Parallelized t-SNE for dimension reduction of big data

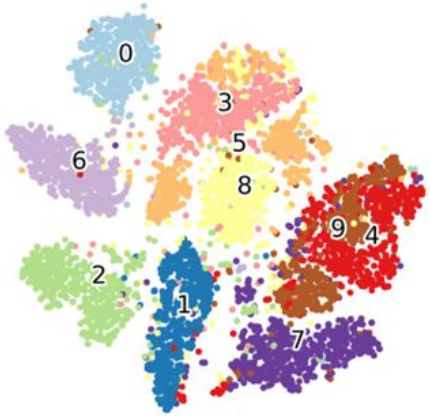
CS205 project final presentation

Eleonora Shantsila, Eric Yang, Daniel Tan, Paul Kramer y Rosado

t-SNE is a popular dimensionality reduction technique

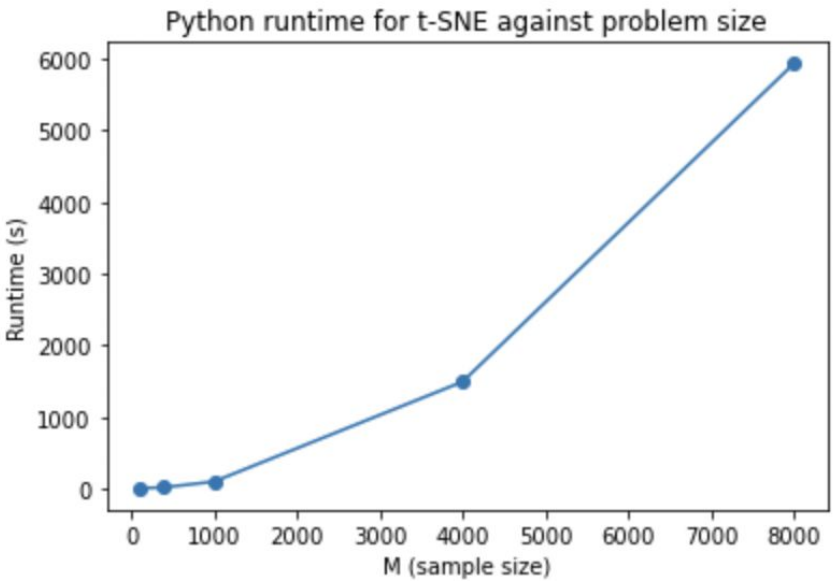
What is t-SNE? :

- “t-distributed stochastic neighbor embedding”
- Commonly used for dimensionality reduction and visualization
- Popular in image processing, computer vision, NLP, speech processing, genomics



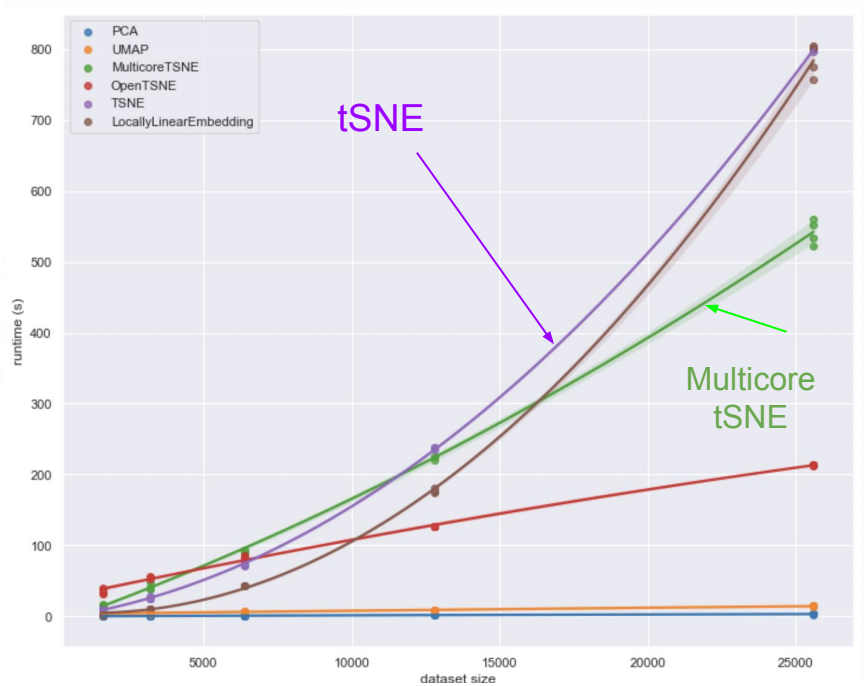

```
tsne (data, no_dims, perplexity) {
        PCA(data);
        calc_pairwise_dist();
        calc_sigmas();
        calc_conditional_probs();
        calc_perplexity_diff();
        calc_stochastic_dist();
        calc_pairwise_dist_embed();
        calc_KL_dist();
        return_2D_embeddings;
}
```

