International Journal of Engineering Trends and Technology
ISSN: 2231 – 5381 /doi:10.14445/22315381/IJETT-V69I6P231

Volume 69 Issue 6, 216-224, June, 2021

Analysis and Design of High Performance Deep Learning Algorithm: Convolutional Neural Networks

Sunil Pandey[#1], Naresh Kumar Nagwani[#2], Shrish Verma[#3]

[#1]*Research Scholar,* [#2]*Associate Professor*

Department of Computer Science and Engineering, NIT Raipur 492010, CG, India
[#3]*Professor, Department of Electronics and Communication Engineering, NIT Raipur 492010, CG, India*

[1]sys_admin@nitrr.ac.in, [2] nknagwani.cs@nitrr.ac.in, [3] shrishverma@nitrr.ac.in

Abstract *— Deep learning algorithms like convolutional neural networks (CNNs) have a multi-layered computational design. The CNN comprises of stacks of different layers which perform feature engineering and training or classification computations on the inputs which are generally 3-D tensor datasets. Training a CNN is very demanding in terms of computational resources and time. Training times of several weeks and even months are not unheard of. This is one of the important reasons limiting widespread adoption of CNNs in new applications. Performance enhancement of CNNs is therefore an active R&D area. In view of this, the design of CNN algorithms for high performance distributed and parallel computing architectures assumes significance. The CNN can be conceptualized as a pipeline system which makes CNNs amenable to pipeline parallelism. In the present work, a pipeline computation design and model of the CNN has been proposed. The performance of the pipeline model of the CNN has been analyzed based on representative data generated through different computational experiments. Analysis shows that a net performance gain of 18X can be achieved on a CNN feature engineering pipeline by combining pipeline parallelism with task parallelism.*

Keywords — *Deep Learning, Convolutional Neural Networks, Pipeline Computing, Pipeline Parallelism, Task Parallelism, High Performance Computing.*

I. INTRODUCTION

Deep learning algorithms are a novel artificial intelligence technology and are being applied towards the solution of a multitude of engineering problems of this era. A glance at the wide spectrum of applications for which different deep learning technology has provided very successful solutions can be made from [1-20]. However, all deep learning algorithms are extremely compute intensive and tend to consume copious amounts of computer resources like computer time, processors, accelerators, primary memory, secondary storage etc. for their training and application which is proving to be a limitation hindering growth and widespread adoption of this otherwise promising technology.

The deep learning technology has been explained in sufficient detail in [21]. The challenges of deep learning technology which include its highly compute intensive nature is a perspective which is shared by different researchers [22-24]. In the effort to keep the training times of deep learning algorithms reasonable, the high performance features of contemporary computer architectures need to be utilized. Multicore processors, symmetric multiprocessor and compute cluster architectures can play a very important role in this context. They are the subject of discussion in [25][26] and [27] respectively.

Distributed and parallel deep learning models have been discussed in [28][29][30]. GPU based hardware accelerator cards and libraries have also been deployed for accelerating deep learning [30][31]. The compute intensive optimization methods which are used in deep learning are explained in [32]. The asynchronous stochastic gradient descent method is the primary optimization algorithm of distributed machine learning and is discussed in [33].

Techniques for improving the speed of neural networks on CPUs are discussed in [34-36]. Matrix methods for deep learning which are important from the perspective of high performance deep learning implementations also are discussed in [37-40]. The remarkable successes achieved by deep learning technology have been highlighted in [41-44].

The deep learning paradigm which has been taken up for analysis in this paper is the convolutional neural network (CNN). The CNN is comprised of multiple stacks of layers which perform feature engineering and training or classification computations on the input data which are generally 3-D tensor datasets. Fig.1 shows the architecture of a typical CNN with the input image of “Sundari”.

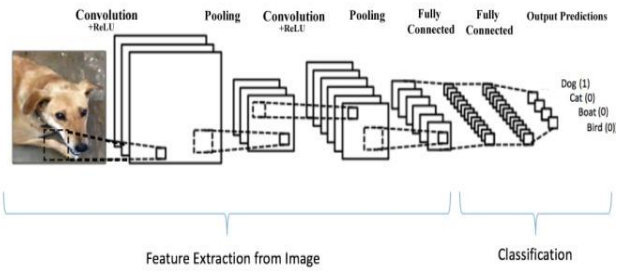

Fig. 1: Architecture of a typical CNN

The convolutional layer is the primary feature engineering layer and is used for generating raw features from the samples of the input 3-D tensor dataset by performing convolution computations on them. The nonlinear activation layer performs a nonlinear transformation mapping of the raw features thereby normalizing them. The pooling or the sub-sampling layer is used for feature space dimension reduction. The fully connected layers comprise of an artificial neural network composed of multilayer perceptrons which is used in the supervised training of the feature dataset through the feedforward backpropagation algorithm. The CNN architecture has been explained in necessary and sufficient detail for the current work in Section VIII on Understanding Deep Neural Networks: Convolutional Type in [45].

The CNN can be visualized as a pipeline computation with the input 3-D tensors flowing through its various stages. Fig. 2 shows two of the popular CNNs, viz., the LeNet (on the left) and the AlexNet (on the right). The flow takes place from the bottom to the top.

