Parallelizing the Interpolation between Latent Space of Autoencoder Networks

Tansin Jahan
School of Computer Science
Carleton University
Ottawa, Canada K1S 5B6
tansinjahan@cmail.carleton.ca

December 16, 2018

Abstract

From early computing, faster execution of instructions is always been one of the major factors and because of that, faster logic devices and hardware improvements has been made to the machines. But focusing only on the hardware issues is not going to overcome this problem. Therefore, recent developments of parallel architecture combines both the hardware capability (ex- improvements on GPU) and implemented libraries (ex- CUDA framework and it's deep learning libraries such as cuDNN, Numba etc.) so that parallelism can be acheived easily. This project executes several matrix addition operations between the results of convolutional neural network's layer and takes advantage of CUDA platform to execute these operations inside GPU using multiple threads. By accessing GPU's shared memory, faster execution time has been achieved which compared to CPU is really significant. Though the result found by the CNN does not correspond to the expected output shape but the analysis of time differences between CPU and GPU was valuable and can lead to the possibilities of future research regarding parallelism inside convolutional neural network.

1 Introduction

The idea of parallel computing infers to execute more than one tasks simultaneously so that the complexity(ex- time, space etc.) in computation can be carried out smoothly. Though this is the simplest definition of parallel computing, in the real world, parallelization means a lot more than just handling the complexity in the computation of algorithms. With the development of Deep Learning Networks, parallel computing has become the essential choice for the implementation of these huge networks as it has to deal with lots of parameters.

Parallelism can be introduced following different taxonomies of parallel architectures. According to Flynn's taxonomoy, parallel architectures can be categorized as - Single Instruction Single Data(SISD), Multiple Instruction Single data(MISD), Single Instructions Multiple Data(SIMD) and Multiple Instruction Multiple Data(MIMD) [7]. In terms of memory parallelization, it can be divided into distributed and shared memory [8].In addition with multicore and massive parallel clusters, GPU has become another predominant choice to aid in parallelism architecture. Using GPU's several core architectures, it becomes possible to run programs in parallel which provides unprecedented computational power for

applications with large data. Now-a-days the ubiquity and easy access to GPU through google cloud computing [1] has made it more popular.

So, I have divided the project into several groups such as - convolution of each input batches, loading data and vector operations for different combinations of sample inputs. Then the time taken by each group is compared so that the parallelization can be introduced in such a way where the total runtime of the program can be optimum. Following the amount of input data, memory parallelization inside GPU to calculate vector operations between input samples seems more optimized and convenient as a parallelization technique. For results, I tried to visually compare the time differences of GPU vs CPU in terms of different input sizes. And GPU performed 1.5x better than CPU as expected.

In summary, for this project several threads will be introduced to compute vector addition in CUDA so that the performance of the whole network can be improved. So, the rest of the paper is organized as follows - In Section 2, we will review the relevant literature. Section 3 and 4 will present the problem and it's proposed solution. In Section 5, we will focus on the evaluation of experiment and computational time comparison between diferent approaches. At last, Section 6 concludes the paper.

2 Literature Review

2.1 Object reconstruction using CNN

With the recent development of Convolutional Neural Network, it has been used to solve several Computer Vision problem. For example, object detection or reconstruction from input image, semantic information extraction from a scenario, object segmentation- these are all recent computer vision application where CNN has been used to produce better result. Likewise, 3D reconstruction of an object is one of the Computer Vision problem and recently multiple approaches (ex: 3D-GAN, 3D-shapenets) has been proposed as a solution to the problem [15]. In general, all these CNN models are considered as generative model as it produces output based on given input. But in terms of novelty, research are now more inclined to find something new rather than having just the same output as input. Following this, *Alec Redford* in his paper [11] experimented about using average mean vectors of input samples to produce new samples which gave very promising results. Another study recently shows that the interpolation between sample space and introducing noise vector can also produce good results for generative models. The experiment proposed a *Generative Latent Optimization* model by exploiting properties of GANs: learning from large data, synthesizing visually-appealing samples, interpolating meaningfully between samples, and performing linear arithmetic with noise vectors[4].

