

# CMPE-220 Homework

Sravan Kumar Gorati

SJSU ID:017441749

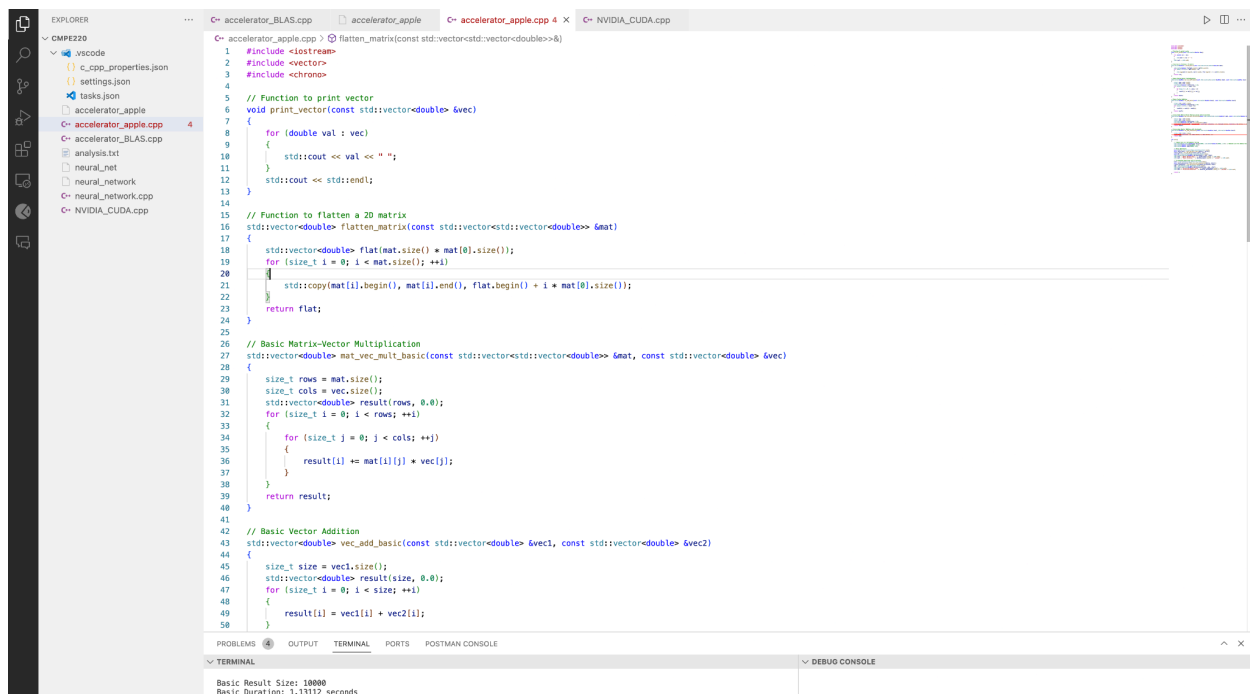
Colab Link: [CMPE220-Sravan.ipynb](#)

Github Link: <https://github.com/sravangorati2001/CMPE220-Homework>

## Performance Analysis of Neural Network Operations in C/C++ and Python

Step1:

Basic Neural Network operation in C/C++:



The screenshot shows a C++ code editor with a file explorer on the left and a code editor on the right. The file explorer shows a project named 'CMPE220' with files like 'c\_cpp\_properties.json', 'settings.json', 'tasks.json', 'accelerator\_apple', 'accelerator\_BLAS.cpp', 'analysis.txt', 'neural\_net', 'neural\_network.cpp', and 'NVIDIA\_CUDA.cpp'. The code editor shows the implementation of a basic neural network operation in C++. The code includes headers for `<iostream>`, `<vector>`, and `<chrono>`. It defines a function `print_vector` to print a vector of doubles. It also defines a function `flatten_matrix` to flatten a 2D matrix into a 1D vector. The main function `mat_vec_mult_basic` performs a basic matrix-vector multiplication. It takes a matrix `mat` and a vector `vec` as input and returns a result vector. The code uses nested loops to calculate the result. The output of the program is shown in the terminal at the bottom, displaying 'Basic Result Size: 10000' and 'Basic Duration: 1.13112 seconds'.

```
1 #include <iostream>
2 #include <vector>
3 #include <chrono>
4
5 // Function to print vector
6 void print_vector(const std::vector<double> &vec)
7 {
8     for (double val : vec)
9     {
10         std::cout << val << " ";
11     }
12     std::cout << std::endl;
13 }
14
15 // Function to flatten a 2D matrix
16 std::vector<double> flatten_matrix(const std::vector<std::vector<double>> &mat)
17 {
18     std::vector<double> flat(mat.size() * mat[0].size());
19     for (size_t i = 0; i < mat.size(); ++i)
20     {
21         std::copy(mat[i].begin(), mat[i].end(), flat.begin() + i * mat[0].size());
22     }
23     return flat;
24 }
25
26 // Basic Matrix-Vector Multiplication
27 std::vector<double> mat_vec_mult_basic(const std::vector<std::vector<double>> &mat, const std::vector<double> &vec)
28 {
29     size_t rows = mat.size();
30     size_t cols = vec.size();
31     std::vector<double> result(rows, 0.0);
32     for (size_t i = 0; i < rows; ++i)
33     {
34         for (size_t j = 0; j < cols; ++j)
35         {
36             result[i] += mat[i][j] * vec[j];
37         }
38     }
39     return result;
40 }
41
42 // Basic Vector Addition
43 std::vector<double> vec_add_basic(const std::vector<double> &vec1, const std::vector<double> &vec2)
44 {
45     size_t size = vec1.size();
46     std::vector<double> result(size, 0.0);
47     for (size_t i = 0; i < size; ++i)
48     {
49         result[i] = vec1[i] + vec2[i];
50     }
51 }
```

