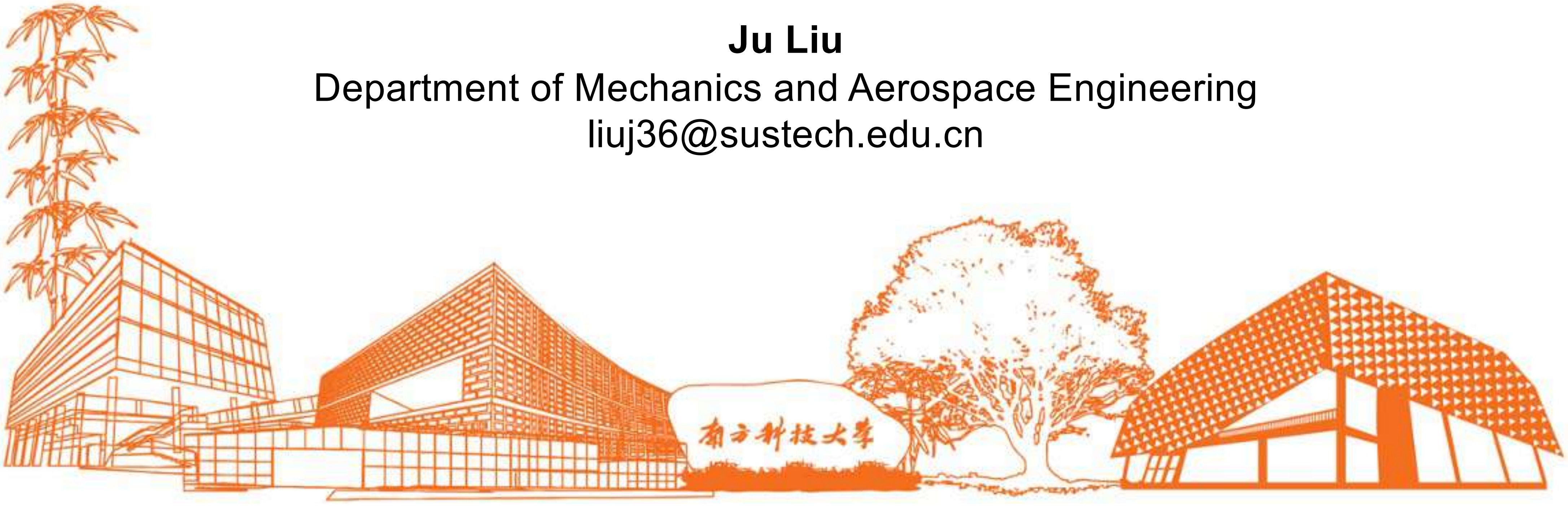
MAE 5032 High Performance Computing: Methods and Applications

