Apetrei Bogdan Gabriel, ISS1

Perceptron training on MNIST low level comparison

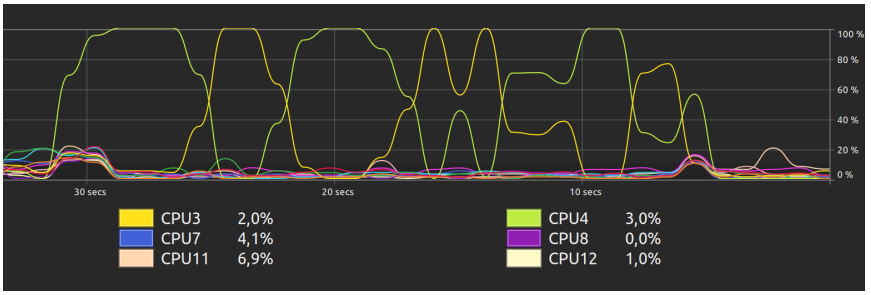
1. Introduction

Even in the best of cases, without relying on the GPU, Python can have difficulties taking advantage of a mulțicore environment to make the calculations often associated with neural networks. In the following paper I will present a simple perceptron implementation in c++ that takes full advantage of the host hardware, along with the lessons learned in implementing it.

As a baseline, those programs are running on Ubuntu 12, on a system with a AVX2 compatible CPU, with 12 hardware threads (the AMD Ryzen 5 5600). We are training a single layer consisting of 10 perceptrons, on the MNIST dataset, with 60 000 training images.The runtime is only considered for the duration of the training function. The exact performance numbers must only be considered from a comparative point of view.

a) Simple python perceptron.

A simple perceptron class which uses numpy matrix operations to train the weights and biases of the model using gradient descent. This implementation is relatively fast because numpy features a mechanism for harnessing SIMD features [1]. What we do notice however is the single threaded nature of the implementation makes it so very little of the hardware is actually used. Also thanks to the many layers of abstraction present in python, it is hard to assert the true memory usage of the program but the OS rapports the python thread as peaking at about 600MB of RAM

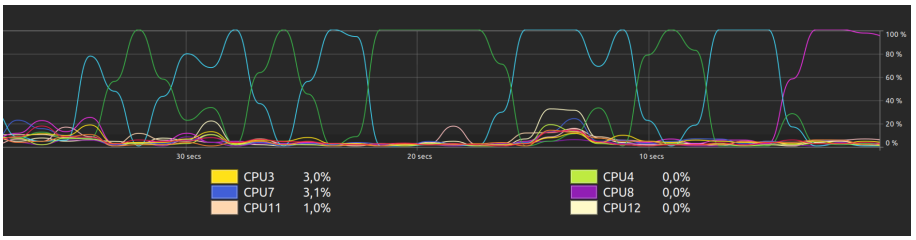



```
Baseline is: 2.90 MB
Memory used in perceptron: 0.00 MB
We are at epoch: 0
We are at epoch: 1
We are at epoch: 2
We are at epoch: 3
We are at epoch: 4
We are at epoch: 5
We are at epoch: 6
We are at epoch: 7
We are at epoch: 8
We are at epoch: 9
Training duration: 22.52 seconds
Memory used after training: 0.00 MB
Accuracy on the testing set: 0.9122
```

b) Simple c++ perceptron.

The structure of the simple c++ perceptron mirrors the structure of the python program, with the same class that holds the biases and weights of all 10 perceptrons. However, because the STL offers no native matrix implementation, we implement the gradient descent for 10 perceptrons, and 60 000 images in a sequential nested for loop. Memory wise, we make use of mostly vectors and references to vectors, as they are stored in contiguous memory and when used in a loop take better advantage of the caching and page caching mechanics. Also their main drawback (slow and unpredictable resizing) is completely negated if we take care to resize them to a known maximum size ahead of time.