Basic Result Size: 10000  
Basic Duration: 1.13112 seconds

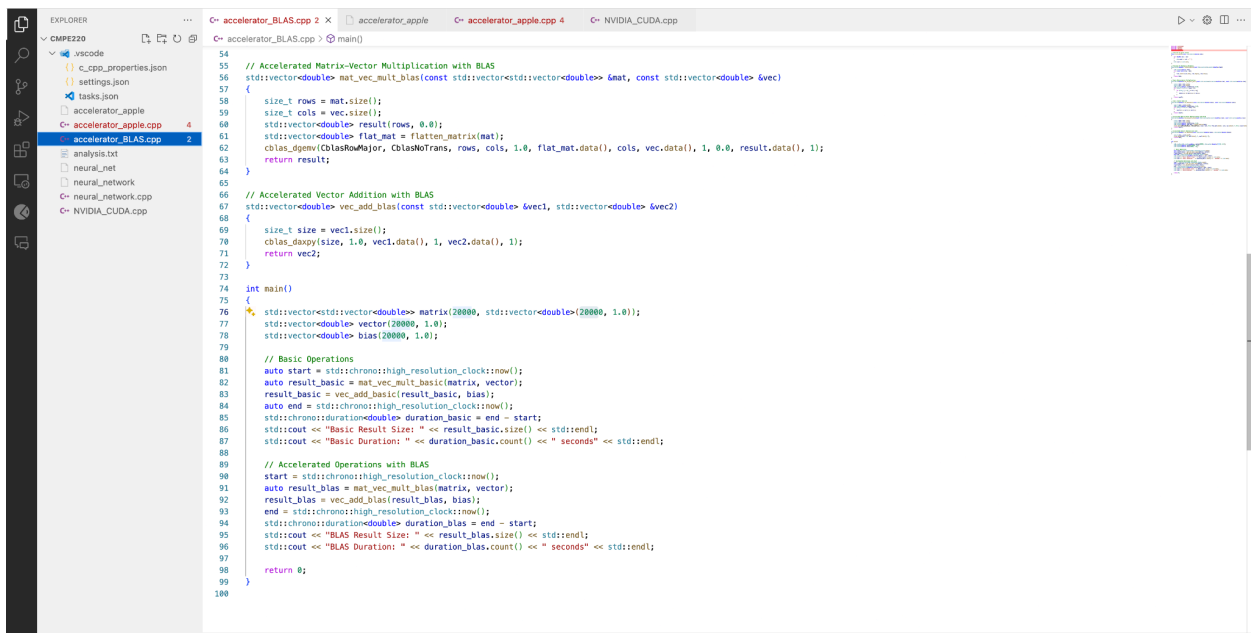
Sample Output for size: 10000

Basic Result Size: 10000  
Basic Duration: 1.13112 seconds

Output for Different Inputs:

Input	Basic
10000	1.0021
20000	1.70555
30000	5.73627
40000	9.89812
45000	12.3204

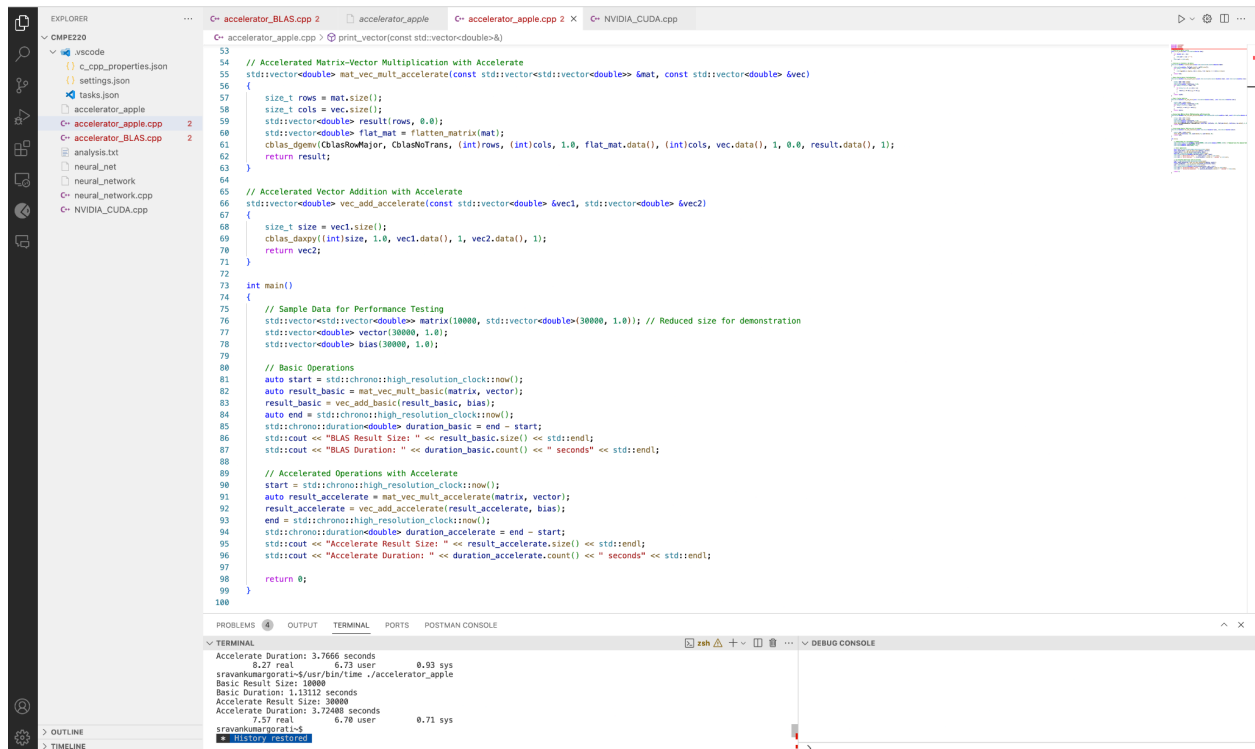
Acceleration using BLAS (Basic Linear Algebra Subprograms):



Output for Different Inputs:

