High Performance Parallel Implementations for Convolutional Neural Networks

Atharva Anand Joshi
Electrical and Computer Engineering
Carnegie Mellon University
Pittsburgh, PA
atharvaa@andrew.cmu.edu

Eugene Min
Electrical and Computer Engineering
Carnegie Mellon University
Pittsburgh, PA
ekmin@andrew.cmu.edu

Ananya Ayasi
Electrical and Computer Engineering
Carnegie Mellon University
Pittsburgh, PA
aayasi@andrew.cmu.edu

***Abstract*—This paper tackles the computational problem of Convolutional Neural Networks (CNN), which is a popular deep learning architecture used with images. The goal is to understand how parallelism benefits the various subroutines corresponding to the convolution layer. To that end, both OpenMP and CUDA are explored. This work experiments with multiple design choices with the goal of optimizing on the speedup over the basic sequential implementation. These design choices are discussed in detail with the reasonings. The paper ends with some directions on extending the current work.**

***Index Terms*—Convolutional Neural Networks, Parallelism, OpenMP, CUDA, High Performance Computing, Deep Learning, Image Processing, CPU, GPU**

I. Introduction

A. Problem Statement

Convolution neural networks (CNNs) have been incorporated into resource-constrained edge devices to intelligently manage and process local data coming from a variety of sensors [1]. To that end, thread parallelism has been used to boost the performance of neural networks as the software architecture of neural networks and the lack of dependency between neurons in each inference layer provide significant opportunity for parallelism in a multiprocessor platform.

The concept of parallelism in the case of deep neural networks is a quite popular research problem. [2] implements a CNN architecture with OpenMP for the detection of corn leaf diseases. Similarly parallelism for real-time object detection by [3] is also implemented using OpenMP and is an important use case, when it comes to systems that require real time detection like autonomous vehicles. Our work has been inspired by [4] that utilizes parallelism for Handwritten Character Recognition (HCR).

Our paper focuses on studying the effect of various configurations on the speedups for the OpenMP and CUDA implementations of a CNN architecture for image classification. Through this exercise, we expect to gain insights on which approaches work and which do not. The implementation choices would then be discussed in detail. In summary, we intend to optimally utilize the hardware to achieve the best possible speedups for CNNs.

B. Convolutional Neural Networks

Convolutional Neural Networks are a type of feed-forward neural networks that are widely used for image processing tasks. With an input layer, hidden layers and an output layer, the hidden layers consist of one or more layers that perform convolutions. As the convolution kernel slides along the input matrix, the convolution operation generates a feature map, which in turn contributes to the input of the next layer. This is followed by other layers such as pooling layers and fully connected layers.

Microprocessor development efforts continue to concentrate on adding cores rather than increasing single-thread performance [5]. Hence, it's essential to equip computationally complex tasks to fully utilize the capabilities of advanced processing units by enabling parallelism.

The parallelism in the convolution operation is highly scalable in terms of the number of threads. Each component can be broken down into a large number of small independent operations. This makes it ideal for GPU implementation which has a lot more threads than the CPU. The main focus of this paper is the parallelization of a convolutional layer that performs an inner product of several filters with the input matrix/ image. For the purpose of exploring the parallelized implementations, the baseline chosen was a codebase that implements CNN in C++. The baseline consists of the inference performance over the MNIST test dataset.

C. Parallelism Analysis

For this paper, the benefits of parallelism for CNNs is mainly analyzed for inference. Typically, CNN inference is a resource-hungry task as intermediate results and weights take a lot of memory footprint, and various operations require massive computation [8]. Hence, parallelism of CNN inference is an interesting problem and this is explored through OpenMP and CUDA programming.

OpenMP (Open Multi-Processing) [9] is an API that supports multi-platform shared-memory multiprocessing programming in C, C++, and Fortran on many platforms, instruction-set architectures and operating systems. It consists of a set of compiler directives, library routines, and environment variables that influence run-time behavior.

Compute Unified Device Architecture (CUDA) [10] is a parallel computing platform and API that allows software to use GPUs for accelerated general-purpose processing, an approach called general-purpose computing on GPUs (GPGPU). CUDA is designed to work with programming languages such as C, C++, Fortran and Python. CUDA-powered GPUs also support programming frameworks such as OpenMP, OpenACC and OpenCL.

In the interest of achieving parallelism, there are different existing methods to achieve the same. Data parallelism is the most common form of parallelism due to its simplicity, for which the dataset is split into several shards, and each shard allocated to a device. However, there could be redundancy issues due to the model weights being shared across multiple devices.