Need for big compute or big data



➔ Python time for MNIST dataset of size **8,000:**
- **tSNE: ~6000s (100min)**

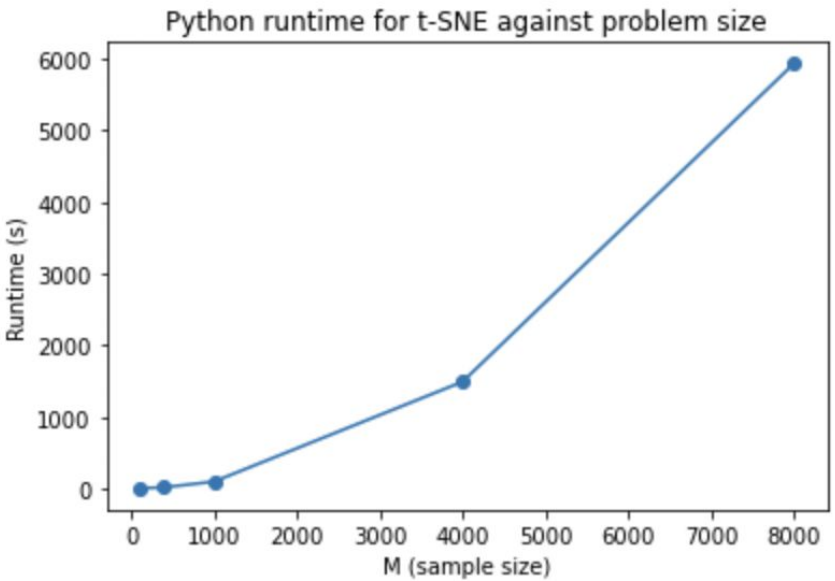
Need for big compute or big data



➔ Python time for MNIST dataset of size **8,000:**
- **tSNE: ~6000s (100min)**

➔ Python multiprocessing code has already been produced. For dataset of size **25,000:**
- **tSNE: 800 min**
- **Multicore tSNE: 550 min**

Need for big compute or big data



➔ Python time for MNIST dataset of size **8,000:**
- **tSNE: ~6000s (100min)**

➔ Python multiprocessing code has already been produced. For dataset of size **25,000:**
- **tSNE: 800 min**
- **Multicore tSNE: 550 min**

➔ Required further optimisation and an implementation in a language more appropriate for parallelisation

➔ No C implementation available (as far as we could find) so we implemented tSNE in C from scratch

C implementation

Two key components
- PCA
- Core t-SNE

X: MxN dataset

K: reduced no. of dimensions after PCA

Reducing to 2 dimensions with t-SNE

PCA

Normalise data X by subtracting column means for each of N columns

Calculate covariance matrix S by multiplying the transpose of the normalised X with X

Perform SVD on matrix S → Implemented through minor adaptations of an implementation found online

Pick K largest eigenvalues and corresponding eigenvectors

Project X to MxN matrix

C implementation

Two key components
- PCA
- Core t-SNE

Working with MxN dataset X

Reducing to K dimensions with PCA

Reducing to 2 dimensions with t-SNE

Core t-SNE

Randomly initialise Y: Mx2 matrix containing the 2D projection

Perform gradient descent to converge to Y

- Use **KL_dist** function to **update the gradient** matrix (Mx2) at each iteration
- Using learning rate alpha **update the Y matrix**
- Repeat until Y matrix converges

