Parallelization of Stochastic Gradient Descent

E4750 2018 - sgd1
James F. Xue jx2181,
Columbia University

Abstract

Stochastic gradient descent is a popular technique in machine learning and deep learning. However, SGD is difficult to parallelize since the algorithm updates a shared weight matrix every time-step, requiring expensive locking for atomic updates. This work explores four modes of parallelization: algorithmic, batch, multiclass, and lock-free. The main challenge lies in aggregating the results into a single loss value, which the lock-free parallelization seeks to address. We find that while all modes of parallelization offers some benefit, additional parameter optimization is needed to maximize the benefit of parallelization.

1. Overview

1.1 Parallelizing SGD in a Nutshell

Stochastic gradient descent (SGD) is a powerful and common technique in machine learning for accelerating optimization of loss functions. It is a modification of gradient descent (GD), which evaluates the loss at all data points per time step. Stochastic gradient descent enables faster learning than gradient descent through evaluating at a single data point per time step [2]. A further variation on GD is batch gradient descent (BGD), which takes a small subset of data points to evaluate upon. BGD often enables more stable convergence than SGD at the cost of evaluating at more points [3].

With the advent of big data, SGD, GD, and BGD have become valuable techniques for training machine learning and deep learning models, especially SGD. Simultaneously, the continual grow of datasets and the pressure to improve model performance demands that these techniques training faster and better. Parallel processing with modern GPUs could be a potential solution to this challenge. This work seeks to explore accelerate SGD and BGD through use of several modes of parallelization.

The first and most straightforward mode of parallelization is parallelizing the SGD algorithm itself (the work will refer to this mode as algorithmic parallelization). Modern data often have hundreds or thousands of dimensions. For SGD to evaluate loss at a single data point, it often requires calculating the dot product of the n-dimensional data point and an n-dimension weight matrix. Parallelizing the dot product calculation could potentially improve SGD running time. The challenge in this mode of parallelization is summing the result of each thread, which potentially bottleneck the running time.

The second mode of parallelization is parallelizing calculating the loss at multiple points in BGD (the work will refer to this mode as batch parallelization). Since BGD calculates the loss at multiple points in each time step, the calculation could be parallelized by tiling over the batch size in each time step [3]. The challenge in this mode is similar to before: the aggregation of the loss in each time step can bottleneck the running time.

The third mode of parallelization is parallelizing multiple SGD's (the work will refer to this mode as multiclass parallelization). Multiple SGDs occur in multiclass classification. A common technique for multiclass classification is one-vs-all, which attempts to find a decision boundary between points of one class and all other classes. Thus, k classes will require k decision boundaries and thus k loss functions. Since all loss functions are separate, there will not be any aggregation between each of the k classes and thus will avoid much of the bottlenecking of the other modes of parallelization.

The last mode of parallelization is an alteration of the first mode of parallelization called 'Hogwild!' (the work will refer to this mode as lock-free parallelization). This technique was described in a work by Feng Niu at the University of Wisconsin-Madison. It is a lock-free version of the first mode of parallelization that throws away the atomic summing aspect of loss calculation [2]. The challenge in this mode of parallelization is that it might not converge with enough stability due to the errors that might accumulate through avoiding the locking process.

1.2 Prior Work

While SGD itself has been popularized by many works, this work focuses specifically on parallel SGD, which encompasses a far smaller body of work.

An early paper by Langford, Smola, and Zinkevich [4] seeks to address the challenge of parallelization through an algorithm they named 'Delayed Stochastic Gradient Descent' (DSGD). Instead of updating the weight vector with the current gradient, it updates the weight with a gradient from t time-steps earlier. The rationale for this delay is that enables a scheduler to manage the usage of the cores for either updating the weight vector or computing the gradient--a scheme that is effective when gradient calculation takes a significant amount of time. Ultimately, this is a pipeline optimization to minimize *latency* rather than a true algorithmic one that improves *throughput*.

Another solution proposed by Zinkevich, Weimer, Smola, and Li is an aggregated SGD technique where each core runs a batch SGD on a varying fraction of a subset of all training data. After each core runs for a fixed time-step or until convergence, the individual weight matrices are averaged and returned. In essence, the results from each batch SGD instance is completely separate until the end. While this algorithm is focused on theoretical guarantees, the speedup is a result of splitting data across cores.

