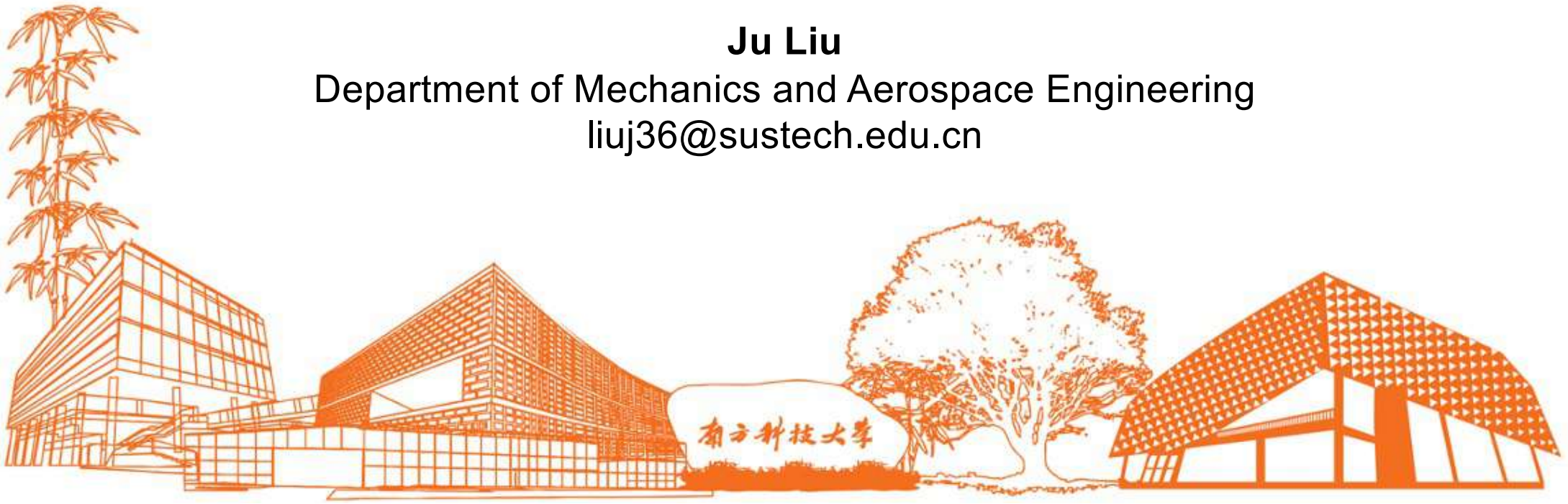
MAE 5032 High Performance Computing: Methods and Applications

Lab 2: Understand Cache

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



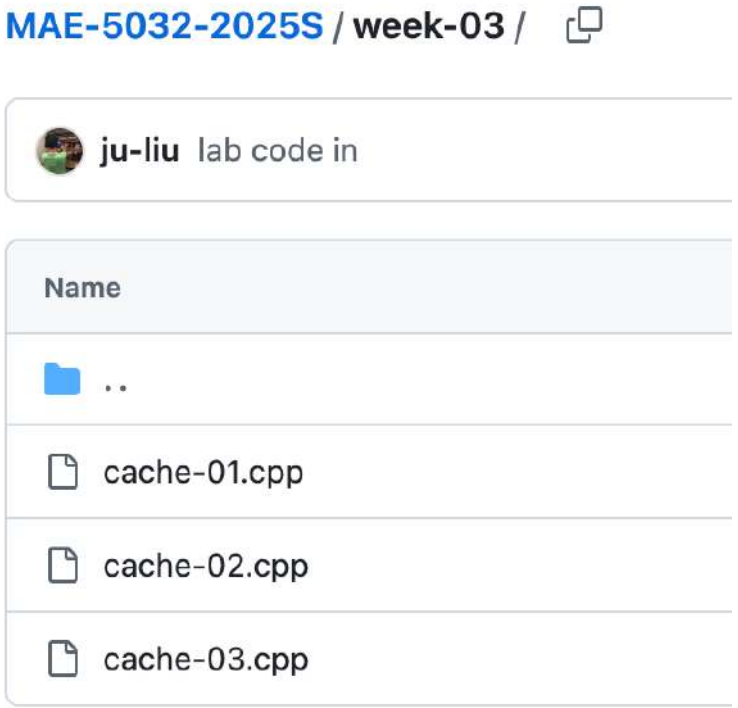
Objective

- You will experiment three codes
  - understand hierarchical structure of the memory system
  - understand the **locality** in practical programming