C implementation

Two key components
- PCA
- Core t-SNE

Working with MxN dataset X

Reducing to K dimensions with PCA

Reducing to 2 dimensions with t-SNE

Core t-SNE

Randomly initialise Y: Mx2 matrix containing the 2D projection

Perform gradient descent to converge to Y

- Use **KL_dist** function to **update the gradient** matrix (Mx2) at each iteration
- Using learning rate alpha **update the Y matrix**
- Repeat until Y matrix converges

➔ Gradient descent **converged** for **normalised MNIST**
➔ Gradient descent **did not converge** for **our genomics data**

C implementation

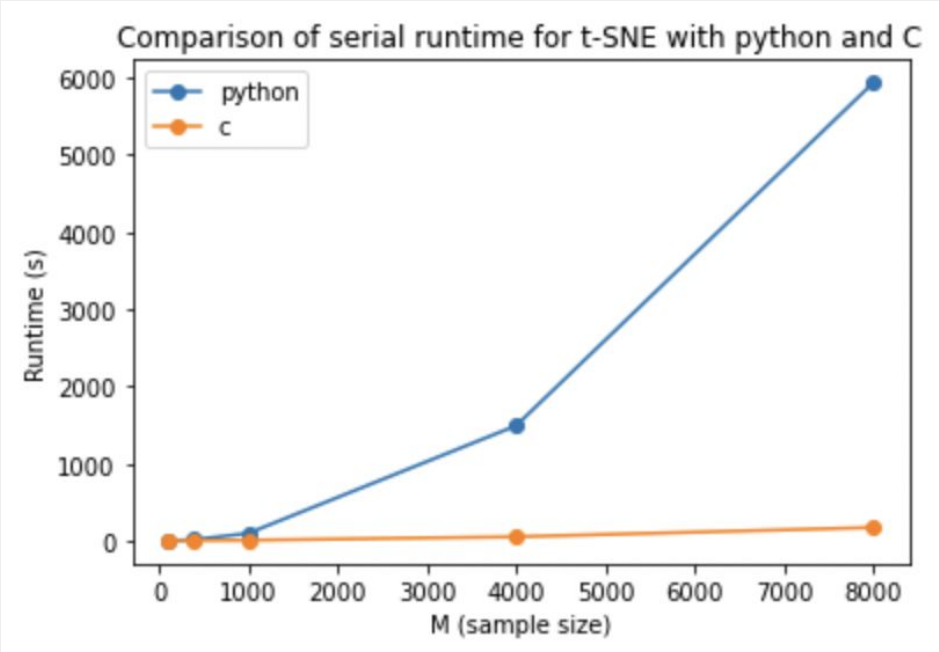
Two key components
- PCA
- Core t-SNE

Working with MxN dataset X

Reducing to K dimensions with PCA

Reducing to 2 dimensions with t-SNE

Comparison with python



Profiling of bottlenecks prior to optimization

Function	Percent Time Taken
PCA - subtract_col_means	0.09%
PCA - Calculating Covariance Matrix	62.21%
PCA - SVD - HouseHolders Reduction to Bidiagonal Form	4.34%
PCA - SVD - Givens Reduction to Bidiagonal Form	1.66%
PCA - SVD -Sort by Decreasing Singular Values	0.008%
tSNE - calc_D	3.69%
tSNE - calc_perplexity_diff	7.91%
tSNE - calc_perplexity_diff while_loop a	0.52%
tSNE - calc_perplexity_diff while_loop b	3.47%
tSNE - calc_perplexity_diff while_loop c	0.018%
tSNE - calc_sigmas total	12.14%
tSNE - calc_pji	1.91%
tSNE - Update P	1.65%

PCA functions take 70.64% of total computation time

Remainder t-SNE functions take 29.36% of total computation time

PCA

Calculating Covariance Matrix takes most of the runtime.