A more interesting solution proposed by Niu, Recht, Re, and Wright implements SGD without locking. The update scheme allows processors access to memory with the possibility of overwriting each other's work. This avoids much of the bottleneck of SGD methods while provides an experimentally performant rate of convergence.

2. Description

This work chooses to benchmark SGD with a standard library (scikit-learn's SGDClassifier module), which computes SGD serially or with a multicore option (vs. our many core approach). This benchmark is then compared with the four modes of parallelization using the MNIST dataset (55000 rows of training data) which is a collection of handwritten images (28x28) of numbers between 0 and 9. Both the training time as well as the test accuracy are charted and compared for up to 100 iterations.

2.1. Objectives and Technical Challenges

The objective of the four different specifications of comparison against the benchmark is to:

1. demonstrate the strengths and weaknesses of each approach
2. balance the trade-offs between run-time, accuracy, and stability associated with the different approaches
3. highlight the limitations of each approach

The challenge in the first two specifications (algorithmic and multiclass) is aggregating the results of each kernel atomically and efficiently. This is the main bottleneck in computing SGD that this work attempts to address. We need to intelligently aggregate results so that running time is efficient while maintaining correctness and stability. The challenge in the last two specifications (lockfree and batch lockfree) is stability and accuracy of results.

2.2. Problem Formulation: MNIST Classification

The problem we are applying to the benchmark as well as our four implementations of gradient descent is classifying MNIST digits dataset. The dataset is composed of handwritten digit images of 28x28 pixels. Furthermore, the dataset has 55000 training images, large enough that a hundred training iterations will take around 40 seconds using a standard package.

The machine learning problem we face across the benchmark and our implementations is a multiclass classifications problem. We have 10 classes, the digits 0 to 9, that we would like to classify each data point into. To tackle this problem we use a *one-vs-rest* approach where we create 10 classifiers that distinguish each class from the rest of the classes. This work will use logistic regression as the classifier that we will use stochastic gradient descent to optimize.

Logistic regression gives a probabilistic measure for each weight by applying the dot product of the weight **w** with the data point **x** into a sigmoid or logistic function [6], which maps the dot product to the range [0,1].

$$h(\mathbf{x}) = \theta\left(\sum_{i=0}^{d} w_i x_i\right) = \theta(\mathbf{w}^{\mathrm{T}}\mathbf{x})$$

$$\theta(s) = \frac{e^s}{1+e^s} = \frac{1}{1+e^{-s}}.$$

For a true label y_true, we want to maximize the dot product. The reverse holds for a negative label y_false. Thus the objective (likelihood) functions that we want to maximize is [6]:

$$E_{\text{in}}(\mathbf{w}) = \frac{1}{N}\sum_{n=1}^{N} \ln(1 + e^{-y_n \cdot \mathbf{w}^{\mathrm{T}}\mathbf{x}_n})$$

The known stochastic gradient descent update to this objective function is [6]:

$$\mathbf{w}(t+1) \leftarrow \mathbf{w}(t) + y_* \mathbf{x}_* \left(\frac{\eta}{1+e^{y_* \mathbf{w}^{\mathrm{T}}\mathbf{x}_*}}\right)$$

A similar weight vector update formula is implemented (with batch tweaks) across my implementations of gradient descent.

2.3 Software Design

The core algorithm can be summarized in the following steps:

Stochastic Gradient Descent:
for 1, 2, 3, … max_epochs:
  for each data point **x** in **x_train**:
    for each class 0, 1, 2 ...9 :
      w = w_class
      find dot product **wx**
      find gradient G(**wx**)
      update: **w = w** - stepsize*G(**wx**)

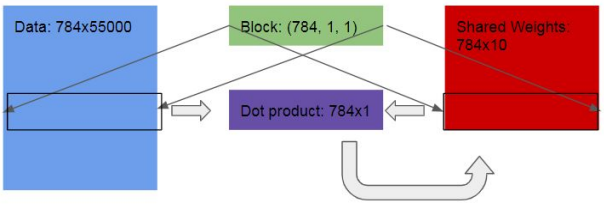
This generic algorithm uses a single point each step to find the gradient G(**wx**), which it uses to update **w** for a particular class.

The main source of shared memory usage in all of the following step is sharing the **w** matrix. Since there are 10 classes and the data dimension is 784, we use a 10x784 **shared_w** matrix to manage bandwidth constraints.

