Lab 2 - Memory & Performance

COM4521/COM6521 - Parallel Computing with Graphical Processing Units (GPUs)

Code

- Starting Code
- Solution

Learning Outcomes

- Understand how to manually allocate and free memory
- Understand how to read from a binary file
- Understand how to implement a linked list data structure in C
- Understand how to apply a number of performance optimisations to code
- Demonstrate an ability to benchmark optimisation performance

Exercise 1

The purpose of this exercise is to modify some existing code to use pointers. Some sample code has been provided for you. The example code will read in a binary file which contains 4 records of information on students. The information consists of their forename, surname and average module mark. A `struct` has been defined to hold the student data, the format of this `struct` matches the `struct` used in the program which created the binary files used in the example.

1. Compile and execute the program. It should print out the information for 4 students.
2. The `print_student()` function is inefficient. It requires passing a structure (by value) which causes all of the data to be duplicated. Amend this so that the structure is passed as a reference. You will need to update both the `print_student()` function declaration and definition.
3. The `main()` function uses a statically defined array to hold our student data. Modify this code so that `students` is a pointer to a student `struct` and then manually allocate enough memory to read in the student records. Don't forget to also `free()` the data at the end of the program.

Exercise 2

Copy your previous code from the first exercise into a new project called `Lab02_Exercise02`.

The student structure uses a statically defined, fixed length `char` array to hold both the forename and surname. This is OK but potentially wasteful when we deal with large records as much of the `char` array will be empty. The file `students2.bin` differs from the file used in the first exercise in that it uses dynamic length `char` arrays to hold strings. Both the forename and surname are written to the binary file in the following format:

`unsigned int n, char[0], char[1], char[2], ..., char[n]; e.g. 5, 'J', 'o', 'h', 'n', '\0'`

Modify the `struct` definition so that forename and surname are pointers to `char`. Now update the code to read the student data. You will need to use `fread()` to read the length of the forename (i.e. n).

Hint: allocate memory for the forename (of length n) and then `fread()` the forename, etc. Don't forget to also update your code to ensure that you `free()` any memory you have allocated.

Exercise 3

Copy your previous code from the last exercise into a new project called `Lab02_Exercise03`.

Both the previous exercises assumed that we knew how many student records were stored in the binary data file. For the next exercise we will update our program to read, store and display an arbitrary number of records. In order to do this we are going to use a linked list data structure. The `linked_list.h` header file contains very basic implementation of a generic linked list. The header file contains a structure `llitems` which defines a pointer to the previous and next item in the list.

1. The implementation of a linked list is incomplete. Complete the function `add_to_linked_list()` by implementing the following:
    1. Check that the `ll_end` item is in fact the end of the list (the next record should be `NULL`). If it is not the end then the function should return `NULL`.
    2. Add the item to the end of the linked list updating the old end of the linked list to reflect the addition.
    3. Return a pointer to the new end of the linked list.
2. In order to use the `print_items()` function the function pointer `print_callback` must be set to a function with the following declaration:

    ```
    void print_function(void *);
    ```

You already have a function `print_student()` which could be used but this function accepts a `const` pointer to student structure. Assign to the `print_callback` function pointer, your print_student function using an explicit cast. You must be careful about your use of brackets here.

3. Update your code to read in `students2.bin` by creating a linked list of student records. You will need a pointer to mark both the start and end of the linked list. To test if your stream is at the end of a file (e.g.. it has read the last record) you should check the return value of `fread()` (if less than the requested number of items are returned this indicates the end of the file). You should use the `create_linked_list()` and `add_to_linked_list()` functions. You can use the `free_linked_list()` function to free your linked list but be careful as this won't free the records which the linked list points to.

Exercise 4

1. A program has been provided for you that performs a basic matrix multiply and writes the result out to a text file. The function `multiply_A()` is the most compute intensive function of the program. *You can confirm this by running the visual studio profiler if you wish.* Timing code has been written to profile the function. Execute the program and compare the answer to the original using the command below (which is similar to unix `diff`) which will print any file differences (you will need to make copies of the output files yourself).

   ```
   FC file1 file2
   ```

   and note the performance results in the following table:

Function	**Optimisation**	**Execution Time (seconds)**
`multiply_A()`	None	
`multiply_B()`	Local write	
`multiply_B()`	Release mode (`/O2`)	
`multiply_C()`	Matrix transpose	

We are going to optimise this function by performing a number of changes to it.

1. The inner loop writes directly to `r[i][j]`. This is unnecessary and will cause the program to be memory bound. Create a copy of `multiply_A()` and call it `multiply_B()`. Modify the inner loop to write to a local variable and make only a single write to `r[i][j]` at the end of the inner loop. Compile and execute your program and record the result.
2. Change the build mode to release. This will set the compiler optimisation to `/O2` (Maximum speed). Confirm this in the project properties and then compile and execute. Note your results in the table.

3. The inner loop of the matrix multiply code access both matrix a and b. One of these matrices is accessed using row wise and the other column wise. To avoid a column wise access we can transpose the matrix. Write a function transpose which swaps elements `[i][j]` with `[j][i]`. Transpose the column accessed matrix and then create a copy of `multiply_B()` called `multiply_C()` where you can update your code so that both matrices are accessed using a row wise pattern. Compile, execute and record the result. How is the performance now?

Not a bad speed up huh? The compiler has most likely already performed some loop unrolling for us. You could try loop unrolling yourself to see if this improves performance. If it does not, then the compiler will have done this for us.