Towards Optimization of Big Numbers Computation through an AI Pre-trained Model and Graph Traversal

Omar T. Mohammed[1], Moeid S. Heidari[2],
Alexey A. Paznikov[3], Mikhail S. Kupriyanov[4]
Department of Computer Science and Engineering
Saint Petersburg Electrotechnical University "LETI"
[1]omar.taha.mohammed@gmail.com, [2]moeidheidari@mail.ru, [3]apaznikov@gmail.com, [4]mskupriyanov@etu.ru

***Abstract*. Nowadays we see that multicore computers are able to easily manipulate digit numbers with a size up to 64 bits however as numbers get bigger the computation becomes more complex, the reason is that the size of both CPU registers and buses are limited. As a result, the arithmetic operations such as addition, subtraction, multiplication and division for CPU become more complex to perform. For solving the problem of how to do computation on big digit numbers, a number of algorithms have been developed. However, the existing algorithms are noticeably slow because they operate on bits individually and are designed to run over single-core computers only. In this paper, an AI model is presented that performs a computation on tokens of 8 digit numbers to assist boost the CPU computation performance.**

Keywords: node iteration algorithm; big digit numbers computation; parallel algorithm; machine learning

I. INTRODUCTION

Modern computer systems (CS) can very well handle and perform computations with numbers that have a length not bigger than (32 or 64) bits [1]. However, when those numbers become larger in size then the time for performing arithmetic operations, such as addition, subtraction, multiplication and division increases. This issue happens due to a number of different constraints related to the architecture of hardware and programming language. Because the majority of nowadays CPU registers are 32 or 64 bits, they can only accommodate numbers of that length [2]. Additionally, data types in programming languages are kind of a culprit, for instance, in the foremost programming languages, data of an integer can hold up to 32 bits, and a long data integer can hold up to 64 bits. Many algorithms and methods have been developed directing to solve this problematic of arithmetic calculations on big numbers; though, all of them implement the same principles: first they convert big numbers from base 10 to base 2, after that they execute bit-wise operations on the bit level [3, 4]. For example, arithmetic addition can be performed using the bit-wise logical operators XOR and OR. Those algorithms have complexity of $O(n)$, where n represents the total number of bits composing each of the big operands.

In this paper, we propose an AI model trained on a set of numbers from a range 1 to 8 digits long with its corresponding targets to predict an accurate result of a specific mathematical computation. The aim behind this model is to reduce the execution and time complexity on CPU.

II. RELATED WORKS

Many programming libraries have been developed to solve one of the most common problems which is performing arithmetic computations on big-integer numbers. Moreover, most of these libraries are not designed to work in a parallel fashion but are to operate over single-core systems [5, 6, 7]. Most of the researches and publications have been done to provide a clarification for arithmetic addition on enormous integer numbers tackle the problem not from a software algorithmic perspective but from a hardware perspective. For example, Fagin [8] suggested an enhancement for the carry look ahead adder. His idea revolves around the use of a massively parallel computer with a big number of processors that reaches thousands, each processor has a connection with local memory and also has a communication network. In order to distribute integers to be added between processors, some techniques of parallel prefix are take on to quickly do addition of large numbers in faster than if conventional machines were in place. Avizienis [9] drafted a representation diagram for binary numbers used in fast parallel calculation. This diagram turns around replicating each input operands so as to get rid of the chain carry propagation during an arithmetic addition and subtraction [10, 11].

This paper is organized as follows. Section 1 is an introduction of how computer systems handle arithmetic operations and then a snapshot of our AI approach. Section 2 is about previous works have been done related to the development of different algorithms that were suggested to improve the performance of CPU handling arithmetic calculations. Section 3 tackles our first approach of performing mathematical calculations which is through a node iteration technique, an example of addition has been described with it's schema. Section 4 drafts the limitation of the provided approach and the reason behind it. Section 5 explains briefly our second approach which is performing arithmetic

* The reported study was funded by RFBR according to the research projects № 19-07-00784, 18-57-34001 and was partially supported by Russian Federation President Council on Grants for governmental support for young Russian scientists (project SP-4971.2018.5).

calculations through an AI pre trained model that can partially enhance CPU performance in handling some repetitive arithmetic tasks such as addition in particular. An example of addition has been provided with the structure of the model dataset is also described in details, plus a comparison in execution time of performing an addition calculation has is mentioned between CPU and the pre trained model. One other thing this section has is an introduction of how neural networks works with some information of how our model was created. Section 6 shares the execution time results of a conducted experiment of an addition operation for both CPU and our AI pre trained model, it shows the difference between them in terms of timing and the reason of getting such results. Section 7 drafts some limitations of our proposed AI approach. Finally, Section 8 presents our conclusion and future works.