It's also relevant that we have GCC as the compiler of choice, and GCC uses argument '-O0' (no optimizations enabled) as the default compilation flag [2]. This mode prevents even simple optimizations from taking place and is only used because of the ease of debugging it provides.




```
/home/unwise/Git/Perceptron/builddir/Perceptron
Baseline memory: 36.6367
Final count: 10000
Final count: 60000
Memory used after loading the dataset: 388.613
Memory used in perceptron: 388.613
We finished epoch: 0
We finished epoch: 1
We finished epoch: 2
We finished epoch: 3
We finished epoch: 4
We finished epoch: 5
We finished epoch: 6
We finished epoch: 7
We finished epoch: 8
We finished epoch: 9
Training duration: 76.478
Memory used at the end point: 388.863
Perceptron accuracy: 0.9075
Process finished with exit code 0
```

Predictably, this configuration results in the worst possible execution time (~76 seconds). This situation highlights the importance of the compiler în the c++ development process. Though there are obvious optimizations to be done in code, none of them have almost any relevance compared to the steps taken by the compiler. Before changing the implementation, we must consider a more proper compiler argument. Simply using '-O1', the most basic form of optimization, we bring the runtime back to 8.6 seconds. Some optimizations include:

- Use of combined instructions (decrement and branch, increments or decrements of addresses with memory accesses…)
- Memory and stack access pattern improvements
- Loop optimizer techniques, separating invariants, using registers for variants

In short, the compiler takes full advantage of the extended instruction set and memory access patterns present în the X86 architecture, while in '-O0' mode it is forced to lay out every instruction in a manner that closely resembles the written source code.

c) Optimised c++ perceptron.

The optimised Perceptron implementation is in fact almost identical to the unoptimized version, but by making a few careful decisions, we can greatly improve the training time of the perceptron.

First of all, we decided to use the '-O3' compiler flag. In our case, one particularly interesting optimization that is not enabled in '-O1' is vectorization and loop vectorization. Through this optimization, the compiler attempts to make use of the SIMD instructions present in the target hardware. We can only speculate, but some loops in our implementation could greatly benefit from this form of parallelization offered by modern X86 CPUs. While it is possible to use SIMD instructions manually, doing so greatly complicates the code, so even if the compiler is unable to get good results every time, it is a much better option for developers.

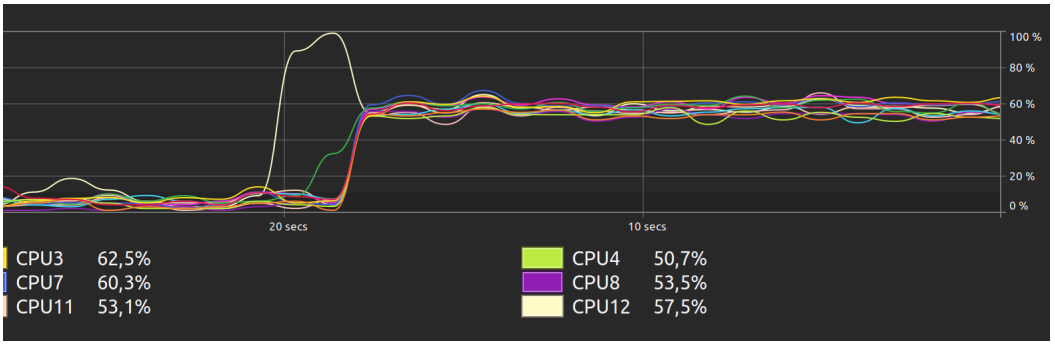
The last optimization we will take a look at using all 12 threads of the CPU. For this, we will make use of the popular library OpenMP, an API for managing mulți-threaded operations.

We can easily find 2 places where parallelization is seamless.

- When calculating the predictions for each perceptron, we can simply calculate all 10 of them in parallel. Because training implies predictions for each and every image, this greatly speeds up the process.
- Calculating the new weights and biases via gradient descent for each perceptron is also a fully independent process with no possible collisions.

