Parallelism and Float Precision: Report

Bryant Har, et al.

Summary

A one-paragraph summary of what you observed during this programming assignment.

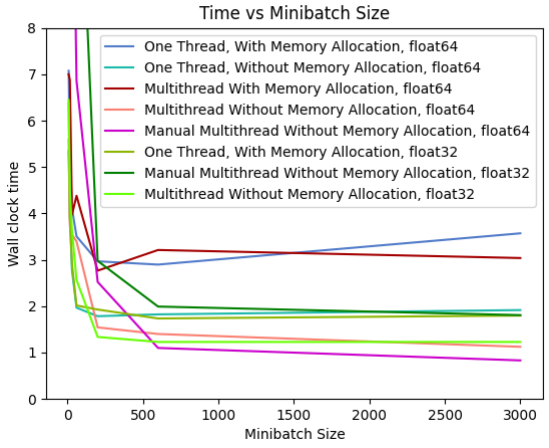
We explored the effects of precision and parallelism in machine learning algorithms and numpy operations. Generally, larger batch sizes, leveraging batching better, led to faster gradient descents. This involved reusing existing numpy arrays instead of reallocating memory, using 32 bit computation instead of 64, using implicit numpy multithreading, and manually multithreading. Overall, comparing across batch sizes, we saw that manually multithreaded without allocation performed the best at very large batch sizes.

The final wall clock times for each type of run are tabulated below against the batch size. Times are given in seconds truncated to 3 decimal places. More precise values are given in the code.

	Batch: 8	**Batch: 16**	**Batch: 30**	**Batch: 60**	**Batch: 200**	**Batch: 600**	**Batch: 3000**
With Allocation	7.074	5.126	4.104	3.505	2.967	2.897	3.571
Without Allocation	5.348	3.937	2.777	1.967	1.785	1.824	1.916
Multithread With Allocation	6.999	6.872	3.975	4.381	2.766	3.211	3.038
Multithread Without Allocation	5.593	3.727	3.544	3.395	1.543	1.401	1.125
Manual Multithread Without Allocation	45.617	24.395	13.322	6.859	2.523	1.100	0.832
With Allocation (32 bit)	6.384	3.948	2.901	2.015	1.928	1.735	1.798
Manual Multithread Without Allocation (32 bit)	84.236	41.927	24.810	13.423	2.976	1.992	1.808
Multithread Without Allocation (32 bit)	6.447	4.513	3.622	2.565	1.337	1.229	1.230

All times given in seconds truncated to three decimal places

This data is graphed shown below. A close up of the best performing runs (under 2s for batch size 3000) is shown at part 4.



The (single) figure displaying the wall-clock time results from all parts.

Part 1

Wall-clock time measurements from Part 1.3, and a short explanation of which function is faster and how much faster it is.

As tabulated above, the results are also summarized below. Results for batch sizes of 16 are clear.

	Batch: 8	Batch: 16	Batch: 30	Batch: 60	Batch: 200	Batch: 600	Batch: 3000
With Allocation	7.074	5.126	4.104	3.505	2.967	2.897	3.571
Without Allocation	5.348	3.937	2.777	1.967	1.785	1.824	1.916

Clearly, **not allocating new arrays** proved to be significantly faster than allocating new arrays. This speedup is evident from the graph below in part 2. Without allocation, for 16 batch size, was ***1.23x*** faster than with allocation.

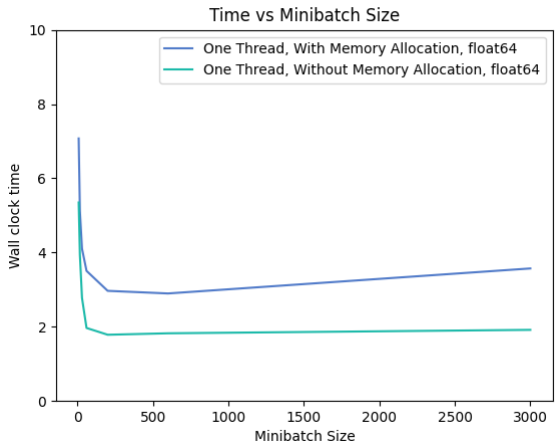
without allocation is 23% faster

Part 1: 2

Answers to questions from Part 1.4. Plot the resulting wall-clock times for both functions together on a single figure with x-axis the minibatch size and y-axis the wall clock time. How does the relative performance of the two functions compare as the minibatch size is changed? Can you explain why you observed this?

