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 of this document.

7.2 Topics Covered

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

7.3 Problems — Part A

- 8.21 [cache parameters]
- 8.20 a b c d [cache example]
- 8.22 [cache misses in algorithm]

7.4 Problems — Part 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.) This matrix multiplication is to be performed by utilizing 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 1×1 matrix is encountered. For example, the base case for the recursion might be chosen to correspond to $mn \leq 64$.

   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 `matrix_transpose` computes the transpose of the matrix having `m` rows, `n` columns, and the element data of type `T` stored in row-major order pointed to by `a`. The resulting transposed element data is written to the buffer pointed to by `b`, with the element data being stored in row-major order. 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>`).

   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`.

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.) This is to be done by utilizing the cache-oblivious algorithm from the lecture slides. Note that, for optimal efficiency, the recursion should not be continued until 1×1 matrices are encountered. For example, the base case for the recursion might be chosen to correspond to $mnp \leq 64$.

   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 computes the matrix product $C = AB$ as described above. The parameter `a` points to the element data for a matrix A with `m` rows and `n` columns. The parameter `b` points to the element data for a matrix B with `n` rows and `p` columns. The parameter `c` points to the element data for a matrix C with `m` rows and `p` columns. All matrix element data is stored in row-major order. The type `T` can be any numeric type for which matrix transposition would be meaningful (e.g., **`int`**, **`double`**, `std::complex<double>`).

   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`.

3. *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(const T* x, std::size_t n);
}
```

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

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