Function	Percent Time Taken
PCA - subtract_col_means	0.09%
PCA - Calculating Covariance Matrix	**62.21%**
PCA - SVD - HouseHolders Reduction to Bidiagonal Form	4.34%
PCA - SVD - Givens Reduction to Bidiagonal Form	1.66%
PCA - SVD -Sort by Decreasing Singular Values	0.008%

Highly parallelizable
Matrix multiplication

Not parallelizable
Data dependencies

```
for (int i=0; i<N; ++i) {
    for (int j=0; j<N; ++j) {
        for (int k=0; k<M; ++k) {
            cov[i][j] += data_t[i][k]*data[k][j];
        }
    }
}
```

PCA - Calculating Covariance Matrix

1) Original code

```
for (int i=0; i<N; ++i) {
    for (int j=0; j<N; ++j) {
        for (int k=0; k<M; ++k) {
            cov[i][j] += data_t[i][k]*data[k][j];
        }
    }
}
```

2) + tmp variable in for loop

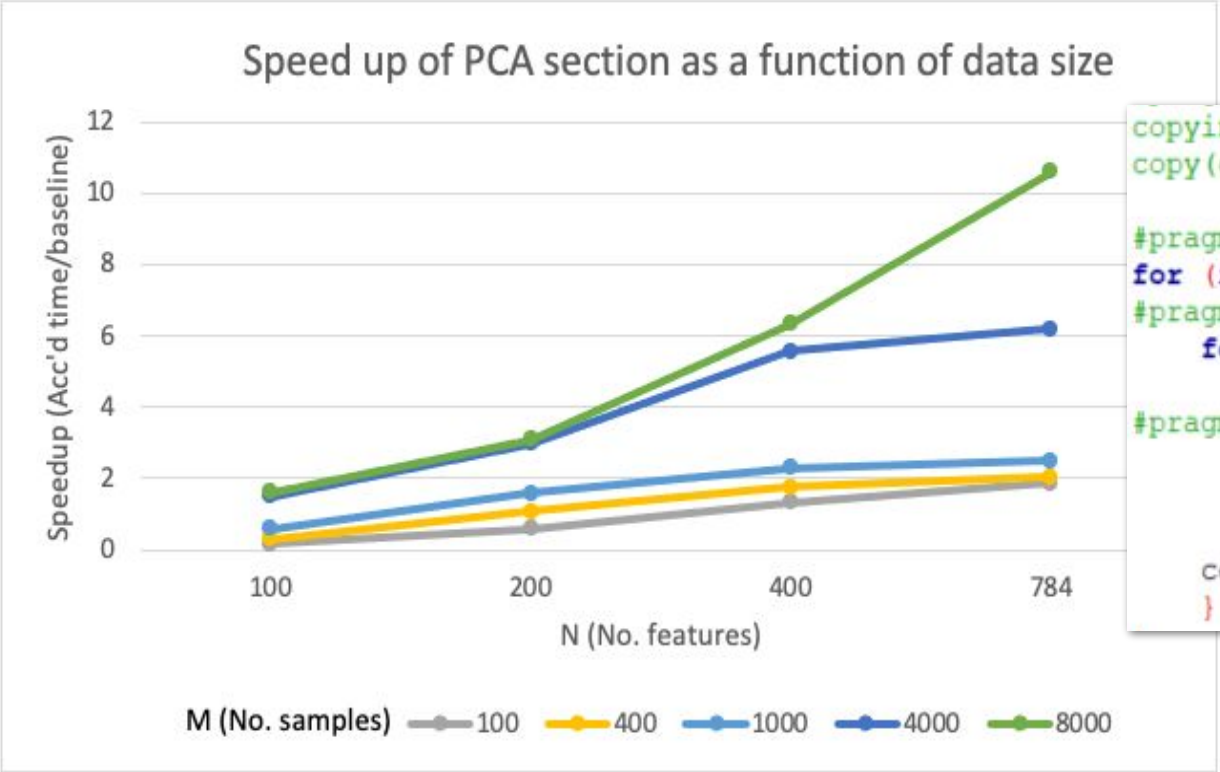
```
for (int i=0; i<N; ++i) {
    for (int j=0; j<N; ++j) {
        double tmp = 0;
        for (int k=0; k<M; ++k) {
            tmp += data_t[i][k]*data[k][j];
        }
    cov[i][j] = tmp;
    }
}
```