Lab 3: SIMD & BLAS

Ju Liu
Department of Mechanics and Aerospace Engineering
liuj36@sustech.edu.cn



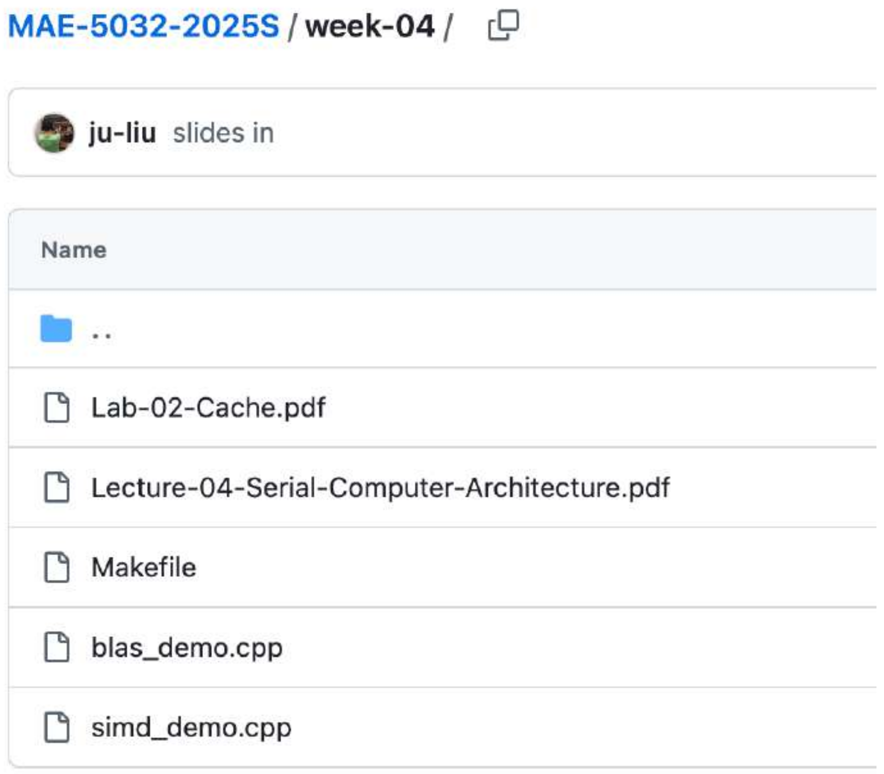
Objective

- You will experiment two codes
  - experience the invocation of SIMD in an explicit manner
  - explore the block matrix multiplication
  - learn the use of BLAS functions

Task 1: SIMD

- Go to https://github.com/ju-liu/MAE-5032-2025S/tree/main/week-04

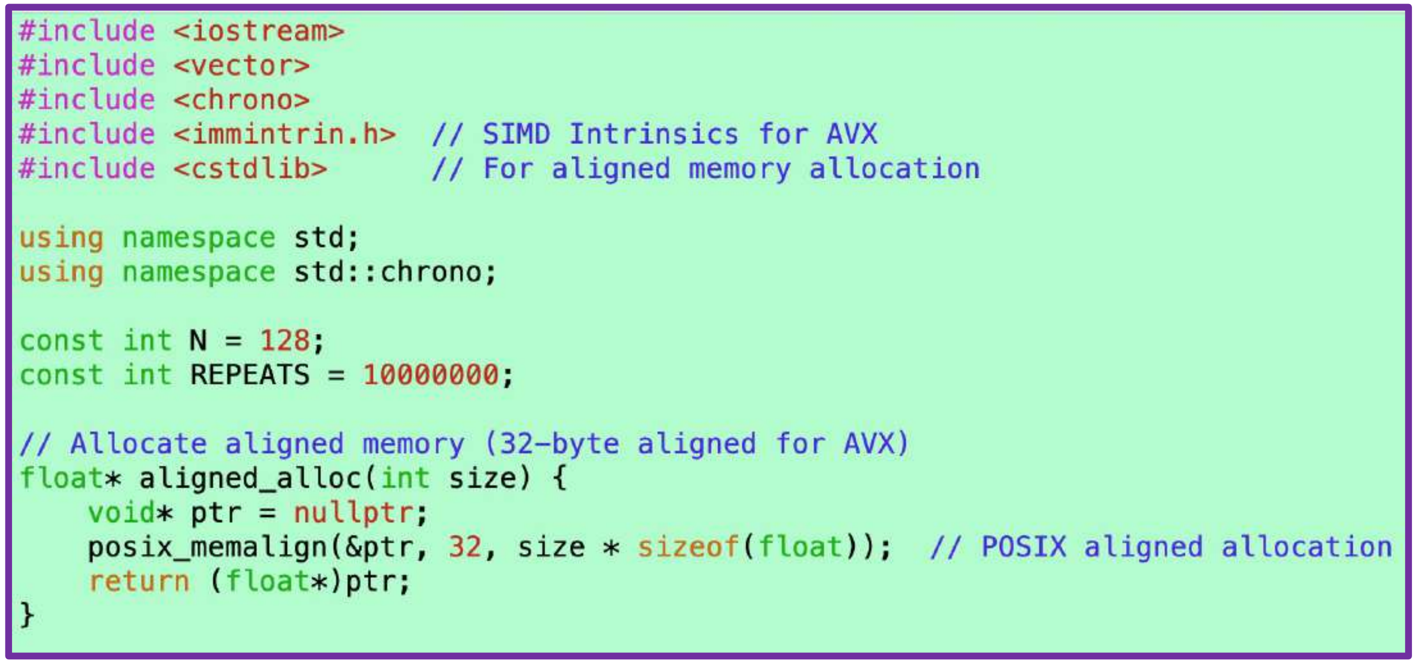
- Download the code


MAE-5032-2025S / week-04 /

ju-liu slides in

Name
..
Lab-02-Cache.pdf
Lecture-04-Serial-Computer-Architecture.pdf
Makefile
blas_demo.cpp
simd_demo.cpp

