Neural Networks

Feed Forward Neural Network in C++ 17 and OpenMP for performance optimization

Andreas Karatzas

April 23, 2022

Instructor: Mohammad Sayeh

Electrical, Computer and Biomedical Engineering

Neural Networks

Hello,

My name is Andreas, and today I'll share with you my work for course ECE 572, "Neural Networks"

Introduction

Introduction

Abstract

Problem Statement

Optimizing the performance of a Feed Forward Neural Network.

Abstract
Problem Statement
Optimizing the performance of a Feed Forward Neural Network.

My work for the course's project was on optimizing the performance of a typical Feed Forward Neural Network, alternatively known as Multi-layer Perceptron.

What

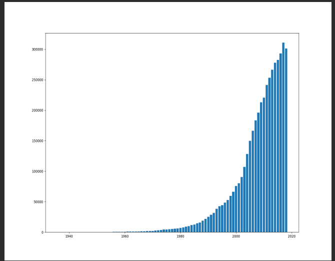
Overview

1. Comparison of state-of-the-art deep learning frameworks
2. Implement a neural network application that utilizes all system cores efficiently

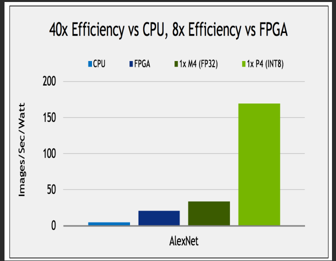
What
Overview
1 Comparison of state-of-the-art deep learning frameworks
2 Implement a neural network application that utilizes all system cores efficiently

To really grasp the magnitude of this work, I also implemented the same neural network using PyTorch. Furthermore, I implemented a serial version of my neural network and then I optimized it using a framework called OpenMP.

Why?

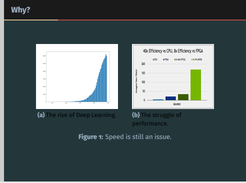


(a) The rise of Deep Learning.

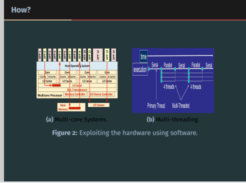


(b) The struggle of performance.

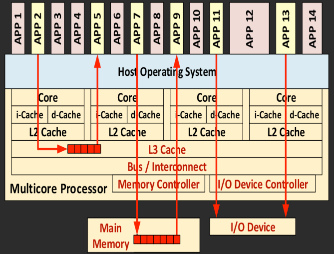
Figure 1: Speed is still an issue.



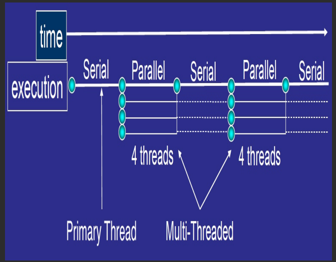
So why does that work matter? First of all, we live in the era of deep learning. There is an exponential growth on the field of AI and that means that work published in the field of machine learning matters to the research community. Furthermore, performance is and probably will always be a subject for discussion. Let's not forget that the very reason that neural networks did not succeed in dominating the field of computer science was when they first appeared was partially because of limited computational power at that time.



How?



(a) Multi-core Systems.



(b) Multi-threading.

Figure 2: Exploiting the hardware using software.

There are some pretty straight forward ways of optimizing an application. First of all, we have to take into consideration today's general computer system architecture. That is basically the CPU architecture. Today's era regarding CPU architectures is the many-core era. This means that there might be quite a lot of individual cores inside a single chip. Therefore, our goal is how to efficiently utilize all that hardware. However, to beat the state-of-the-art deep learning frameworks, we have to take it to the next level. That is to implement the whole project using low level programming languages, which were created in the first place for performance purposes.

Background

Background

Feed Forward Neural Networks

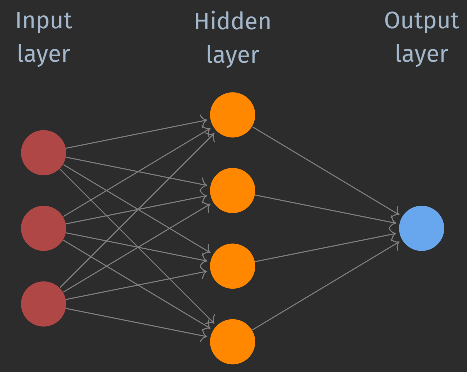
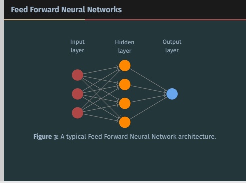


Figure 3: A typical Feed Forward Neural Network architecture.