For LeNet, the pipeline flow is described as follows. The input is a 28 pixel x 28 pixel image. The first stage is the convolutional stage which comprises of 6 convolutional filters of size 5 x 5 with a padding of 2. The second stage is the pooling or sub-sampling stage which comprises of non-overlapping average pooling windows of size 2 x 2 with a stride of 2. The third stage is once again a convolutional layer which comprises of 16 convolutional filters of size 5 x 5 with no padding. The fourth stage is similar to the second stage. The fifth, sixth and seventh stages constitute the input, hidden and output layers of a three-layer artificial neural network having 120, 84 and 10 artificial neurons respectively. The pipeline flow of the AlexNet may be understood in similar terms.

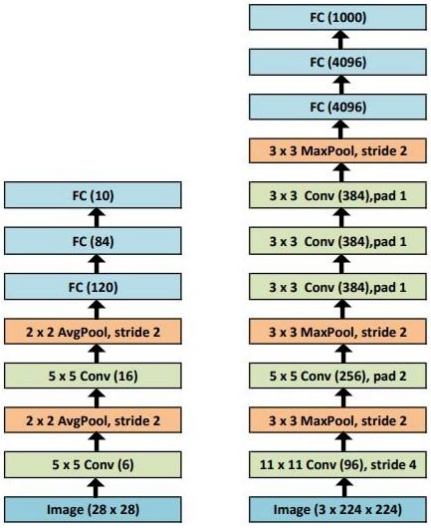

Fig.2: LeNet (left) and AlexNet (right)

II. MATERIALS AND METHODS

In the present work, a pipeline computation design and model of the CNN has been proposed. The performance of the pipelined CNN model has been analyzed using representative data obtained from different computational experiments. In this section, the pipeline computation model has been elucidated. The major advantage of conceptualizing the CNN as a high level computational pipeline of functional computational blocks is that it becomes possible to apply high performance techniques from the time tested pipeline parallel model. It then becomes possible to map the individual blocks or even entire pipelines on the multiple cores of one or more CPUs spread over the multiple compute nodes of a high performance computing cluster, for example.

Pipelined Computations

The concept of a software pipeline is of significance in this paper. The software pipeline is a generic and a higher level concept than the instruction pipelining model which is a familiar model from the field of computer architecture and processor design.

In a software pipeline, a larger computational problem is divided into a series of ordered sequential tasks which can be completed one after the other. The output of the previous task becomes the input for the current task. The individual tasks are then executed as a distinct process or on separate processors. Fig. 3 illustrates a larger problem P which has been partitioned into a series of 5 ordered sequential tasks P_0 through P_4.

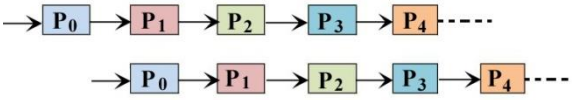

Fig. 3: Pipeline with 5 ordered, sequential tasks P_0 through P_4

Pipeline Types

A pipelined algorithm can be designed for a specific computational problem, with a resultant increase in the speed of execution and higher performance if one of the following conditions is satisfied:

(a) Multiple instances of the entire problem are to be executed, or,

(b) A sequence of data samples are to be processed through multiple independent computational operations, or,

(c) If the information which is required to start the next process in the queue can be passed prior to the current process completing all its internal operations.

The conditions in (a), (b), and (c) above lead to Type 1, Type 2 and Type 3 pipeline computations respectively. The timing or the space-time diagrams of the three types of

pipelines are highlighted in Figs. 4 – 7 with the Figs. 4 and 5 being alternate representations of Type 1 pipeline model.