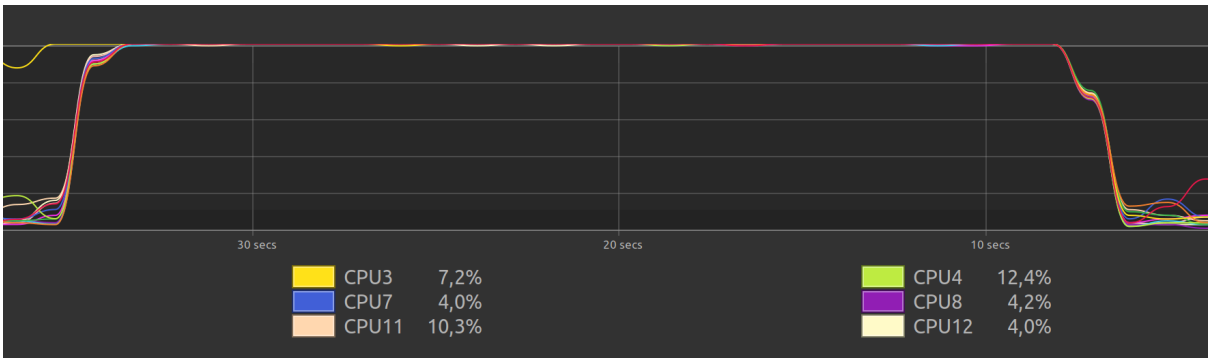
The only detail left to decide is the optimal number of threads OpenMP should be allowed to create. Here we can come to a few conclusions:

Using 24 threads (double the amount of hardware threads present) results în severe thread starvation and excessive switching, concluding with a run time of 58 seconds, much worse that what we started with.

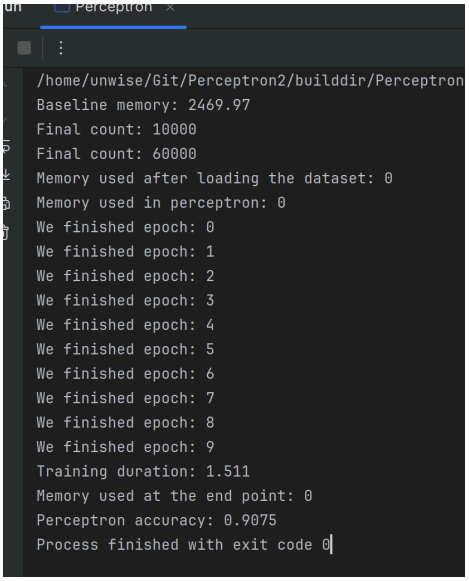


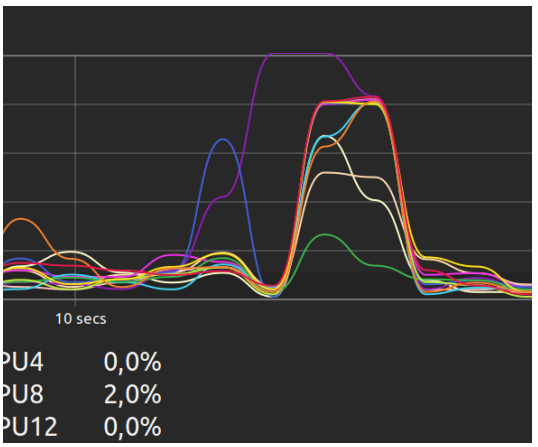
We can reduce the number of 12 and we attain full use of the CPU cores, and a good runtime of ~3 seconds, but we still have to wonder if there might still be excessive switching

going on as the OS prioritises its own threads.



Finally, by utilising 10 threads, we can obtain enough breathing room required for a training process that only takes ~1.5 seconds. As for the accuracy results on all projects, they are in line with common results such as this and illustrate the limitations of linear models. Even if now we have the performance headroom for a significantly greater number of epochs, the model is unable to accomplish accuracy beyond ~9.2





[1] CPU/SIMD Optimizations — NumPy v1.26 Manual
[2] GCC optimization - Gentoo wiki
[3] OpenMP - Wikipedia