Input	BLAS
10000	2.9876
20000	4.7325
30000	14.387
40000	24.0987
45000	33.1647

## Acceleration using Apple's accelerate:



```
53 // Print vector (const std::vector<double>& v)
54
55 // Accelerated Matrix-Vector Multiplication with Accelerate
56 std::vector<double> mat_vec_mult_accelerate(const std::vector<double>& mat, const std::vector<double>& vec)
57 {
58     size_t rows = mat.size();
59     size_t cols = vec.size();
60     std::vector<double> result(rows, 0.0);
61     std::vector<double> flat_mat = flatten_matrix(mat);
62     cblas_dgemv(CblasRowMajor, CblasNoTrans, (int)rows, (int)cols, 1.0, flat_mat.data(), (int)cols, vec.data(), 1, 0.0, result.data(), 1);
63     return result;
64 }
65
66 // Accelerated Vector Addition with Accelerate
67 std::vector<double> vec_add_accelerate(const std::vector<double>& vec1, std::vector<double>& vec2)
68 {
69     size_t size = vec1.size();
70     cblas_daxpy((int)size, 1.0, vec1.data(), 1, vec2.data(), 1);
71     return vec2;
72 }
73
74 int main()
75 {
76     // Sample Data for Performance Testing
77     std::vector<double> matrix(10000, std::vector<double>(30000, 1.0)); // Reduced size for demonstration
78     std::vector<double> vector(30000, 1.0);
79     std::vector<double> bias(30000, 1.0);
80
81     // Basic Operations
82     auto start = std::chrono::high_resolution_clock::now();
83     auto result_basic = mat_vec_mult_basic(matrix, vector);
84     result_basic = vec_add_basic(result_basic, bias);
85     auto end = std::chrono::high_resolution_clock::now();
86     std::chrono::duration<double> duration_basic = end - start;
87     std::cout << "BLAS Result Size: " << result_basic.size() << std::endl;
88     std::cout << "BLAS Duration: " << duration_basic.count() << " seconds" << std::endl;
89
90     // Accelerated Operations with Accelerate
91     start = std::chrono::high_resolution_clock::now();
92     auto result_accelerate = mat_vec_mult_accelerate(matrix, vector);
93     result_accelerate = vec_add_accelerate(result_accelerate, bias);
94     end = std::chrono::high_resolution_clock::now();
95     std::chrono::duration<double> duration_accelerate = end - start;
96     std::cout << "Accelerate Result Size: " << result_accelerate.size() << std::endl;
97     std::cout << "Accelerate Duration: " << duration_accelerate.count() << " seconds" << std::endl;
98
99     return 0;
100 }
```

Terminal Output:

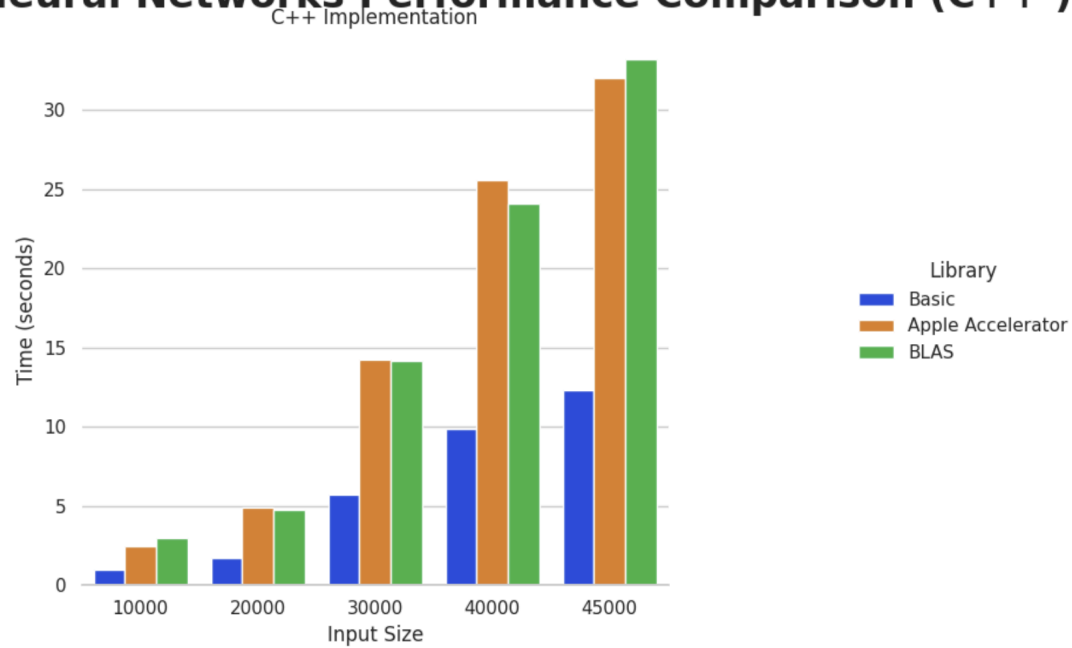
```
Accelerate Duration: 3.7666 seconds
0.27 real    6.73 user    0.93 sys
sravanakumar@prl:~/bin/line ~/accelerator_apple
Basic Result Size: 10000
Basic Duration: 1.1312 seconds
Accelerate Result Size: 30000
Accelerate Duration: 3.7248 seconds
7.57 real    6.78 user    0.71 sys
sravanakumar@prl:~/bin/line ~/accelerator_apple
```

## Output for Different Inputs:

Input	Apple Accelerator
10000	2.4378
20000	4.92506
30000	14.2296
40000	25.5829
45000	32.0233