2.2 Current approaches for 3D object reconstruction

Though most of these models chose 2D image for their dataset but very few actually experimented with depth of an image. Previous work shows that given a depth image as input, the volumetric representation can be produced [16]. Following these works, an approach to reconstruct 3D voxelized object from different viewpoint of one or more images of that object (i.e. single-view or multi-view) is proposed where recurrent neural network has been

used [6]. In total 50,000 model is used to train and test the proposed network. Training the network with this large amount of data is really time consuming and therefore introducing parallelism between the layers of the model can help to improve the performance of the network.

2.3 Parallelism in GPU

GPU's parallel processor architecture has made it an essential choice for several applications where parallelism is needed. Such most common areas where GPU computing can be used are - Bioinformatics, Data Science, Analytics, and Databases, Defense and Intelligence, Machine Learning, Imaging and Computer Visions etc. [13]. Using GPU's multithreaded processors, several threads can be introduced to perform matrix operations which is highly efficient for both graphics and general-purpose parallel computing applications [10]. To execute programs in GPU written by high-level languages such as C, C++ or Python - CUDA provides a high level interface by dividing CPU as host memory and GPU as device memory. In CUDA programming model, the whole program is divided into several phases where the phase with data-parallelism is executed inside GPU using kernel functions and others implemented inside CPU. [12]. In addition with interface CUDA also provides several GPU acclerated libraries such as - cuDNN for deep learning, NCCL for parallel algorithms, cuBLAS for linear algebra and math. These CUDA libraries are designed by Nvidia and 3x faster than CPU in terms of runtime [2].

For this project, I followed GPU shared memory parallelization to calculate 2D-vector ad-

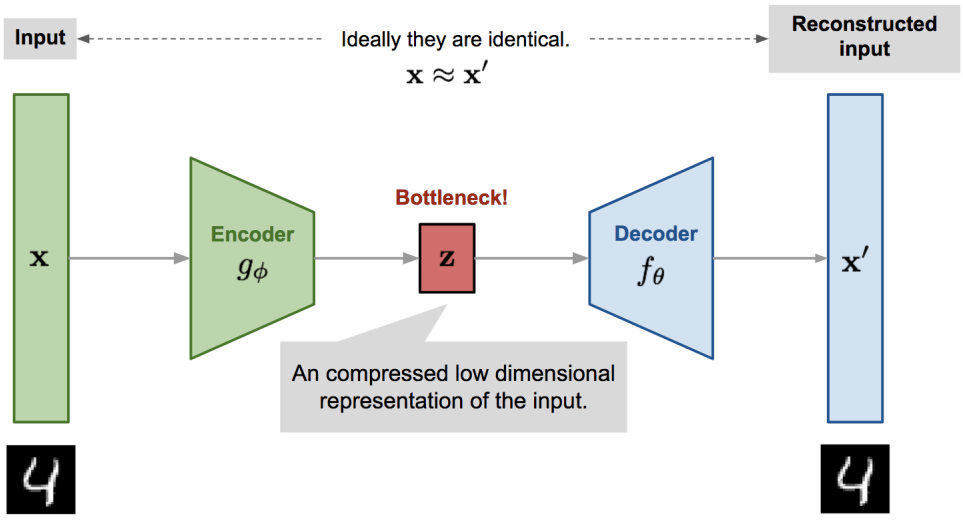


Figure 1: A simple autoencoder architecture
[14]

dition operation inside the autoencoder network shown in figure 1. The implemented CNN for this project feeds on 3D volumes as input involving higher dimension(i.e. 32x32x32) and then reduces that into lower dimension [17] which is called as latent space(also Z). In latent space, represented as Z vector, the input object has minimum dimension with maximum features. From the lower dimension, the Z vector passes to deconvolution and therefore

produces the same 3D volumes. But we can combine Z vector of two input volumes to interpolate new points and give it to the decoder so that it can produce new 3D objects based on this interpolation. This interpolation of Z vector (addition of two Z vectors) can be an overhead for the performance of the Autoencoder. Let us consider we have 50 inputs. Then for each input, we have to compare it with another 49 inputs and calculate interpolation each time. So, for 50 inputs the vector addition will be 50 x 50 which is in total 2500 times addition of vectors.This calculation can take much time compared to the convolution and therefore we can parallelize this computation in GPU using Multithreading.

