

## 7 Assignment 5 [Assignment ID: cpp\_cache]

### 7.1 Preamble (Please Read Carefully)

Before starting work on this assignment, it is **critically important** that you **carefully** read Section 1 (titled “General Information”) which starts on page 1-1 of this document.

### 7.2 Topics Covered

This assignment covers material primarily related to the following: cache-oblivious algorithms, matrix transposition, matrix multiplication, FFT.

### 7.3 Problems — Part A — Nonprogramming Exercises

- 8.32 [cache parameters]
- 8.33 a b c d [cache]
- 8.34 [cache misses in algorithm]
- 8.36 [virtual memory parameters]
- 8.37 a b c [address translation]

### 7.4 Problems — Part B — Linear Algebra

B.1 *Cache-oblivious matrix transposition.* In this exercise, a function template is to be developed that performs a matrix transposition using a particular cache-oblivious algorithm. Given an  $m \times n$  matrix  $A$ , we wish to compute  $B = A^T$  (where  $B$  is an  $n \times m$  matrix). (Note that, as a matter of notation, an  $m \times n$  matrix is a matrix with  $m$  rows and  $n$  columns.)

The function template to be developed is called `matrix_transpose` and has the following declaration:

```
namespace ra::cache {
    template <class T>
    void matrix_transpose(const T* a, std::size_t m, std::size_t n,
        T* b);
}
```

The function template `matrix_transpose` computes the transpose of the matrix having  $m$  rows,  $n$  columns, and the element data of type  $T$  pointed to by  $a$ . The resulting transposed element data is written to the matrix buffer pointed to by  $b$ . All matrices are stored in row-major order (with no padding between rows). The value of  $b$  is permitted to be equal to  $a$ . If  $b$  equals  $a$ , the matrix named by  $a$  is replaced by its transpose. Note that an auxiliary buffer can be used by the implementation to handle this case. The type  $T$  can be any numeric type for which matrix transposition would be meaningful (e.g., `int`, `double`, `std::complex<double>`). The `matrix_transpose` function is to utilize the cache-oblivious algorithm from the lecture slides. This algorithm uses a divide and conquer strategy and is based on recursion. Note that, for optimal efficiency, the recursion should not be continued until a  $1 \times 1$  matrix is encountered. For example, the base case for the recursion might be chosen to correspond to  $m, n \leq 64$ .

For comparison purposes, a second function template called `naive_matrix_transpose` must be provided that computes the matrix transpose using a straightforward naive approach that does not consider the effects of the cache. This function template has the following declaration:

```
namespace ra::cache {
    template <class T>
    void naive_matrix_transpose(const T* a, std::size_t m,
        std::size_t n, T* b);
}
```

The interface for this function template is identical to the one for `matrix_transpose`.

All of the code for the `matrix_transpose` and `naive_matrix_transpose` function templates must be placed in the header file `include/ra/matrix_transpose.hpp`.

The code used to test the `matrix_transpose` function template should be placed in a file called `app/test_matrix_transpose.cpp`.

**B.2 Cache-oblivious matrix multiplication.** In this exercise, a function template is to be developed that performs matrix multiplication using a particular cache-oblivious algorithm. Given an  $m \times n$  matrix  $A$  and an  $n \times p$  matrix  $B$ , we wish to compute the matrix product  $C = AB$ , where  $C$  is  $m \times p$ . (Note that, as a matter of notation, an  $m \times n$  matrix is a matrix with  $m$  rows and  $n$  columns.)

The function template to be developed is called `matrix_multiply` and has the following declaration:

```
namespace ra::cache {
    template <class T>
    void matrix_multiply(const T* a, const T* b, std::size_t m,
                       std::size_t n, std::size_t p, T* c);
}
```

The `matrix_multiply` function template computes the matrix product  $C = AB$ . The parameter `a` points to the element data for the matrix  $A$  with  $m$  rows and  $n$  columns. The parameter `b` points to the element data for the matrix  $B$  with  $n$  rows and  $p$  columns. The parameter `c` points to the element data for the matrix  $C$  with  $m$  rows and  $p$  columns. The storage pointed to by `a`, `b`, and `c` is not permitted to overlap. All three matrices have elements of type `T`. The type `T` can be any numeric type for which matrix multiplication would be meaningful (e.g., `int`, `double`, `std::complex<double>`). All matrix element data is stored in row-major order (with no padding between rows). The `matrix_multiply` function is to utilize the cache-oblivious algorithm from the lecture slides. Note that, for optimal efficiency, the recursion should not be continued until  $1 \times 1$  matrices are encountered. For example, the base case for the recursion might be chosen to correspond to  $mnp \leq 64$ .

For comparison purposes, a second function template called `naive_matrix_multiply` must be provided that computes the matrix product using a straightforward naive approach that does not consider the effects of the cache. This function template has the following declaration:

```
namespace ra::cache {
    template <class T>
    void naive_matrix_multiply(const T* a, const T* b,
                             std::size_t m, std::size_t n, std::size_t p, T* c);
}
```

The interface for this function template is identical to the one for `matrix_multiply`.

All of the code for the `matrix_multiply` and `naive_matrix_multiply` function templates must be placed in the header file `include/ra/matrix_multiply.hpp`.

The code used to test the `matrix_multiply` function template should be placed in a file called `app/test_matrix_multiply.cpp`.

## 7.5 Problems — Part C — Fast Fourier Transform

**C.1 Cache-oblivious fast-Fourier transform (FFT).** In this exercise, a function template is to be developed that computes a fast-Fourier transform (FFT) using a particular cache-oblivious algorithm.

The function template to be developed is called `forward_fft` and has the following declaration:

```
namespace ra::cache {
    template <class T>
    void forward_fft(T* x, std::size_t n);
}
```

The `forward_fft` function template computes the DFT of the sequence of  $n$  elements of type  $T$  pointed to by  $x$ . This computation is done in place (i.e., the input sequence is overwritten with the DFT result). The `forward_fft` function is to utilize the cache-oblivious algorithm from the lecture slides. In order to simplify the selection of an appropriate factorization of  $n$  (which is needed in the DFT algorithm), the implementation may impose the constraint (on the user) that  $n$  must be a power of two. The type  $T$  can be any complex number class that has an interface compatible with `std::complex`. For example, the code should work with  $T$  chosen as `std::complex<float>` and `std::complex<double>`. Note that, for optimal efficiency, the recursion should probably not be continued until a 1-point DFT is encountered (i.e.,  $n = 1$ ). For example, the base case for the recursion might be chosen to correspond to  $n \leq 4$ . In order to perform matrix transposition (which is needed in the DFT algorithm), the `matrix_transpose` template function developed in Exercise B.1 should be used.

All of the code for the `forward_fft` function template must be placed in the header file `include/ra/fft.hpp`.

The code used to test the `forward_fft` function template should be placed in a file called `app/test_fft.cpp`.

*Mathematical constants.* Mathematical constants, such as  $\pi$  and  $e$ , are available in the C++ Standard Library (e.g., `std::numbers::pi_v` and `std::numbers::e_v`). If you need to use  $\pi$  or  $e$  in your code, you should use the values provided by the C++ Standard Library; otherwise, your code could potentially fail test cases due to errors introduced by insufficiently accurate values for mathematical constants.