Task 1: SIMD

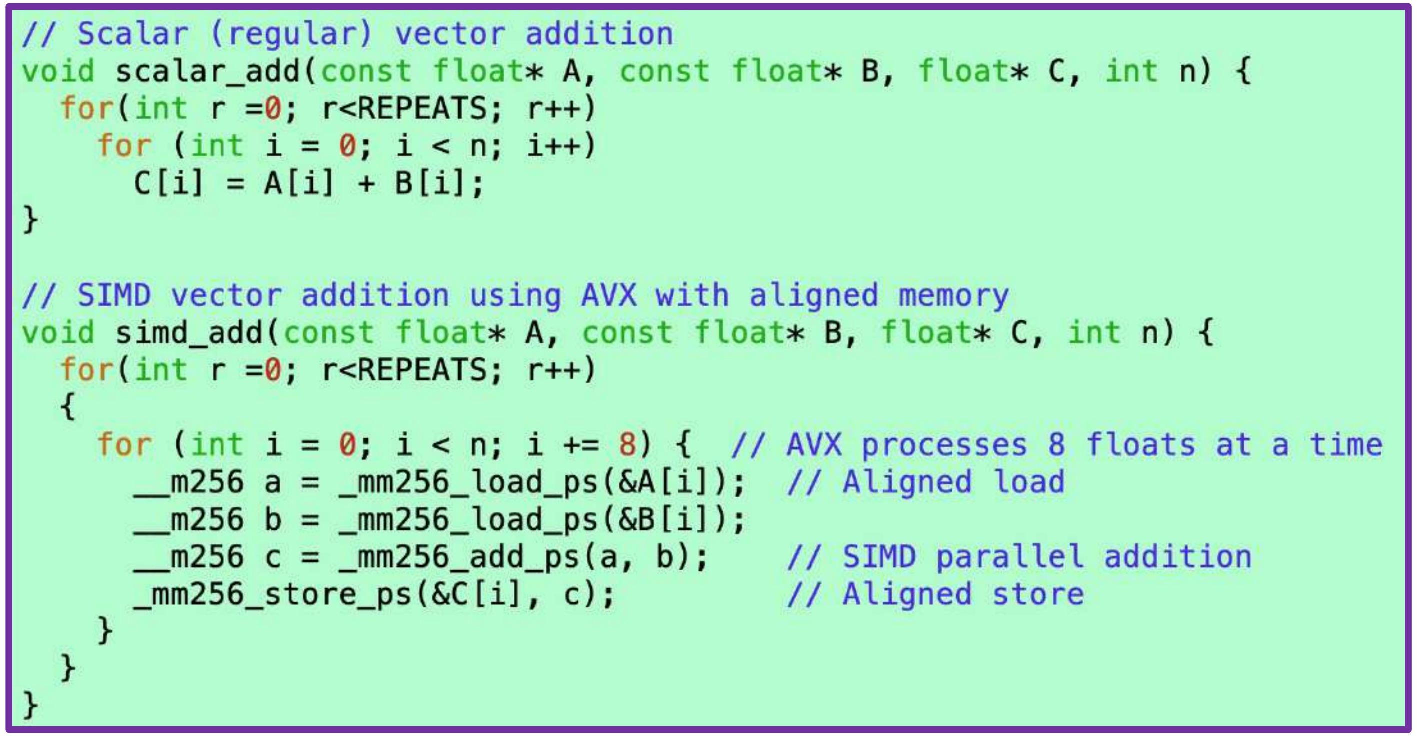


```cpp
#include <iostream>
#include <vector>
#include <chrono>
#include <immintrin.h>  // SIMD Intrinsics for AVX
#include <cstdlib>      // For aligned memory allocation

using namespace std;
using namespace std::chrono;

const int N = 128;
const int REPEATS = 10000000;

// Allocate aligned memory (32-byte aligned for AVX)
float* aligned_alloc(int size) {
    void* ptr = nullptr;
    posix_memalign(&ptr, 32, size * sizeof(float));  // POSIX aligned allocation
    return (float*)ptr;
}
```

Allocate the memory with size x 4 bytes whose starting address is guaranteed to be a multiple of 32 bytes = 256 bits.

This is necessary for AVX.

