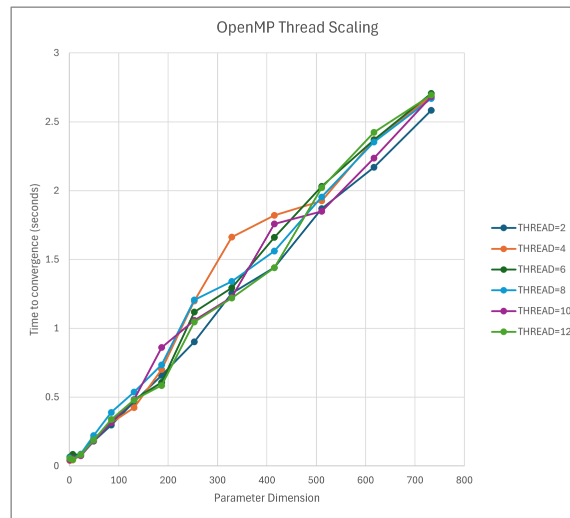
Efficient Gradient Descent - Vectorization, Parallelism, and CUDA deployment in C

James Utley
utleyj@bu.edu



1. Introduction

Gradient Descent is the backbone of machine learning and the AI boom that has occurred over the last several years. Several variants of the algorithm offer better performance at the cost of precision or better precision at the cost of performance. Almost all of the most effective deployments of gradient descent have their roots in C and its derivatives.

This project set out to deploy gradient descent from scratch in C and in CUDA C#. The choice to only examine gradient descent and none of its variants is deliberate: in order to understand the effects of coding optimization techniques, it is important that the algorithm remains the same while the coding techniques change. Comparisons with other types of gradient descent would not be worthwhile comparisons. While it would be interesting for future projects to similarly examine the effects of coding optimization variants of gradient descent, this project remains restricted in scope for clarity and consistency.

2. Related Work

There has been a lot of work done on gradient descent optimization, most of which have created the many variants of gradient descent. Gradient descent is an optimization algorithm very closely associated with machine learning so research on its efficiency is many and varied.

There is also a lot of work done on the relationship between martian crater diameter and depth. Millot et al. [1] noted that there exists a linear relationship between the diameter and depth. This is further corroborated by earlier work from Barlow and Bradley [2] about relationships between martian crater geography. This relationship thus makes this an applicable problem for gradient descent.

While there have been several uses of machine learning algorithms for crater detection [3], the dataset that this project uses comes from Stuart Robbins [4], a research scientist with a passion for scientific education. The dataset that he compiled is detailed in the following section.

3. Method

Dataset:

This project uses a dataset of Martian craters created by Stuart Robbins. The dataset stores data for 384,344 craters, including information on crater name/identifier, crater depth, crater diameter, its number of layers, and its coordinates on Mars. For the purposes of this project, we consider crater depths and diameters that are both greater than 0, which reduces the number of usable data to 76,804 craters.

Functions:

This project uses three gradient descent implementations: serial, OpenMP, and CUDA. Each function runs 500 epochs. To validate the accuracy of the functions, each implementation initially uses scalar parameters, and records its weight, loss, and bias at each epoch. After the validation that each function computes the same final and intermediate values, the dimension of the parameter vector is increased according to the equation:

$$dimension = Ax^2 + Bx + C$$

Where A, B, and C are constants and x is the iterator counter variable.

All of the functions use the following algorithm:

```
X = Vector of length n
F =function/dataset
Tolerance = 1e-4 = 0.0001
η = Step Size
Repeat{
  //Predict value based on
parameters
  //Compute loss
  //Compute gradient
  //Update parameter
  X_new=X_old-η*F'(X_old)
  if |F(X_new)- F(X_old)|<Tolerance
  break;
}
```

Unlike gradient descent for a known function, this deployment of gradient descent computes the gradient of an unknown function from data. This deployment of gradient descent was chosen because of its academic rigor as compared to the trivial gradient descent solution for a known function.

Parameters, Optimization, and Descent:

Based on existing research, there exists a linear relationship (see above) between Martian crater diameter and crater depth. Thus this project considers the following function:

$$depth = w * diameter + b$$

In this equation, *w* is an *n* dimensional weight parameter vector and *b* is an *n*-dimensional bias parameter vector.

It is important to note that the weight and bias parameters that apply to predicting the depth are one dimensional by default from the data. In order to assess the scalability of OpenMP and CUDA, the dimensionality of the parameters was artificially increased by performing the same calculation n times, and therefore not impacting the converged output. This allows for scalability experimentation without changing the dataset.

4. Description of the Theory

This project relies on the nature of parallelism and the physical structure of GPUs to optimize gradient descent.

The optimized CPU code relies on OpenMP as its primary optimization. Other methods were also considered such as loop unrolling and vectorization, however the nature of gradient descent already made it fairly vectorized. Therefore, early deployments of vectorization and unrolling produced either a similar performance to the

serial function or a worse one. However, the gradient descent algorithm is very parallelizable, the effects of which are most pronounced as the dimension of the parameters increases. Exploiting the independent parameter dimension updates allows for the deployment of parallelism in the prediction step, the loss calculation, the gradient computation step, and the parameter update step. This was accomplished via OpenMP, the thread toolbox of C that allows for thread scheduling with little syntactic overhead.