3 Problem Statement

From the studies of generative models, it is seen that most of these models focused on generating 2D images and does not take into account the depth or volume of any image. Moreover, it is also discussed in Section 2.2 that, recent approaches for 3D object reconstruction using recurrent neural network [6] or simple convolutional neural network [16] produces the same output as input . Hence, the novelty gets suppressed by the regular design of generative models. This project intends to address this issue by leveraging generative models composition to introduce novelty in object reconstruction. Therefore the research question arises as following -

1. Can we propose an interpolation technique inside convolutional neural network to change the expected output explicitly by means of matrix addition between two sample inputs?

2. As the interpolation technique involves matrix addition inside it, would shared memory parallelism be useful to reduce the total time of execution in terms of larger dataset?

At each step, finding this interpolation between any two sample inputs might be an overhead if the matrix size becomes larger (example - 50,000 shapes of object). A general overview of time taken by GPU vs CPU for different sizes of matrix multiplication is shown in figure 2 [9]. A CUDA platform for GPU programming will be an ideal option to speed up this computation.

4 Proposed Solution

In this Section, we will discuss the approach taken to solve the research questions presented in Section 3. So, subsection 4.1 will present an algorithm to address research question of interpolation technique. Following that, subsection 4.2 will describe the parallelization technique used for implementation of the algorithm specifically.

4.1 Algorithm for interpolation

The formal definition of Linear Interpolation is to generate new points given two known points. As for the autoencoder network of this project, Linear Interpolation would mean to interpolate new vector(i.e. latent space or $\mathbf{z}$) by adding two other vectors produced by two known shapes. Then we will provide this new $\mathbf{z}$ to the network so that it can create new

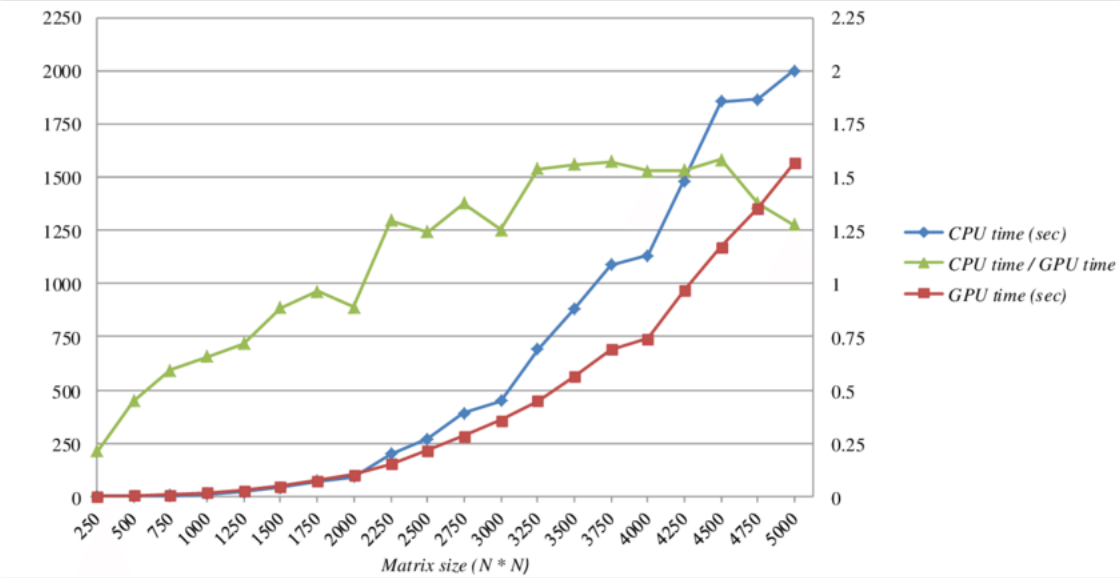


Figure 2: CPU vs GPU time for Matrix Multiplication
[9]