III. Node Iteration Approach

In this approach, we will generate a directional graph of operands and results as nodes of the graph and edges for directions. For the next operations we will iterate the generated graph to find the result, at first if we find the result we will return it and if not then it means that we didn't calculate such an operation before, o we need to do the operation in tradition way and store the operands and results as new nodes in the graph with specified directed edges for next usage. We will store the graph in a shared mid extra memory with high access speed to be useful for all threads. In this approach, the learned and generated graph will be shared with other computer resources. We need to find the most efficient algorithm to iterate the generated graph with minimal time. In this approach, each graph has two list of pointers a list of directed operation and a list for result pointers. Let us do the operation for 2+2+1+542+543+6 (Fig. 1).

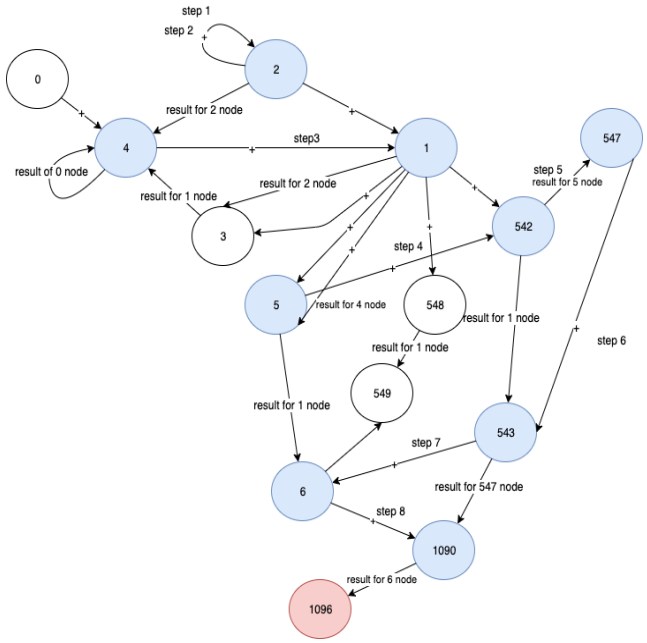

Fig. 1. Example schema 1

In this example we find the first operation node in the graph which is 2. After that we find a pointer pointing to the second operation as it is node 1. And from the second operand node we find a result pointer which is specified to the node 2 result which is node 3. And in this way, we can find the result of 2+1 = 3 by graph iteration.

We can illustrate our approach as below (Fig. 2, 3).

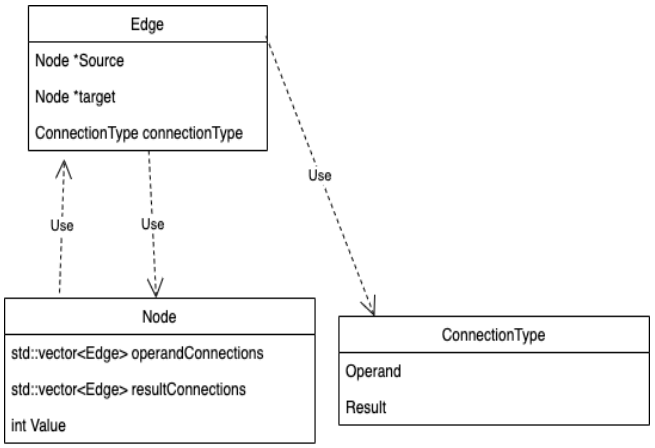

Fig. 2. Class diagram

```
1: template <typename T>
2: bool Contains(std::vector<T> &Vec, int &number){
3: bool found = false;
4: std::find_if(Vec.rbegin(), Vec.rend(),
5:  [number, &found](const T &n) -> bool {
6:   found = (((Node)n).getValue() == number);
7:   return found; })
8: return found; }
```

```
9: template <typename T>
10: bool Contains(std::vector<T> &Vec, int &number){
11: bool found = false;
12: std::find_if(Vec.rbegin(), Vec.rend(),
13:  [number, &found](const T &n) -> bool {
14:   found = (((Node)n).getValue() == number);
15:   return found; })
16: return found; }
```

Fig. 3. Hash table code implemented in C++

IV. Limitations

In the traditional way of calculation, we have just two memory access and because of this reason we can obtain the result much faster. In our approach we have slower results because we need some other memory accesses. So, if we want to implement such approach in reality we need to store our graph in a much faster memory with higher speed of access to get the desired result. And calculating the time needs some time in this example and memory access time also is variable in any system.