Furthermore, the structure of GPUs allows for CUDA to run gradient descent efficiently. Graphical Processing Units (GPUs) were originally built for displaying video and videogames at high frame rates. Because of their ability to process pixel outputs, their structure optimizes the processing of matrices in parallel. Therefore, the gradient descent algorithm, which utilizes vector parameter updates, is ideal for the gradient descent algorithm.

These two parallelism techniques are the theory this project seeks to assess, namely how much better they do than a serial implementation of gradient descent.

5. Parallel Partitioning

As the previous section noted, the gradient descent algorithm is highly parallelizable. However, there are significant performance bottlenecks in the algorithm that can lead to lower performance as compared to other optimization algorithms.

To understand these bottlenecks, one must understand that there are four main parts of the gradient descent algorithm: Predict y value, calculate loss, calculate gradient, and update parameters.

The prediction of y and the calculation of the gradient handle m vectors of size $1 \, x \, n$ where n is the dimension of the parameter. As such, the computational complexity of these two steps is $O(mn)$. Updating the parameters and computing the loss deal only with the parameter vectors and therefore they each have a computational complexity of $O(n)$. Updating the parameters depends on the computation of the gradient, which depends on the prediction of the y value, leading to the performance bottleneck between these three steps. Because of these dependencies, the algorithm has inherent costs that cannot be simplified by code optimization. This bottleneck inherent in the algorithm forms the partitioning of the parallel functions. The parallel operations happen on the $1 \, x \, n$ vectors m times during the prediction and gradient computation steps, and $1 \, x \, n$ parameter updates and los calculations.

It is important to note that of the four part of the gradient descent algorithm, the loss calculation is the most unnecessary and is only used as a benchmark for the programmer; in this project, the loss calculation is only included in the validation of the functions, and it is excluded during the scaling of the parameter dimensions to increase performance.

6. Results

The results of this project were very promising. The functions were first validated using scalar parameter vectors. The losses, weights, and biases were recorded at each epoch and were cross-referenced against one another to ensure that the convergences followed the same paths. In figure 1, the average difference of the bias, weight, and loss are recorded. From this

Avg Diff	
Bias	2.5264E-05
Weight	2E-09
Loss	3.92E-07

Figure 1: Average respective difference between parameters

figure we can see that the average difference of the loss and the two parameters is low enough to be negligible and can easily be attributed to the slight imprecision of parallel floating point operations in the OpenMP optimized code on the cpu and CUDA on the gpu. The resulting graphs of the loss, weight, and bias can be seen at the end of this report.

The result to which the functions converged at was:

$$depth = 0.0001 * diameter + 0.3783$$

In order to test the scalability of the gradient descent algorithm and limited by the dataset, the number of parameters was artificially increased by performing the same parameter update *n* times. With this, this project was able to assess the scalability of the gradient descent algorithm to higher parameter dimensionality without compromising the computation of the converged values or changing the dataset used. The number of parameters was increased quadratically with each test according to the equation:

$$dimension = Ax^2 + Bx + C$$

Where A, B, and C are constants defined in the program. The initial setting of A, B, C set them to 5, 1, 1 where x increased from 0 to 12 in step sizes of 1, resulting in 13 total tests. The parameter vectors, weight and bias, reached a minimum dimension of 1 and a maximum dimension of 733. The results of this scaling are shown in figure 2.

Notice the effect of increasing the dimension of the parameter vector: as the dimension grows larger, the optimized functions perform better than the serial version, with the CUDA function performing the best consistently. This was expected and aligned with the hypothesis of the project.

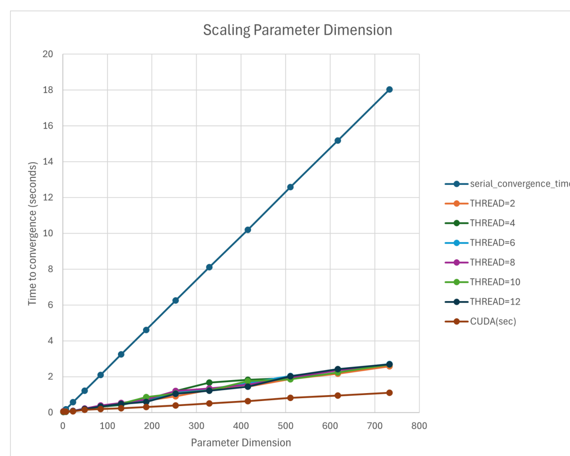


Figure 2: Effects of scaling parameter dimension

One interesting outcome is that the performance of the individual thread amounts was roughly equivalent, with slight variations. Figure 3 examines this relationship, but even Figure 3 does not present a clear picture of the effects of OpenMP threading. It is difficult to identify which of the threads performs the best consistently since the graph is very dynamic and the different thread counts contribute erratically. However, it does seem that the threads converge to similar values both as the parameter dimension increases to large values. Because of the similarity of the values, I believe that this is likely due to the machine the program ran on was not allocating threads as was directed, and a future direction would be to run the same

simulation only on OpenMP on a completely clear machine.