Task 1: Sequential vs. Random access

- Go to https://github.com/ju-liu/MAE-5032-2025S/tree/main/week-03

- Download the codes to your local computer
  - put them in a separate folder
  - make sure you have a compiler (`which gcc` or `which g++`)
  - `sudo apt install build-essential`



```cpp
#include <iostream>
#include <vector>
#include <chrono>
#include <cstdlib>

#define ARRAY_SIZE (1024 * 1024 * 64) // 64MB array

int main() {
  std::vector<int> arr(ARRAY_SIZE, 1); // Allocate a large array
  volatile int sum = 0;

  // Measure sequential access time
  auto start = std::chrono::high_resolution_clock::now();
  for (size_t i = 0; i < arr.size(); i++) {
    sum += arr[i];
  }
  auto end = std::chrono::high_resolution_clock::now();
  std::cout << "Sequential Access Time: "
    << std::chrono::duration_cast<std::chrono::milliseconds>(end - start).count()
    << " ms" << std::endl;

  // Shuffle indices for random access
  std::vector<size_t> indices(arr.size());
  for (size_t i = 0; i < indices.size(); i++) indices[i] = i;
  std::random_shuffle(indices.begin(), indices.end());

  // Measure random access time
  start = std::chrono::high_resolution_clock::now();
  for (size_t i = 0; i < arr.size(); i++) {
    sum += arr[indices[i]];
  }
  end = std::chrono::high_resolution_clock::now();
  std::cout << "Random Access Time: "
    << std::chrono::duration_cast<std::chrono::milliseconds>(end - start).count()
    << " ms" << std::endl;

  return sum;
}
```

Volatile keyword tells the compiler not to optimize a variable

size_t: an unsigned integral type

std::chrono: C++ standard library for dealing with time and duration measurements

```cpp
#include <iostream>
#include <vector>
#include <chrono>
#include <cstdlib>

#define ARRAY_SIZE (1024 * 1024 * 64) // 64MB array

int main() {
  std::vector<int> arr(ARRAY_SIZE, 1); // Allocate a large array
  volatile int sum = 0;

  // Measure sequential access time
  auto start = std::chrono::high_resolution_clock::now();
  for (size_t i = 0; i < arr.size(); i++) {
    sum += arr[i];
  }
  auto end = std::chrono::high_resolution_clock::now();
  std::cout << "Sequential Access Time: "
    << std::chrono::duration_cast<std::chrono::milliseconds>(end - start).count()
    << " ms" << std::endl;

  // Shuffle indices for random access
  std::vector<size_t> indices(arr.size());
  for (size_t i = 0; i < indices.size(); i++) indices[i] = i;
  std::random_shuffle(indices.begin(), indices.end());

  // Measure random access time
  start = std::chrono::high_resolution_clock::now();
  for (size_t i = 0; i < arr.size(); i++) {
    sum += arr[indices[i]];
  }
  end = std::chrono::high_resolution_clock::now();
  std::cout << "Random Access Time: "
    << std::chrono::duration_cast<std::chrono::milliseconds>(end - start).count()
    << " ms" << std::endl;

  return sum;
}
```

Compile the code by

```
g++ -O2 -std=c++11 -o
cache01 cache-01.cpp
```

You will get an executable named cache01

```
./cache01
```

Task 2: Stride-based memory access

```cpp
#include <iostream>
#include <vector>
#include <chrono>

#define ARRAY_SIZE (256 * 1024 * 1024) // 256MB array to test L1, L2 effects
#define NUM_ACCESSES 10000000 // Fixed number of accesses

int main() {
  std::vector<int> arr(ARRAY_SIZE, 1);
  volatile int sum = 0;

  std::cout << "Testing cache effects with fixed number of accesses:\n";

  for (size_t stride = 1; stride <= 1024; stride *= 2)
  {
    auto start = std::chrono::high_resolution_clock::now();

    for (size_t i = 0, j = 0; i < NUM_ACCESSES; i++)
    {
      sum += arr[j];
      j = (j + stride) % arr.size(); // Wrap around when exceeding array size
    }

    auto end = std::chrono::high_resolution_clock::now();
    std::cout << "Stride: " << stride << ", Time: "
      << std::chrono::duration_cast<std::chrono::microseconds>(end - start).count()
      << " us" << std::endl;
  }

  return sum; // Prevent compiler optimization
}
```

- This code investigate how different stride sizes affect cache performance by iterating over a large array.
- The number of access time is maintained as the same.