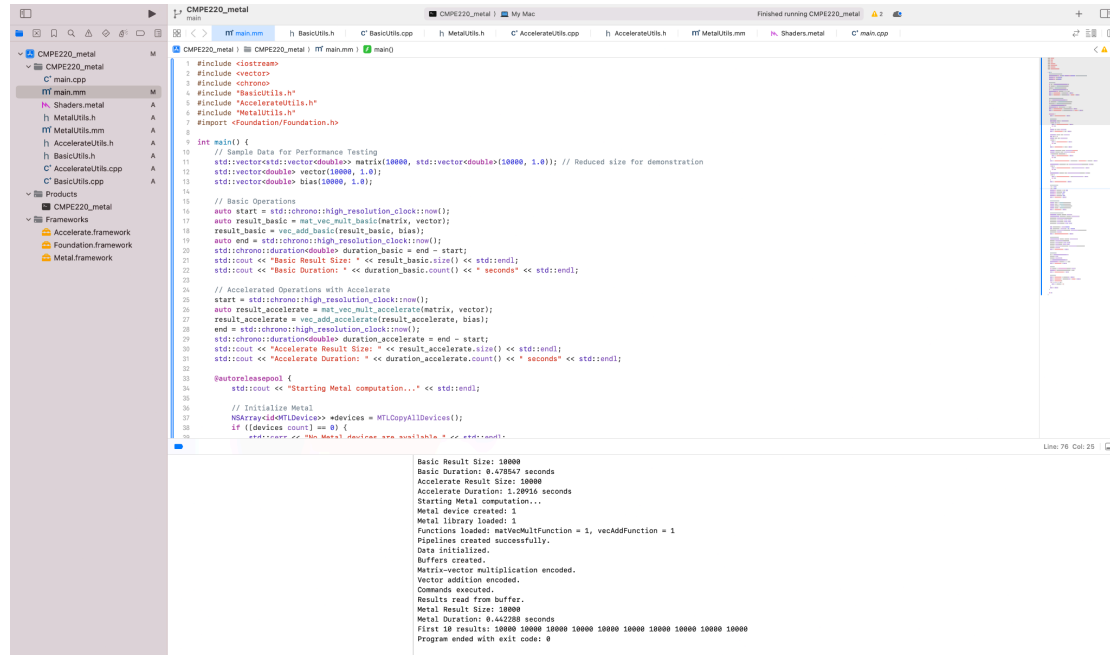
Performance Analysis of Basic, BLAS, Apple's Accelerator:

Basic Neural Networks Performance Comparison (C++ )



## Step2:

### Neural network operation using Apple's Metal Framework:



```
1 #include <iostream>
2 #include <vector>
3 #include <chrono>
4 #include "BasicUtils.h"
5 #include "AccelerateUtils.h"
6 #include "MetalUtils.h"
7 #import <Foundation/Foundation.h>
8
9 int main() {
10     // Sample Data for Performance Testing
11     std::vector<std::vector<double>>> matrix(10000, std::vector<double>(10000, 1.0)); // Reduced size for demonstration
12     std::vector<double> vector(10000, 1.0);
13     std::vector<double> bias(10000, 1.0);
14
15     // Basic Operations
16     auto start = std::chrono::high_resolution_clock::now();
17     auto result_basic = mat_vec_mult_basic(matrix, vector);
18     result_basic = vec_add_bias(result_basic, bias);
19     auto end = std::chrono::high_resolution_clock::now();
20     std::chrono::duration<double> duration_basic = end - start;
21     std::cout << "Basic Result Size: " << result_basic.size() << std::endl;
22     std::cout << "Basic Duration: " << duration_basic.count() << " seconds" << std::endl;
23
24     // Accelerated Operations with Accelerate
25     start = std::chrono::high_resolution_clock::now();
26     auto result_accelerate = mat_vec_mult_accelerated(matrix, vector);
27     result_accelerate = vec_add_accelerated(result_accelerate, bias);
28     end = std::chrono::high_resolution_clock::now();
29     std::chrono::duration<double> duration_accelerate = end - start;
30     std::cout << "Accelerate Result Size: " << result_accelerate.size() << std::endl;
31     std::cout << "Accelerate Duration: " << duration_accelerate.count() << " seconds" << std::endl;
32
33     @autoreleasepool {
34         std::cout << "Starting Metal computation..." << std::endl;
35
36         // Initialize Metal
37         MTRuntimeDevice* devices = MTRuntimeDeviceCopyAllDevices();
38         if ([devices count] == 0) {
39             std::cout << "No Metal hardware was available. Try again!" << std::endl;
40         }
41     }
```

Basic Result Size: 10000  
Basic Duration: 0.478547 seconds  
Accelerate Result Size: 10000  
Accelerate Duration: 1.20916 seconds  
Starting Metal computation...  
Metal device created: 1  
Metal library loaded: 1  
Functions loaded: matVecMultFunction = 1, vecAddFunction = 1  
Pipelines created successfully.  
Data initialized.  
Buffers created.  
Matrix-vector multiplication encoded.  
Vector addition encoded.  
Commands executed.  
Results read from buffer.  
Metal Result Size: 10000  
Metal Duration: 0.442288 seconds  
First 10 results: 10000 10000 10000 10000 10000 10000 10000 10000 10000 10000  
Program ended with exit code: 0