The plot is shown below. As minibatch size changes, the advantages and speedup of not allocating the new array becomes clearer and clearer. The with memory allocation plot shows significantly slower performance than without memory allocation and seems to take more time as batch size increases. By contrast, the without allocation plot stays roughly stable for all batch sizes. Clearly, without allocation is superior across all batch sizes. With allocation slows down with batch size, while without allocation stays stable.

We can explain this, since we save time by not re-allocating space for new arrays and instead reusing old arrays. This means that without memory allocation will be faster than with memory allocation, since with memory allocation has to reallocate new memory proportional to the batch size at every iteration. This is why its graph seems to trend up, while without memory allocation seems stable. With memory allocation must allocate a 3000 batch array at every iteration which is costly, while without simply reuses the old array and saves that time.



Part 2

The number of cores on your CPU in Part 2.1 and the method you used to identify that, and answers to questions from Part 2.3.

I have ***8 cores***. On Mac, opening "System Information" reads,

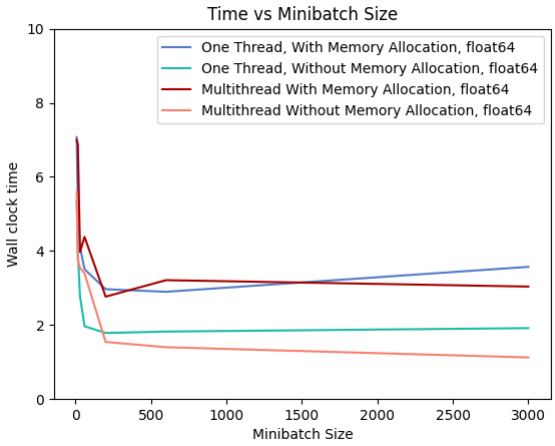
```
Total Number of Cores: 8 (4 performance and 4 efficiency)
```

Therefore, I used 4 threads when implicitly multithreading, as I thought that was a solid number.

8 Cores

Q2.3: How does the speed of the algorithm using multithreading compare to the speed when no multithreading is used? Note that there should now be four series on this figure, two for 1-thread execution and two for multi- threaded execution

The wall clock times are shown below. Clearly multithreading induces a significant speedup over no multithreading. Specifically, for 3000 batch size, multithreading induced a **15%** speedup for with allocation and a **41%** speedup for without allocation. This is because we parallelize the operations implicitly.

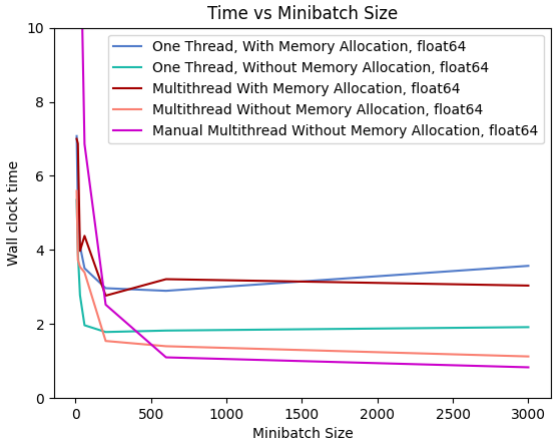


Part 3

Answers to questions from Part 3.2.
How does the wall-clock time of your manually multithreaded code compare to that of the other algorithms? Can you give an explanation for any trends you observed?