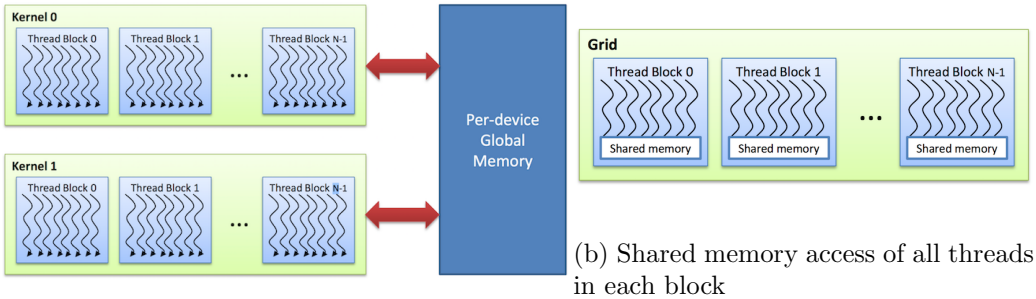
shape between the range of the given vectors. Generally these vectors can be of any shape but for the simplicity we created it as a 2D matrix. So the algorithm is as follows -

Algorithm 1 Algorithm to generate new **z** as latent space for Autoencoder

```
for i = first input shape do
    for j = second input shape do
        z1 = z vector for i
        z2 = z vector for j
        for t = 0 to 1 do
            newz = (1-t) * z1 + t * z2
            run decoder network
            t = t + 0.1
        end for
    end for
end for
```

4.2 Parallelism

The innermost loop of our algorithm contains vector addition and scalar multiplication between range of [0,1]. This computation can be executed in parallel manner by leveraging the power of GPU. For clarification, the dimension of both z1 and z2 is [8 by 8]. GPU provided by NVIDIA has now become the most common choice for high performance parallel computing application. And CUDA architecture takes this advantage of GPU computing to run highly parallelized program [3]. NVIDIA CUDA architecture provides two interfaces - i) A device-level programming interface and ii) A language integration programming interface. Through second interface, we can write programs using high-level language(ex-



(a) Thread blocks in Kernel

(b) Shared memory access of all threads in each block

Figure 3: CUDA parallel programming architecture

C,C++,Python) and C Runtime for CUDA automatically handles setting up the GPU and executing the compute functions [3]. Another important objective of CUDA is that it divides the execution of program between **host** and **device** which represents comsecutively **CPU** and **GPU**. Multiple threads can be initialized inside CPU and can be sent to GPU for executing in parallel manner. Threads can be grouped together in a block and mutiple blocks build together grids of thread blocks. Beacuse of the hardware chip configuration, shared memory is much faster than local and global memory. And each thread block is allocated this shared memory, so all threads in the block can access to the same shared memory.

5 Experimental Evaluation

For this experiment, one particular object(i.e. chair) has been chosen as dataset. But it can also be extended for other objects. Firstly, Section 5 will focus on explaining dataset and the configuration of CNN. Then parallel thread architecture used for the experiment will be discussed followed by the time comparison between CPU and GPU for total input shapes.

5.1 Dataset and GPU configuration

As I mentioned in Section 2.3, 50 different shapes of chair in total has been used for this experiment which has been downloaded from ShapeNet [5]. Among those, 40 shapes has been used as training data and 10 shapes has been used for testing data. Each of these shapes has dimension of [32 x 32 x 32]
The GPU used here is NVIDIA Titan V with 8 virtual cores. Total RAM space is 30GB and the version of OS is UBUNTU 16.04