Let us first study the problem from a high level overview. In this figure, we can see the general layout of a feed forward neural network. We can immediately trace some points where we can reconfigure the execution to run in parallel. For example, since forward propagation is really just a GEMM operation we can split the workload of each neuron to a separate task.

The Perceptron

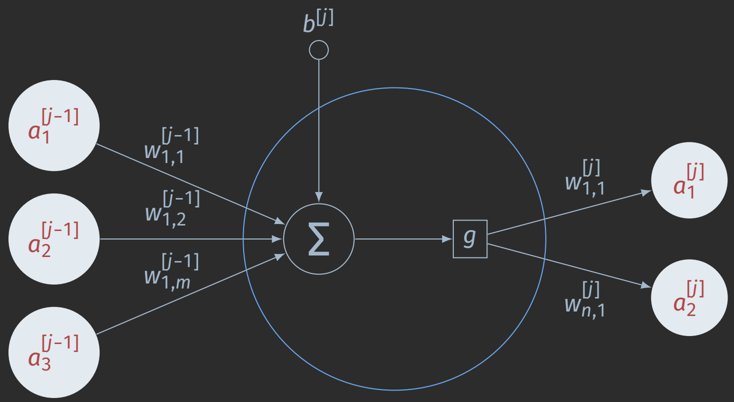
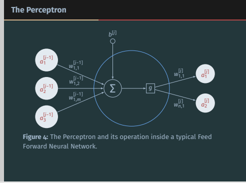


Figure 4: The Perceptron and its operation inside a typical Feed Forward Neural Network.



Before we proceed to the implementation of the aforementioned project, let us remind ourselves the architecture of a single neuron. Throughout the lectures, we studied a single model of neuron for simplicity. That neuron was activated if the load was more than a certain threshold which was usually 0 in or case. However, neural networks today might be engineered with a variety of ways. For example, there is a large set of activation functions, instead of that simple model that we went through during the semester. For our case, the activation function will be the Sigmoid function. At this point I would like to make a note. There are some serious problems that arise in the attempt to implement numerically challenging applications in such a low level. Computers use some easily exploitable arithmetic protocols, that can cause all sorts of disasters when it comes to implementing projects such as this.

Tools and Technical details

Tools and Technical details

Software

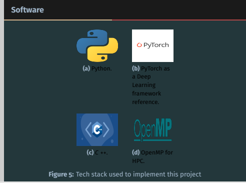
(a) Python.

(b) PyTorch as a Deep Learning framework reference.

(c) C ++.

(d) OpenMP for HPC.

Figure 5: Tech stack used to implement this project



For this work, I used Python and PyTorch to implement a neural network trained using the fashion-MNIST dataset, which is a set of 32 by 32 gray-scale images. This model was implemented to be compared with the optimized one. The optimized model was implemented in C++. Finally, to enable high performance computing (or HPC) I used OpenMP, which made the code run in parallel.

PyTorch

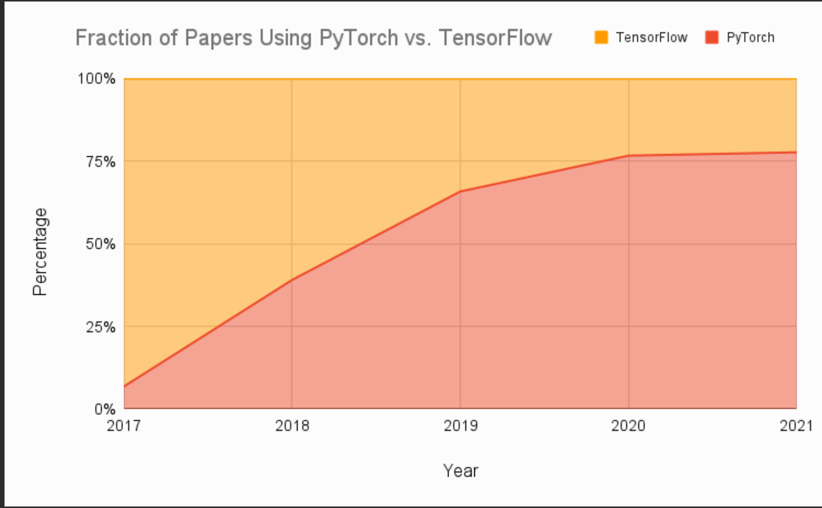
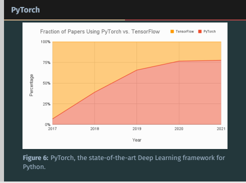


Figure 6: PyTorch, the state-of-the-art Deep Learning framework for Python.