3) + OpenACC pragmas

```
copyin(data_t[0:N][0:M], data[0:M][0:N]) \
copy(cov[0:N][0:N])

#pragma acc loop independent vector(32)
for (int i=0; i<N; ++i) {
#pragma acc loop independent
    for (int j=0; j<N; ++j) {
        double tmp = 0;
#pragma acc loop reduction(+:tmp)
        for (int k=0; k<M; ++k) {
            tmp += data_t[i][k]*data[k][j];
        }
    cov[i][j] = tmp;
    }
```

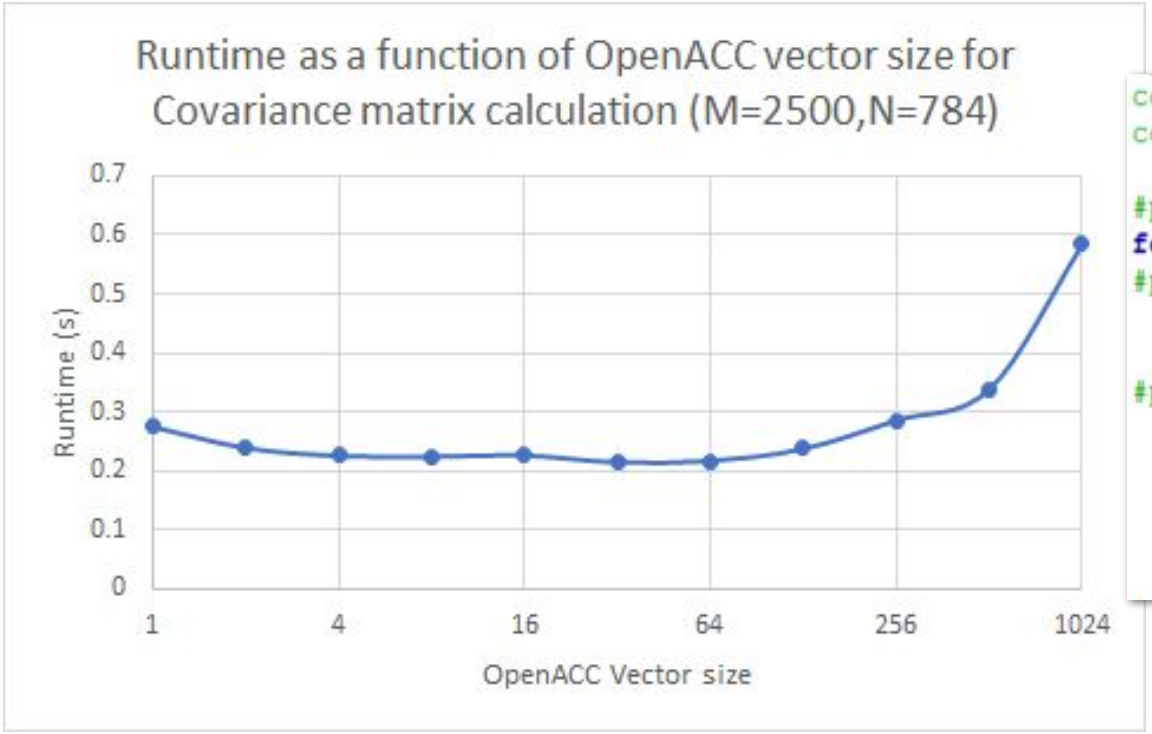
PCA - Parallelization speedup




```
copyin(data_t[0:N][0:M], data[0:M][0:N]) \
copy(cov[0:N][0:N])

#pragma acc loop independent vector(32)
for (int i=0; i<N; ++i) {
#pragma acc loop independent
    for (int j=0; j<N; ++j) {
        double tmp = 0;
#pragma acc loop reduction(+:tmp)
        for (int k=0; k<M; ++k) {
            tmp += data_t[i][k]*data[k][j];
        }
    cov[i][j] = tmp;
    }
```