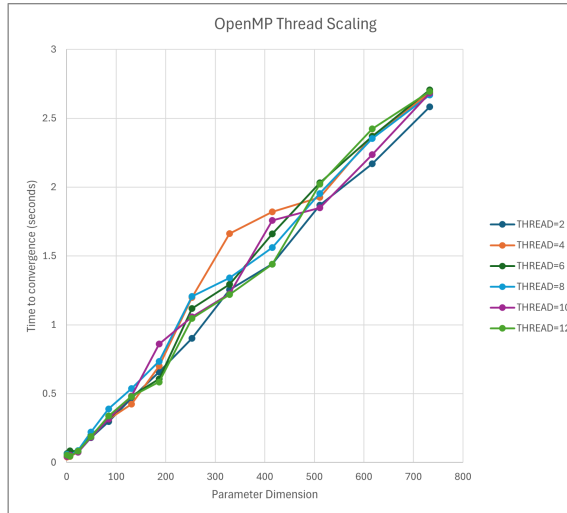


Figure 3: Parameter vector dimension scaling effect with different OpenMP Thread counts

The more interesting scalability relationship to investigate is the parameter dimension where CUDA or OpenMP become more efficient to run than the serial function. Figure 4 looks at this relationship.

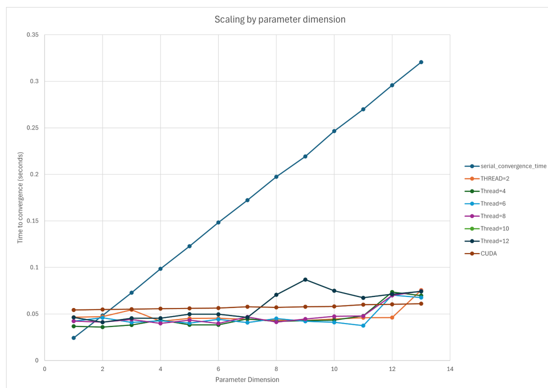


Figure 4: Small increment scaling of parameter dimensions

As Figure 4 notes, the OpenMP optimization outperforms the serial function when the parameter dimension equals two, and the CUDA optimization outperforms the serial when the dimension size equals three. That is to say, the serial code only performs better than the parallel optimizations when the parameters are scalars.

Another important observation from Figure 4 is the size of dimension where CUDA outperforms the OpenMP optimization. OpenMP dominates speed until when the parameter dimension reaches 8, where CUDA outperforms thread size 12. Then at dimension 12, CUDA outperforms threads 4,6,8, and 10, and at dimension 13 CUDA outperforms all of the OpenMP values.

The results of this project are as expected: the serial control outperformed OpenMP and CUDA while the parameter dimension was small due to the overhead costs of allocating threads and CUDA memory, but ultimately both OpenMP and CUDA were faster than the serial. This result makes sense with our knowledge of the gradient descent algorithm. Since gradient descent is very parallelizable, the parallel modalities of OpenMP and CUDA get great performance from the algorithm. What this project has shown is that parallelism and GPU processing are very powerful implements for gradient descent and any application using high dimensional parameters or data ought to use parallelism in their gradient descent deployments or face linearly increasing computation costs with serial implementations.

7. Bibliography

1. Millot, C., Quantin-Nataf, C., Dehouck, E., Torres, I., & Volat, M. (2025). Depth to diameter relationships for <50 m diameter Martian craters. Journal of Geophysical Research: Planets, 130, e2024JE008844. https://doi.org/10.1029/2024JE008844
2. Di, K., Li, W., Yue, Z., Sun, Y., & Liu, Y. (2014). A machine learning approach to crater detection from

topographic data. Advances in Space Research, 54(11), 2419–2429. https://doi.org/10.1016/j.asr.2014.08.018

3. Barlow, N. G., & Bradley, T. L. (1990). Martian impact craters: Correlations of ejecta and interior morphologies with diameter, latitude, and terrain. Icarus, 87(1), 156–179. https://doi.org/10.1016/0019-1035(90)90026-6
4. Robbins, S. Mars Crater Dataset. https://about.sjrdesign.net/research_mars.html.

