Benchmark Suites:
Developing and Implementing a Simple Benchmark Suite *mmult* to perform the Matrix Multiplication

By\ Md Siddiqur Rahman Tanveer
g202417180
COE501 Term 242

1. Introduction

The technique of planning, reasoning, implementing, testing, and fine-tuning computer systems using hierarchical and structural approaches that focus on various abstraction layers in the compute stack is known as computer architecture. Computer architects can test and assess various parts of the architecture and micro-architecture designs by benchmarking. They can then remedy any shortcomings in the architecture and micro-architecture to enhance performance, which leads to innovation.

It is standard practice to use benchmark suites to facilitate efficient benchmarking. A benchmark suite consists of:

1) a build system that is adaptable and extensible
2) a number of functions and techniques to enable efficient benchmarking; and
3) a number of independent benchmarks.

All specified benchmarks will be passed into the build system, which will then compile them into separate binaries for execution and profiling. The benchmark suites offer helpful functions and methods to help profile the benchmarks on various architectures and operating systems and to manage how the benchmarks are conducted.

The aim of this assignment to implement a new benchmark and add it to the provided educational benchmark suite YABMS [1]. It is our responsibility to comprehend this benchmark suite's build system. We need to comprehend the new benchmark's algorithm and every aspect of its implementation. In order to observe the advantages of specific implementations, we must run and profile the benchmark on our computer.

2. Collaboration

In this assignment at first I tried to understand by reading the task. After attending the hackathon I got the idea about the environment in WSL (Windows Subsystem for Linux) [2] and how to add the new benchmark to the existing YABMS and build and run. Also, we are the class mates organized a team meetings to discuss the task. One of us Moneer Kaid Hasan Al-Bokhaiti also gave details description on the team meeting what is the input and what will be the output as we have already understood from the hackathon and reading manual. He also explained the *vvadd* too. I also got help from Nafisa Tabassum regarding the pointer, and input argument passing to the command line. I helped Nafisa Tabassum, Sakibul Islam, and Irfan Rashid regarding installing WSL.

3. Na¨ıve Implementation

At first I tried to understand the code first. Since, it’s long I didn’t code in C. I have to rewind my mind to do a simple code in C and compile it through command prompt. Then I came to know that to work with this benchmark suite I need to install the WSL. Then I installed the WSL in my windows 11 operating system. I installed Ubuntu on WSL. The first thing I did was to build the system using *make* in the WSL terminal and run the *vvadd* benchmark first. I ran it and then tried to figure out how this suite works.

There is a *template* provided in the benchmark suite and I copied that and renamed this to *mmult*. So that a new empty benchmark has been created. I had to change the following files to create this *mmult* functional.

1. Main.c
2. common\macros.h
3. impl\naive.c
4. include\types.h

At first I developed and implemented the benchmark for *int* to check whether the benchmark is working properly or not. I included the type of my variables in the *types.h* file as per below

```
#ifndef __INCLUDE_TYPES_H_
#define __INCLUDE_TYPES_H_

typedef struct {
  byte*   input;
  byte*   output;


  int*   A;
  int*   B;
  int*   R;

  int*   M;
  int*   N;
  int*   P;

  size_t size;

  int     cpu;
  int     nthreads;
} args_t;

#endif //__INCLUDE_TYPES_H_
```

So that in my whole benchmark will know that I will deal the int type values. I added a function in macros.h to initialize the matrix as below

```
#define __ALLOC_INIT_MATRIX_DATA(type, nelems) ({      \
  unsigned int nbytes = (nelems) * sizeof(type);       \
  nbytes = ((nbytes + 63) / 64) * 64;                  \
  type* temp = (type*)aligned_alloc(512 / 8, nbytes);  \
                                                       \
  if (temp == NULL) {                                  \
    printf("\n");                                      \
    printf("  ERROR: Cannot allocate memory!");        \
    printf("\n");                                      \
    printf("\n");                                      \
    exit(-2);                                          \
  }                                                    \
                                                       \
  /* Generate data */                                  \
  for(int i = 0; i < nelems; i++) {                    \
    temp[i] = i % (0x1llu << (sizeof(type) * 8));      \
  }                                                    \
  temp;                                                \
})
#endif
```