2.3.1: Algorithmic Parallelization Design

The algorithmic parallelization scheme parallelize the calculation of the gradient at each step. The main area of parallelization is calculating the dot product **wx**, which has a dimension of 784 (28x28) in our MNIST classification problem. It uses a block of dimension (784, 1, 1) so that each thread calculates a single dimension of **wx** per timestep per class. It then aggregates all the single dimensional wx into **wx** with a tree based reduction where the results of two threads add until there is only the aggregate **wx** left.

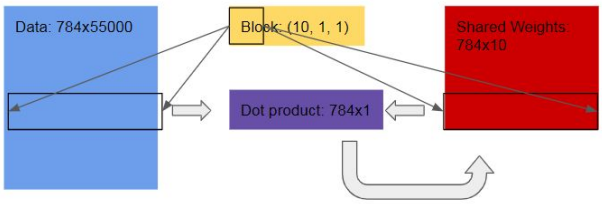
Algorithmic Gradient Descent:
for 1, 2, 3, … max_epochs:
  for each data point **x** in **x_train**:
    for each class 0, 1, 2 ...9 :
      w = w_class
      find dot product **wx**
        map thread tx to data dimension
        find wx for each dimension
        reduce all wx into **wx** using a tree
      find gradient G(**wx**)
      update: **w = w** - stepsize*G(**wx**)



2.3.2 Multiclass Parallelization Design

Since we have 10 classes in our multiclass classification problem, we essentially have 10 logistic regressions to optimize under the *one-vs-rest* scheme. Instead of looping each thread through the ten classes, we can have separate threads handle the stochastic gradient descent of a different logistic regression classifier. This approach doesn't parallelize the calculation of the gradient itself, but takes advantage of the multiclass nature of our problem.

Multiclass Gradient Descent:
for 1, 2, 3, … max_epochs:
  for each data point **x** in **x_train**:
    w = w_class where class = threadIdx.x
    find dot product **wx**
    find gradient G(**wx**)
    update: **w = w** - stepsize*G(**wx**)

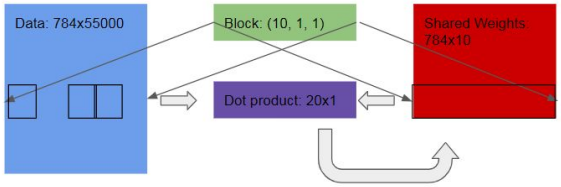


2.3.3 Lock-Free Parallelization Design

The lock-free parallelization is based on the Hogwild! paper. Per timestep, the Hogwild! algorithm takes a subset of the {1, 2, 3, … 784} dimensions and calculates gradient and updates the weights of only those selected dimensions. Part of the Hogwild! implementation is the randomization of the dimension to update. Due to the absence of rand() in pycuda, this work hashed the threadIdx.x to generate 20 semi-random dimensions for each thread to optimize the gradient over. Similar to the multiclass module, it uses a block of dimension (10, 1, 1) to separate out the 10 classes, with each thread handling a different set of 20 semi-random dimensions.

Lock-Free Gradient Descent:

for 1, 2, 3, … max_epochs:
    for each data point **x** in **x_train**:
        select 20 dimensions **e**
        w = **w_class** where class = threadIdx.x
        find dot product wx for each e in **e**
        aggregate into **wx**
        find gradient G(**wx**)
        update: **w** = **w** - stepsize*G(**wx**)

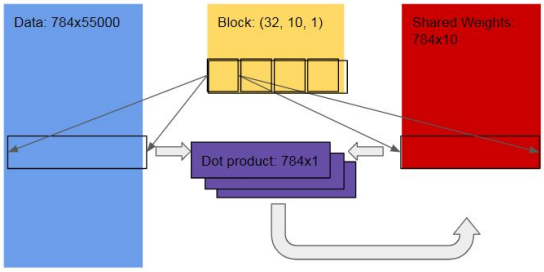


2.3.4 Lock-Free Batch Parallelization Design

Due to the instability of the Lock-Free gradient descent methodology, we choose to implement a batch version of lock-free gradient descent. We choose to use all dimensions for each data point and ingest 32 data points (batchsize = 32) each time step. The lock-free aspect comes from updating the **w** for each class without aggregating or locking. This methodology essentially averages (with possibility of loss) the gradient of 32 points per iterations while reducing the number of iterations per epoch a factor of 32 folds.