Another paradigm of parallelism is model parallelism [7], where the model is split and distributed over an array of devices. This methodology will be the core of our CUDA implementation. The CNN subroutine we mainly focused on parallelizing is the Convolution Layer. It has been found that implementing parallelism for the other subroutines including fully connected layer and pooling, did not really improve the performance, as will be discussed further in the paper.

The Convolution operation[1] is visualized in figure 1. It can be broken down into two components - First is the element wise multiplication of the filter weights with pixels in the window and summing the products. Second is the movement of the filters across the dimensions of the image. Both the components are completely independent since they are computing different segments on the image. Each window leads to a new pixel in the resulting image which allows us to compute each pixel in parallel. Hence, the convolution operation is a great candidate for implementing parallelism for CNNs.

D. Dataset

Convolutional Neural Networks have gained popularity for their image processing applications over the years. Hence, to explore parallelism, we used CNN for the simple task of image classification. To that end, the MNIST database (Modified National Institute of Standards and Technology database) which is a large database of handwritten digits, is used.

The Modified NIST database [6] was constructed from NIST's Special Database 3 and Special Database 1 which contain binary images of handwritten digits. Being a subset of a larger set available from NIST, the database has a training set of 60,000 examples and a test set of 10,000 examples. The digits have been size-normalized and centered in a fixed-size image. The images were centered in a 28×28 image by computing the center of mass of the pixels, and translating the image so as to position this point at the center of the 28×28 field. The dataset is popular for pattern recognition given that minimal effort is required for preprocessing and formatting.

[1]Image Credits: Intuitively Understanding Convolutions for Deep Learning

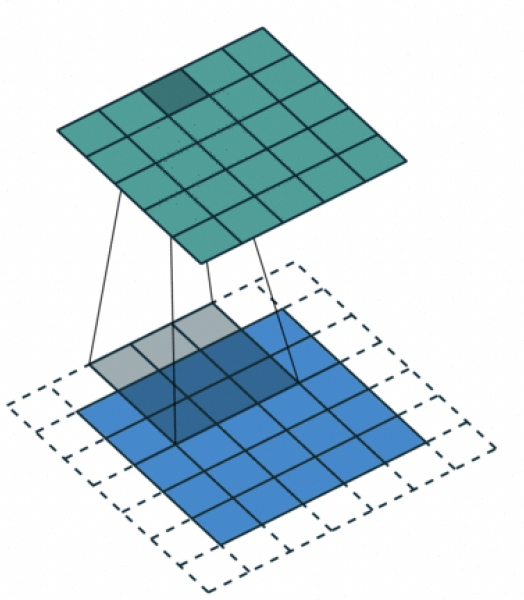

Fig. 1: Visualization of the Convolution Operation

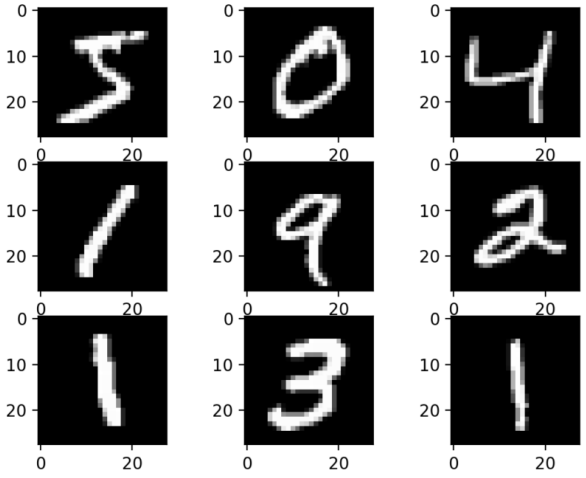

Fig. 2: MNIST Dataset

II. Final Design

A. Hardware Resources

Throughout this paper, our hardware platform for the baseline as well as our parallel implementations is the ECE CMU machine (ECE019). All the authors have used the same machine for development and experimentation. This machine is equipped with the Intel(R) Xeon(R) Silver 4208 CPU and the Nvidia Tesla T4 GPU [11]. The details of the underlying resources are included in tables I and II.