So, average memory access time = miss rate0 + hit time0 * (miss rate1* miss penalty1+ hit time1)

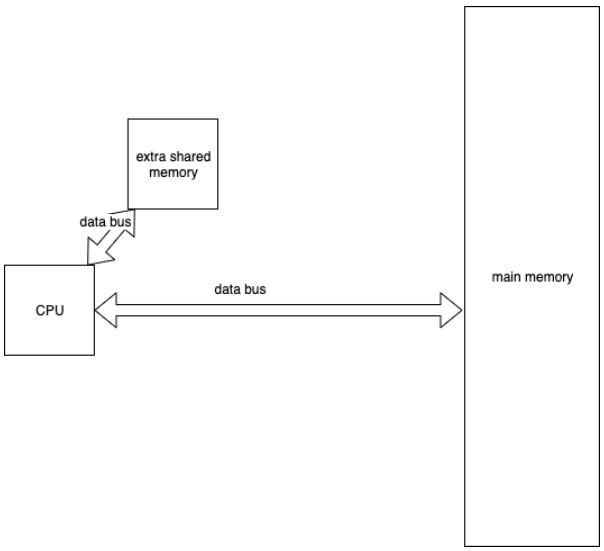

Fig. 4. Current computation process between CPU and Memory

ResultTime=Result time – (memory access time * number of new accesses). this approach is not going to be useful for single operation calculation but also for any other complex operations to avoid repeated calculations.

V. The Proposed AI Approach

As we know artificial neural network and deep learning models, are considered among the most powerful prediction tools that machine learning can offer, they are suited to solve perceptual problems. Artificial intelligence (AI) is very good also at handling some repetitive tasks. For example in a workplace AI will help hiring managers and recruiters to better match potential candidates for jobs. Based on this principle we found to use AI not to help human as a goal in this paper but to help the computer itself particularly CPU with preforming some repetitive tasks such as some frequent arithmetic calculations specially when it's getting done on big numbers. In this paper, we focused on the addition arithmetic and how to use machine learning to best help CPU with performing repetitive summations.

Taking an example the CPU can easily compute addition of two numbers, for instance, 17 + 54 in a matter of nanoseconds, while computing a bigger number with another big number e.g. 1697458765 + 2536844568 is definitely heavier on CPU and will take longer time to execute. With the development of deep learning, we see that there have been a big success for such kinds of problems. In this paper, and for the sake of easy understanding and simplicity, we attempted to train a neural network on a simple arithmetic addition, the addition of two numbers and then we tried to predict the value answer through our trained model. The structure of the dataset consisted of two inputs and one output. The model has been trained on 1 million entries of integer type with its corresponding target outputs starting from zero incrementing to one million, and it was tested to provide accurate results on integer numbers ranging from 1 million to 10 million regardless to the trained set itself. The result is that we created a model that is able to accurately predict the addition of two integer numbers ranging from zero to ten million that is ready to save the CPU from performing addition calculations within this range, especially when CPU is in need to do a huge amount of tasks of such kind repeatedly. One interesting fact we have noticed was that after testing the execution time of CPU for performing a summation of two small integers was slightly more than the time our trained model took to predict the summation of the same numbers see (Fig.7). Nevertheless, the main goal is not to compete with CPU abilities but to assist it to perform faster.

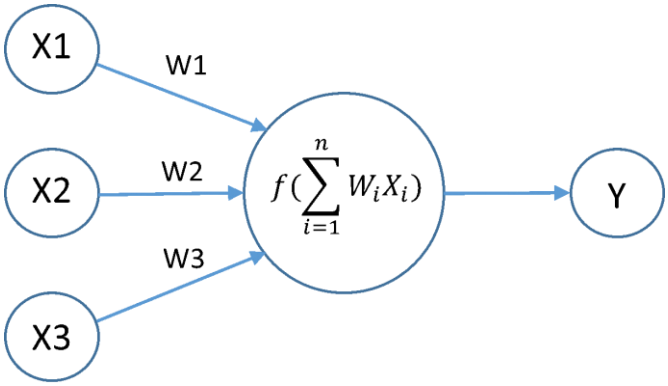

Fig. 5. Neural network

As we can see in (Fig. 5) it's basically the simplest neural network there is, a perceptron, with no hidden layers and the identity function $f(x) = x$ as activation function. It's really just linear regression in disguise. As long as you pass in empty values as 0, this network can learn how to add an arbitrary number of values up to N by finding a weight vector W of length N containing all 1s which we used for an experiment the addition of two digits. Of course, more complex networks could learn other arithmetic operations as well, but linear regression is already overkill for this task. Keras library has been used for creating the model which is a part of Tensorflow library see (Fig. 6).