Lock-Free Batch Gradient Descent:

for 1, 2, 3, … max_epochs:
    index = 0
    for 32 data points starting at index in **x_train**:
        w = **w_class** where class = threadIdx.y
        aggregate into **wx**
        find gradient G(**wx**)
        update: **w** = **w** - stepsize*G(**wx**)
        index += 32



2.4 Hardware Specifications

Due to the nature of our problem, hardware specification ican be an influencing factor for our experimental results. The experiments were conducted on an Nvidia GTX1070ti gpu and an Intel Core i5-8600 cpu. Their specifications are provided below:

Nvidia GTX1070ti gpu:

- Base Clock: 1607 MHz
- Cuda Cores: 2432
- Memory Size: 8GB GDDR5
- Memory Bandwidth: 256GB/s

Intel Core i5-8600

- Clockspeed: 3.1 GHz
- Cores/Threads: 6
- Cache: 9MB

3. Results

Since all runs are compared with a benchmark with scikit-learn, we begin with a description of our parameters for the benchmark. All experiments contain a comparable run of SGDClassifier from

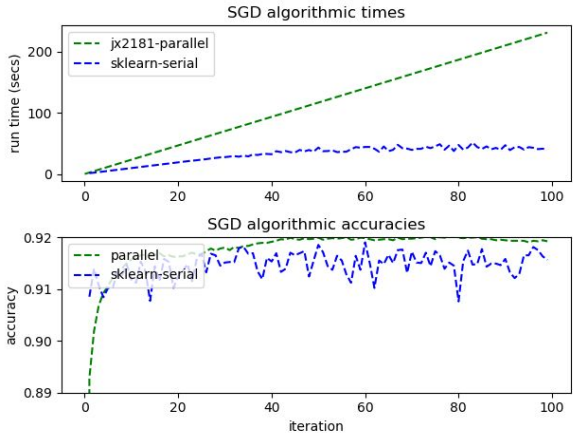
scikit-learn. The scikit-learn benchmark is run with the same specifications in each comparison: eta (learning rate) of 0.01, tolerance of 0.0001, a logistic regression as the classifier, and varying maximum iterations. An important note is that the SGDClassifier is not serial. It is run with a setting that uses all 6 cpu cores.

The benchmarks performed well and consistently. All benchmarks max out at around 40 seconds to reach 100 iterations. The stability is due to the tolerance which ends training. See the note below for additional information. The performance is quite good with accuracy consistently over 0.9.

It is important to note that the running time of the benchmark stabilizes/remains nearly fixed at iteration 30. This is due to the tolerance parameter that stops the training early if the improvement in the objective function is less than that of the tolerance. This 'early stopping' feature is not implemented in our SGD algorithms, hence the linear increase in running time. All subsequent speed comparisons will be using the runtime at iteration 30.

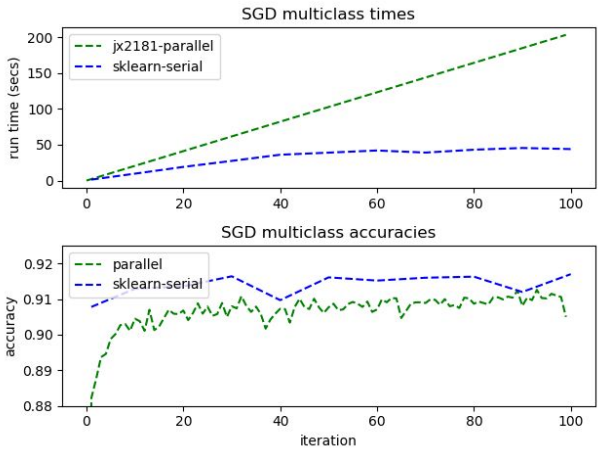
A note regarding the benchmark runtimes, the benchmark was run with varying max_iteration parameters. Thus, the results are not actually the results from a single run. Adding to the tolerance early stopping, there is some variation in the runtime event though the max_iteration has increased. Overall, it is stable around 40 seconds.