5.2 CNN and training

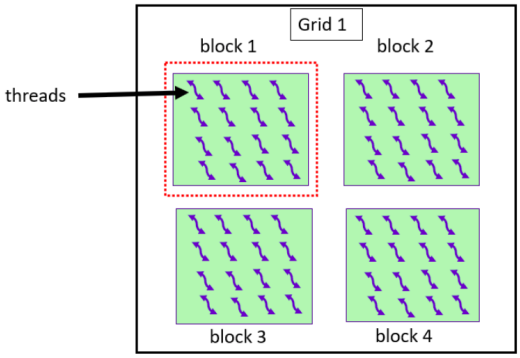
The CNN model named Autoencoder used here consists of **Encoder**(for convolution) and **Decoder**(for deconvolution) and both has exactly 3 layers to convolute over the input object. Table 1 indicates the output structure after convolution in each of the layers. So, encoder takes an input volume of [32 x 32 x 32 x 1] and produce an output shape of [2 x 2 x 2 x 8] which is designated as **z**(i.e. latent space). This **z** then passed to the kernel where interpolation takes on and **newz** returned to the decoder which later on gives the output based on the changes made to that **z** vector.

Table 1: The highlighted output of encoder [2 x 2 x 2 x 8] represents **z** and reshaped into [8 x 8] to perform vector operation conveniently inside kernel. No feature is lost or modified in this reshape operation as dimension remains unchanged. After interpolation **z** again reshaped into [2 x 2 x 2 x 8] to follow the model architecture and feed to the Decoder.

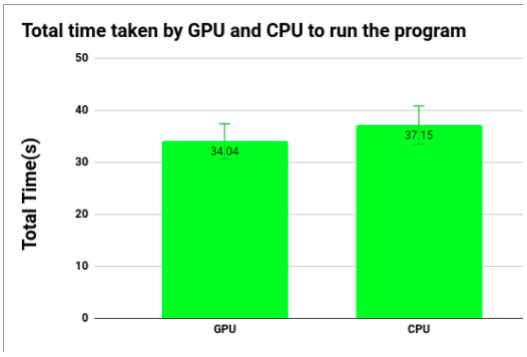
Encoder	Interpolation	Decoder
32 x 32 x 32 x 1	[8 x 8] + [8 x 8]	2 x 2 x 2 x 8
16 x 16 x 16 x 32	newz = [8 x 8]	8 x 8 x 8 x 16
8 x 8 x 8 x 16	reshaped into = [2 x 2 x 2 x 8]	16 x 16 x 16 x 32
2 x 2 x 2 x 8		32 x 32 x 32 x 1

5.3 Threads and results comparison

To utilize GPU's computational power, 4 blocks of thread has been initialized. The dimension of threads are [4 x 4]. The kernel function takes two input vectors(**z1** and **z2**), one output vector(**newz**) and a scalar value(**t** as number inside given interpolation range) as arguments for calculation. In figure 4a, implemented threads and blocks architecture for this experiment is shown.



(a) Threads architecture for the experiment



(b) Time comparison between CPU and GPU

Figure 4: First subfigure 4a shows how many threads are implemented and subfigure 4b represents the total time taken by CPU and GPU to complete the program using shared memory

5.4 CPU vs GPU analysis

Two versions of the program has been implemented. In the first version, threads performed the vector addition using global memory and in another one using shared memory. The time for each of the execution has been compared. Table 2 represents time taken by different operations inside the program. It is observed from Table 2 that for input size 2, shared memory took 0.17s to complete the vector operation where global memory took 0.24s. As for CNN running time, it remains same for both CPU and GPU which is 11 seconds.

Another CPU version of the program is also implemented to compare with GPU. The results are described in Table 2 and Table 3. For input size 2, total time taken by GPU

is 35.04 seconds whereas, CPU took 37.15 seconds. So, GPU speedup over CPU is 1.06 seconds which is not very significant. But surprisingly, when the input size grows, the differences also became visible significantly. For example, table 3 shows that for 50 inputs, GPU takes only 0.09s but CPU took 0.9s. We can see from line graph 5 that, GPU tends to take less time as input grows and become steady in opposed to CPU. In summary, we can claim that GPU is not certainly robust enough for computation in terms of smaller input. But when the input size is large enough than GPU will surely outperform CPU by providing faster computation time.