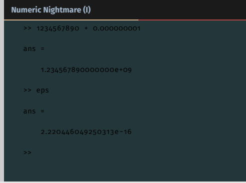
My choice over the deep learning framework was not at random. Pytorch has both the best performance and the greatest outreach in the research community. In this figure, we can observe that it's main rival, Tensorflow, has experienced a decay over the years. That is mainly due to the capabilities of one over the other. Pytorch is better in terms of speed, rapid prototyping and debugging. Therefore, it is the framework that I chose to study as my point of reference for this project.

Challenges

Challenges

Numeric Nightmare (I)

```
>> 1234567890 + 0.000000001

ans =

     1.234567890000000e+09

>> eps

ans =

     2.220446049250313e-16

>>
```

We are going to elaborate now on the problems that an engineer is required to solve when implementing such applications. Let's start with this example. How much is one billion plus one to the power of minus 10? An uninitiated would answer a billion point 9 zeros and 1. But people like us must know that this sum is equal to one billion. Therefore, you can't just keep adding and multiplying numbers because the sum will eventually become huge. And you are going to run to overflow errors. The maximum value of an image in the dataset is 255. Can that be reduced? The answer is yes. We can use well known normalization methods and scale our data to belong between 0 and 1.

Numeric Nightmare (II)

```
>> digits(30);
>> f = exp(sqrt(sym(163))*sym(pi));
>> vpa(f)

ans =

    262537412640768743.999999999999

>>
```


Numeric Nightmare (II)
>> digits(30);
>> f = exp(sqrt(sym(163))*sym(pi));
>> vpa(f)
ans =
262537412640768743.999999999999
>>

Our operations are all defined in the Real Numbers set. That means that we are dealing with floating points. Floating points in theory offer us a way to representing quantities with infinite numerical precision. However, in practice floating point numbers are nothing but a nightmare. How are you going to back-propagate the model error if you cannot accurately represent it? The answer is that you can't. To be able to do so, you would have to have an infinite memory, and that is not happening any time near. In fact the problem of the vanishing gradient is a well known issue when it comes to neural networks.

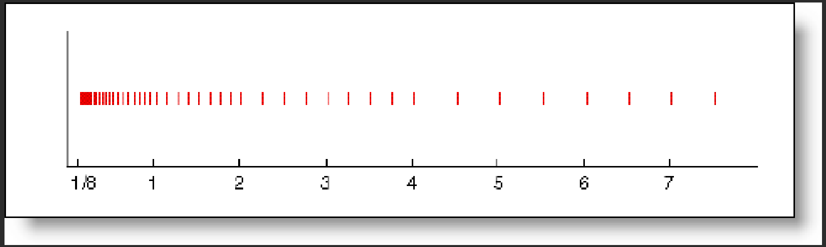
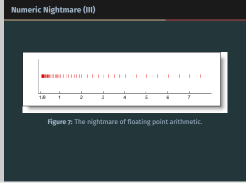


Figure 7: The nightmare of floating point arithmetic.



But what's even worse than that? How can you protect your model from another well known numerical issue called underflow? First of all, you have to be prepared for such a problem. But of course when this is your first project, you don't foresee problems like that. And here I did have that problem. And it took me a week to find it. You see the sigmoid function here depends on the ϵ constant to perform forward propagation.

Numeric Nightmare (IV)

std::exp, std::expf, std::expl

Defined in header `<cmath>`

```
float       exp ( float arg );                  (1)
float       expf( float arg );                      (since C++11)
double      exp ( double arg );                 (2)
long double exp ( long double arg );            (3)
long double expl( long double arg );                (since C++11)
double      exp ( IntegralType arg );           (4) (since C++11)
```

1-3) Computes *e* (Euler's number, `2.7182818...`) raised to the given power `arg`

4) A set of overloads or a function template accepting an argument of any integral type. Equivalent to (2) (the argument is cast to `double`).

Parameters

arg - value of floating-point or Integral type

Return value

If no errors occur, the base-*e* exponential of `arg` (e^{arg}) is returned.