3.1. Algorithmic Parallelization



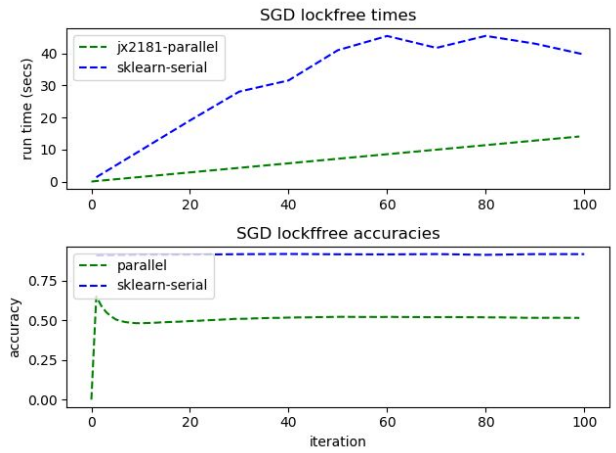
Algorithmic parallelization consistently outperforms the benchmark in terms of accuracy. It beat the benchmark by the 10th iteration and stayed above the benchmark accuracy. However, it is slower than SGDClassifier, with almost twice the run time for the same number of iterations (under 30). This is due to the fact that the main parallelization, dot product, exhibits low arithmetic intensity. Each core has only one multiplication when obtaining single dimensional wx. Furthermore, the reduction to a single **wx** is not well parallelizable. It is not surprising that it obtained comparable accuracy results while being slower than SGDClassifier.

3.2. Multiclass Parallelization



Multiclass Parallelization slightly underperforms scikit-learn in terms of accuracy, with accuracy values closer to 0.91 whereas scikit-learn accuracy hovers closer to 0.92. The main difference is that Multiclass Parallelization is twice as slow as scikit-learn. This difference is unsurprising given that we used a block of (10, 1, 1) and grid of (1, 1, 1) for Multiclass Parallelization. Furthermore, recall from earlier that the scikit-learn SGDClassifier was run with 6 cpu cores. The comparison is really between 10 gpu kernels and 6 cpu cores, and in this case the cpu cores win.

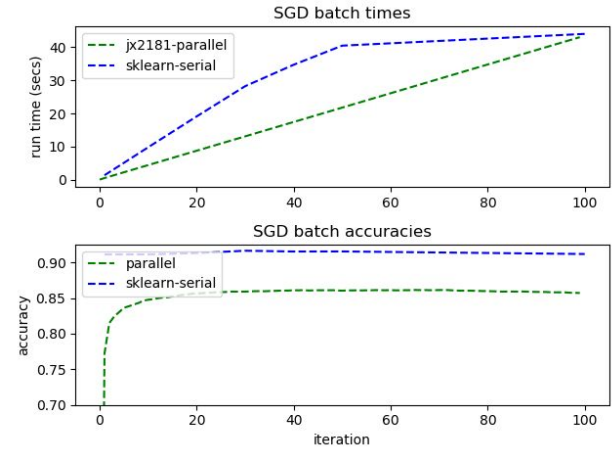
3.3. Lock-free Parallelization



The lock-free implementation is a modification on the Hogwild! algorithm from the Niu paper. Running time improvement was actually quite similar to the Hogwild! results. The Hogwild! paper saw between a 6x and 8x running speed increase for SVM, MC, and Cuts [2] while our implementation saw a 9x running speed increase for logistic regression. These increases are around the same magnitude.

Unlike the paper, we did not see comparable accuracy scores from running the lock-free parallelization. While most of the accuracy score of the implementation hovers around 0.5 is far better than a random guess (expected accuracy score 0.1), it is nowhere near 0.9.

3.4 Lock-free Batch Parallelization



The log-free batch parallelization offers better running time without over sacrificing running time. The running speed (prior to epoch 30) was about 2.5x faster than scikit-learn, while the accuracy of 0.85 is not terribly far from 0.90. While we would've liked the accuracy to be better, this implementation provides enough speed increase that it might be worthwhile in certain situations where accuracy is not paramount.

3.5 Summary Table

	speedup vs scikit	**best accuracy**
scikit-learn	1x	0.915
algorithmic	0.5x	0.922
multiclass	0.5x	0.911
lockfree	9x	0.700
batch lockfree	2.5x	0.850

4. Discussion and Further Work

Since this paper was first inspired by the Hogwild! paper, it is important to first note the results of this work was similar running time-wise but different accuracy-wise. The 8x speed increase in Hogwild! is very much comparable to the 9x increase in the lock-free implementation. However, while the accuracy did not collapse to 0.10 as I feared, it did not produce the results Hogwild! found. While Hogwild! had comparable results between the atomic implementation and their lock-free implementation, this work's lock-free implementation had 0.6 accuracy compared to the benchmark 0.9.