Table 2: Time differences on GPU in terms of Shared and Global memory

Different Methods	GPU(s)	CPU(s)
Vectoradd time(shared)	0.17	0.0017
Vectoradd time(global)	0.24	0.0017
CNN run time	11	11
Total program run time	35.04	37.15

Table 3: GPU and CPU time differences vs input size

	Time(s)	
Input size	GPU	CPU
2	0.17	0.0017
10	0.15	0.0024
20	0.12	0.034
30	0.10	0.54
40	0.09	0.78
50	0.09	0.9

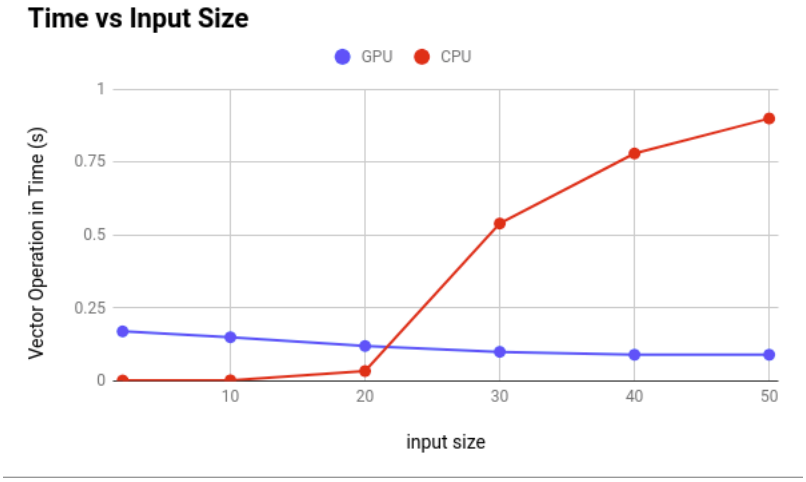


Figure 5: The variance of GPU and CPU time is shown in compared to the input sizes

5.5 Autoencoder output

In this section, we will discuss the results provided by the **Autoencoder** network after interpolation. As preliminary results, it can be seen from figure 6 that the output shape niether biased to any of the input shape nor it corresponds to a new shape as expected from the network. There can be several issues involved such as networks learning parameters, range of interpolated values etc. As future work, these issue can be certainly focused on which will help to detect the problem associated with the output.

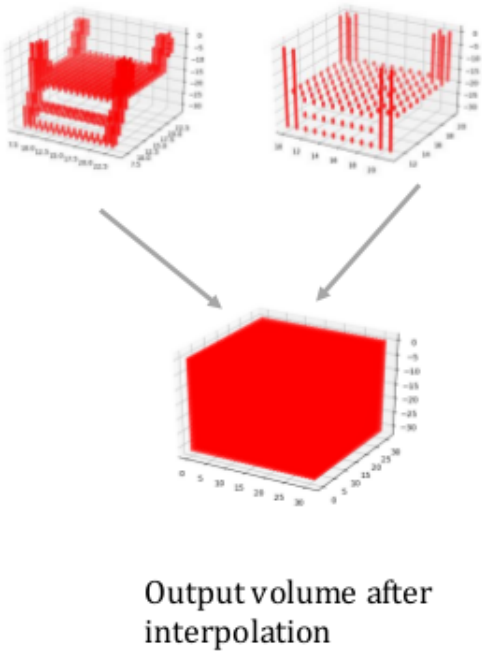


Figure 6: Output shape from two given input shape through interpolation inside latent space

6 Conclusion

To keep compute time minimized for any task is really important. Now-a-days recent improvements on GPU's are specially focused to maintain this overhead. GPU's are always preferred for deep learning as CNN and it's parameter computation can be complex sometimes. The approaches taken in this project by leveraging GPU's shared memory parallelism contributes to improve the runtime of the program despite of involving 6 layers of Convolutional Neural Network. In addition, the project also takes input of 3D volumes and tries to make some new shape out of it is not very common for most of the CNN. Moreover, several analysis like - global or shared memory time comparison, CPU or GPU time differences in terms of input sizes has given very interesting insightful which can lead to further research in deep learning to minimize the time complexity of heavy networks.
For future work, several other parallelism approaches can also be taken and compared with this one to find the better solution. And other CUDA deep learning libraries such as cuDNN

can be used to implement the CNN. So that it can help to improve the overall performance of the network. To conclude, GPU parallelism supports to minimize runtime of the program if the computational operation is large enough. Otherwise, CPU might outperform GPU and GPU's computability might not be utilized properly.