If a range error due to overflow occurs, `+HUGE_VAL`, `+HUGE_VALF`, or `+HUGE_VALL` is returned.

If a range error occurs due to underflow, the correct result (after rounding) is returned.

Error handling

Errors are reported as specified in `math_errhandling`.

If the implementation supports IEEE floating-point arithmetic (IEC 60559),

- If the argument is ±0, 1 is returned
- If the argument is -∞, +0 is returned
- If the argument is +∞, +∞ is returned
- If the argument is NaN, NaN is returned

Notes

For IEEE-compatible type `double`, overflow is guaranteed if $709.8 < arg$, and underflow is guaranteed if $arg < -708.4$

Figure 8: The `std::exp()` function.

Let's therefore have a closer look at the provided `exp` function. And once you see it you know that you are in a bad situation. What is this in the notes section down? overflow is guaranteed if 709.8 < *arg*, and underflow is guaranteed if *arg* < −708.4. Nice ah? For this purpose there are some very handy mathematical infinite series, called Taylor series.

The exponential function

$$e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!} = 1 + \frac{x^2}{2!} + \frac{x^3}{3!} + \dots$$

Numeric Nightmare (V)
The exponential function
$e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!} = 1 + \frac{x^2}{2!} + \frac{x^3}{3!} + \dots$

Here you can see how to compute the exponential using a special subset of the Taylor series called Maclaurin series.

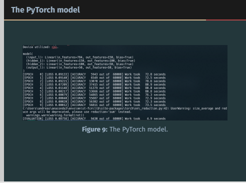
Experiments

Experiments

The PyTorch model

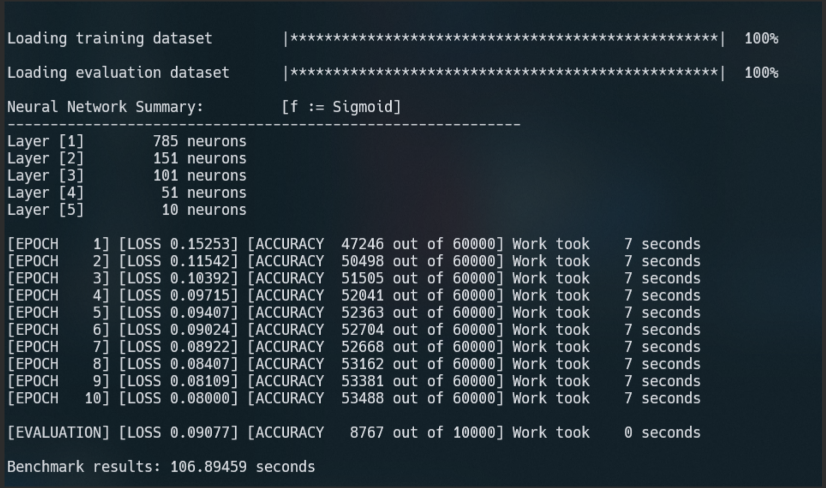
```
Device utilized: cpu.

model(
  (input_l): Linear(in_features=784, out_features=150, bias=True)
  (hidden_1): Linear(in_features=150, out_features=100, bias=True)
  (hidden_2): Linear(in_features=100, out_features=50, bias=True)
  (output_l): Linear(in_features=50, out_features=10, bias=True)
)
[EPOCH    0] [LOSS 0.09133] [ACCURACY   5943 out of  60000] Work took   72.8 seconds
[EPOCH    1] [LOSS 0.09140] [ACCURACY   6549 out of  60000] Work took   72.5 seconds
[EPOCH    2] [LOSS 0.09221] [ACCURACY  13070 out of  60000] Work took   78.6 seconds
[EPOCH    3] [LOSS 0.07209] [ACCURACY  37415 out of  60000] Work took   80.9 seconds
[EPOCH    4] [LOSS 0.01148] [ACCURACY  51379 out of  60000] Work took   80.0 seconds
[EPOCH    5] [LOSS 0.00217] [ACCURACY  53666 out of  60000] Work took   80.8 seconds
[EPOCH    6] [LOSS 0.00079] [ACCURACY  54865 out of  60000] Work took   76.3 seconds
[EPOCH    7] [LOSS 0.00040] [ACCURACY  55687 out of  60000] Work took   72.9 seconds
[EPOCH    8] [LOSS 0.00020] [ACCURACY  56302 out of  60000] Work took   72.3 seconds
[EPOCH    9] [LOSS 0.00012] [ACCURACY  56811 out of  60000] Work took   73.5 seconds
C:\Users\andreas\anaconda3\envs\mnist-fcn\lib\site-packages\torch\nn\_reduction.py:42: UserWarning: size_average and red
uce args will be deprecated, please use reduction='sum' instead.
  warnings.warn(warning.format(ret))
[EVALUATION] [LOSS 0.08758] [ACCURACY   9430 out of  60000] Work took    4.9 seconds
```

Figure 9: The PyTorch model.