Parameter	Value
CPU Architecture	x86_64
Number of CPUs	32
Number of Threads per core	2
Number of Cores per socket	8
Number of Sockets	2
Number of NUMA nodes	2

TABLE I: Details of the CPU architecture

Parameter	Value
GPU Architecture	Turing Architecture
GPU Memory	15360 MiB
Number of streaming multiprocessor (SMs)	40
Number of CUDA cores	2560
Number of Tensor cores	320
Peak single precision performance	8.1 TFLOPS

TABLE II: Details of the GPU architecture

B. Baseline

The baseline implementation used in this paper is simple_cnn[2]. As the name suggests, this is a minimal sequential C++ implementation of the convolutional neural networks. The codebase does not parallelize the application and we look into every opportunity to speed up using our OpenMP and CUDA extensions. The comparison is based on the inference performance over the MNIST test dataset which consists of 10,000 samples. The baseline yields deterministic results for inference, therefore we compare our parallelized implementation for correctness with this. The accuracy of the model trained using the baseline is 96.55%. We obtain the same results using our parallelized implementations. Figure 3 displays the performance numbers across different data sizes. As expected, the computation time grows linearly with the size of the dataset. The total computation time for the baseline over the complete test dataset is 6543.54 milliseconds.

C. OpenMP Implementation

In the context of OpenMP, we primarily explore data parallelism with respect to the convolution and the fully-connected layers. This involves parallelly computing the output for each input image. The performance is analysed across three dimensions: (1) Number of Threads, (2) Thread Affinity, and (3) OpenMP Places. Since we saw limited benefits of parallelism on the fully-connected layer, we focused on the convolution layer for (2) and (3).

D. CUDA Implementation

As explained in Section 1 (C), the Convolution operation has great scope for parallelism. Moreover, the parallelism is highly scalable in terms of the number of threads. We can break each component into a large number of small independent operations. In the CUDA programming model, the GPU is treated as a co-processor onto which an application running on a CPU can launch a massively parallel compute kernel [12]. In this implementation, we use the banked shared memory, which is the second fastest memory on the GPU after the registers.

[2]Baseline Link: https://github.com/can1357/simple_cnn

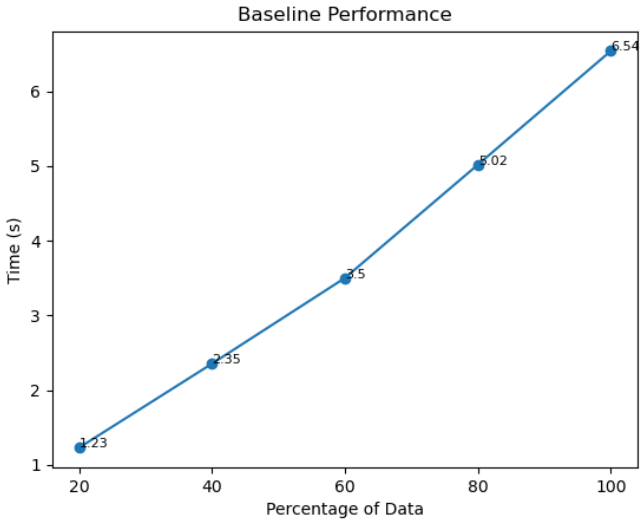

Fig. 3: Baseline Performance

1) Parallelizing Input Pixels: The initial direction was to use each thread to perform computations corresponding to one input pixel in a window. Hence, each thread performs the scalar multiplication, followed by summation across the window. However, this presents three problems:

(a) The reduction of sum across the window is expensive. It offsets the benefits provided by the parallel computation for the window by introducing an implicit barrier.
(b) This requires an extremely large number of threads: 16×625 pixels $\times 8$ filters $= 80000$ threads for a 28×28 input image convolved with 4×4 filters, 8 in number.
(c) Resolving the bank conflicts in shared memory becomes intractable. This results in limited benefits of using shared memory to speedup the read/write access.

2) Parallelizing Output Pixels: Due to the challenges caused by the previous approach, we instead decided to parallelize each output pixel. This implies that each thread now sequentially traverses through a single window and computes its corresponding output pixel. Although this might seem to be less parallel than the previous approach, it has several benefits which leads to an overall improved performance with lesser requirement for resources.