PCA - Calculating Covariance Matrix




```
copyin(data_t[0:N][0:M], data[0:M][0:N]) \
copy(cov[0:N][0:N])

#pragma acc loop independent vector(32)
for (int i=0; i<N; ++i) {
#pragma acc loop independent
    for (int j=0; j<N; ++j) {
        double tmp = 0;
#pragma acc loop reduction(+:tmp)
        for (int k=0; k<M; ++k) {
            tmp += data_t[i][k]*data[k][j];
        }
    cov[i][j] = tmp;
    }
```

t-SNE acceleration using OpenACC

for each row of matrix D:

initialise a and b

while perplexity differences greater than zero:

calculate perplexity differences for sigma equal to a

a=a/2

while perplexity differences less than zero:

calculate perplexity differences for sigma equal to a

b=2b

update sigma[i] with min x of perplexity difference function

using the bisection method and a, b

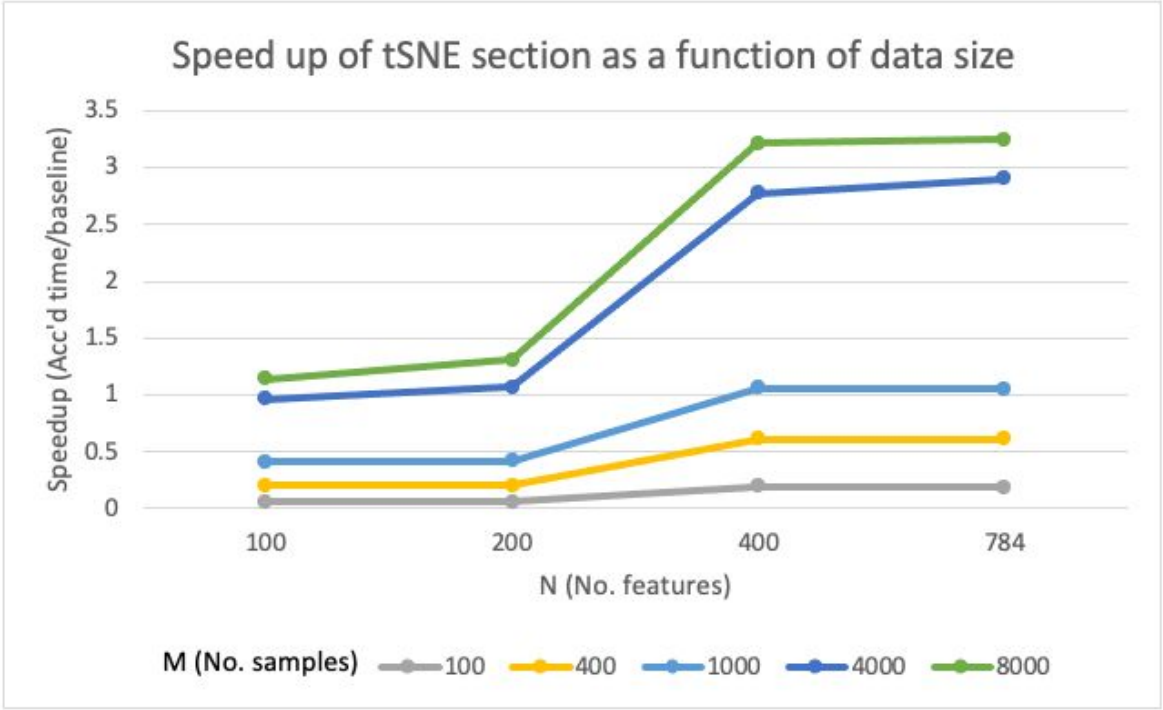
```
float calc_perplexity_diff(double sigmai, double target_perplexity, int d1, double Di[d1])
    // d1=main
    // Di is row i of D of size M
    double Z=0;
    double denom=0;
    double pji_var=0;
    double H=0;
    double diff=0;

    denom=2*pow(sigmai,2);
    #pragma acc parallel vector_length(32) loop reduction(+:Z)
    for (int i=0; i<d1; i++){
        Z+=exp(-pow(Di[i],2)/denom);
    }
    #pragma acc parallel
    for (int j=0; j<d1; j++){
        pji_var=exp(-pow(Di[j],2)/denom)/Z;
        if (pji_var>0) {
            H-=pji_var*log2(pji_var);
        }
    }
    diff=pow(2,H)-target_perplexity;
    return diff;
}
```