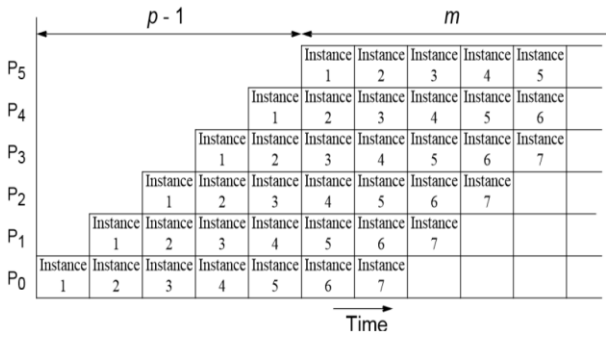

Fig. 4: Type 1 Pipeline Space Time Diagram

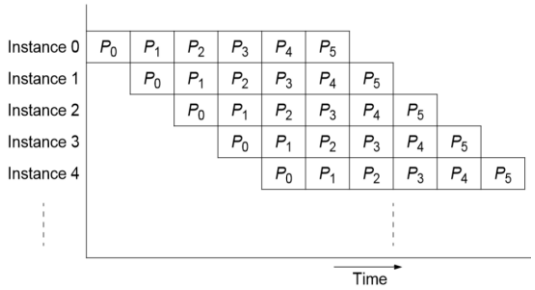

Fig. 5: Alternate Type 1 Pipeline Diagram

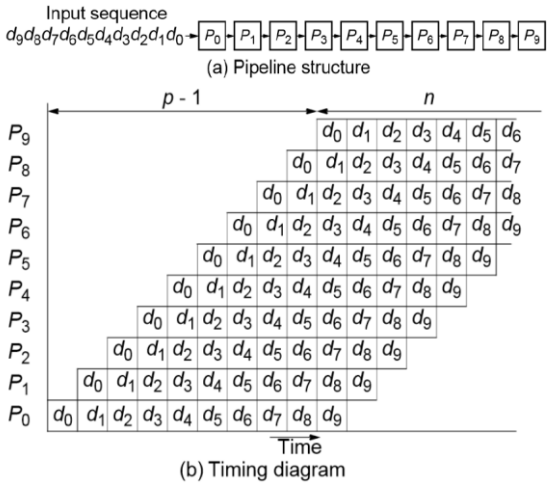

Fig. 6: Type 2 Pipeline Diagram

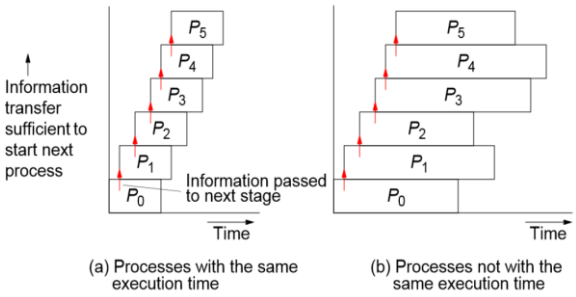

Fig. 7: Type 3 Pipeline Space-Time Diagram

Computing Platform

The ideal computing platform for pipelined computations is a multiprocessor system with the processors arranged in a line configuration as shown in Fig. 8.

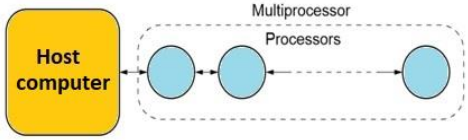

Fig. 8: Computing platform for pipeline computing

A group of pipeline stages can be assigned to each processor as illustrated in Fig. 9.

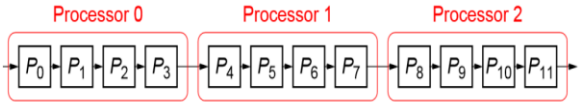

Fig. 9: Mapping of groups of pipeline stages to processors for pipelined computation

The pipeline parallel programming technique as a high level parallel composition is discussed in [46]. Contemporary pipeline parallelism is discussed in [47]. The issues related to the construction of computational pipelines are discussed in [48]. Troubleshooting of failed computational pipelines is the matter of discussion in [49]. Pipeline scheduling has been discussed in [50]. An analytical model of pipeline parallelism is given in [51]. Pipeline parallelism for streaming data has been exposited in [52].

III. PIPELINE COMPUTATIONS IN CONVOLUTIONAL NEURAL NETWORKS

The computations of the convolutional neural network can be modelled as a software pipeline. The CNN software pipeline can be separated into 5 distinct stages as mentioned in Table I. Each of these layers corresponds to a function.

The "LOAD" stage is used for loading an image sample from the secondary storage, which is typically the hard disk, to the primary memory. The "CONV" stage performs the convolution operation on the loaded image sample. The "NLMAP" stage is the nonlinear mapping stage which performs a nonlinear mapping or transform on the convolved image. The "POOL" stage performs the pooling or the subsampling operation on the image obtained from the "CONV" and "NLMAP" stages. The "ANN" refers to the classifier stage which is a fully connected stage used for supervised training on the features from the "POOL" stage.

TABLE I
The Five Stages of the CNN Software Pipeline

Stage	Code	Input
LOAD	P_0	Filename
CONV	P_1	Image file, Filter Size, Filter Nos., Padding
NLMAP	P_2	Input from P_1
POOL	P_3	Input from P_2, Pooling Window Size, Stride
ANN	P_4	No. of Layers, MLPs in each layer

A hypothetical CNN assumed to have a single stage each of P_0 through P_4 is modelled as a CNN structure.

The pipeline structure of the hypothetical CNN can be modelled as shown in Fig. 10 below

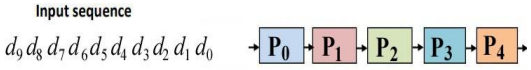

Fig.10: Pipeline structure of the CNN pipeline

In the above figure, d_0, d_1, d_2, etc. represent the image samples. The above model is a typical Type 2 pipeline wherein a sequence of data samples are to be processed through multiple independent computational operations.

Alternately, it is possible to group or consolidate the stages P_0 through P_4 as a single instance of the complete problem as illustrated in Fig. 11.

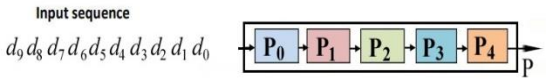

Fig.11: Pipeline structure of the CNN pipeline with a single consolidated stage P

In this alternate scenario, multiple instances of the entire problem are to be executed on individual data samples, d_0, d_1, d_2, etc. This alternate model is a typical Type 1 pipeline.

The pipelined computation format affords considerable flexibility when modelling the CNN. The CNN can be modelled as a Type 1 or Type 2 pipeline. The stages can be merged to form a single consolidated stage. It is also possible to split a stage into two or more stages.

A CNN can be considered to be a high level pipeline composition of the five stages mentioned in Table I above. The LeNet CNN of Fig. 2 (left) is revisited for an example of the composition described above.