(a) There is no reduction of sum involved. This is because each thread computes its own sum which leads to truly independent operations across the threads.
(b) This approach requires a total of only 625 pixels $\times$ 8 filters $=$ 5000 threads for a 28×28 input image convolved with 4×4 filters, 8 in number.
(c) Each thread now computes its own output pixel and the image is stored in a row major order in the shared memory. As a result each thread in a warp accesses elements from different banks, resulting in no bank conflict. This does not require a sophisticated design of the intermediate layout of the shared memory, extracting

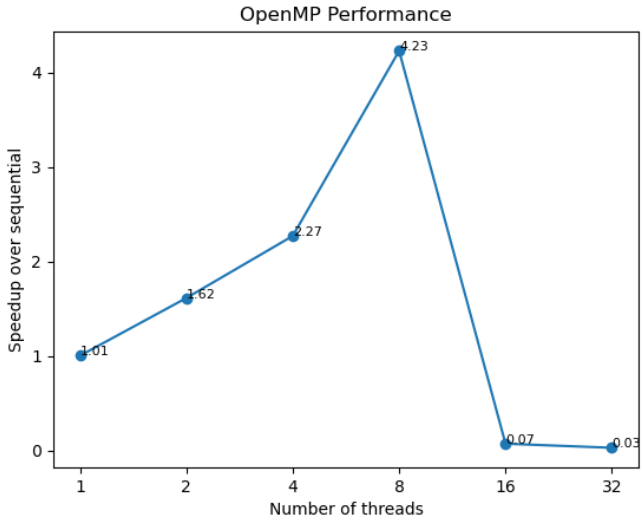

Fig. 4: OpenMP Performance

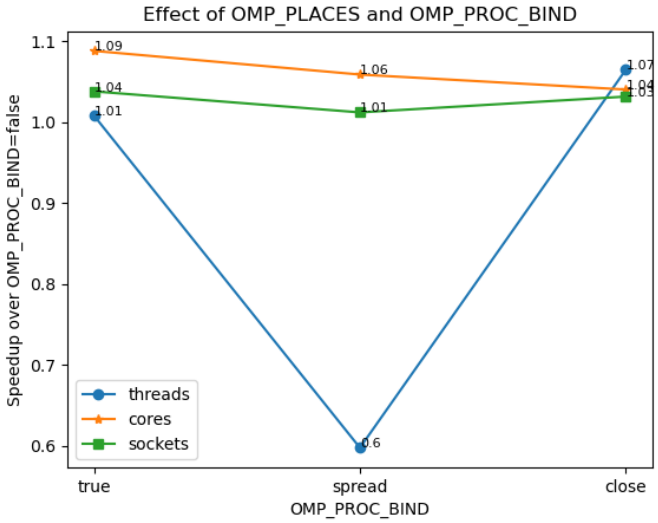

Fig. 5: Thread Affinity and OpenMP Places

the best performance out of it.

3) Using multiple streaming multiprocessors: Our intermediate CUDA implementation uses a single streaming multiprocessor (SM) for computing 8 filter convolutions. However, each filter convolution is again independent of each other and can further be parallelized. Hence, we analysed the speedup achieved by parallelizing the filter convolutions on multiple SMs, with the help of thread blocks.

III. Performance Results

A. OpenMP Implementation

Figure 4 shows the plot for the speedup with the number of threads. We observe that the speedup increases with the addition of threads upto 8 threads, after which it drops. This implies that the ideal balance between computation workload size and overhead of parallelism seems to be at 8 threads in which afterwards, the overhead starts to overshadow any benefits. Morever, as we further increase the number of threads, they could also be possibly competing for resources. The plot only shows the performance for parallelizing the convolutional layer. We attempted to add more threads to ReLU and fully connected layers but we observed either degraded or nearly equal performance from before.

The results for our experiments with Thread Affinity and OpenMP Places are summarized in figure 5. The only mapping of software to hardware thread that contributes a significant difference in speedup, with 4 threads, are threads that are spread out. This makes intuitive sense because this configuration places threads as far as possible hardware wise which likely narrows the possibility of threads taking advantage of locality from L1 cache or the LLC. All threads need to retrieve data from main memory. The rest of the configurations are nearly identical since the number of possible places are narrowed and the mappings are essentially the same.