The times are plotted below. Clearly, **manually multithreaded code in magenta performs much faster than the other algorithms**. It is by far the lowest curve by the end, meaning it took the least amount of time. For smaller batches, it seems to perform not as well. However, we can explain this trend since manually multithreading suffers if the task is small enough that it's difficult to parallelize. Extremely small batch sizes are much less efficient to parallelize and require more iterations of setting up threads, and as such, the overhead of repeatedly setting up the thread dominates the runtime, making it perform much more slowly than unthreaded code. We also observe the trend that it's faster than our implicitly parallelized code. This is clearly because manually threaded code is more specialized and tailored to our purpose. It can therefore leverage our knowledge of the task to make it more efficient than a general purpose

multithreading schema.



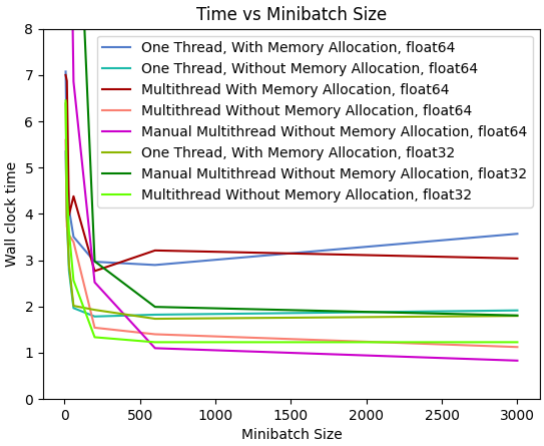
Part 4

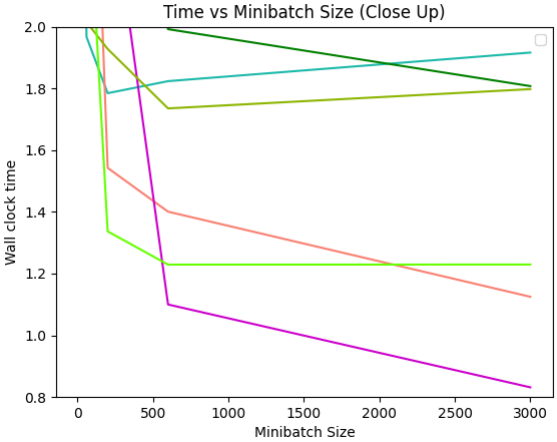
Answers to questions from Part 4.3. How does the wall-clock time of your 32-bit code compare to that of the other algorithms? Can you give an explanation for any trends you observed?

The plots and the blown up plots are shown below. Interestingly, the 32-bit float computation seems to perform as well or sometimes worse than our 64-bit version. Generally, we observe the trend that 32-bit versions of our above algorithms are **equally fast or slightly slower.**

We can rationalize this fairly well. We could expect 32-bit float to be faster, since single precision lends to a greater data throughput, less memory to store each number, lower power requirements, and a wider "data highway". 32 bit float can fit more data, and data transmits faster.

However, this trend we observe can be explained. Modern CPUs trend toward 64 bit computation, since that modern programs typically work in double precision. My computer, for instance, was designed in 2021. As such, it likely has specialized hardware and extra registers for 64 bit computation that lends itself to faster computations. Modern hardware is specialized and tailored to perform and accelerate 64-bit computations, and this explains why 32 bit fails to perceptibly improve our runtime. The plots follow:


Time vs Minibatch Size
One Thread, With Memory Allocation, float64
One Thread, Without Memory Allocation, float64
Multithread With Memory Allocation, float64
Multithread Without Memory Allocation, float64
Manual Multithread Without Memory Allocation, float64
One Thread, With Memory Allocation, float32
Manual Multithread Without Memory Allocation, float32
Multithread Without Memory Allocation, float32
Wall clock time
Minibatch Size
0
1
2
3
4
5
6
7
8
0
500
1000
1500
2000
2500
3000


Time vs Minibatch Size (Close Up)
Wall clock time
Minibatch Size
0.8
1.0
1.2
1.4
1.6
1.8
2.0
0
500
1000
1500
2000
2500
3000