List of Figures
Enlarged for readability

Figure 1: Average respective difference between parameters

Avg Diff	
Bias	2.5264E-05
Weight	2E-09
Loss	3.92E-07

Figure 2: Effects of scaling parameter dimension

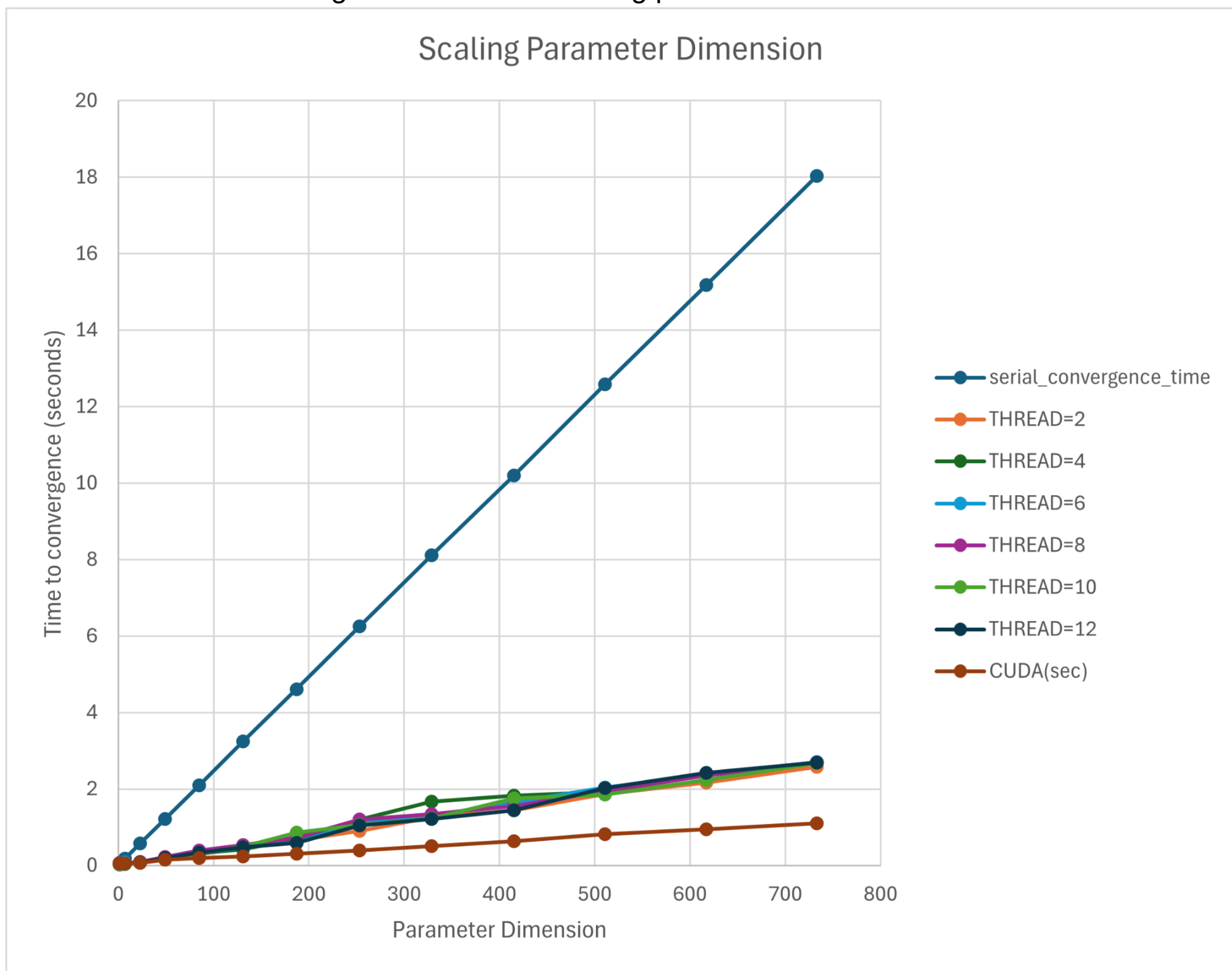


Figure 3: Parameter vector dimension scaling effect with different OpenMP Thread counts