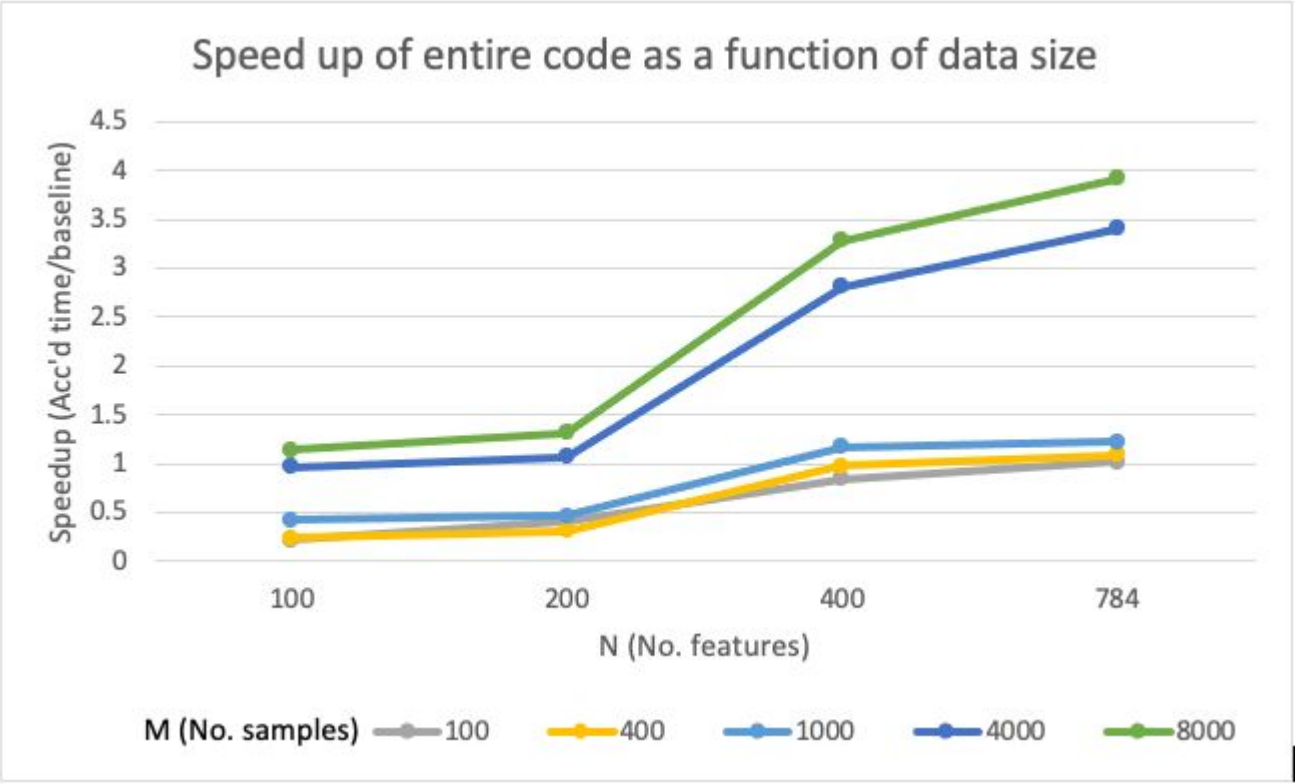
t-SNE acceleration using OpenACC

```
void calc_Q(int d1, double Y[][2], double Q[][d1]){
    double Z=0.0;
    double norm=0.0;
    #pragma acc parallel
    #pragma acc data copyin(Y[:d1][:]) copyout(Q[:d1][:])
    for (int i=0; i<d1; i++) {
        #pragma acc loop reduction(+:Z)
        for (int j=0; j<i; j++) {
            norm=0.0;
            for (int k=0; k<2; k++){
                norm+=pow(Y[i][k]-Y[j][k],2);}
            Q[i][j]=1.0/(1.0+norm);
            Q[j][i]=Q[i][j];
            Z+=2*Q[i][j];
        }
        }
    for (int i=0; i<d1; i++) {
        for (int j=0; j<d1; j++) {
            Q[i][j]=Q[i][j]/Z;
        }
    }
}
```