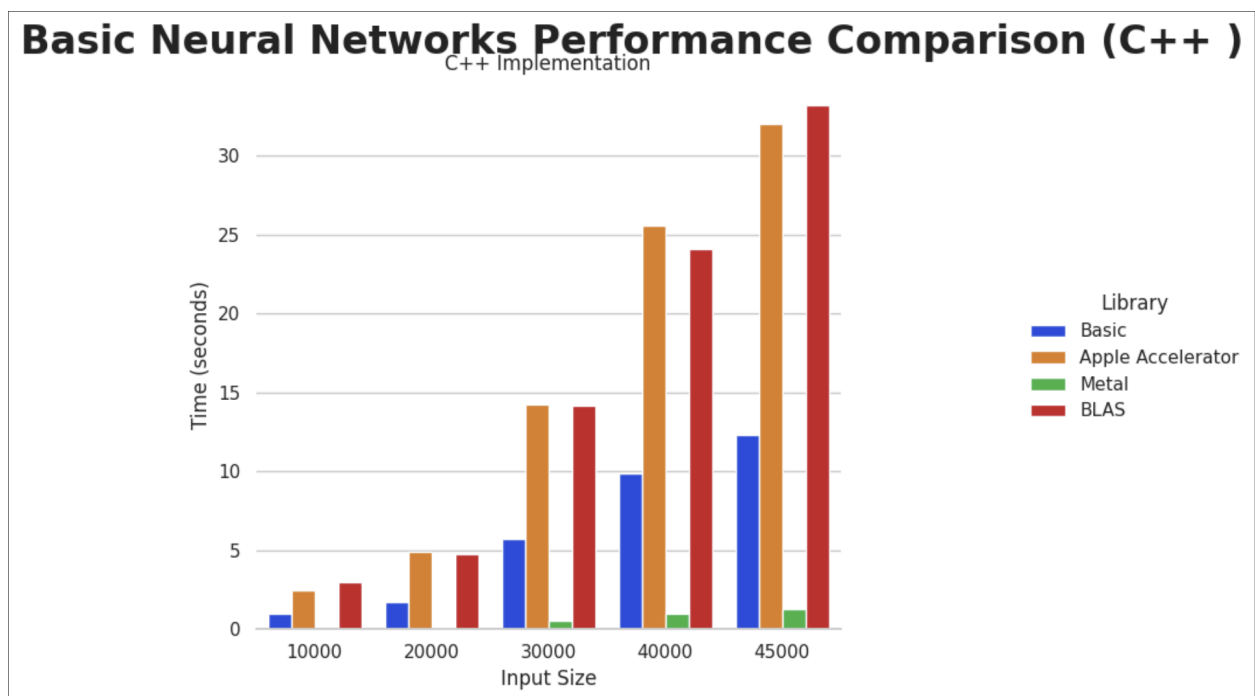
### Sample output:

```
Basic Result Size: 10000
Basic Duration: 0.478547 seconds
Accelerate Result Size: 10000
Accelerate Duration: 1.20916 seconds
Starting Metal computation...
Metal device created: 1
Metal library loaded: 1
Functions loaded: matVecMultFunction = 1, vecAddFunction = 1
Pipelines created successfully.
Data initialized.
Buffers created.
Matrix-vector multiplication encoded.
Vector addition encoded.
Commands executed.
Results read from buffer.
Metal Result Size: 10000
Metal Duration: 0.442288 seconds
```

Output for different inputs:

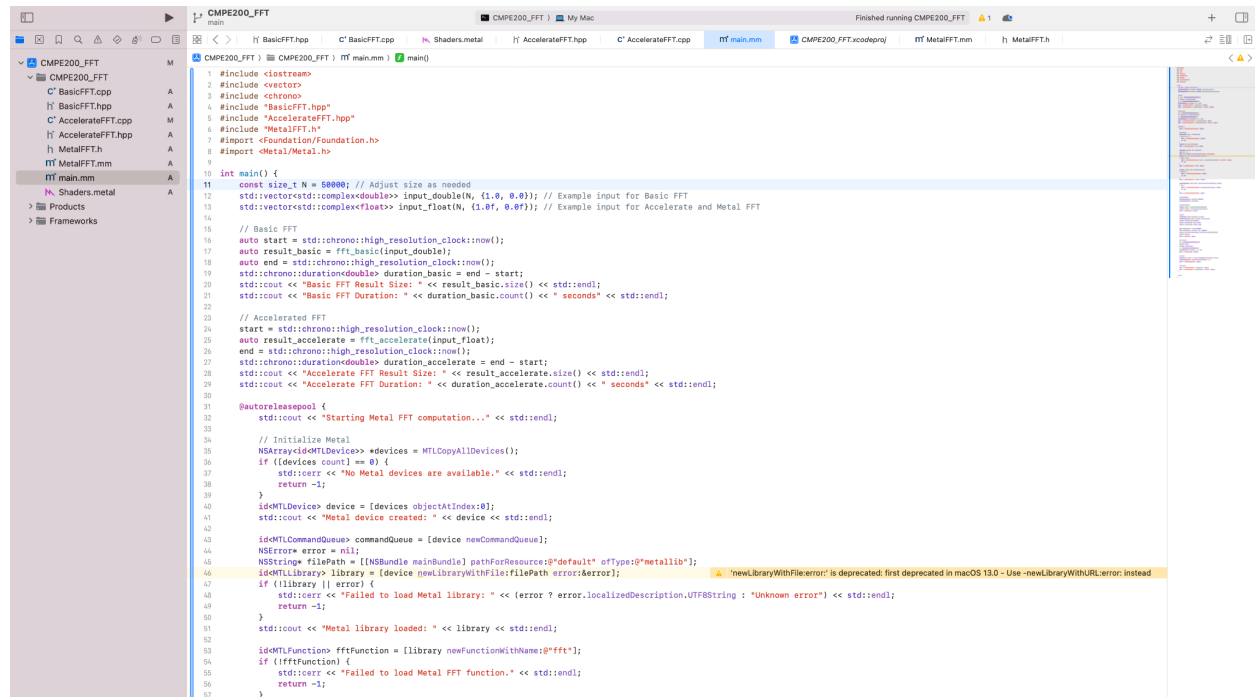
Input	Metal
10000	0.01965
20000	0.108957
30000	0.549007
40000	0.973443
45000	1.26005

Performance Analysis of Basic, BLAS, Apple's Accelerator, Metal:



### Step3:

## FFT (Fast Fourier Transform) Performance in C++:



```
1 #include <iostream>
2 #include <vector>
3 #include <chrono>
4 #include "BasicFFT.hpp"
5 #include "AccelerateFFT.hpp"
6 #include "MetalFFT.h"
7 #import <Foundation/Foundation.h>
8 #import <Metal/Metal.h>
9
10 int main() {
11     const size_t N = 50000; // Adjust size as needed
12     std::vector<std::complex<double>> input_double(N, {1.0, 0.0}); // Example input for Basic FFT
13     std::vector<std::complex<float>> input_float(N, {1.0f, 0.0f}); // Example input for Accelerate and Metal FFT
14
15     // Basic FFT
16     auto start = std::chrono::high_resolution_clock::now();
17     auto result_basic = fft_basic(input_double);
18     auto end = std::chrono::high_resolution_clock::now();
19     std::chrono::duration<double> duration_basic = end - start;
20     std::cout << "Basic FFT Result Size: " << result_basic.size() << std::endl;
21     std::cout << "Basic FFT Duration: " << duration_basic.count() << " seconds" << std::endl;
22
23     // Accelerated FFT
24     start = std::chrono::high_resolution_clock::now();
25     auto result_accelerate = fft_accelerate(input_float);
26     end = std::chrono::high_resolution_clock::now();
27     std::chrono::duration<double> duration_accelerate = end - start;
28     std::cout << "Accelerate FFT Result Size: " << result_accelerate.size() << std::endl;
29     std::cout << "Accelerate FFT Duration: " << duration_accelerate.count() << " seconds" << std::endl;
30
31     @autoreleasepool {
32         std::cout << "Starting Metal FFT computation..." << std::endl;
33
34         // Initialize Metal
35         NSArray<id<MTLDevice>> *devices = MTLCopyAllDevices();
36         if ([devices count] == 0) {
37             std::cerr << "No Metal devices are available." << std::endl;
38             return -1;
39         }
40         id<MTLDevice> device = [devices objectAtIndex:0];
41         std::cout << "Metal device created: " << device << std::endl;
42
43         id<MTLCommandQueue> commandQueue = [device newCommandQueue];
44         NSError *error = nil;
45         NSString *filePath = [[NSBundle mainBundle] pathForResource:@"default" ofType:@"metallib"];
46         id<MTLLibrary> library = [device newLibraryWithFile:filePath error:&error];
47         if ([library == nil]) {
48             std::cerr << "Failed to load Metal library: " << [error ? error.localizedDescription.UTF8String : "Unknown error"] << std::endl;
49             return -1;
50         }
51         std::cout << "Metal library loaded: " << library << std::endl;
52
53         id<MTLFunction> fftFunction = [library newFunctionWithName:@"fft"];
54         if (!fftFunction) {
55             std::cerr << "Failed to load Metal FFT function." << std::endl;
56             return -1;
57         }
58     }
```

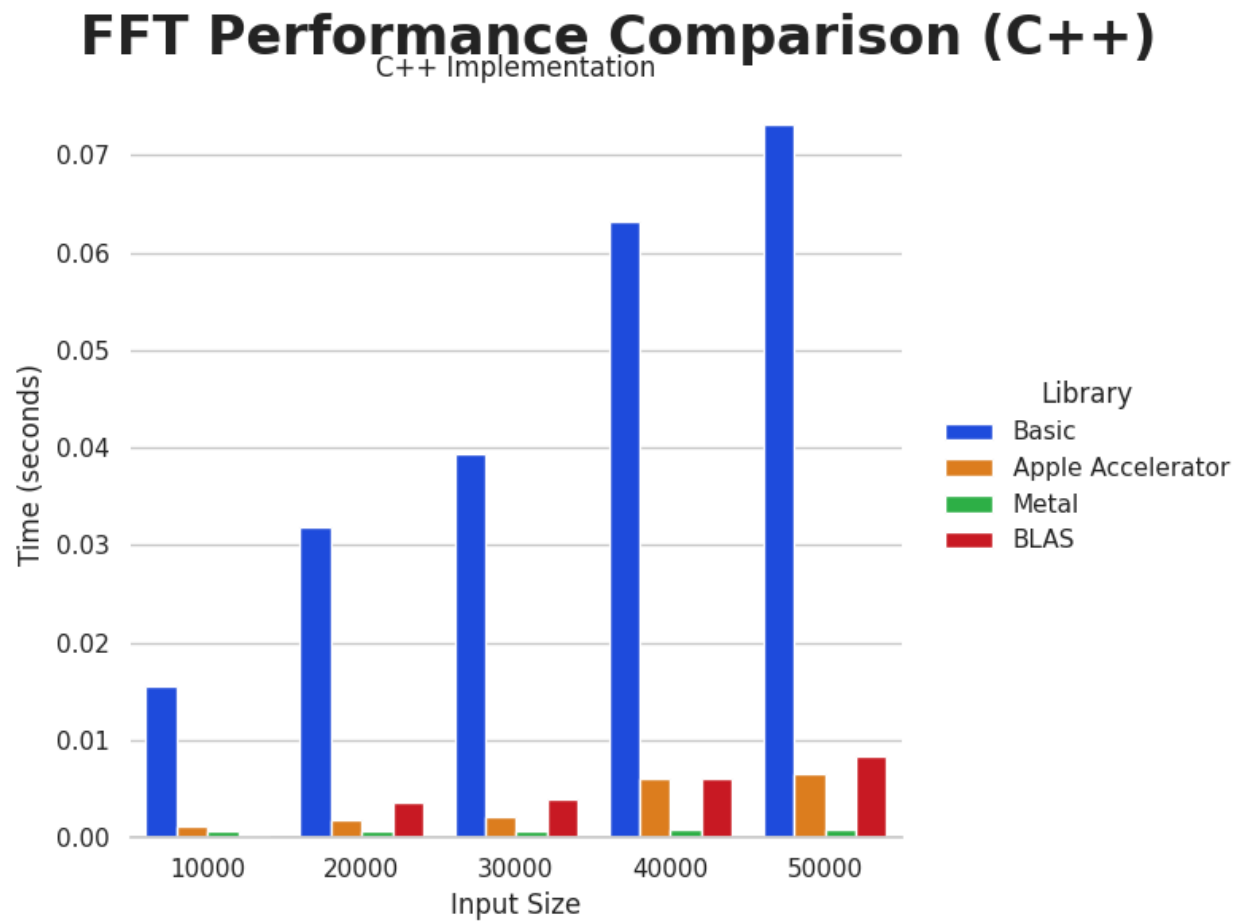
### Sample output for input 10000:

```
Basic FFT Result Size: 10000
Basic FFT Duration: 0.0160458 seconds
Accelerate FFT Result Size: 10000
Accelerate FFT Duration: 0.00114825 seconds
Starting Metal FFT computation...
Metal device created: 1
Metal library loaded: 1
FFT Function loaded: 1
FFT Pipeline created successfully.
Buffers created.
FFT encoded.
Commands executed.
Results read from buffer.
Metal FFT Result Size: 10000
Metal FFT Duration: 0.000606125 seconds
```

Output for Different Inputs:

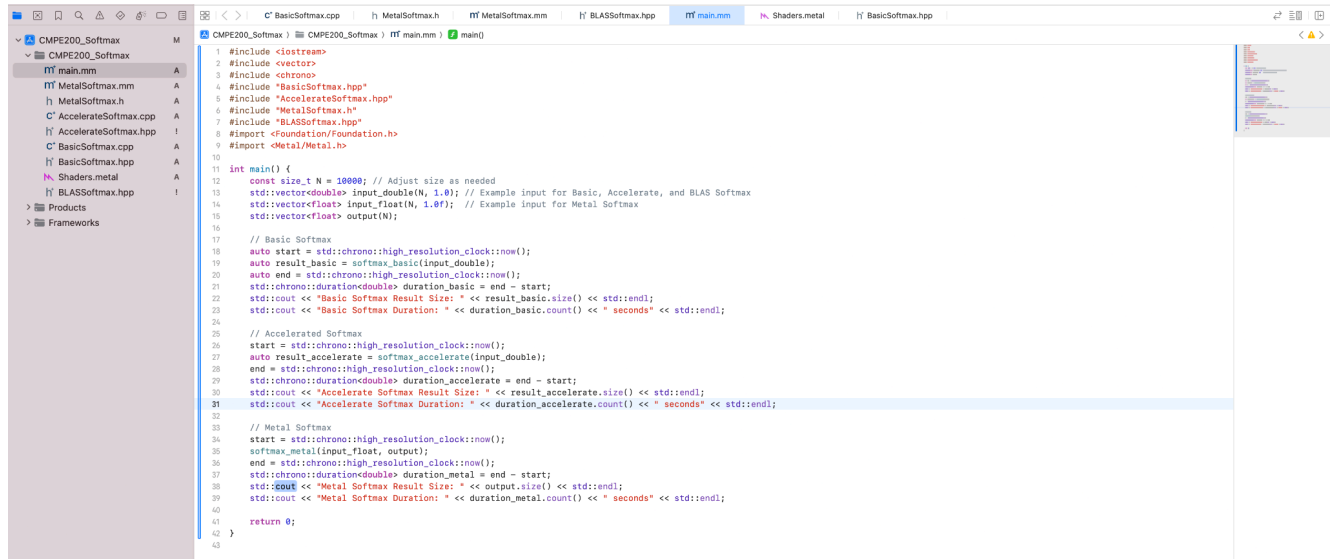
FFT					
Input	Metal	Apple Accelerator	BLAS	Basic	
10000	0.000716042	0.00112067	0.0015234	0.0156052	
20000	0.0007319929	0.00191117	0.0359372	0.0319372	
30000	0.000757792	0.00219592	0.03956	0.039306	
40000	0.000790084	0.006104	0.06145	0.063168	
50000	0.000813459	0.00659729	0.08354	0.0730943	

Performance Analysis of FFT in C++:





## Softmax Implementation in C++:



```
1 #include <iostream>
2 #include <vector>
3 #include <chrono>
4 #include "BasicSoftmax.hpp"
5 #include "AcceleratedSoftmax.hpp"
6 #include "MetalSoftmax.h"
7 #include "BLASSoftmax.hpp"
8 #import <Foundation/Foundation.h>
9 #import <Metal/Metal.h>
10
11 int main() {
12     const size_t N = 10000; // Adjust size as needed
13     std::vector<double> input_double(N, 1.0); // Example input for Basic, Accelerate, and BLAS Softmax
14     std::vector<float> input_float(N, 1.0f); // Example input for Metal Softmax
15     std::vector<float> output(N);
16
17     // Basic Softmax
18     auto start = std::chrono::high_resolution_clock::now();
19     auto result_basic = softmax_basic(input_double);
20     auto end = std::chrono::high_resolution_clock::now();
21     std::chrono::duration<double> duration_basic = end - start;
22     std::cout << "Basic Softmax Result Size: " << result_basic.size() << std::endl;
23     std::cout << "Basic Softmax Duration: " << duration_basic.count() << " seconds" << std::endl;
24
25     // Accelerated Softmax
26     start = std::chrono::high_resolution_clock::now();
27     auto result_accelerate = softmax_accelerate(input_double);
28     end = std::chrono::high_resolution_clock::now();
29     std::chrono::duration<double> duration_accelerate = end - start;
30     std::cout << "Accelerate Softmax Result Size: " << result_accelerate.size() << std::endl;
31     std::cout << "Accelerate Softmax Duration: " << duration_accelerate.count() << " seconds" << std::endl;
32
33     // Metal Softmax
34     start = std::chrono::high_resolution_clock::now();
35     softmax_metal(input_float, output);
36     end = std::chrono::high_resolution_clock::now();
37     std::chrono::duration<double> duration_metal = end - start;
38     std::cout << "Metal Softmax Result Size: " << output.size() << std::endl;
39     std::cout << "Metal Softmax Duration: " << duration_metal.count() << " seconds" << std::endl;
40
41     return 0;
42 }
43
```