Task 1: SIMD



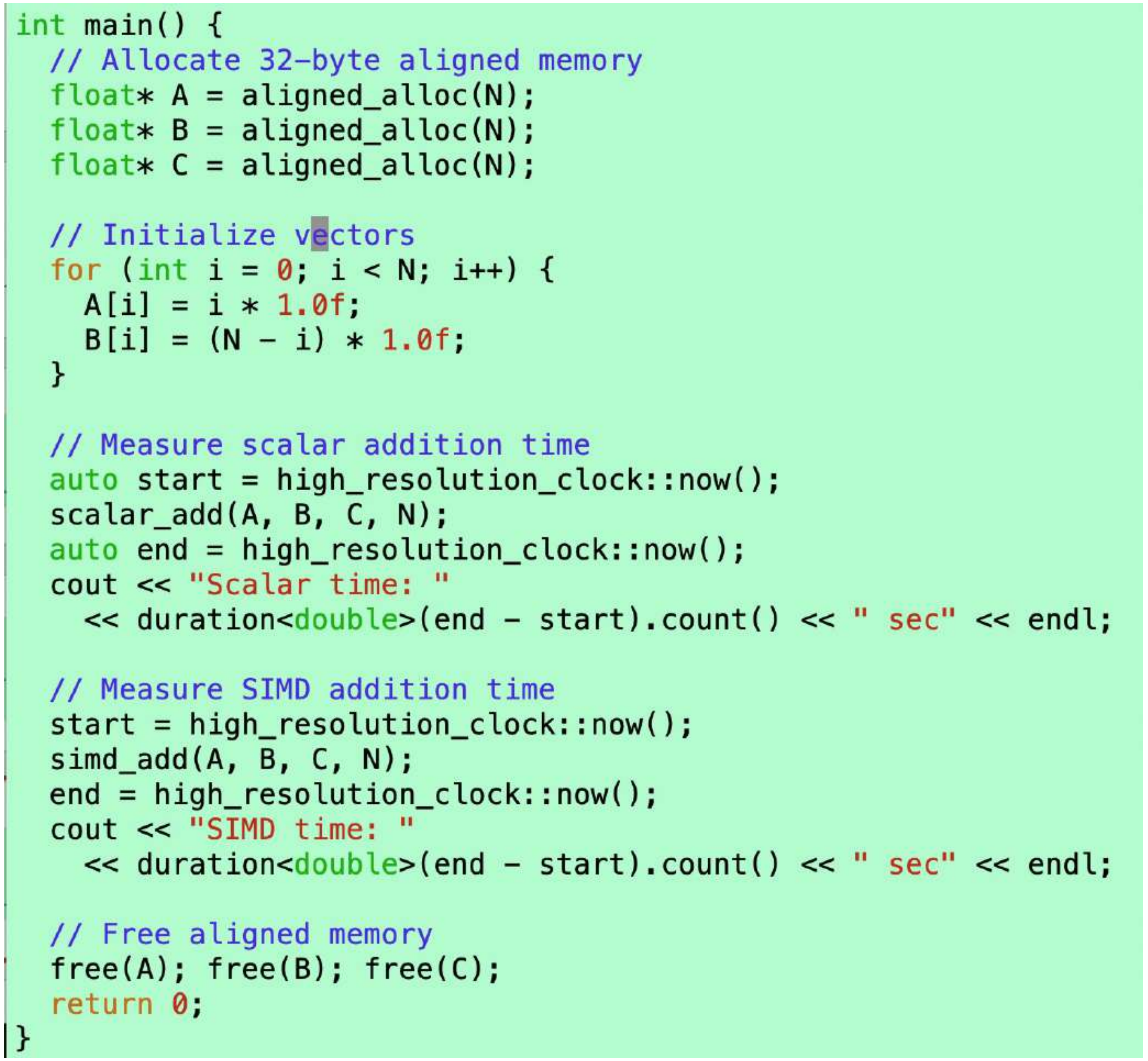
```c
// Scalar (regular) vector addition
void scalar_add(const float* A, const float* B, float* C, int n) {
  for(int r =0; r<REPEATS; r++)
    for (int i = 0; i < n; i++)
      C[i] = A[i] + B[i];
}

// SIMD vector addition using AVX with aligned memory
void simd_add(const float* A, const float* B, float* C, int n) {
  for(int r =0; r<REPEATS; r++)
  {
    for (int i = 0; i < n; i += 8) {  // AVX processes 8 floats at a time
      __m256 a = _mm256_load_ps(&A[i]);  // Aligned load
      __m256 b = _mm256_load_ps(&B[i]);
      __m256 c = _mm256_add_ps(a, b);    // SIMD parallel addition
      _mm256_store_ps(&C[i], c);         // Aligned store
    }
  }
}
```

_mm256_load_ps is part of the AVX instruction set.