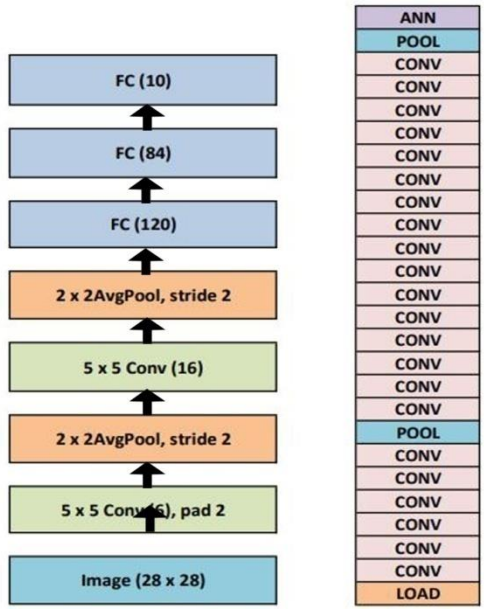

Fig. 12: LeNet (left) LeNet Pipeline (right)

In order to analyse the CNN pipeline performance, the execution times of the five stages of the CNN pipeline are required as the input parameters and have been determined through computational experiments. Table II shows the experimentally determined times of the five stages of the CNN software pipeline. These are the average compute times for the processing of one data element, i.e., one input image sample through the CNN pipeline.

TABLE II
Input Parameters: Experimentally determined times of the stages of the CNN software pipeline

S. No.	Stage	Execution Time, T (s)	Remarks
1.	P_0	0.1	HD image 1280 pixels x 720 pixels
2.	P_1	46	Convolutions of image with 100 3x3 filters
3.	P_2	19.9	Nonlinear hypertangent activation function
4.	P_3	24.1	Average pooling 2x2 window
5.	P_4	29.84	50 inputs, 100 hidden layer neurons, 10 outputs, 10000 epochs

The times seen in Table II are experimentally determined times of the different stages of a CNN software pipeline for a single HD image of 1280 pixels x 720 pixels. Same is graphically represented in Fig. 13.

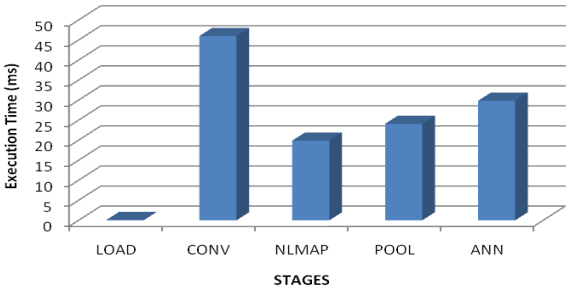

Fig. 13: Experimentally Determined Times of the stages of the CNN software pipeline

A real-world dataset from the domain of Materials Science has been taken up for analysis using the CNN. The primary motivation here was the design and analysis of a pipelined version of the CNN for application on this dataset. This dataset is the North Eastern University's Steel Surface Defect Dataset. This dataset comprises of a total of 6 classes. Each of these classes corresponds to one of six surface defects which are common in hot-rolled steel strips. These defects are the rolled-in scale (RS), patches (Pa), crazing (Cr), pitted surface (PS), inclusion (In) and scratches (Sc). Each class has 300 grayscale images. Size of each image in the dataset is 200 pixels x 200 pixels. Classification strategies applied on this dataset are discussed in [53-55].

Fig. 14 shows the master CNN pipeline architecture designed for the solution of this problem. The feature engineering part of the CNN has been considered here for this demonstration. The complete pipeline can be seen to be a composition of six different pipelines. For example, pipeline 1 comprises of loader1 followed by cnp11, cnp12, cnp13, cnp14 and flattening1.

The convolutional, nonlinear mapping and pooling stages have been consolidated into a single "cnp" stage. An additional "flattening" stage can be seen. The flattening operation is used for mapping of the final pooled two-dimensional matrices into one-dimensional vectors. These are the feature vectors which serve as the feature vector input to the "classifier" stage. Four convolutional filter stages with 25, 25, 25 and 25 numbers of 11 x 11 filters have been considered. These filters can be observed to be common for all the six CNN pipelines. These convolutional filter perform the convolutional operation on each individual image of the CNN dataset. For the experiments, 100 images of each class were considered.

The master CNN pipeline architecture for the NEU surface defect dataset is of a much higher level of sophistication than the simple pipeline of Fig. 10. The master pipeline is composed of six parallel pipelines. Each of the pipelines corresponds to one of the classes of the dataset. The six pipelines have been designed for performing the feature engineering computations on the input images corresponding to the six different classes of the dataset in parallel. This provides the basis for much higher performance as this approach combines pipeline parallelism with task parallelism on CNN computations. The net outcome is a multiplicative gain in speedup which can be theoretically seen to be the product of the speedups due to pipeline parallelism and the speedup due to task parallelism.

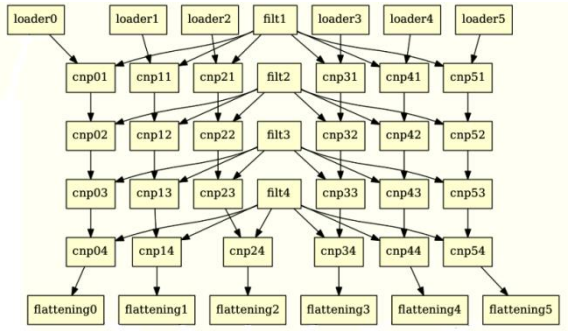

Fig. 14: Master CNN pipeline architecture for NEU Dataset composed of six pipelines

The representative times taken for the different stages of the master CNN Pipeline architecture for NEU Dataset is mentioned in Table III below

TABLE III
Input Parameters: Experimentally determined times of the stages of the master CNN pipeline

Stage	Execution Time, T in seconds					
LOAD	5.89	5.63	5.91	5.83	5.71	5.67
CNP1	23.2	22.5	22.4	22.7	22.7	22.8
CNP2	6.56	6.43	6.44	6.46	6.36	6.22
CNP3	1.81	1.76	1.84	1.76	1.78	1.75
CNP4	0.68	0.66	0.64	0.66	0.66	0.65
FLAT	0.05	0.05	0.05	0.05	0.04	0.05

The graphical representation of the experimentally determined representative times of the stages of the master CNN pipeline is graphically depicted in Fig. 15.