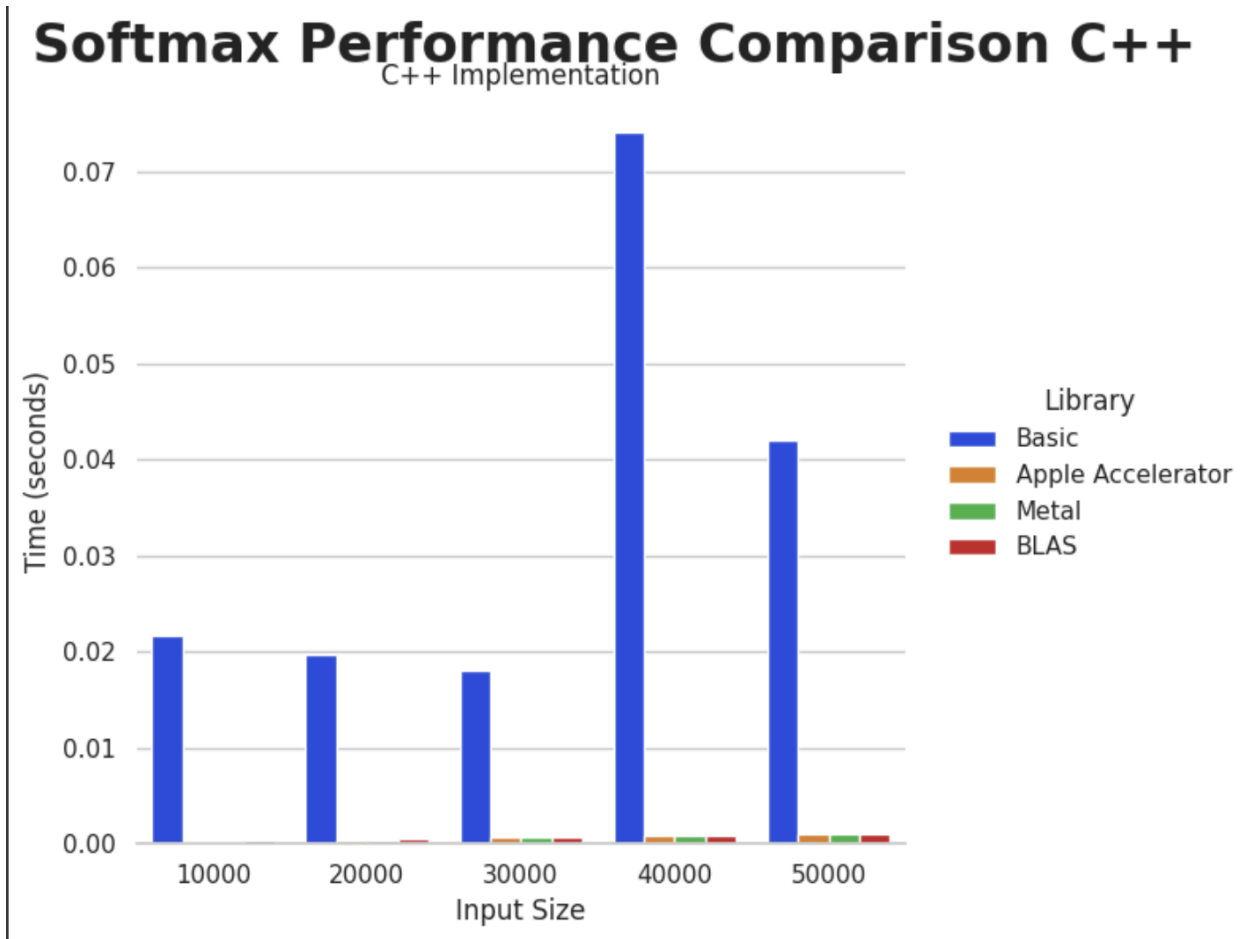
### Sample output for input 10000:

```
Basic Softmax Result Size: 10000
Basic Softmax Duration: 0.0002065 seconds
Accelerate Softmax Result Size: 10000
Accelerate Softmax Duration: 0.000217084 seconds
Metal Softmax Result Size: 10000
Metal Softmax Duration: 0.0182341 seconds
Program ended with exit code: 0
```

### Output for Different Inputs:

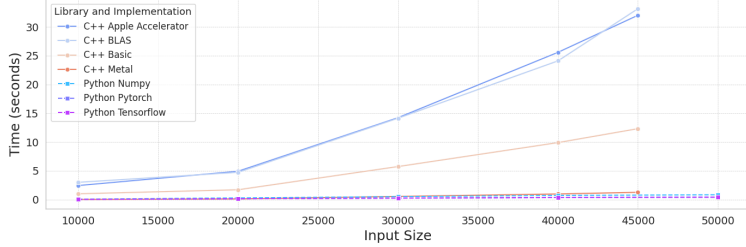
Softmax					
Input	Metal	Apple Accelerator	BLAS	Basic	
10000	0.000201667	0.000254	0.000314	0.0217576	
2000	0.000414084	0.000392458	0.000352458	0.019743	
3000	0.000624542	0.000621583	0.00071583	0.0180205	
4000	0.000825709	0.000878542	0.000867542	0.074053	
5000	0.00103075	0.00109221	0.00199521	0.0421108	

Performance Analysis of Softmax in C++:

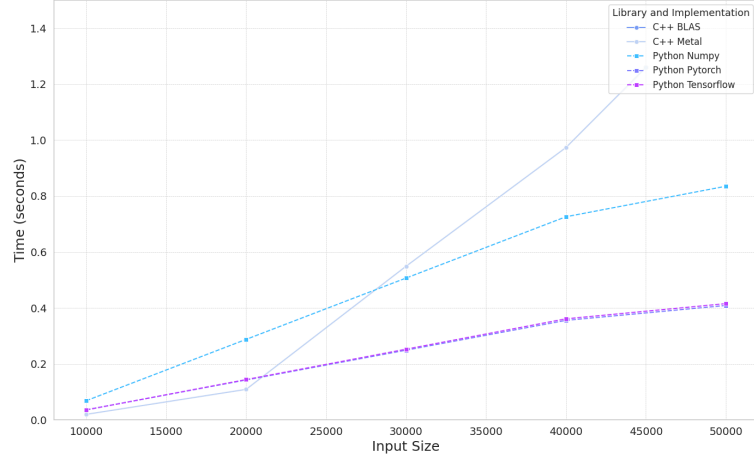


# C++ vs Python Performance Analysis

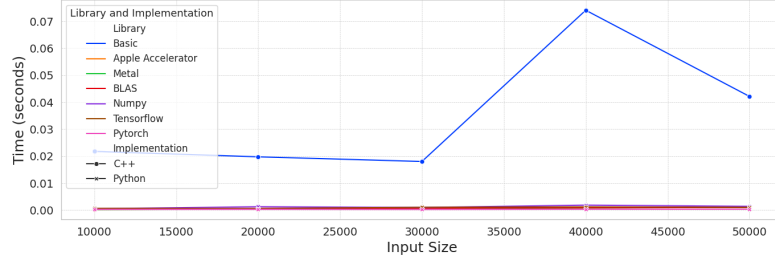
## Basic Neural Networks Performance Comparison (C++ vs Python) - Full Range



## Basic Neural Networks Performance Comparison (C++ vs Python) - Zoomed In



## Softmax Performance Comparison (C++ vs Python) - Full Range



## Softmax Performance Comparison (C++ vs Python) - Zoomed In