It load 8 single-precision floating-point values from memory into AVX 256-bit register

_mm256_add_ps: addition in parallel

_mm256_store_ps: writes 8 floating point values from an AVX 256-bit register into memory



```cpp
int main() {
  // Allocate 32-byte aligned memory
  float* A = aligned_alloc(N);
  float* B = aligned_alloc(N);
  float* C = aligned_alloc(N);

  // Initialize vectors
  for (int i = 0; i < N; i++) {
    A[i] = i * 1.0f;
    B[i] = (N - i) * 1.0f;
  }

  // Measure scalar addition time
  auto start = high_resolution_clock::now();
  scalar_add(A, B, C, N);
  auto end = high_resolution_clock::now();
  cout << "Scalar time: "
    << duration<double>(end - start).count() << " sec" << endl;

  // Measure SIMD addition time
  start = high_resolution_clock::now();
  simd_add(A, B, C, N);
  end = high_resolution_clock::now();
  cout << "SIMD time: "
    << duration<double>(end - start).count() << " sec" << endl;

  // Free aligned memory
  free(A); free(B); free(C);
  return 0;
}
```

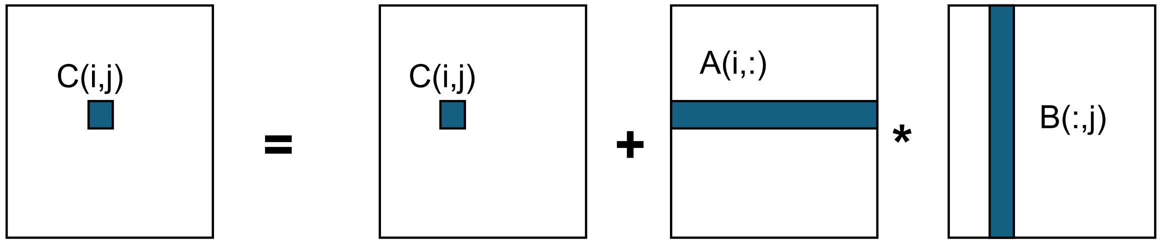
Compile the code by

```
g++ -O0 -std=c++11 -mavx
  -o simd simd_demo.cpp
```

You will get an executable named simd

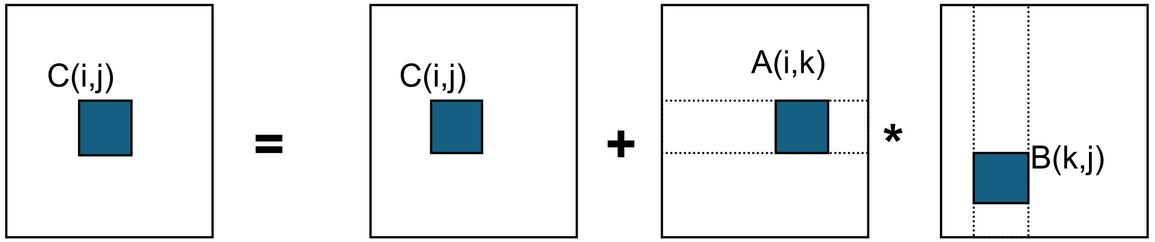
```
./simd
```

Task 2: BLAS



```
// Naïve matrix multiplication
void matrix_mult_naive(const std::vector<double>& A,
                       const std::vector<double>& B,
                       std::vector<double>& C) {
    for (int i = 0; i < N; i++)
        for (int j = 0; j < N; j++)
            for (int k = 0; k < N; k++)
                C[i * N + j] += A[i * N + k] * B[k * N + j];  // Row-major
}
```