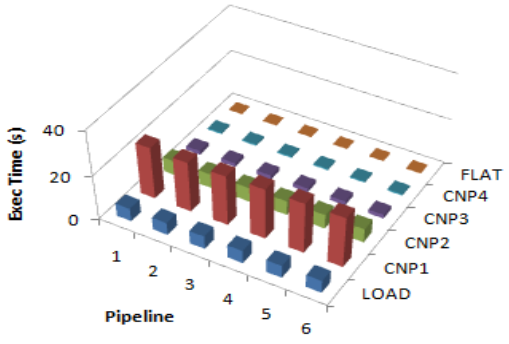

Fig. 15: Experimentally Determined Times of the stages of the CNN software pipeline

IV. RESULTS AND DISCUSSION

In this section, the speedup in computation obtained through the high performance pipelined version of the CNN algorithm vis-a-vis the non-pipelined version is discussed.

Speedup

The speedup is defined as the ratio of the time taken by the non-pipelined version to the time taken by the pipelined version on a given problem.

Pipeline with Equal Stage Execution Times:

For a p-stage pipeline with m samples and equal time taken by the individual stages, the speedup can be derived as follows:

$$Speedup = \frac{mp}{m+(p-1)} \tag{1}$$

Pipeline with Unequal Stage Execution Times:

For a p-stage pipeline with m samples and unequal time taken by the individual stages, speedup can be derived as follows:

$$Speedup = \frac{m\sum_{1}^{p}\tau_i}{\sum_{1}^{p-1}\tau_i + m\tau_p} \tag{2}$$

where τ_i is the time taken by stage P_i.

From Table II and Fig. 13, it is clearly observed that the CNN pipeline involves stages with unequal times.

For a problem with 10000 samples, which represents the size of typical datasets, the relation between speedup and the number of equal pipeline stages is shown in Fig. 16.

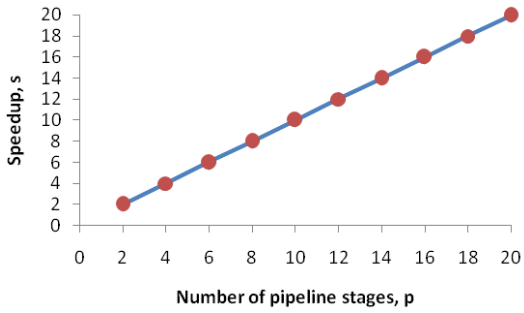

Fig. 16: Relationship between speedup and number of equal pipeline stages with 10k instances

It can be seen that in this ideal scenario where the time taken by individual pipeline stages is the same, the speedup is nearly equal to the number of pipeline stages. The relationship is linear and scales excellently with the number of pipeline stages.

This may lead to the conclusion that the non-pipelined algorithm should be decomposed into a large number of small pipeline stages for maximum performance. Such a conclusion would be premature and naïve, however. In the real world, the depth of a pipeline cannot be increased indefinitely. This is because the individual blocks or pipelines have to be mapped to different CPU cores during parallel execution. These blocks and pipelines are required to communicate the output as well as any intermediate data between each other. This may involve inter-core, inter-CPU as well as inter-node communication. Presence of many such blocks would increase the communication leading to a point where the communication overheads start dominating computation and the performance starts deteriorating.

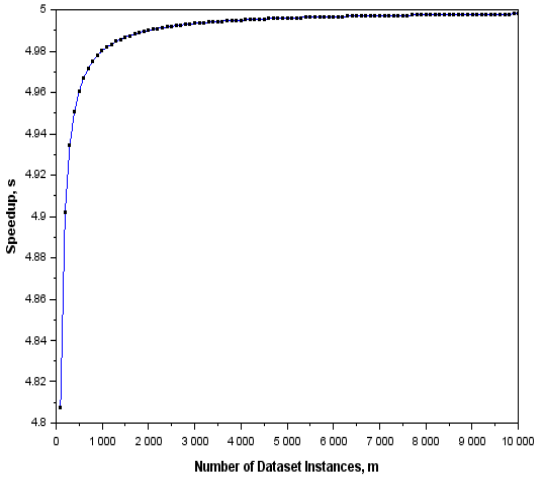

Fig. 17: Relation between speedup and number of instances in a pipeline with 5 equal stages

For a problem which has five equal pipeline stages, the relation between speedup and the number of instances is shown in Fig. 17. It can be seen that for a certain fixed number of equal stages in a pipeline which is five in this example, the speedup approaches the number of equal pipeline stages with an increase in the number of instances, m.

For a problem with 10000 samples and five unequal pipeline stages, e.g., 1, 1.5, 1.75, 2 and 2.2 time units, the speedup can be calculated to be 3.8. From this example and the formula for speedup in the case of unequal pipeline stages, it can be seen that the speedup and therefore the performance of the pipelined algorithm is higher if the different stages have as equal times as possible. The possiblity of aggregating small consecutive pipeline stages into a single pipeline stage and the partitioning of a larger stage into two or more smaller stages allows for some degree of flexibility in dealing with unequal pipeline stage times.