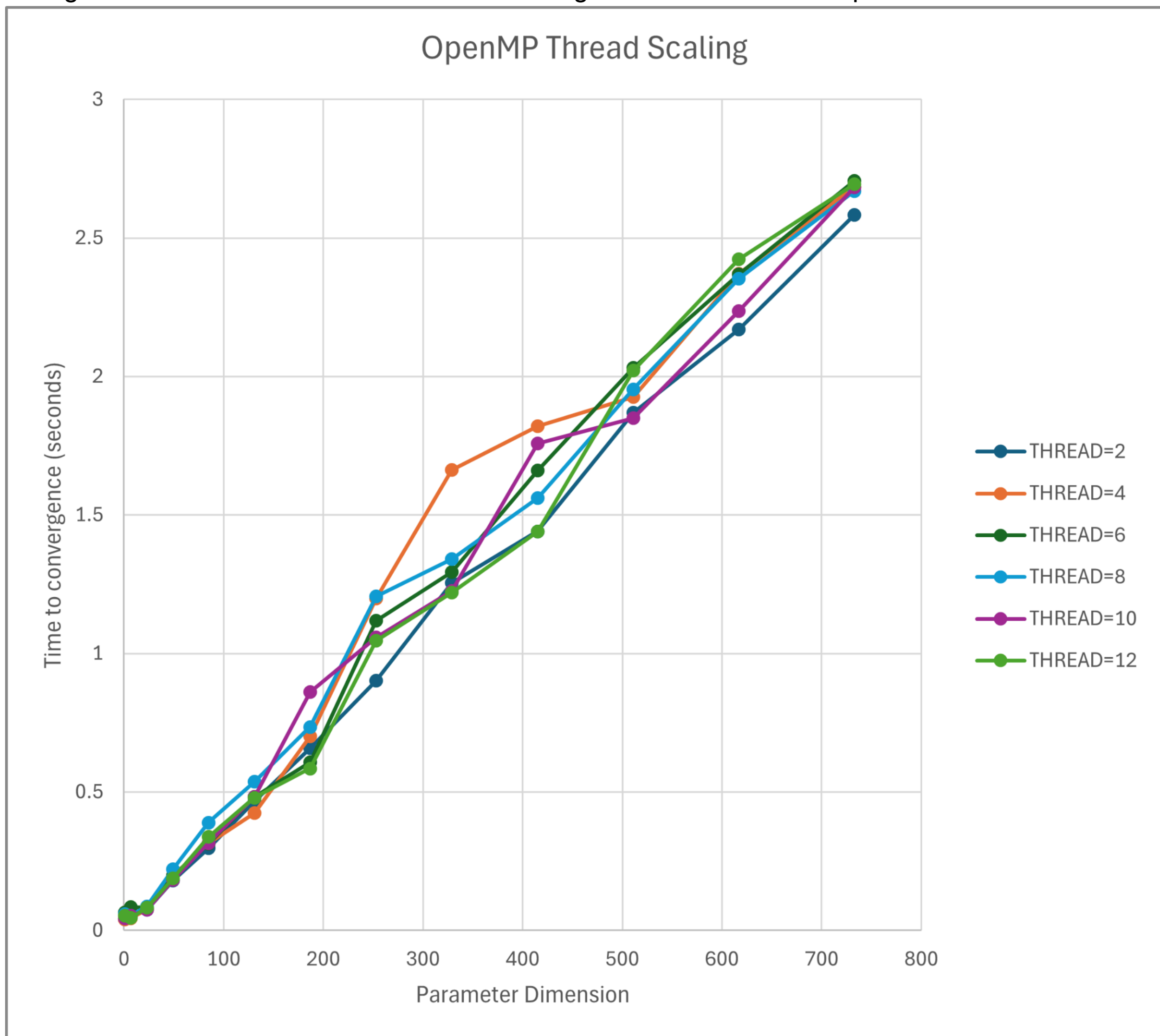


Figure 4: Small increment scaling of parameter dimensions

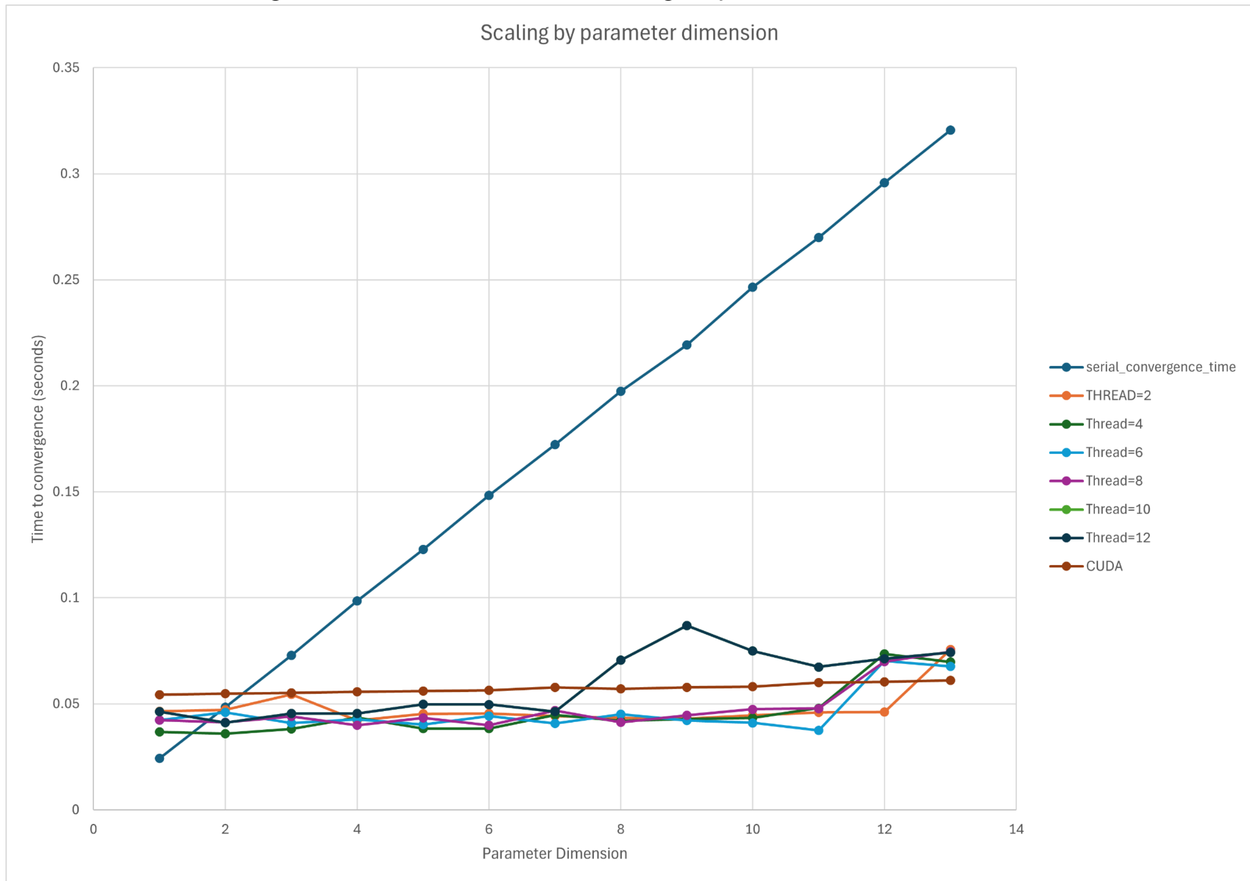