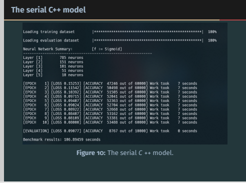
Now to the exiting part, seeing the project actually working. First of all, let us see how did the PyTorch model go. This is 72 seconds per epoch. At this point, I would like to mention that Pytorch has actually some utilities that are connected which OneAPI and MPI all which optimize and parallelize the load of the model. So all I'm saying is that this is a fair fight. I'm not comparing my work with a frameworks that runs serial.

The serial C++ model



```
Loading training dataset       |**************************************************|  100%

Loading evaluation dataset     |**************************************************|  100%

Neural Network Summary:        [f := Sigmoid]
------------------------------------------------------------
Layer [1]        785 neurons
Layer [2]        151 neurons
Layer [3]        101 neurons
Layer [4]         51 neurons
Layer [5]         10 neurons

[EPOCH    1] [LOSS 0.15253] [ACCURACY  47246 out of 60000] Work took    7 seconds
[EPOCH    2] [LOSS 0.11542] [ACCURACY  50498 out of 60000] Work took    7 seconds
[EPOCH    3] [LOSS 0.10392] [ACCURACY  51505 out of 60000] Work took    7 seconds
[EPOCH    4] [LOSS 0.09715] [ACCURACY  52041 out of 60000] Work took    7 seconds
[EPOCH    5] [LOSS 0.09407] [ACCURACY  52363 out of 60000] Work took    7 seconds
[EPOCH    6] [LOSS 0.09024] [ACCURACY  52704 out of 60000] Work took    7 seconds
[EPOCH    7] [LOSS 0.08922] [ACCURACY  52668 out of 60000] Work took    7 seconds
[EPOCH    8] [LOSS 0.08407] [ACCURACY  53162 out of 60000] Work took    7 seconds
[EPOCH    9] [LOSS 0.08109] [ACCURACY  53381 out of 60000] Work took    7 seconds
[EPOCH   10] [LOSS 0.08000] [ACCURACY  53488 out of 60000] Work took    7 seconds

[EVALUATION] [LOSS 0.09077] [ACCURACY   8767 out of 10000] Work took    0 seconds

Benchmark results: 106.89459 seconds
```

Figure 10: The serial *C* ++ model.

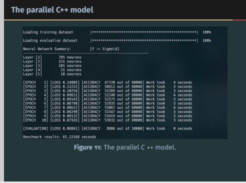


Wow ... what happened here? 72 seconds per epoch down to 7 seconds per epoch. Ok ... that's going well! Can we go better?

The parallel C++ model

```
Loading training dataset        |**************************************************|  100%

Loading evaluation dataset      |**************************************************|  100%

Neural Network Summary:         [f := Sigmoid]
------------------------------------------------------------
Layer [1]        785 neurons
Layer [2]        151 neurons
Layer [3]        101 neurons
Layer [4]         51 neurons
Layer [5]         10 neurons

[EPOCH    1] [LOSS 0.14809] [ACCURACY  47720 out of 60000] Work took    4 seconds
[EPOCH    2] [LOSS 0.11222] [ACCURACY  50811 out of 60000] Work took    3 seconds
[EPOCH    3] [LOSS 0.10256] [ACCURACY  51589 out of 60000] Work took    3 seconds
[EPOCH    4] [LOSS 0.09621] [ACCURACY  52240 out of 60000] Work took    4 seconds
[EPOCH    5] [LOSS 0.09143] [ACCURACY  52575 out of 60000] Work took    3 seconds
[EPOCH    6] [LOSS 0.08790] [ACCURACY  52935 out of 60000] Work took    3 seconds
[EPOCH    7] [LOSS 0.08612] [ACCURACY  53087 out of 60000] Work took    4 seconds
[EPOCH    8] [LOSS 0.08290] [ACCURACY  53347 out of 60000] Work took    3 seconds
[EPOCH    9] [LOSS 0.08133] [ACCURACY  53419 out of 60000] Work took    3 seconds
[EPOCH   10] [LOSS 0.07926] [ACCURACY  53631 out of 60000] Work took    3 seconds

[EVALUATION] [LOSS 0.08861] [ACCURACY   8806 out of 10000] Work took    0 seconds

Benchmark results: 65.22168 seconds
```

Figure 11: The parallel C ++ model.