The data of Table II mentions the experimentally determined times of the different stages of a CNN pipeline. This data is useful for the determination of the relative performance of the pipelined version of the CNN algorithm in relation to the non-pipelined version of the same. From the above data, the speedup is calculated as 4.02.

From the experimentally determined timing data given in Table III related to the master CNN pipeline architecture of Fig. 14, for North Eastern University's surface defect dataset, the speedup of the six individual pipeline stages can be computed to be 3 approximately. From a look at the design of the master CNN pipeline architecture of Fig. 14, it is clear that the six pipelines which compose this architecture are identical in all respects including the stages, their number and arrangement, and the convolutional filters as well. It is therefore a reasonable expectation that the time taken for the execution of the individual stages or blocks and the pipelines would be identical. This fact is borne out through the data revealed on experimentation.

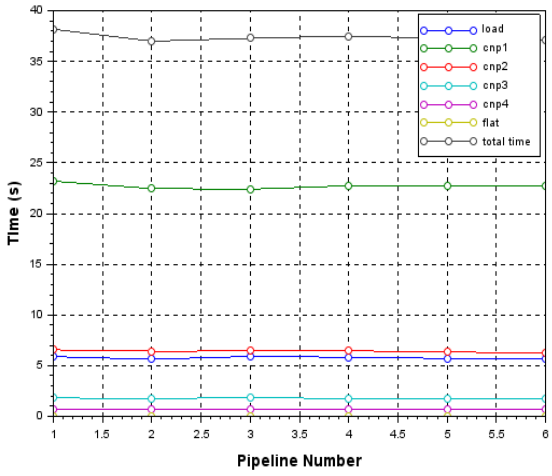

Fig. 18: Stage and Total Times of CNN Pipeline

Two important observations can be made in this context. First is that the pipelines have been designed to be identical and therefore their execution times are nearly equal. Second is that the pipelines are completely independent of each other as can be seen from the stage or task reliance graph of Fig. 14. In view of the two reasons above, it is possible to achieve task parallelism in this master CNN pipeline. Task parallelism by mapping individual pipelines to different compute nodes can therfore lead to nearly six fold performance gain over the serial counterpart. This combined with the three fold performance gain from pipelined operations results in an 18 times performance gain over the serial equivalent since the gains are multiplicative.

Performance gain of 18 times is remarkable considering the fact that there is no other performance enhancing parameter which has been included in the discussion or analysis as yet. For instance, no parallelization of individual computational stages or blocks has been discussed on multicore CPUs except the indpendent mapping and execution of the blocks on individual CPU cores. Specialized and esoteric hardware accelerators like graphic processor units and tensor processing units, etc. are also entirely excluded from the current discussion.

From the above analysis it is reasonably inferred that the pipelined convolutional type deep learning algorithm has a markedly higher perfornance than the conventional non-pipelined or monolithic CNN algorithm. If the design is for high performance then the pipelined CNN algorithm of Table II and Fig. 10 is a clear winner having a performance gain of up to four times when compared with the equivalent monolithic traditional CNN algorithm. This performance gain can be attributed solely to pipeline parallelism and is not dependent or influenced by any other factor.

Further, in the context of the master pipeline CNN architecture designed for the NEU surface defects dataset, the performance gain due to pipeline parallelism is approximately 3. The performance gain due to task parallelism is approximately 6. These gains are multiplicative in nature and amplify each other to result in a net performance gain of 18 times over the equivalent monolithic serial non-pipelined CNN algorithm.

An 18-fold increase in the performance through pipelined redesign of the convolutional neural network and task mapping is considerable given the fact that this performance gain is purely the result of pipelining the algorithm design, domain partitioning and mapping with no other performance enhancing parameter included in the analysis.

V. CONCLUSIONS

The concept of computational pipelines and the speedups resulting from pipeline implementations have been explained in sufficient detail. A high performance pipelined design of a deep learning algorithm of convolutional type has been proposed in the current work. The ideal from the perspective of maximum performance is to design the software pipeline using the principle of nearly equal pipeline stage times. In the limiting cases, the speedup tends to approach the number of pipeline stages which can be treated as the upper bound of the pipeline performance. While equally timed stages may be very desirable from the perspective of performance, they are seldom practicable considering the nature of most algorithms. Two examples of pipeline implementations of CNNs have been designed and analysed. A five stage CNN pipeline comprising of the instance or image loader, the convoluter, the nonlinear transformer, the pooler and the trainer stages has been elucidated. Another master CNN computational pipeline has been designed for a real-world problem of hot rolled steel strip surface defects classification. This CNN pipeline specifically for feature engineering consists of six identical and independent

pipelines. The CNN pipelines have been analyzed on the basis of data obtained from computational experiments by timing the CNN stages. Analysis shows that the pipelined CNN algorithm design in itself results in 4X better performance than its conventional non-pipelined monolithic counterpart for the configuration considered. In the context of the CNN designed for surface defects problem, the performance gain due to pipeline parallelism is approximately 3X while that due to task parallelism is approximately 6X which result in a net performance gain of 18X over the equivalent monolithic serial non-pipelined CNN algorithm. Speedups of the individual blocks on account of other reasons like code optimizations, block parallelization, hardware accelerators, etc. shall also result in a multiplicative effect on the overall speedup.

REFERENCES

[1] Zezhou Cheng, Qingxiong Yang, and Bin Sheng, Deep Colorization, in 2015 IEEE International Conference on Computer Vision (ICCV), Santiago, (2015) 415-423.