Task 2: BLAS

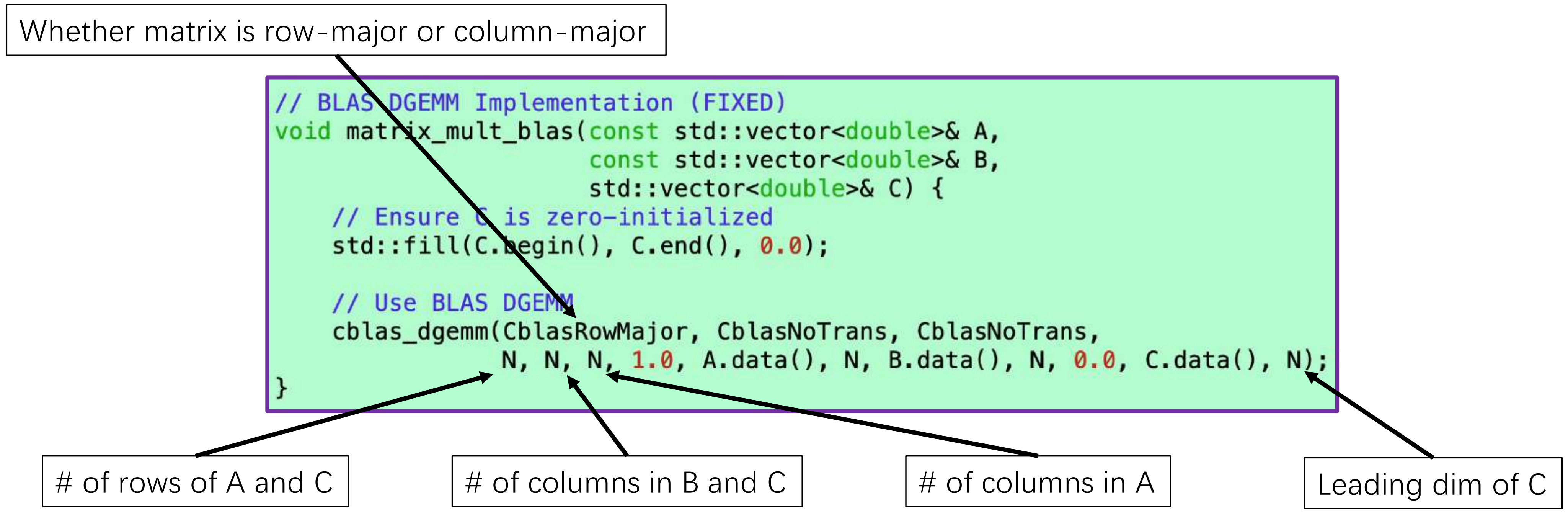


```
// Blocked matrix multiplication (cache-friendly)
void matrix_mult_blocked(const std::vector<double>& A,
                         const std::vector<double>& B,
                         std::vector<double>& C, int block_size) {
    for (int bi = 0; bi < N; bi += block_size)
        for (int bj = 0; bj < N; bj += block_size)
            for (int bk = 0; bk < N; bk += block_size)
                for (int i = bi; i < std::min(bi + block_size, N); i++)
                    for (int j = bj; j < std::min(bj + block_size, N); j++)
                        for (int k = bk; k < std::min(bk + block_size, N); k++)
                            C[i * N + j] += A[i * N + k] * B[k * N + j];  // Row-major
}
```

Task 2: BLAS

cblas_dgemm is part of the BLAS library and is used for matrix-matrix multiplication
C = alpha AB + beta C

A is m-by-k, B is k-by-n, and C is m-by-n. alpha and beta are scalars.



```
// BLAS DGEMM Implementation (FIXED)
void matrix_mult_blas(const std::vector<double>& A,
                      const std::vector<double>& B,
                      std::vector<double>& C) {
    // Ensure C is zero-initialized
    std::fill(C.begin(), C.end(), 0.0);

    // Use BLAS DGEMM
    cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans,
                N, N, N, 1.0, A.data(), N, B.data(), N, 0.0, C.data(), N);
}
```

Linux:

Install openblas

```
sudo apt update
Sudo apt install openblas
```

Check to see if the lib is installed

```
ls /usr/lib/x86_64-linux-gnu | grep openblas
ls /usr/include/x86_64-linux-gnu | grep openblas
```

Compile the code

```
g++ -std=c++11 blas_demo.cpp -I/usr/include/x86_64-linux-gnu -
L/usr/lib/x86_64-linux-gnu -lopenblas
```

Mac:

Install homebrew if not already installed

```
/bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)"
```