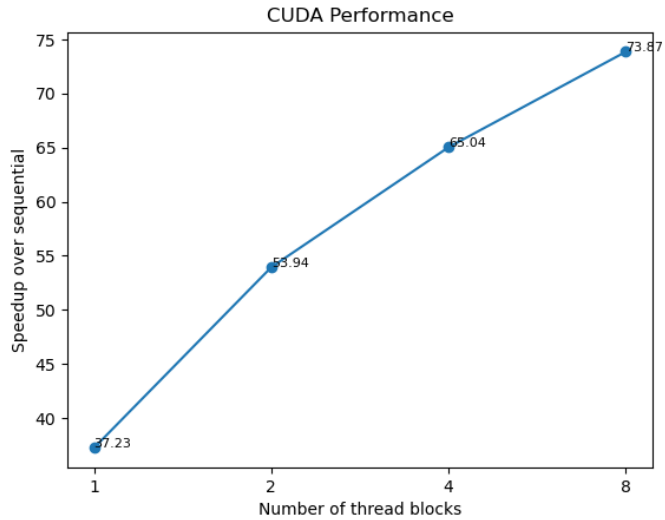

Fig. 6: CUDA Performance

B. CUDA Implementation

Figure 6 displays the speedup achieved through the use of multiple streaming multiprocessors. Though we cannot determine whether each thread block runs on a separate SM every time, we observe consistent results in every run. Note that the use of n thread blocks implies that $8/n$ filters are computed by each thread block. There is a clear increasing trend on the speedup with the addition of thread blocks, implying that our approach is scalable with respect to the number of SMs. This increase is not linear though, which we attribute to the overhead of data transfer on the GPU. The copy from the main memory to the shared memory cannot be sped up with the addition of more SMs because each SM has its own shared memory.

Implementation	Time(ms)	Speedup
Baseline	6543.54	1
OpenMP	1546.85	4.23
CUDA	88.5772	73.87

TABLE III: Summary of best performances

In table III, we summarize the best performances achieved through our OpenMP and CUDA implmentations.

IV. Future Directions

A. Size of problem

Through this work, we would like to restate that "Parallelism is not free" and it is important that the amount of work should be large enough for the overhead from spawning and coordinating multiple threads to amortize. Currently, the size of the images is 28×28 which is not very large compared to those seen in daily life today. While significant speedups are observed through our implementations, these would be much greater if we work with larger images. For instance, [13] exploits parallelism within the spatial domain allowing scaling to continue beyond the mini-batch size. Training deep nets on large data is an HPC problem, and tackling it requires exploiting as much parallelism as possible. It might be interesting to assess the scalability of this work over the ImageNet [14] dataset whose images are of size 469×387.

B. Data and Model Parallelism

In this paper, data parallelism is explored in the OpenMP implementation and model parallelism is explored in the CUDA implementation. This work can be extended by combining the two approaches in a heterogeneous setup as explored in [15]. This could lead to an even better performance and can be achieved with the help of task parallelism on the CPU and streams on the GPU.

References

[1] G. Abich, R. Garibotti, J. Gava, R. Reis and L. Ost, "Impact of Thread Parallelism on the Soft Error Reliability of Convolution Neural Networks," 2022 IEEE 13th Latin America Symposium on Circuits and System (LASCAS), Puerto Varas, Chile, 2022, pp. 1-4, doi: 10.1109/LASCAS53948.2022.9789088. keywords: Performance evaluation;Convolution;Instruction sets;Neural networks;Memory management;Parallel processing;Software reliability;Thread Parallelism;Soft Error Reliability;Machine Learning;Edge Devices,

[2] D. A. Padilla, R. A. I. Pajes and J. T. De Guzman, "Detection of Corn Leaf Diseases Using Convolutional Neural Network With OpenMP Implementation," 2020 IEEE 12th International Conference on Humanoid, Nanotechnology, Information Technology, Communication and Control, Environment, and Management (HNICEM), Manila, Philippines, 2020, pp. 1-6, doi: 10.1109/HNICEM51456.2020.9400004. keywords: Process control;Production;Agriculture;Convolutional neural networks;OpenMP;Convolutional Neural Network;Deep Learning;Image Processing,

[3] M. Rakhimov, J. Elov, U. Khamdamov, S. Aminov and S. Javliev, "Parallel Implementation of Real-Time Object Detection using OpenMP," 2021 International Conference on Information Science and Communications Technologies (ICISCT), Tashkent, Uzbekistan, 2021, pp. 1-4, doi: 10.1109/ICISCT52966.2021.9670146. keywords: Information science;Multicore processing;Operating systems;Image processing;Streaming media;Parallel processing;parallel processing;object detection;convolutional neural networks;image processing;OpenMP;multicore computing,

[4] M. Musaev and M. Rakhimov, "Accelerated Training for Convolutional Neural Networks," 2020 International Conference on Information Science and Communications Technologies (ICISCT), Tashkent, Uzbekistan, 2020, pp. 1-5, doi: 10.1109/ICISCT50599.2020.9351371. keywords: Training;Handwriting recognition;Neural networks;Graphics processing units;Parallel processing;Acceleration;Character recognition;Handwritten character recognition;Artificial Intelligence;Machine learning;Deep Learning;Convolutional Neural Networks;Parallel processing;OpenMP,