[2] Zhang R., Isola P., and Efros A.A., Colorful Image Colorization, European Conference on Computer Vision, 2016 - Springer, vol. 9907, 2016.

[3] Larsson G., Maire M., and Shakhnarovich G., Learning Representations for Automatic Colorization, Computer Vision(ECCV), Lecture Notes in Computer Science - Springer, 9908, 2016.

[4] Hwang, Jeff, and You Zhou, Image Colorization with Deep Convolutional Neural Networks, Stanford University, 2016.

[5] Andrew Owens et al., Visually Indicated Sounds, in 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), Las Vegas, (2016) 2405 - 2413.

[6] I Sutskever, O. Vinyals, and Q.V. Le, Sequence to Sequence Learning, in Proc. Advances in Neural Information Processing Systems 27 (2014) 3104–3112.

[7] K. Cho et. al., Learning phrase representations using RNN encoder-decoder for statistical machine translation, in Proc. Conference on Empirical Methods in Natural Language Processing, (2014) 1724–1734.

[8] Zhang Jiajun and Zong Chengqing, Deep Neural Networks in Machine Translation: An Overview, IEEE Intelligent Systems, 30(5) (2015) 16-25

[9] A. Krizhevsky, I. Sutskever, and G. Hinton, "ImageNet classification with deep convolutional neural networks, in NIPS Proceedings, 2012.

[10] A. G. Howard, Some improvements on deep convolutional neural network based image classification, in International Conference on Learning Representation (ICLR), Banff, Canada, 2014

[11] D. Erhan, C. Szegedy, A. Toshev, and D. Anguelov, Scalable Object Detection Using Deep Neural Networks, in 2014 IEEE Conference on Computer Vision and Pattern Recognition, Columbus, (2014) 2155-2162.

[12] D. Erhan, C. Szegedy, and A. Toshev, Scalable object detection using deep neural networks, in CVPR, 2014.

[13] Alex Graves, Generating Sequences With Recurrent Neural Networks , 2014. [Online]. https://arxiv.org/pdf/1308.0850.pdf

[14] Ilya Sutskever, James Martens, and Geoffrey E Hinton, Generating text with recurrent neural networks, in Proceedings of the 28th International Conference on Machine Learning (ICML-11), New York, NY,(2011) 1017-1024.

[15] Andrej Karpathy and Li Fei-Fei, Deep Visual-Semantic Alignments for Generating Image Descriptions, in 2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), Boston, Massachusetts, (2015) 3128 - 3137.

[16] Ayushi Chahal, Preeti Gulia, Deep Learning: A Predictive IoT Data Analytics Method, International Journal of Engineering Trends and Technology, 68(7) 2020.

[17] P. Seetha Subha Priya, S. Nandhinidevi, M. Thangamani, S. Nallusamy, A Review on Exploring the Deep Learning Concepts and Applications for Medical Diagnosis, International Journal of Engineering Trends and Technology, 68(10) (2020).

[18] Sangeeta, Preeti Gulia, Deep learning based combating strategy for COVID-19 induced increased video consumption, International Journal of Engineering Trends and Technology, 68(7) (2020).

[19] Ferdinand Kartriku, Robert Sowah, Charles Saah Deep Neural Network: An Efficient and Optimized Machine Learning Paradigm for Reducing Genome Sequencing Error, International Journal of Engineering Trends and Technology, 68(9) (2020).

[20] Ramya T.E., Marikkannan, M. Investigations on Combinational Approach for Processing Remote Sensing Images Using Deep Learning Techniques, International Journal of Engineering Trends and Technology, 67(8) (2019).

[21] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton, Deep Learning, 521 (2015) 436.

[22] X. W. Chen and X. Lin, Big Data Deep Learning: Challenges and Perspectives, IEEE Access, 2 (2014) 514-525.

[23] M.M. Najafabadi et. al., Deep learning applications and challenges in big data analytics, Journal of Big Data, 1 (2015).

[24] P. Angelov and A. Sperduti, Challenges in Deep Learning, in European Symposium on Artificial Neural Networks, Computational Intelligence and Machine Learning, Bruges (Belgium), (2016) 27-29.

[25] Xian-He Sun, Yong Chen, and Surendra Byna, Scalable Computing in the Multicore Era, in Proceedings of the Inaugural Symposium on Parallel Algorithms, Architechures and Programming, Hefei: University of Science and Technology of China Press, 2008.

[26] M. Tanveer, M.A. Iqbal, and F. Azam, Using Symmetric Multiprocessor Architectures for High Performance Computing Environments, International Journal of Computer Applications, 27(9)(2011)

[27] M. B. Giles and I. Reguly, Trends in high-performance computing for engineering calculations, in Phil.Trans.R.Soc.A, 2014.

[28] J. Dean et. al., Large Scale Distributed Deep Networks, in 25th International Conference on Neural Information Processing Systems, Lake Tahoe, Nevada, (2012) 1223-1231.

[29] J. Hauswald et. al., DjiNN and Tonic: DNN as a service and its implications for future warehouse scale computers, in Proceedings 42nd Annual International Symposium on Computer Architecture (ISCA), Portland, OR, USA, (2015) 27-40.

[30] V. Hegde and S. Usmani. (2016) Parallel and Distributed Deep Learning, Stanford University Online Report.

[31] M. Bouache and J. Glover, Deep Learning GPU-Based Hardware Platform Hardware and Software Criteria and Selection, in ICS-2016, Istanbul, Turkey, 2016.