Or go to https://brew.sh/

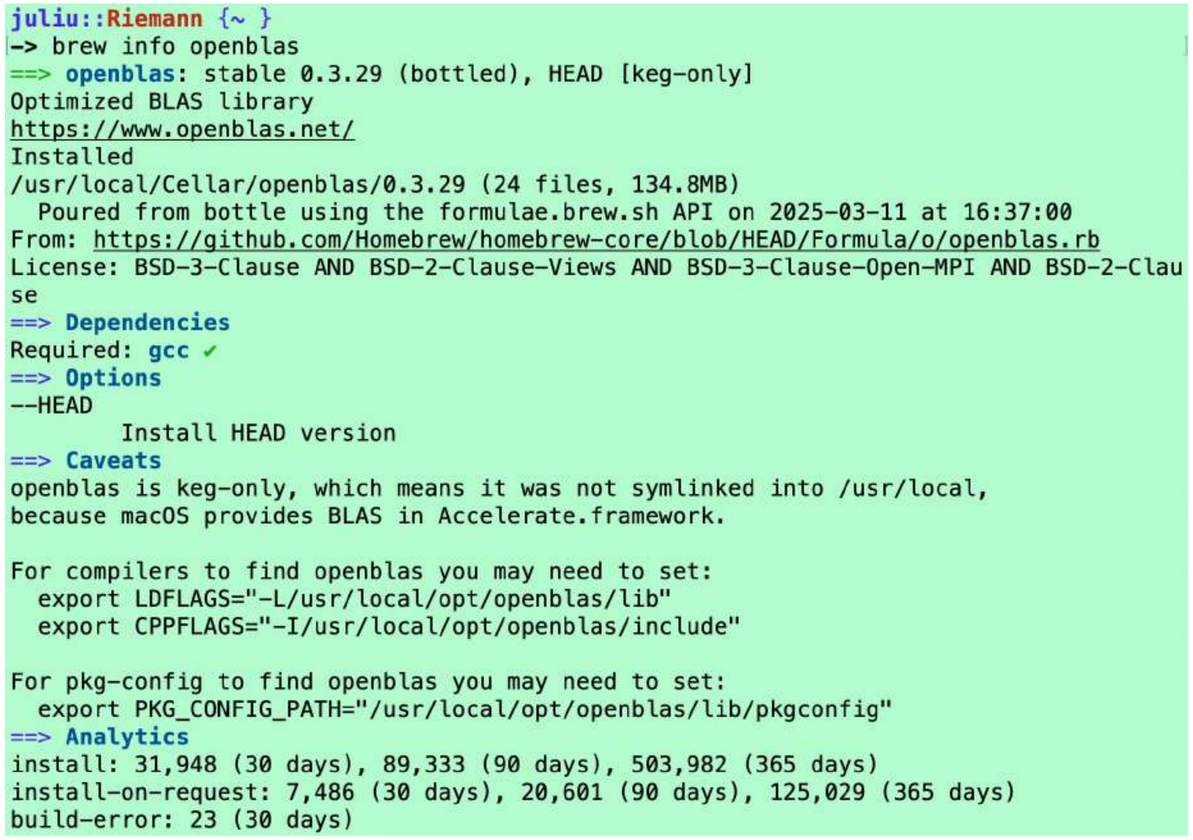
Install openblas

```
brew install openblas
```

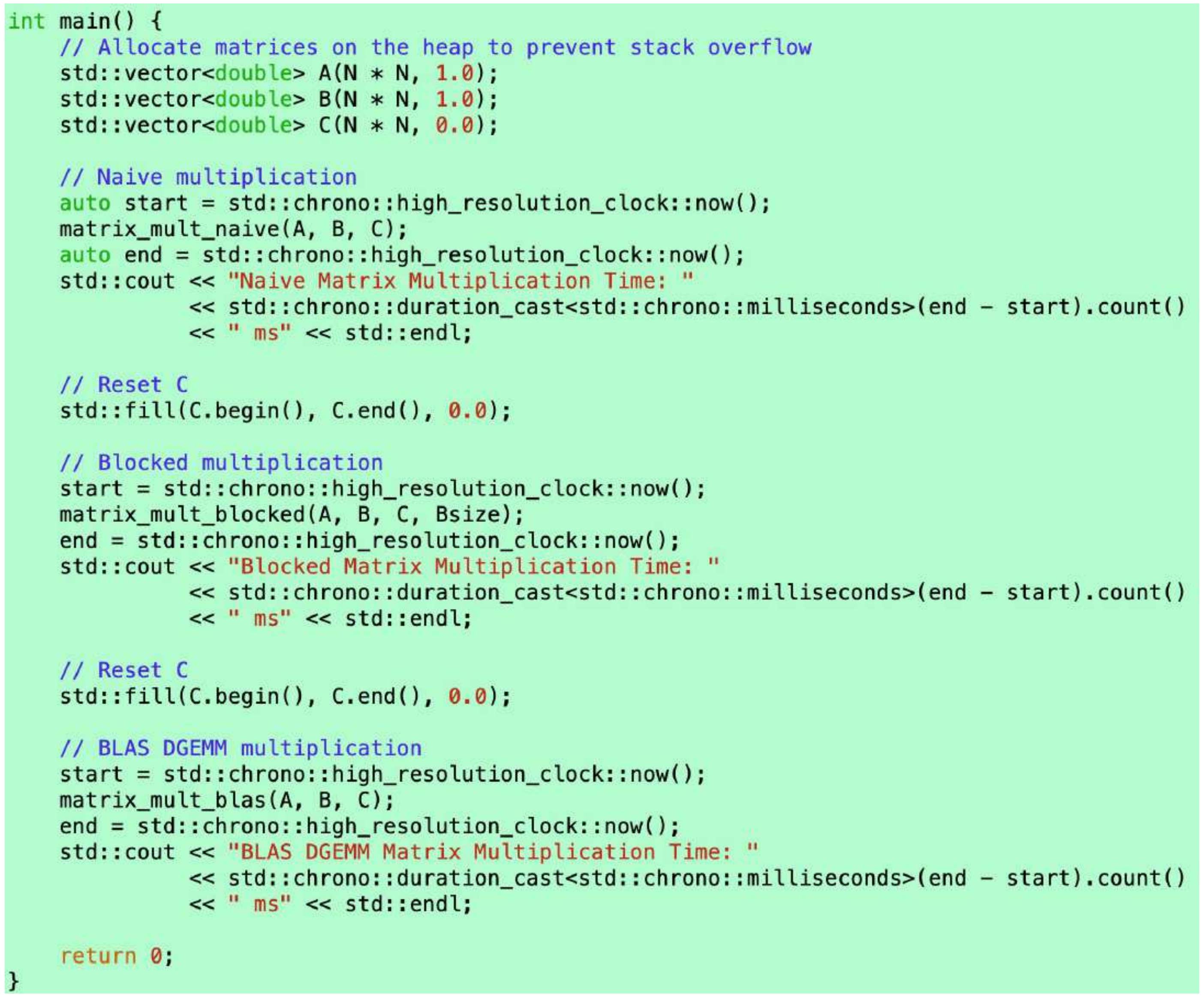
Mac:

Check install info

```
brew info openblas
```



```
juliu::Riemann {~ }
-> brew info openblas
==> openblas: stable 0.3.29 (bottled), HEAD [keg-only]
Optimized BLAS library
https://www.openblas.net/
Installed
/usr/local/Cellar/openblas/0.3.29 (24 files, 134.8MB)
  Poured from bottle using the formulae.brew.sh API on 2025-03-11 at 16:37:00
From: https://github.com/Homebrew/homebrew-core/blob/HEAD/Formula/o/openblas.rb
License: BSD-3-Clause AND BSD-2-Clause-Views AND BSD-3-Clause-Open-MPI AND BSD-2-Clau
se
==> Dependencies
Required: gcc ✔
==> Options
--HEAD
        Install HEAD version
==> Caveats
openblas is keg-only, which means it was not symlinked into /usr/local,
because macOS provides BLAS in Accelerate.framework.

For compilers to find openblas you may need to set:
  export LDFLAGS="-L/usr/local/opt/openblas/lib"
  export CPPFLAGS="-I/usr/local/opt/openblas/include"

For pkg-config to find openblas you may need to set:
  export PKG_CONFIG_PATH="/usr/local/opt/openblas/lib/pkgconfig"
==> Analytics
install: 31,948 (30 days), 89,333 (90 days), 503,982 (365 days)
install-on-request: 7,486 (30 days), 20,601 (90 days), 125,029 (365 days)
build-error: 23 (30 days)
```



```cpp
int main() {
    // Allocate matrices on the heap to prevent stack overflow
    std::vector<double> A(N * N, 1.0);
    std::vector<double> B(N * N, 1.0);
    std::vector<double> C(N * N, 0.0);

    // Naive multiplication
    auto start = std::chrono::high_resolution_clock::now();
    matrix_mult_naive(A, B, C);
    auto end = std::chrono::high_resolution_clock::now();
    std::cout << "Naive Matrix Multiplication Time: "
              << std::chrono::duration_cast<std::chrono::milliseconds>(end - start).count()
              << " ms" << std::endl;

    // Reset C
    std::fill(C.begin(), C.end(), 0.0);

    // Blocked multiplication
    start = std::chrono::high_resolution_clock::now();
    matrix_mult_blocked(A, B, C, Bsize);
    end = std::chrono::high_resolution_clock::now();
    std::cout << "Blocked Matrix Multiplication Time: "
              << std::chrono::duration_cast<std::chrono::milliseconds>(end - start).count()
              << " ms" << std::endl;

    // Reset C
    std::fill(C.begin(), C.end(), 0.0);

    // BLAS DGEMM multiplication
    start = std::chrono::high_resolution_clock::now();
    matrix_mult_blas(A, B, C);
    end = std::chrono::high_resolution_clock::now();
    std::cout << "BLAS DGEMM Matrix Multiplication Time: "
              << std::chrono::duration_cast<std::chrono::milliseconds>(end - start).count()
              << " ms" << std::endl;

    return 0;
}
```

Compile the code by

```
g++ -std=c++11 -
I/usr/local/opt/openblas/include
-L/usr/local/opt/openblas/lib -
lopenblas -O3 blas_demo.cpp
```

You will get an executable named simd

```
./a.out
```