On Handling AI Tasks in CPU with Low Latency and High Performance

Protiva Das
BRAC University
protiva.das@g.bracu.ac.bd and
Annajiat Alim Rasel
BRAC University
annajiat@gmail.com

No Institute Given

Abstract. Artificial Intelligence (AI) is becoming the striking force of this era. Machine Learning (ML), deep learning (DL), and reinforcement learning (RL) are the three methods to train AI-based systems to elicit intelligence from row data. All these three methods need massive computation power to train AI tasks. CPUs can handle normal ML, DL, or RL tasks, but normally, it is difficult to handle complex tasks. Especially for any DL tasks, it needs a Graphics Processing Unit(GPU) or Tensor Processing Unit(TPU) to accomplish the AI tasks. GPU and TPU are costly and inconvenient for all AI enthusiasts or developers; sometimes, it might hinder their research and development process. This research aims to pave the way for optimally handling ML and DL tasks in the CPU. One complementary method to convey AI tasks besides personal GPU is free or paid cloud systems like Google Colab, AWS, IBM, or Azure Cloud. This option might not ensure data security and confidentiality, and they are sometimes insufficient with respect to the need. However, the proposed method ensures data privacy and integrity, unlike the traditional cloud-based system, which does not have a suitable GPU/TPU. After implementing the developed algorithm for calculating latency and speed, we found the lowest training time for 16 CPU cores(DL:40s, ML: 5s) and the highest training time for 1 CPU core (DL:58s, ML:58s). This approach can be the best reference for deploying models to the end systems.

Keywords: Artificial intelligence, · Machine learning, · Deep learning, · Reinforcement learning, · Cluster computing.

1 Introduction

Artificial Intelligence (AI) is the striking force of this present era. AI technologies are producing astounding results. Machine, deep, and reinforcement learning (ML, DL, RL) are the three branches of AI-based systems. Among them, ML and DL are the data-driven methods, and RL is an event-driven method. Whatever method we choose it needs substantial computation power to accomplish the

given AI tasks of these three types. Usually, CPU computation power is sufficient for ordinary ML and RL tasks, but complexity and performance issues arise when the tasks become complex ML or RL tasks. Inherently, wider complexity arises while the DL task is fetched into a CPU. For any DL tasks, it needs a different kind of computation engine namely Graphics Processing Unit(GPU) or Tensor Processing Unit(TPU). Usually, DL tasks are optimized in GPU/TPU for their massive parallelism and architecture, but GPU and TPU are costly and inconvenient for all AI enthusiasts. Usually, AI personnel handle their ML and RL tasks in local or cloud CPU and DL tasks in cloud GPUs (such as Google Colab, Kaggle, and so on) in the absence of local or personal GPU. However, this way has many drawbacks; data privacy and model integrity are the main concerns. This research provides a way to illustrate how to optimize ML and DL tasks in the CPU. This method ensures data privacy and integrity, unlike the traditional cloud-based system, which does not have a suitable GPU/TPU.

There are few instances in the literature of using CPU and its cluster as the processing unit for AI tasks such as [1], [2], [3], [4], [5], [6]. However, [7] used a method to use CPU and GPU to process the AI tasks. This method must be expressed and demonstrated more to extend the work to a higher dimension and help AI enthusiasts. This research focuses on using CPUs to train ML/DL tasks efficiently in the CPU. This method can be considered a unique implementation of AI tasks using CPUs and their cluster.

The rest of the paper advances as follows: section II presents the methodology section, section III presents the results and discussion, and Section IV covers the conclusion of the research.

2 Methodology

2.1 System Architecture

Normally, CPUs are single-core to multi-core architectures, depending on the vendor. On the other hand, GPUs are multi-core, and the number of cores is more than hundreds to thousands depending on vendor model and architecture. The computation speed of GPU lies in the inherent multi-cores, and performance slowness lies in the fewer cores inside CPUs. However, we can optimize the CPU core utilization to reduce the latency and increase the performance of the CPU-based AI tasks. The proposed architecture provides a thorough description and implementation of the utilization of CPU cores in uplifting AI tasks. Figure 1 presents the various architectures of AI task-compatible devices. We used these devices for AI model training, evaluation, and inference.

Depending on various vendors, CPUs, and GPUs can usually inherit the following architecture and constituents from the vendors.

uCPU This paper defines uCPU as the single-core processor with only an ALU, control unit, and registers. These kinds of devices are primitive computers used during the 2000s or later. It has less usage in accommodating AI tasks. Figure 1 (a) presents this architecture.