[32] Q. Le et. al., On Optimization Methods for Deep Learning, in Proceedings of the International Conference on Machine Learning, Washington, 2011.

[33] J. Keuper and F.J. Pfreundt, Asynchronous Parallel Stochastic Gradient Descent A Numeric Core for Scalable Distributed Machine Learning Algorithms, in Proceedings of the Workshop on Machine Learning in High-Performance Computing Environments, Austin, TX, USA, 2015.

[34] V. Vanhoucke, A. Senior, and M. Z. Mao, Improving the speed of neural networks on CPUs, in Proceedings of the Deep Learning and Unsupervised Feature Learning NIPS Workshop, Granada Spain, 2011.

[35] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan, Deep Learning with Limited Numerical Precision, Journal of Machine Learning Research, 37 (2015).

[36] S Chetlur et. al. (2014) cuDNN: Efficient Primitives for Deep Learning. [Online]. https://arxiv.org/pdf/1410.0759.pdf

[37] A. Delong. Practical Guide to Matrix Calculus for Deep Learning,http://www.psi.toronto.edu/~andrew/papers/matrix_calculus_for_learning.pdf [Online]

[38] Baoyuan Liu, Min Wang, H. Foroosh, M. Tappen, and M. Penksy, Sparse Convolutional Neural Networks, in 2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), Boston, MA, (2015) 806-814.

[39] C. Ionescu, O. Vantzos, and C. Sminchisescu, Matrix Backpropagation for Deep Networks with Structured Layers, in 2015 IEEE International Conference on Computer Vision (ICCV), Santiago, (2015) 2965-2973.

[40] Y. Zhang and S. Zhang, Optimized Deep Learning Architectures with Fast Matrix Operation Kernels on Parallel Platform, in 2013 IEEE 25th International Conference on Tools with Artificial Intelligence, Herndon, VA, (2013) 71-78.

[41] Ciresan, Dan, Ueli Meier, and Jürgen Schmidhuber. 2012. Multi-column deep neural networks for image classification. In 2012 IEEE Conference on Computer Vision and Pattern Recognition 3642-3649.
[42] Ciresan, Dan, Ueli Meier, Jonathan Masci, Luca M. Gambardella, and Jurgen Schmidhuber. 2011. Flexible, High Performance Convolutional Neural Networks for Image Classification. in 2013 International Joint Conference on Artificial Intelligence, 1237–1242.
[43] Lawrence, Steve, C. Lee Giles, Ah Chung Tsoi, and Andrew D. Back. Face Recognition: A Convolutional Neural Network Approach, 1997 IEEE Transactions on Neural Networks, 8(1) 98-113.
[44] Russakovsky, O., Deng, J., Su, H. et al. ImageNet Large Scale Visual Recognition Challenge. Int J Comput Vis vol. 115**,** 2015
[45] Sunil Pandey, Naresh Kumar Nagwani, Shrish Verma. Parallel and Scalable Deep Learning Algorithms for High Performance Computing Architectures International Journal of Engineering Trends and Technology, 69(4) (2021) 236-246.
[46] Mario Rossainz-López, Manuel I. Capel, Odon D. Carrasco-Limón, Fernando Hernández-Polo, Bárbara E. Sánchez-Rinza, Implementation of the Pipeline Parallel Programming Technique as an HLPC: Usage, Usefulness and Performance, Annals of Multicore and GPU Programming, 4 (1). ISSN: 2341-3158.
[47] I-Ting Angelina Lee, Charles E. Leiserson, Tao B. Schardl, Jim Sukha, Zhunping Zhang, On-the-Fly Pipeline Parallelism, ACM Transactions on Parallel Computing, 2(3).
[48] Halling-Brown M, Shepherd AJ. Constructing computational pipelines. Methods Mol Biol. 2008;453:451-70. doi: 10.1007/978-1-60327-429-6_24. PMID: 18712319
[49] Vivien Marx, When Computational Pipelines Go Clank. Nature Methods, vol 17, 659–662 (2020)
[50] Saurav Chatterjee and Jay Strosnider, Distributed Pipeline Scheduling: A Framework for Distributed, Heterogeneous Real-Time System Design, The Computer Journal, 38(4) (1995).
[51] A. Navarro, R. Asenjo, S. Tabik and C. Cascaval, Analytical Modeling of Pipeline Parallelism, 2009 18th International Conference on Parallel Architectures and Compilation Techniques, (2009) 281-290, doi: 10.1109/PACT.2009.28.
[52] Michael I. Gordon, William Thies, and Saman Amarasinghe. 2006. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. In Proceedings of the 12th international conference on Architectural support for programming languages and operating systems (ASPLOS XII). Association for Computing Machinery, New York, NY, USA, 151–162.
[53] K. Song and Y. Yan, “A noise robust method based on completed local binary patterns for hot-rolled steel strip surface defects,” Applied Surface Science, 285 (2013) 858-864.
[54] Yu He, Kechen Song, Qinggang Meng, Yunhui Yan, “An End-to-end Steel Surface Defect Detection Approach via Fusing Multiple Hierarchical Features,” IEEE Transactions on Instrumentation and Measuremente, 69(4) (2020) 1493-1504.
[55] Hongwen Dong, Kechen Song, Yu He, Jing Xu, Yunhui Yan, Qinggang Meng, PGA-Net: Pyramid Feature Fusion and Global Context Attention Network for Automated Surface Defect Detection, IEEE Transactions on Industrial Informatics, 2020.