Task 2: Stride-based memory access

```cpp
#include <iostream>
#include <vector>
#include <chrono>

#define ARRAY_SIZE (256 * 1024 * 1024) // 256MB array to test L1, L2 effects
#define NUM_ACCESSES 10000000 // Fixed number of accesses

int main() {
  std::vector<int> arr(ARRAY_SIZE, 1);
  volatile int sum = 0;

  std::cout << "Testing cache effects with fixed number of accesses:\n";

  for (size_t stride = 1; stride <= 1024; stride *= 2)
  {
    auto start = std::chrono::high_resolution_clock::now();

    for (size_t i = 0, j = 0; i < NUM_ACCESSES; i++)
    {
      sum += arr[j];
      j = (j + stride) % arr.size(); // Wrap around when exceeding array size
    }

    auto end = std::chrono::high_resolution_clock::now();
    std::cout << "Stride: " << stride << ", Time: "
      << std::chrono::duration_cast<std::chrono::microseconds>(end - start).count()
      << " us" << std::endl;
  }

  return sum; // Prevent compiler optimization
}
```

Compile the code by

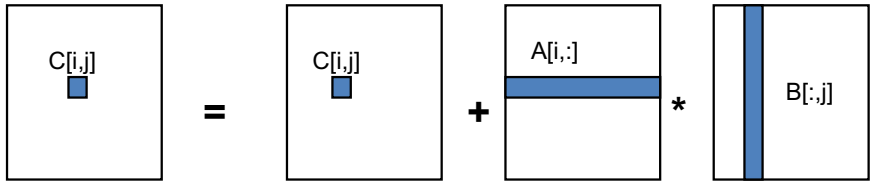
```
g++ -O2 -std=c++11 -o
cache02 cache-02.cpp
```

You will get an executable named cache02

```
./cache02
```

Task 3: Optimize matrix multiplication

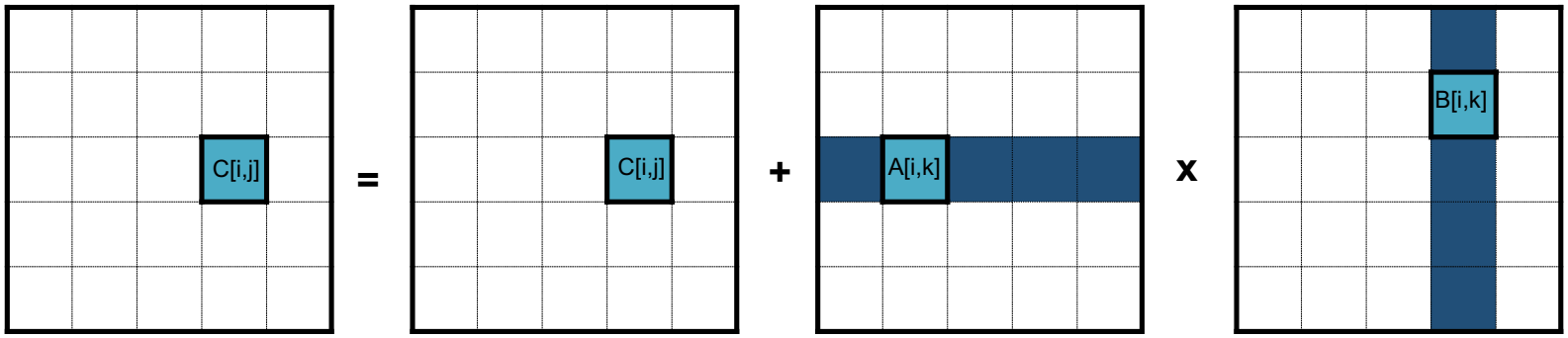
Naïve matrix multiplication:




```
void matrix_mult_naive(std::vector<std::vector<double>> &A,
                       std::vector<std::vector<double>> &B,
                       std::vector<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][j] += A[i][k] * B[k][j];
}
```

Task 3: Optimize matrix multiplication

Blocked (tiled) matrix multiplication:




```
void matrix_mult_blocked(std::vector<std::vector<double>> &A,
                         std::vector<std::vector<double>> &B,
                         std::vector<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][j] += A[i][k] * B[k][j];
}
```

Task 3: Optimize matrix multiplication

```
int main() {
    std::vector<std::vector<double>> A(N, std::vector<double>(N, 1.0));
    std::vector<std::vector<double>> B(N, std::vector<double>(N, 1.0));
    std::vector<std::vector<double>> C(N, std::vector<double>(N, 0.0));

    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;

    C.assign(N, std::vector<double>(N, 0.0)); // Reset C

    start = std::chrono::high_resolution_clock::now();
    matrix_mult_blocked(A, B, C, 32);
    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;

    return 0;
}
```

Summary

- Locality affects memory access speed
- Blocking technique is a widely used strategy in improving performance and optimizing data movement
- Cache misses and cache trashing significantly affect the code performance.
- There are profiling tools that give cache statistics: valgrind