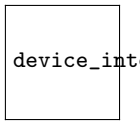

Fig. 1. Internal organizations of AI-tasks compatible devices

mCPU Here, we mean mCPU as the multi-core processor having more than ALU, control unit, and registers inside a CPU chip. These devices started the multi-core computers that were inaugurated after the 2004s or later. One example of this CPU is Intel dual core to core i3 to core i11 CPUs. We want to use these CPUs as the main component in accommodating AI tasks. Figure 1 (b) and (c) present this architecture.

mpCPU The CPU is not a kind of off-the-shelf CPU type. We mean mpCPU as the parallel cluster of the multi-core CPUs mounted in a single motherboard or in different motherboards or PCs to form the cluster to handle AI tasks. It needs to be built as a rack of clusters. Depending on their tasks, this formation is usually found in low-end and high-end servers. Here, mpCPU is the other focus of this research for accomplishing AI tasks. Figure 3 (b) presents this architecture.

uGPU GPU is a single GPU having several inherent cores. These devices are the primitive GPUs invented by NVIDIA [8] in 1999 or later. It has the minimum usage in accommodating AI tasks. Figure 1 (d) presents this architecture.

mGPU mGPU is the multiple copies of uGPU installed in the GPU chip. This kind of architecture is normal in every motherboard of high-configuration computers with a built-in GPU from vendors like NVIDIA or AMD GPU. It is a common scenario for computers or laptops to have mGPUs these days. User can install more GPUs as per their need depending on the availability of expansion slots in the motherboard. As seen from the figure

mpGPU mpGPU is the multiple cluster of uGPU or mGPU installed in a rack or similar platform. This kind of installation is called High Performance Computing Platform(HPC). This configuration can usually be found in cloud computing service providers or a university that has availed it. It is not the common scenario of GPU configuration.

2.2 Device Selection Strategy

Once we have a AI task at hand, one can chose the device using the decision tree strategy a shown in Figure 2. It is common to use GPU or its close cosines for complex ML or normal to complex DL tasks. GPU-based model training,

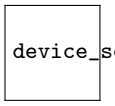

Fig. 2. Device Selection Strategy for Pursuing AI-tasks

Table 1. System configurations

Vendor	CPU Type	Cores	RAM	Conf No
Acer	Core i3	8	4	Conf 1
Asus	Core i5	4	16	Conf 2
Hasee	Core i5	8	24	Conf 3
Hasee	Core i9	16	32	Conf 4

evaluation, and inference always inherently provide high performance and low latency. But what if GPU system is not at hand and not affordable? In such a case CPU driven proposed methodology can pave the ways to accomplish the AI task at hand for both training, evaluation and most possibly model inference. As seen from the figure, if one chooses to complete the AI tasks using uCPU, it might take the slowest possible time, and even more, it might not handle AI tasks. The same process can be handled moderately faster using mCPU, fast way using mpCPU, faster way using uGPU, fastest way using mGPU and superior fastest way using mpGPU. The time cost was reduced from uCPU to mpGPU, but the financial cost increased from uCPU to mpGPU. These authors suggest using the moderate case of the mpCPU if the financial cost is not affordable to maintain. The rest of the work is done with the mCPU and CPU architecture.

2.3 Implementation

While an ML/DL task is fetched into the CPU, only a single core responds to the task normally; multi-core CPU utilization is not assured by default without programming. We utilized all possible CPUs (uCPU, mCPU, mpCPU) for training, testing, and inferring the system data. The uCPU and mCPU are utilized as remain in the CPU package itself. To implement the mpCPU architecture, a cluster of 4 laptops is made using the gigabit Ethernet switch. We used Intel MKL [9] to provide parallelism during training tasks. We utilized the TensorFlow CPU library for CPU-based training, testing, and inference. The detailed MKL system installation can be found here[10]. The installed system works in the parallel architecture to process the data fetched into the neural networks and ordinary machine learning models through muti-threading and multi-tasking based on CPU scheduling organization. We used all possible CPU cores available inside CPUs. The configuration of all devices used in the cluster is given in Table 1.

After installation, we started two sets of AI-based tasks assigned to the system: i) using each device to run the same task individually to the CPU cores

Fig. 3. A typical cluster commuting systems

and ii) running the same task into the CPU cluster. The ML task is documented in the Table 2. We used a sample classification task using the Random Forest algorithm, which had 10000 rows of data, 20 features, and 500 estimators. On the other hand, We utilize a DL task as documented in Table 3. Here, we utilized a sample classification task on the same data using a 3 layers-deep sequential neural network algorithm having network parameters of ReLu activation, binary cross-entropy loss, Adam optimizer, sigmoid activation for the output layer, 100 epochs, and batch size 10. All other DL algorithms can be adopted in the same process. All the ML and DL tasks were accomplished using the Algorithm 1. This algorithm automatically operates based on the type of AT tasks(ML or DL). For each dataset and each ML/DL algorithm, the same modeling tasks(training, testing, and inference) were done as per various CPU cores. Each time, results are saved based on time and accuracy scores. Later, time and performance scores were saved for visualization, and every model was saved accordingly.