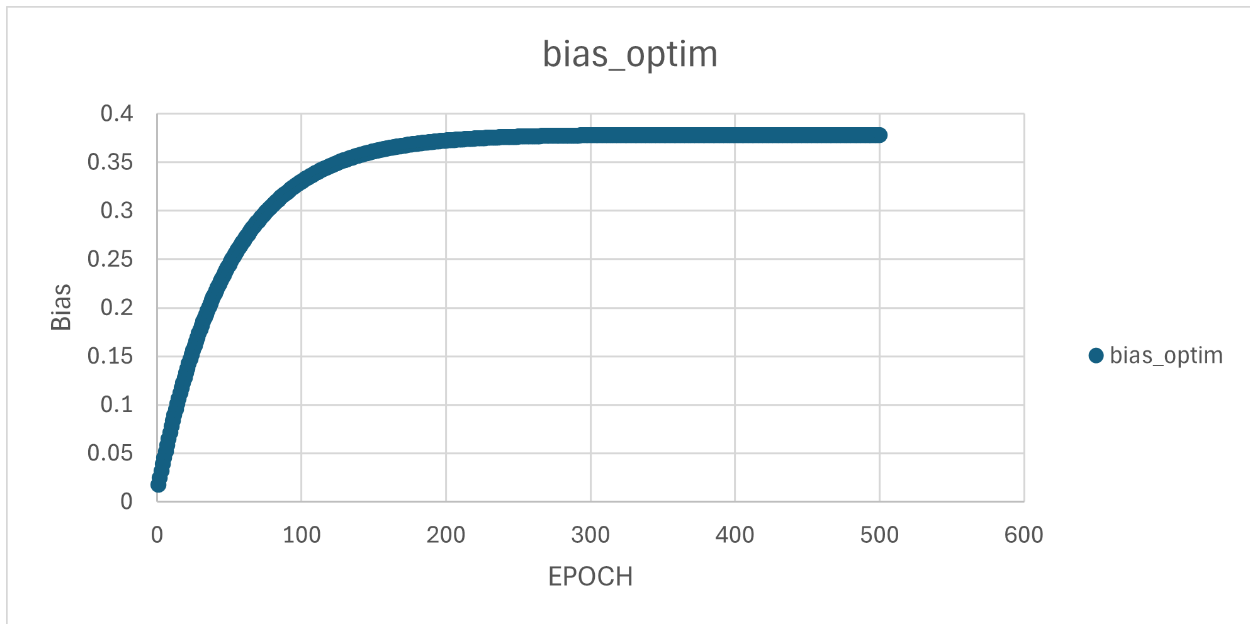
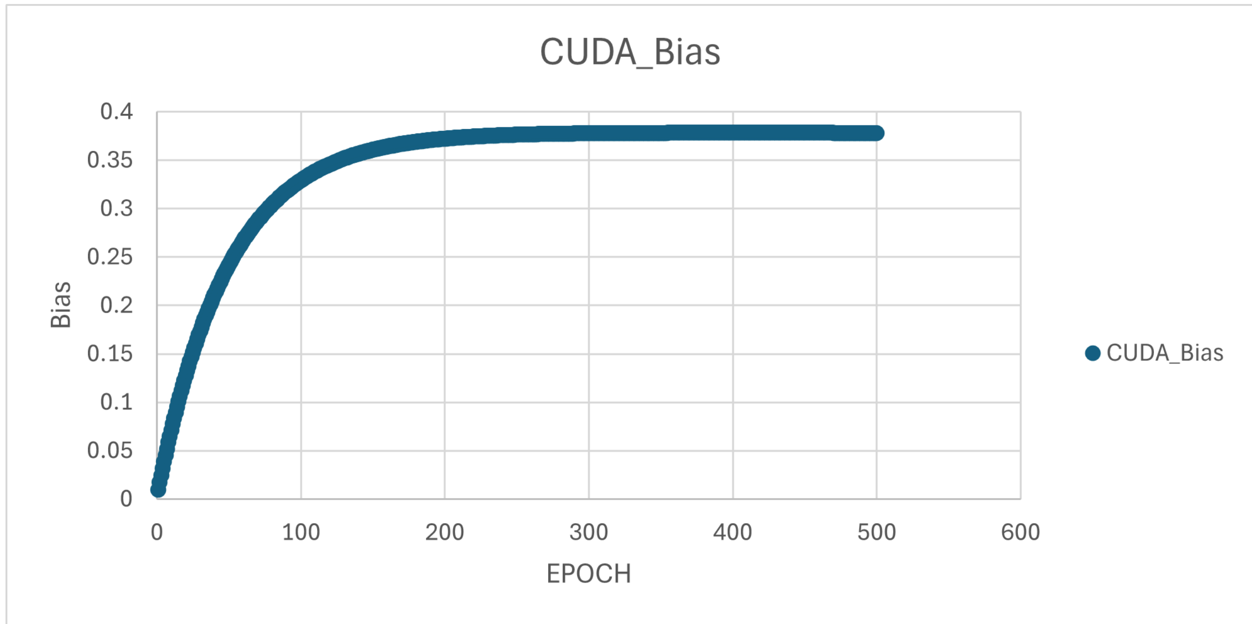
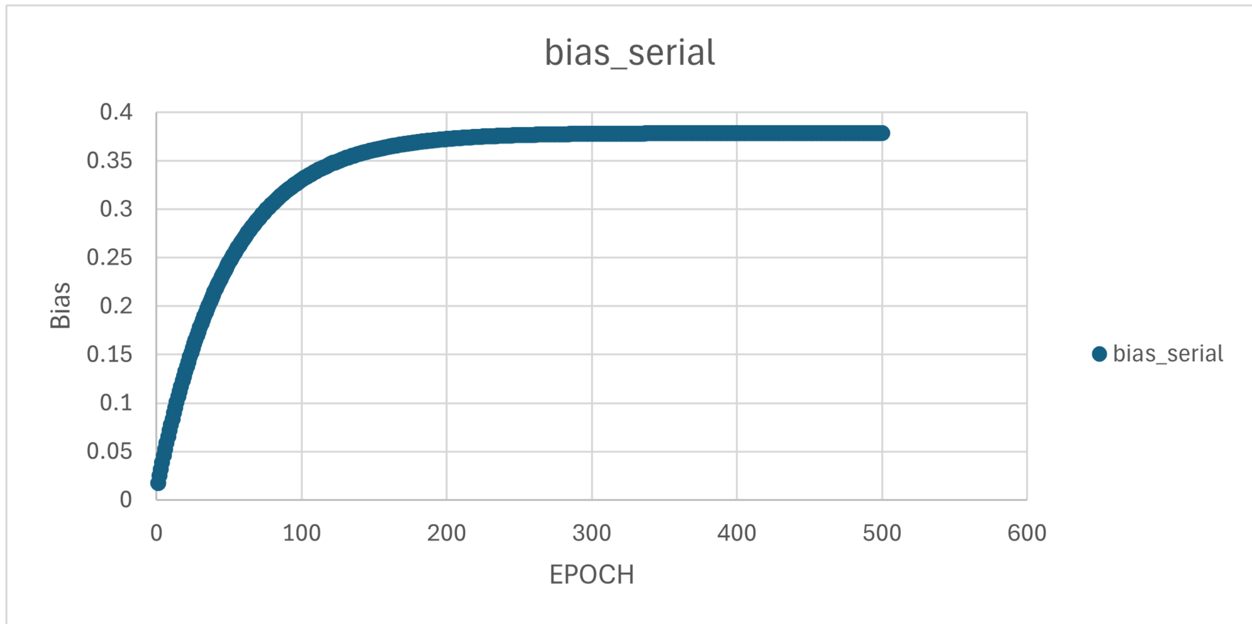


