C.shape[1]:  
        for k in range(A.shape[1]):  
            tmp = (1-t) * A[row, k] + t *  
B[row, k]  
            C[row, k] = tmp
```

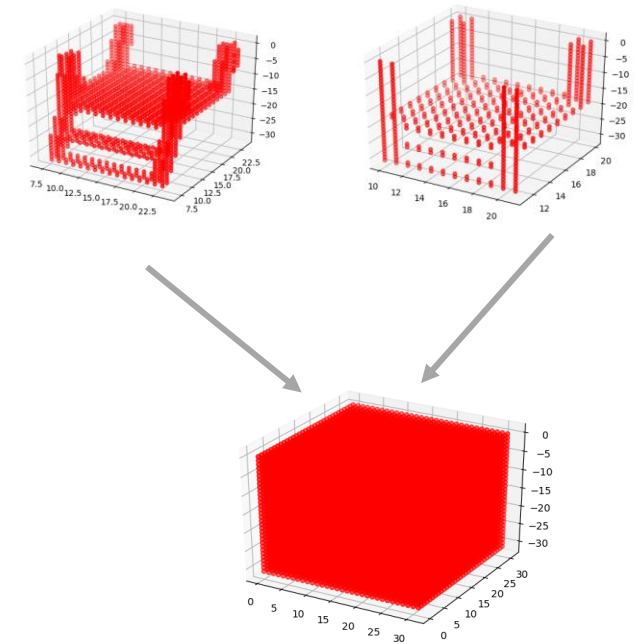
Global memory

@numba.cuda.jit

```
def interpolationBetnLatentSpace(A,B,C,t):  
    sA = cuda.shared.array(shape=(8, 8),  
dtype=float32)  
    sB = cuda.shared.array(shape=(8, 8),  
dtype=float32)  
    x, y = cuda.grid(2)  
    tx = cuda.threadIdx.x  
    ty = cuda.threadIdx.y  
    if x < C.shape[0] and y < C.shape[1]:  
        tmp = 0  
        for i in range(int(A.shape[1] / TPB)):  
            # Preload data into shared memory  
            sA[tx, ty] = A[x, ty + i * TPB]  
            sB[tx, ty] = B[tx + i * TPB, y]  
            cuda.syncthreads()  
            for k in range(8):  
                tmp = (1-t) * A[tx, k] + t * B[tx, k]  
            cuda.syncthreads()  
            C[tx, k] = tmp
```

Shared memory

Results



Output volume after
interpolation

CPU vs GPU analysis

For 5 epochs :

Total time CPU = 37.15s

Vector Add time CPU = 0.0017s

Time taken by CNN layers = 11s

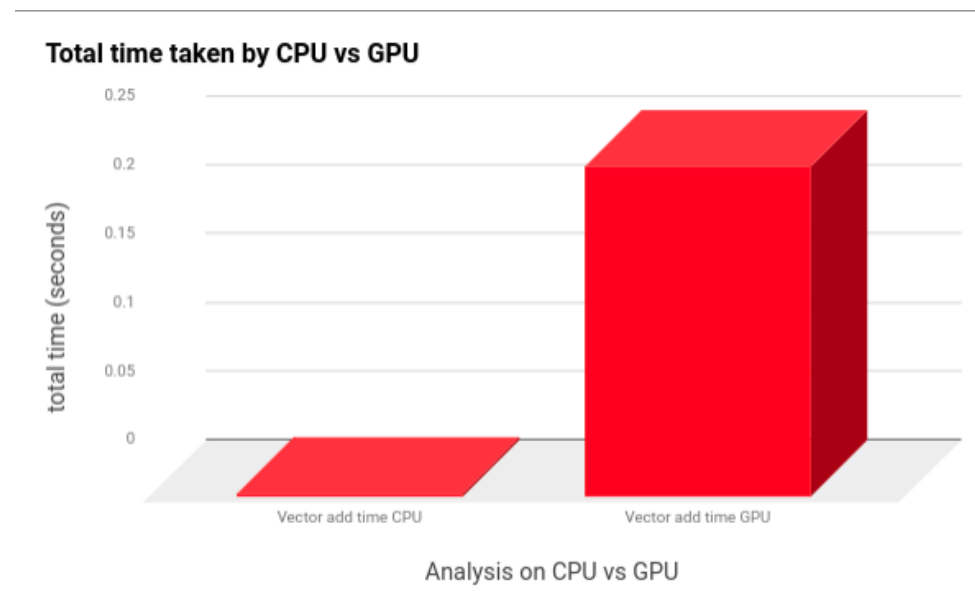
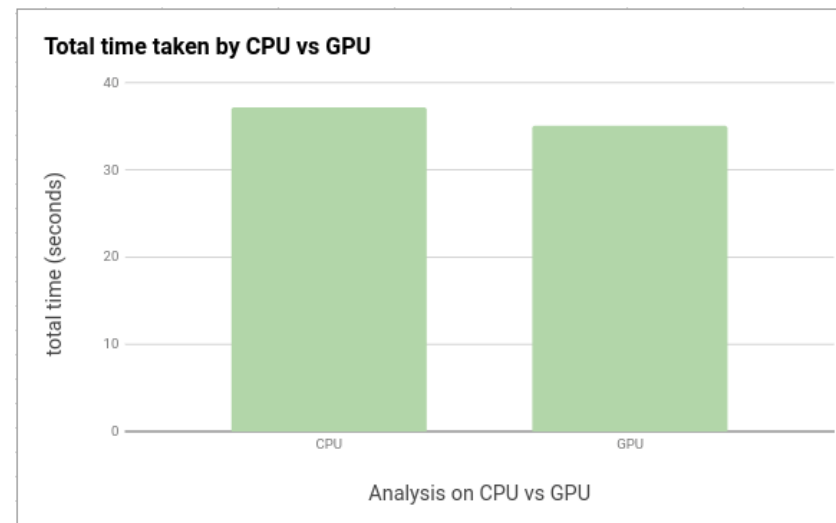
Total time GPU = 35.04s

Vector Add time GPU (global memory) = 0.24s

Vector Add time GPU (shared memory) = 0.17s

Time taken by CNN layers = 11s

GPU speedup over CPU = $37.15 / 35.04$ seconds = 1.06 seconds



Limitation

1. For one pair comparison of volumes, CPU performs better than GPU
2. Very low dimension of **Z** could effect GPU performance
3. Running threads in shared memory of GPU is taking 1.5x more than global memory

Challenges

1. As a dynamic type system, explicit memory allocation in Python is not possible
2. Less example and documentation of Python-CUDA compared to CUDA-C or CUDA-C++
3. Inside Kernel function, debugging is not available



Questions

1. What is Linear Interpolation ?
2. How does Numba work ?
3. Considering the results, does increasing the dimension of Z vectors or volumes of input shape would increase the GPU performance ?