Once training was done, the model training performance parameters were saved for each item. The training performance of the ML algorithm case was saved in Table 4, where we can see the performance for each device case individually. As seen from the table, the lowest ML performance was revealed for each device using a single CPU core, and the highest possible performance was yielded for the utilization of most CPU cores.

Similarly, the training performance of DL algorithm cases was saved in Table 5, where we can see each device case's performance individually. The table shows that the lowest DL performance was revealed for each device using a single CPU core, and the highest possible performance was yielded for utilizing the highest CPU cores. However, it also found that the ML and DL tasks took similar training time since the dataset was the same. The results may vary depending on the varying parameters and different datasets.

We will cover a detailed explanation of the outcomes in the results and discussion sections.

3 Results and Discussions

While training ongoing for each cases of AI tasks (ML, DL) related results were saved and documented accordingly for each steps of training process as per devices. For time and space limitations, the evaluation and inference results were not presented here. As seen from Table 4 and Table 5, the lowest training time was observed for the utilization of 16 CPU cores(DL:40s, ML: 5s) and the highest training time for 1 CPU core (DL:58s, ML:58s) in the Hasee core i9 CPU

Algorithm 1: Algorithm for CPU-based optimal handling of AI tasks for any system

```
function dataCure (systemdata)
  Input  : Selected data for any system
  Output: Dataset consisting of target variables for the candidate system
if datasetnotEmpty then
    if hasNaull(dataset) then
        callfillNull()
    else if hasEmpty(dataset) then
        callfillMean()
    else if hasNoise(dataset) then
        callremoveNoise()
    else if hasOutLier(dataset) then
        callzTreatment()
    else
        returnDataset(system)
return generateDataset(system)

function doOptML (dataset)
  Input  : Healthy datasets of the system
  Output: Optimized ML model of the System
put features in a X,y
foreach i, model, data in nCPU, mlModels, dataset do
    train ← model(i, trn:data)
    test ← model(i, tst:data)
    infer ← model(i, inpdata)
    lat ← time(train/test/infer)
    perf ← acc(train/test/infer)
    outML=saveModel(model, i)
end
return lat, perf
function doOptDL (datasets)
  Input  : Healthy datasets of the system
  Output: Optimized DL model of the System
put features in a tensor
foreach i, model, data in nCPU, dlModels, dataset do
    train ← model(i, trn:data)
    test ← model(i, tst:data)
    infer ← model(i, inpdata)
    doCompile(model)
    lat ← fitDL(model,i)
    scores=doEval(model, i)
    perf ← scores
    outDL=saveModel(model, i)
end
return lat, perf
```

Table 2. Parameters used in AI (ML) task

Item	Value
Model	Random Forest
Sample size	10000
Features	20
Estimator	500
Task type	Classification

Table 3. Parameters used in AI (DL) task

Item	Value
Model	Sequential
Activation	Relu
Layers	8,12,1
Batch Size	10
Epoch	100
Loss	Binary Cross-Entropy
Optimizer	Adam

Table 4. Utilized CPU cores with AI (ML) training tasks

CPU Type	Cores	Total CPU	Utilized Core:Training Time(s)
Acer	Core i3	8	1:63, 2:35, 4:18, 8:10
Asus	Core i5	4	1:66, 2:38, 4:26
Hasee	Core i5	8	1:62, 2:32, 4:17, 8:9
Hasee	Core i9	16	1:58, 2:28, 4:16, 8:8
Hasee	Core i9	16	10:7, 12:6, 16:5

Table 5. Utilized CPU cores with AI (DL) training tasks

CPU Type	Cores	Total CPU	Utilized Core:Training Time(s)
Acer	Core i3	8	1:67, 2:62, 4:58, 8:52
Asus	Core i5	4	1:69, 2:64, 4:59
Hasee	Core i5	8	1:62, 2:59, 4:54, 8:49
Hasee	Core i9	16	1:58, 2:55, 4:44, 8:43
Hasee	Core i9	16	10:42, 12:41, 16:40

based machine. The worst case was observed for the Acer CPU case. The results

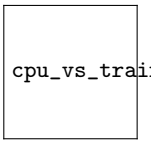

Fig. 4. Training results for the ML task

Fig. 5. Training results for the DL task

reveal the CPU-based machines are optimized for the ML and DL training cases. However, it was seen from the data that DL-based algorithms are taking much more time than ML-based algorithms on numerical data. This is because ML algorithms are readily optimized, but DL algorithms need special case-by-case tuning for optimization like ML algorithms. Moreover, we used an arbitrary DL network structure to compare the results only. A proper hyperparameter tuning might solve this issue.

The overall performance of the algorithms can be depicted by the figures below.

Figure 4 presents the training performance on ML algorithms over different machines. The lowest-performance device was the Asus, and the highest-performance device was the Hasee core i9 processed-based 16-cored machine. Other two machines were performed well as per their configurations.