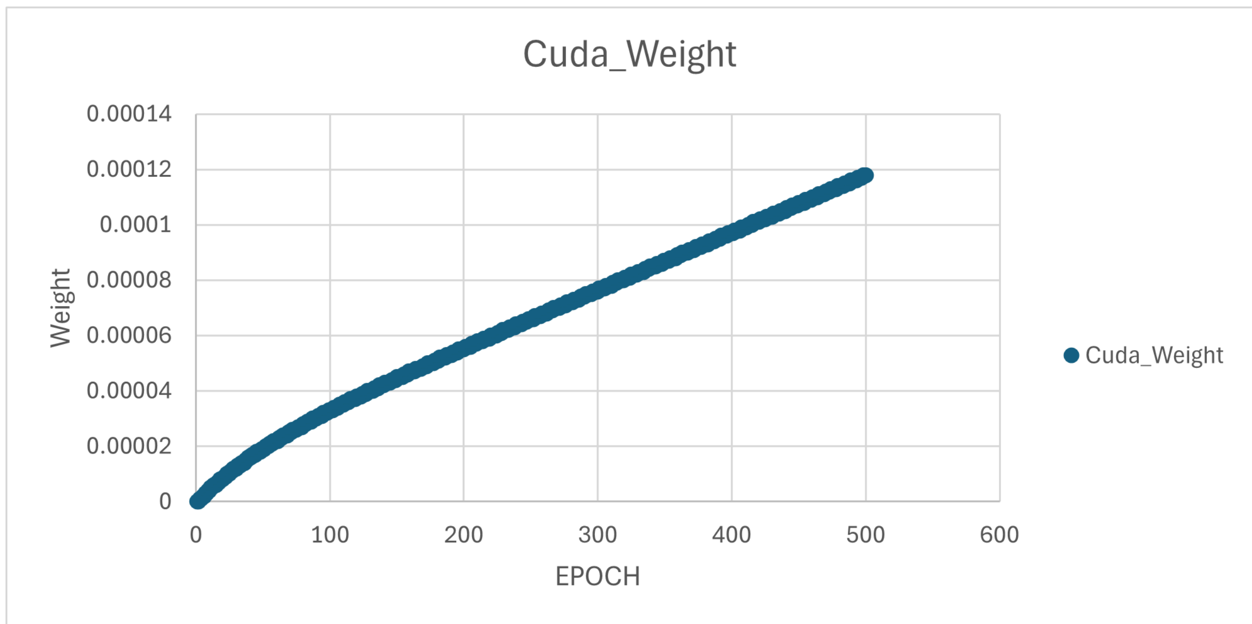
Figure 5, 6, 7: Bias parameter validation data for OpenMP optimization, CUDA, and serial control