After training the model on 1 million input data we have tried to measure the execution time that both CPU and the model is taking to perform one calculation by testing it on a calculation of a single addition of two integer numbers. The test was done using timeit library in Python programming language, it was executed in windows 10 MSI notebook, core i7-8750H 8th Gen 2.2GH (12 CPUs), 16 GB Ram, As we can see in the result table (Fig. 7) the time that the CPU took to perform that calculation was longer by 10 nano-seconds compare to our AI trained model execution time. We believe the reason of getting such results is related to the method that CPU is using to perform an addition operation which is different and more complex from our neural network method. Both CPU and neural network methods have been explained in this paper already.

```
1: from tensorflow import keras
2: import numpy as np
3: import dataset as dc
4: model = keras.Sequential([
5:  keras.layers.Flatten(input_shape=(2,)),
7:  keras.layers.Dense(25, activation='relu'),
8:  keras.layers.Dense(25, activation='relu'),
9:  keras.layers.Dense(25, activation='relu'),
10  keras.layers.Dense(1)])
11:
model.compile(optimizer='adam',loss='mse',metrics=['mae'])
12: model.fit(dc.train_data, dc.train_targets, epochs=20,
13:batch_size=1)
14: setupcode = "s = 0"
15: function = "'a = 13+56 '"
17: print ("CPU execution time of two digits addition =
18:",(timeit.timeit(setup = setupcode, number = 100)))
19: setupcode1 = "r = 0"
20: function = "'b = model.predict(np.array([[13,56]])'"
21: print ("AI pre-trainded model execution time of two digits
22:addition = ",(timeit.timeit(setup = setupcode1, number =
23:100)))
```

Fig. 6. Hash table code implemented in Python

VI. Experimental results

Test Case	Operation	Results	Execution Time in Nano-Seconds
CPU	A+B	Y	1.30 ns
AI pre-trained model	A+B	Y	1.10 ns

Fig. 7. Result table

VII. Limitations

The model needs to be accurately and specifically trained, it also requires big amounts of structured training data and learning have to be generally supervised. Also it require offline batch data for training. Plus it cannot learn effectively in real time.

VIII. Conclusion and Future Works

The discussed approach of node iteration of graph data, were examined in practice. We see that our approach has some limitations. Based on our first experiment see (Fig. 3) we can state that the increase above this threshold does not result in faster processing. This phenomenon was caused due to the necessity of more data exchange that are extensive in case of a big numbers of parallel machines. And the second approach as we can see in the experiment there is a difference in time between CPU execution and AI pre-trained model by approximately 20 nanoseconds which gave a faster performance, we conclude this paper by stating that involving artificial intelligence in arithmetic operation is still evolving and this project might be a step towards improving such interaction.

Future research can improve upon our proposed approaches by combining them together so that other arithmetic operations such as subtraction, multiplication, and division are added. Besides, a disseminated version of the same approach could be designed so that it can be implemented over a network of regular machines, making the execution less expensive and more accessible, also we will target to develop our model to learn in real time.

References

[1] C. Maxfield and A. Brown, "The Definitive Guide to How Computers Do Math: Featuring the Virtual DIY Calculator," Wiley-Interscience. 2004.

[2] J. Hennessy and D. Patterson, "Computer Architecture: A Quantitative Approach," 4th Edition, Morgan Kaufmann, 2006.

[3] D. Knuth, "Art of Computer Programming, Volume 2: Seminumerical Algorithms," 3rd Edition, Addison-Wesley Professional, 1997.

[4] I. Koren, "Computer Arithmetic Algorithms," 2nd Edition, AK Peters Ltd, 2001.

[5] A. Paznikov and Y. Shichkina, "Algorithms for Optimization of Processor and Memory Affinity for Remote Core Locking Synchronization in Multithreaded Applications," Information, vol. 9, i. 1, 2018, pp. 21.

[6] E.A. Goncharenko, A.A. Paznikov, and A.V. Tabakov, "Evaluating the performance of atomic operations on modern multicore systems," Journal of Physics: Conference Series, 2019, vol. 1399, no. 3, pp. 033107

[7] A.A. Paznikov, V.A. Smirnov, and A.R. Omelnichenko, "Towards Efficient Implementation of Concurrent Hash Tables and Search Trees Based on Software Transactional Memory," in Proc. Of the 2019 International Multi-Conference on Industrial Engineering and Modern Technologies (FarEastCon), 2019, pp. 1-5.

[8] B. Fagin, "Fast Addition of Large Integers," IEEE Transactions on Computers. 1992.

[9] A. Avizienis, "Signed-digit number representations for fast parallel arithmetic," IRE Trans. Elec. Comput, 2003, vol. 10, pp. 737-740.

[10] Youssef Bassil, Aziz Barbar, "Sequential & Parallel Algorithms For the Addition of Big-Integer Numbers," vol. 4, no. 1, 2010, pp. 52-69.

[11] A.V. Tabakov and A.A. Paznikov. "Algorithms for Optimization of Relaxed Concurrent Priority Queues in Multicore Systems," in Proc. of the 2019 IEEE Conference of Russian Young Researchers in Electrical and Electronic Engineering (EIConRus), 2019, pp. 360-365.