References

[1] GRAPHICS PROCESSING UNIT (GPU). `https://cloud.google.com/gpu/`. [Online; accessed 12-December-2018].

[2] Home NVIDIA Developer. `https://developer.nvidia.com/cudnn`. [Online; accessed 13-December-2018].

[3] CUDA Architecture Overview. `http://developer.download.nvidia.com/compute/cuda/docs/CUDA_Architecture_Overview.pdf`, 2009. [Online; accessed 13-December-2018].

[4] Piotr Bojanowski, Armand Joulin, David Lopez-Paz, and Arthur Szlam. Optimizing the latent space of generative networks. *arXiv preprint arXiv:1707.05776*, 2017.

[5] Angel X. Chang, Thomas Funkhouser, Leonidas Guibas, Pat Hanrahan, Qixing Huang, Zimo Li, Silvio Savarese, Manolis Savva, Shuran Song, Hao Su, Jianxiong Xiao, Li Yi, and Fisher Yu. ShapeNet: An Information-Rich 3D Model Repository. Technical Report arXiv:1512.03012 [cs.GR], 2018.

[6] Christopher Bongsoo Choy, Danfei Xu, JunYoung Gwak, Kevin Chen, and Silvio Savarese. 3d-r2n2: A unified approach for single and multi-view 3d object reconstruction. *CoRR*, abs/1604.00449, 2016.

[7] Ralph Duncan. A survey of parallel computer architectures. *Computer*, (2):5–16, 1990.

[8] Vipin Kumar, Ananth Grama, Anshul Gupta, and George Karypis. *Introduction to parallel computing: design and analysis of algorithms*, volume 400. Benjamin/Cummings Redwood City, 1994.

[9] Panon Latcharote and Yoshiro Kai. High performance computing of dynamic structural response analysis for the integrated earthquake simulation. In *Proceedings of the Thirteenth East Asia-Pacific Conference on Structural Engineering and Construction (EASEC-13)*, pages H–1. The Thirteenth East Asia-Pacific Conference on Structural Engineering and , 2013.

[10] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with cuda. In *ACM SIGGRAPH 2008 classes*, page 16. ACM, 2008.

[11] Alec Radford, Luke Metz, and Soumith Chintala. Unsupervised representation learning with deep convolutional generative adversarial networks. *arXiv preprint arXiv:1511.06434*, 2015.

[12] Shane Ryoo, Christopher I Rodrigues, Sara S Baghsorkhi, Sam S Stone, David B Kirk, and Wen-mei W Hwu. Optimization principles and application performance evaluation of a multithreaded gpu using cuda. In *Proceedings of the 13th ACM SIGPLAN*

Symposium on Principles and practice of parallel programming, pages 73–82. ACM, 2008.

[13] Bhavana Samel, Shubhrata Mahajan, and AM Ingole. Gpu computing and its applications. *International Research Journal of Engineering and Technology (IRJET)*, 3(04), 2016.

[14] Lilian Weng. From Autoencoder to Beta-VAE. `https://lilianweng.github.io/lil-log/2018/08/12/from-autoencoder-to-beta-vae.html`, 2018. [Online; accessed 16-December-2018].

[15] Jiajun Wu, Chengkai Zhang, Tianfan Xue, William T. Freeman, and Joshua B. Tenenbaum. Learning a probabilistic latent space of object shapes via 3d generative-adversarial modeling. *CoRR*, abs/1610.07584, 2016.

[16] Zhirong Wu, Shuran Song, Aditya Khosla, Fisher Yu, Linguang Zhang, Xiaoou Tang, and Jianxiong Xiao. 3d shapenets: A deep representation for volumetric shapes. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1912–1920, 2015.

[17] Zhuotun Zhu, Xinggang Wang, Song Bai, Cong Yao, and Xiang Bai. Deep learning representation using autoencoder for 3d shape retrieval. *Neurocomputing*, 204:41–50, 2016.