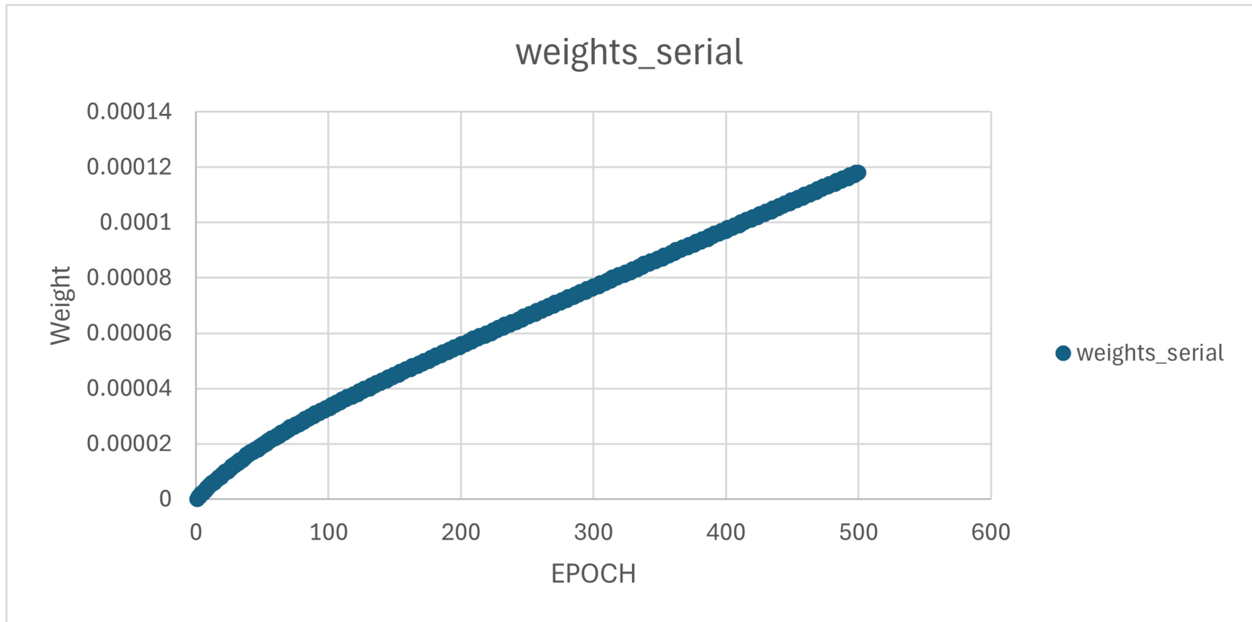



CUDA_Bias
Bias
0.4
0.35
0.3
0.25
0.2
0.15
0.1
0.05
0
0
100
200
300
400
500
600
EPOCH
CUDA_Bias


bias_serial
Bias
0.4
0.35
0.3
0.25
0.2
0.15
0.1
0.05
0
0
100
200
300
400
500
600
EPOCH
bias_serial

Figures 8, 9, 10: Weight parameter validation data for OpenMP optimization, CUDA, and serial control




weights_serial
0.00014
0.00012
0.0001
0.00008
0.00006
0.00004
0.00002
0
Weight
0
100
200
300
400
500
600
EPOCH
weights_serial

Figures 11 and 12: Loss validation data for OpenMP optimization, CUDA, and serial control