Apparently we can! So let's recap. From 72 seconds, down to 7 and then down to 3 ... That means that we achieved over 20 times performance boost. What does the profiler have to say about that?

Congrats, signed by Intel

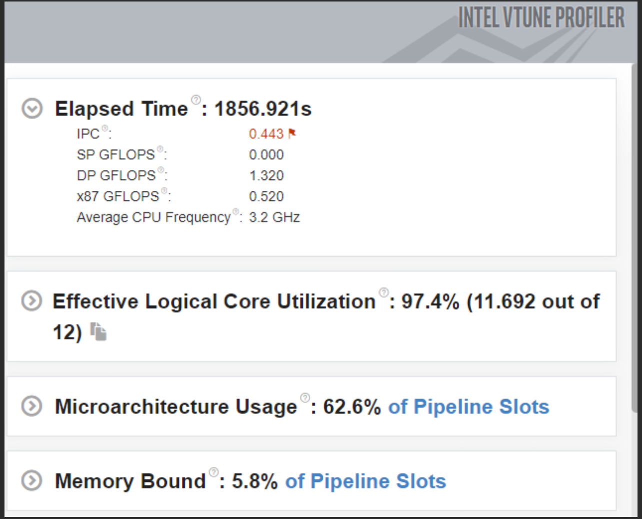
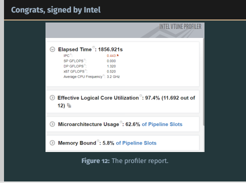
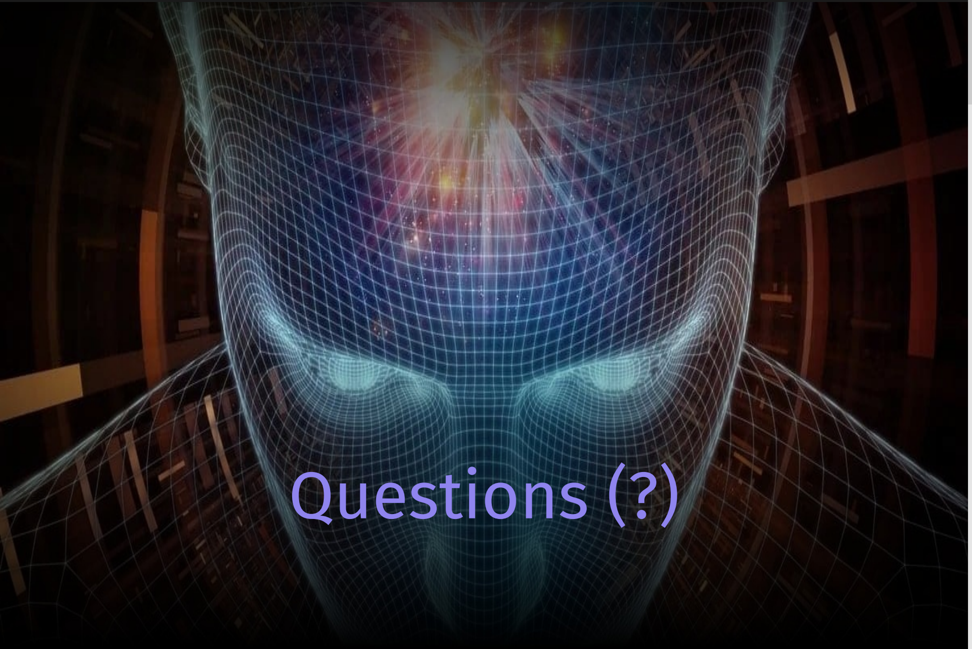


Figure 12: The profiler report.



And this basically means that our application is not memory bound and has an effective core utilization percentage of approximately 98 percent.

Thanks for your patience and time !



Questions (?)

Thanks for your patience and time !
Questions (?)