[5] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone and J. C. Phillips, "GPU Computing," in Proceedings of the IEEE, vol. 96, no. 5, pp. 879-899, May 2008, doi: 10.1109/JPROC.2008.917757. keywords: Graphics;Central Processing Unit;Physics computing;Engines;Arithmetic;Bandwidth;Microprocessors; Hardware;Computational biophysics;Performance gain;General-purpose computing on the graphics processing unit (GPGPU);GPU computing;parallel computing,

[6] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," Proc. IEEE, vol. 86, no. 11, pp. 2278–2324, Nov. 1998. keywords: Neural networks;Pattern recognition;Machine learning;Optical character recognition software;Character recognition;Feature extraction;Multi-layer neural network;Optical computing;Hidden Markov models;Principal component analysis,

[7] H. Zhou, M. Li, N. Wang, G. Min and J. Wu, "Accelerating Deep Learning Inference via Model Parallelism and Partial Computation Offloading," in IEEE Transactions on Parallel and Distributed Systems, vol. 34, no. 2, pp. 475-488, 1 Feb. 2023, doi: 10.1109/TPDS.2022.3222509. keywords: Computational modeling;Parallel processing;Adaptation models;Task analysis;Processor scheduling;Kernel;Internet of Things;Mobile edge computing;fused-layer;DNN inference;partial offloading;model parallelism,

[8] J. Du et al., "Model Parallelism Optimization for Distributed Inference Via Decoupled CNN Structure," in IEEE Transactions on Parallel and Distributed Systems, vol. 32, no. 7, pp. 1665-1676, 1 July 2021, doi: 10.1109/TPDS.2020.3041474. keywords: Parallel processing;Kernel;Convolution;Computational modeling;Optimization;Performance evaluation;Task analysis;Intelligent applications;distributed deep learning;distributed inference;model parallelism;decoupled CNN structure,

[9] Barbara Chapman; Gabriele Jost; Ruud van der Pas, "Under the Hood: How OpenMP Really Works," in Using OpenMP: Portable Shared Memory Parallel Programming , MIT Press, 2007, pp.277-305.

[10] NVIDIA Corporation. NVIDIA CUDA Programming Guide, 1.1 edition, 2007.

[11] E. Lindholm, J. Nickolls, S. Oberman and J. Montrym, "NVIDIA Tesla: A Unified Graphics and Computing Architecture," in IEEE Micro, vol. 28, no. 2, pp. 39-55, March-April 2008, doi: 10.1109/MM.2008.31. keywords: Graphics;Computer architecture;Parallel processing;Pipelines;Concurrent computing;Load management;Multicore processing;Parallel programming;Portable computers;Workstations;Hot Chips 19;GPU;parallel processor;SIMT;SIMD;unified graphics and parallel computing architecture;graphics processing unit;cooperative thread array;Tesla,

[12] A. Bakhoda, G. L. Yuan, W. W. L. Fung, H. Wong and T. M. Aamodt, "Analyzing CUDA workloads using a detailed GPU simulator," 2009 IEEE International Symposium on Performance Analysis of Systems and Software, Boston, MA, USA, 2009, pp. 163-174, doi: 10.1109/ISPASS.2009.4919648. keywords: Analytical models;Yarn;Graphics;Parallel processing;Process design;Parallel programming;Computational modeling,

[13] N. Dryden, N. Maruyama, T. Benson, T. Moon, M. Snir and B. Van Essen, "Improving Strong-Scaling of CNN Training by Exploiting Finer-Grained Parallelism," 2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS), Rio de Janeiro, Brazil, 2019, pp. 210-220, doi: 10.1109/IPDPS.2019.00031.

[14] J. Deng, W. Dong, R. Socher, L. -J. Li, Kai Li and Li Fei-Fei, "ImageNet: A large-scale hierarchical image database," 2009 IEEE Conference on Computer Vision and Pattern Recognition, Miami, FL, USA, 2009, pp. 248-255, doi: 10.1109/CVPR.2009.5206848.

[15] J. Yang, X. Long, Z. Ma and C. Yang, "An Automatic Pipeline Parallel Acceleration Framework for Neural Network Models on Heterogeneous Computing Platforms," 2022 5th International Conference on Pattern Recognition and Artificial Intelligence (PRAI), Chengdu, China, 2022, pp. 1154-1158, doi: 10.1109/PRAI55851.2022.9904009.