Overall, the results were promising. While slower than the benchmark, the algorithmic parallelization and multiclass parallelization had better and comparable results to the benchmark. More classes and higher arithmetic intensity can potentially improve the relative runtime of these implementations. Furthermore, the batch lock-free was promising since it offered a speed increase for a small accuracy decrease.

One limitation to the results is that the parameters were not optimized. While the scikit-learn benchmark was optimized over learning rate, this work's four implementations used the same learning rate (0.001) for comparability, but were not optimized. Furthermore, several implementations had settings used for convenience rather than optimality. The batch implementation had a batch size of 32. It is possible with a different batch size, its 0.85 accuracy can stably reach 0.90. The lock-free implementation

updates 20 dimensions for each time step. Perhaps with additional dimensions, it can perform better without sacrificing too much speed.

A feature that can be implemented is early stopping. In each step, it is easy to detect the accuracy change and stop if that change is too small (since we are calculating the accuracy each epoch already). However, we wanted to see how the parallel algorithms run in their entirety. With early stopping, it is very possible the max iteration for parallel implementations will be much less than 100.

Ultimately, it is difficult to directly compare this method to the existing papers. The existing papers focused on different machine learning methodologies (SVM, MC, and Cuts) whereas this work focused on logistic regression. Furthermore, their approaches were multi-core approaches implemented on different datasets than ours. Nevertheless, the speedup from the lock-free batch shows that there is promise in many-core implementations of SGD.

5. Conclusion

Recall from earlier that our goals were to:

1. demonstrate the strengths and weaknesses of each approach
2. balance the trade-offs between run-time, accuracy, and stability associated with the different approaches
3. highlight the limitations of each approach

Overall, this work explored four modes of parallelization: algorithmic, multiclass, lock-free, and batch. We saw a trade-off in running time and accuracy. Whereas algorithmic and multiclass had good accuracy and poor running time, lock-free and batch had accuracy worse than scikit-learn and far better running time.

Furthermore, the algorithmic approach isn't scalable to higher dimensional data. Since each block was set to (784, 1, 1) in our approach, higher dimensional data will exceed the pycuda limitation on blocksize. The multiclass approach was highly intensive arithmetically on each gpu core. For problems with fewer classes, it might be better to use CPU implementations. Lock-free was severely hampered by accuracy and stability issues. However, lock-free batch is promising because it offers similar accuracy with higher speeds.

We learned that it is important to maintain a moderate arithmetic intensity for each thread. Furthermore, we learned that updates without locking is much less stable (accuracy-wise) than updates with locking.

Further improvements to the capabilities of this work is necessary. This work can adapted to optimize gradient descent over multiple parameter settings. Furthermore, an implementation of adaptive step-size could be effective and useful. Future improvements can potentially transform these initial explorations to production level quality useful for industry level operations.

6. Acknowledgements

Special acknowledgement to the instructors of the E4750 course for providing advice, commentary, and computing resources used to test and refine this work.

7. References

[1] https://github.com/jx2181/parallel-sgd
[2] F. Niu, et al. Hogwild!: A Lock-Free Approach to Parallelizing Stochastic Gradient Descent. Neural Information Processing Systems, 2011.
[3] M. Zinkevich, M. Weimer, A. Smola, L. Li. Parallelized Stochastic Gradient Descent. Neural Information Processing Systems, 2010.
[4] J. Langford, A.J. Smola, and M. Zinkevich. Slow learners are fast. Neural Information Processing Systems, 2009.
[5] C.T. Chu, S.K. Kim, Y. A. Lin, Y. Y. Yu, G. Bradski, A. Ng, and K. Olukotun. Map-reduce for machine learning on multicore. In B. Scholkopf, J. Platt, and T. Hofmann, editors, ¨ Neural Information Processing Systems 19, 2007.
[6] M. Magdon-Ismail. Logistic Regression and Gradient Descent. 2018.

Individual Student Contributions

Task	**% James Xue**
Overall	100%
Paper	100%
Coding	100%
Diagrams	100%
Other	100%