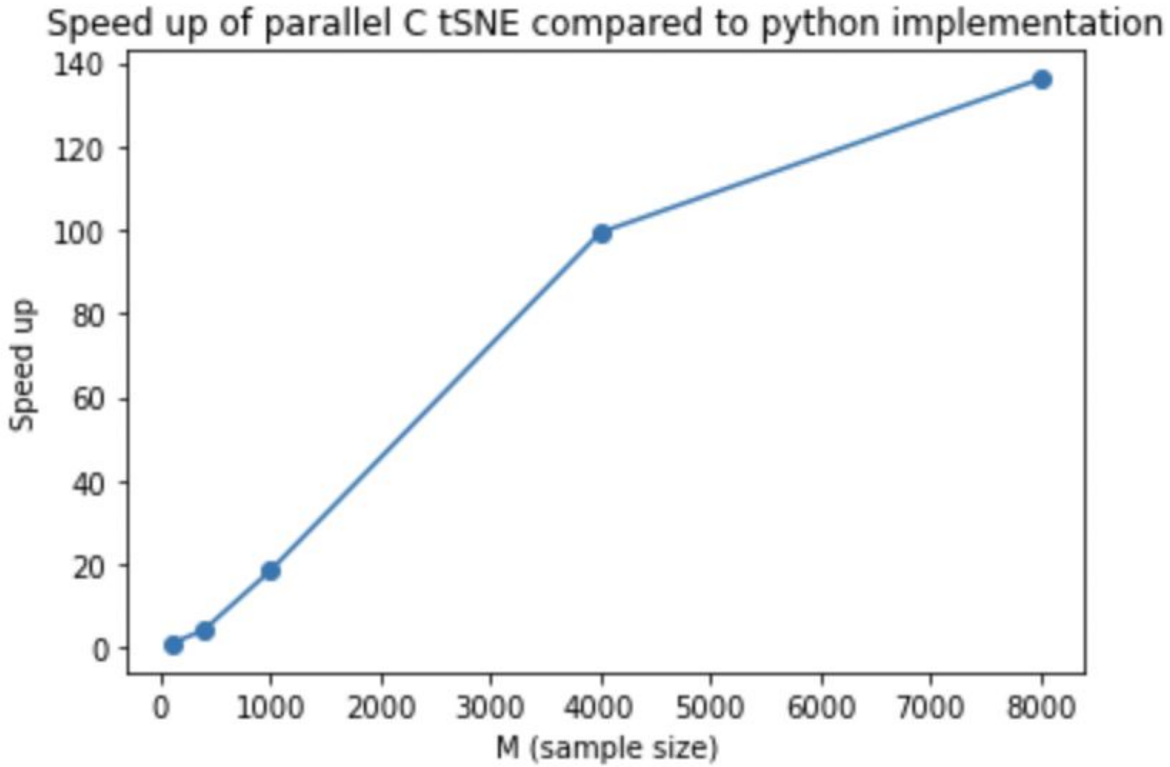
t-SNE parallelisation



Entire code parallelisation



Plot comparing our time with python time



In Summary

- Conclusion
  - Implemented a complicated t-SNE algorithm in C (none existed previously) from scratch
  - Achieved desired speedup that validated design intentions
- Next steps
  - Implement more complex optimization algorithms to converge more complex, heterogeneous datasets
  - Explore other levels of parallelization techniques