Figure 5 presents the training performance on DL algorithms over different machines. The lowest-performance device was also the Asus and the highest-performance device was the Hasee core i9 processed-based 16-cored machine. The other two machines were performed similarly.

After all, it was seen that training time drastically decreased for each case by increasing CPU cores, but the rate of decrement became slower while CPU cores increased over half of the total cores available in that machine. The figures below show that the machine having 4 cores has seen a slower decrement in training time after increasing 2 cores for 4 core CPU, 4 cores for 8 core CPU, and 8 cores for 16 core CPU.

The overall accuracy increment dynamics also saved during the training of AI tasks. It was seen that training accuracy increased after the increment of CPU cores. But this increment gets slower after a while, and the core increment gets higher than half. The increment of accuracy for the 16 CPU-based Hasee machine is presented in Figure 7 for the ML case and Figure 7for the DL case. The results of other machines are not documented for the rational aspect. Whatever the case is the accuracy shows increasing dynamics during training over the CPU core. This is also a ratio for the training in the GPU case. Rationally, GPU holds

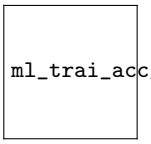

Fig. 6. Training accuracy increment per core for ML task

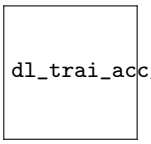

Fig. 7. Training accuracy increment per core for DL task

more cores than CPU; thus, GPU, GPU-based training yields better accuracy in training and inference.

The overall results show that CPU multi-core utilization also makes it easier to handle AI tasks in the absence or affordability of GPU access. By using such a method, users' confidential data can be protected from cloud data leaks and hacks. On the other hand, it is cost-effective to buy a costly GPU rather than some mid-range CPU to build a cluster.

4 Conclusion

In this research, we conducted a CPU-driven AI task (ML, DL) accomplishment using CPU multi-core and multi-threading facility. We developed a parallel processing platform to coordinate over the CPU core to train AI models using identical data. The results show that CPU-based parallel processing is also sufficient to handle ML and DL tasks with better performance and low latency. The lowest training time for 16 CPU cores was DL:40s and ML: 5s and the highest training time for 1 CPU core was DL:58s and ML:58s. This method is better than the GPU-driven training or other AI tasks. This is a substitute process in the absence or unavailability of GPU access to continue the AI tasks at your own facility by confirming data security. Free or premium cloud service is not free from data theft or leaks anymore. So, this method can be a minimum viable way to conduct his/her research at their own facility.

Acknowledgment

The authors would like to thank to the BAISCO and its infrastructure to conduct the experiment setup of this research. We also grateful to our colleagues in correcting the paper.

References

1. Serpa, Matheus S., et al. "Optimizing machine learning algorithms on multi-core and many-core architectures using thread and data mapping." 2018 26th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP). IEEE, 2018.
2. Chu, Cheng-Tao, et al. "Map-reduce for machine learning on multicore." Advances in neural information processing systems 19 (2006).
3. Mason, Karl, et al. "Predicting host CPU utilization in the cloud using evolutionary neural networks." Future Generation Computer Systems 86 (2018): 162-173.
4. Cassales, Guilherme, et al. "Improving the parallel performance of ensemble learners for streaming data through data locality with mini-batching." 2020 IEEE 22nd International Conference on High-Performance Computing and Communications; IEEE 18th International Conference on Smart City; IEEE 6th International Conference on Data Science and Systems (HPCC/SmartCity/DSS). IEEE, 2020.
5. Hegde, Vishakh, and Sheema Usmani. "Parallel and distributed deep learning." May 31 (2016): 1-8.
6. Saleem, Muhammad Fahad. "Benchmarking Processor Performance by Multi-Threaded Machine Learning Algorithms." arXiv preprint arXiv:2109.05276 (2021).
7. Hossain, N., Sun, G., & Islam, F.,Data Science and Machine Learning Processes for IoT Based Pulsed Plasma Thruster Research. Journal of Data Science and Intelligent Systems. (2023).
8. J. Zhao, T. Qi, and C. Wang, "Efficient GPU accelerated topology optimization of composite structures with spatially varying fiber orientations," Computer Methods in Applied Mechanics and Engineering, vol. 421. Elsevier BV, p. 116809, Mar. 2024. doi: 10.1016/j.cma.2024.116809.
9. Stone JE, Gohara D, Shi G (2010) OpenCL: a parallel programming standard for heterogeneous computing systems. Computi Sci Eng 12(3):66–73.
10. Z. Ye et al., "Deep Learning Workload Scheduling in GPU Datacenters: A Survey," ACM Computing Surveys, vol. 56, no. 6. Association for Computing Machinery (ACM), pp. 1–38, Jan. 22, 2024. doi: 10.1145/3638757.