In the naive.c file I included the matrix multiplication operation as below.

```
/* Get the argument struct */
args_t* parsed_args = (args_t*)args;

/* Get all the arguments */
register       int*   dest = (      int*)(parsed_args->output);


register       int*   R = (      int*)(parsed_args->R);
register const int*   A = (const int*)(parsed_args->A);
register const int*   B = (const int*)(parsed_args->B);

register       int*   M = (      int*)(parsed_args->M);
register const int*   N = (const int*)(parsed_args->N);
register const int*   P = (const int*)(parsed_args->P);

int m = *M;
int n = *N;
int p = *P;

printf("M: %d\n", m);
printf("N: %d\n", n);
printf("P: %d\n", p);
```

I am getting matrix dimension from the input parameters in the command line. Then initialize matrix *matA* from 1 to M×N and matrix *matB* from 1 to N×P.

```
//decalre three matrix here
int matA[m][n];
int matB[n][p];
int matR[m][p];
int value = 1;
int index = 0;
//printf("printing A: \n");
for (register int i = 0; i < m; i++) {
  for (register int j = 0; j < n; j++) {
      matA[i][j] = value;
      //printf("%d ", matA[i][j]);
      value++;
  }
  //printf("\n");
}
value = 1;
printf("printing B: \n");
for (register int i = 0; i < n; i++) {
  for (register int j = 0; j < p; j++) {
      matB[i][j] = value;
      //printf("%d ", matB[i][j]);
      value++;
  }
  //printf("\n");
}
```

Also I declared the matrix *matR* which will contain the multiplication result of these two matrices. I initialized this one to zero.

```
for (register int i = 0; i < m; i++) {
  for (register int j = 0; j < p; j++) {
      matR[i][j] = 0;
      //printf("%d ", matR[i][j]);
      value++;
  }
  //printf("\n");
}
```

Then I performed the matrix multiplication as per Algorithm 1 provided in the manual.

Algorithm 1 Naïve `mmult` Implementation

```
1:  procedure MMULT(A, B)
2:      R ← empty_matrix(M, P)
3:      for i = 1 → M do
4:          for j = 1 → P do
5:              R[i][j] ← 0
6:              for k = 1 → N do
7:                  R[i][j] += A[i][k] * B[k][j]
8:              end for
9:          end for
10:     end for
11:     return R
12: end procedure
```

The code implementation is shown below

```
// Perform matrix multiplication: C = A * B
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < p; j++) {
            for (int k = 0; k < n; k++) {
                matR[i][j] += matA[i][k] * matB[k][j];

            }
        }
    }
```

I build the benchmark and then generated the golden reference (matrix multiplication result for matrices A(16×12) and B (12×8) using python script.

I ran the benchmark for testing dimension matrix A(16×12) and B (12×8) and found it matched with golden reference and found that it matched. Here is the simulation outcomes:

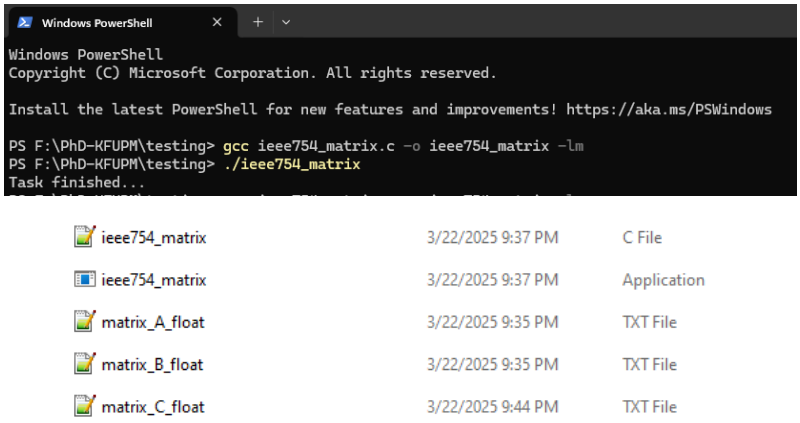
```
 * Verifying results .... Success
 * Running statistics:
  + Starting statistics run number #1:
    - Standard deviation = 277467
    - Average = 2173505
    - Number of active elements = 10000
    - Number of masked-off = 107
  + Starting statistics run number #2:
    - Standard deviation = 219381
    - Average = 2159093
    - Number of active elements = 9893
    - Number of masked-off = 108
  + Starting statistics run number #3:
    - Standard deviation = 206478
    - Average = 2151004
    - Number of active elements = 9785
    - Number of masked-off = 50
  + Starting statistics run number #4:
    - Standard deviation = 201806
    - Average = 2147709
    - Number of active elements = 9735
    - Number of masked-off = 25
  + Starting statistics run number #5:
    - Standard deviation = 199652
    - Average = 2146130
    - Number of active elements = 9710
    - Number of masked-off = 8
  + Starting statistics run number #6:
    - Standard deviation = 198983
    - Average = 2145881
    - Number of active elements = 9702
    - Number of masked-off = 4
  + Starting statistics run number #7:
    - Standard deviation = 198653
    - Average = 2145635
    - Number of active elements = 9698
    - Number of masked-off = 3
  + Starting statistics run number #8:
    - Standard deviation = 198407
    - Average = 2145450
    - Number of active elements = 9695
    - Number of masked-off = 0
 * Runtimes (MATCHING):  2145450 ns
 * Dumping runtime informations:
  - Filename: scalar_naive_runtimes.csv
  - Opening file .... Succeeded
  - Writing runtimes ... Finished
  - Closing file handle .... Finished
```

Then I moved to update this benchmark for matrices which should be IEEE754 [3] floating-point 32-bit single. I didn't generate the dataset for the int operation but I need to generate the dataset according to the dataset information mentioned in the reading manual as per below.

Table 1: **Datasets Information**–table showing dimensions for input datasets and the golden reference.

Dataset	Matrix A		Matrix B		Matrix R	
	M	N	M	N	M	N
testing	16	12	12	8	16	8
small	121	180	180	115	121	115
medium	550	620	620	480	550	480
large	962	1,012	1,012	1,221	962	1,221
native	2,500	3,000	3,000	2,100	2,500	2,100

I used a c program contained in the **ieee754_matrix.c** file to generate the golden references and dataset to perform matrix multiplication operation. I just declare the dimension of the matrices in the program and run this from command prompt.



It randomly generated the matrices A and B data and stored in text files (*matrix_A_float.txt* and *matrix_B_float.txt*) and multiplication result stored *matrix_C_float.txt* file in row-major-matrix [4] format.

I created another benchmark named it ***mmultfloat*** to perform the matrix multiplication for IEEE754 floating point number. Then to read these files, I included the functionality in the *macros.h* file

```
#ifndef __APPLE__
#define __ALLOC_INIT_MATRIX_DATA_FILE_A(type, nelems) ({      \
  unsigned int nbytes = (nelems) * sizeof(type);              \
  nbytes = ((nbytes + 63) / 64) * 64;                         \
  type* temp = (type*)aligned_alloc(512 / 8, nbytes);         \
                                                              \
  if (temp == NULL) {                                         \
    printf("\n");                                             \
    printf("  ERROR: Cannot allocate memory!");               \
    printf("\n");                                             \
    printf("\n");                                             \
    exit(-2);                                                 \
  }                                                           \
                                                              \
    FILE *file = fopen("matrix_A_float.txt", "r");            \
    if (file == NULL) {                                       \
        printf("Error opening file matrix_A_float.txt\n");    \
        return 1;                                             \
    }                                                          \
    /* Reading data from txt file*/                                  \
    printf("Showing the float value from matrix_A_float.txt file\n"); \
    for(int i = 0; i < nelems; i++) {                          \
        if (fscanf(file, "%f", &temp[i]) != 1) {\
            printf("Error reading matrix_A_float.txt file\n");  \
                fclose(file);  \
                return 1; \
    }\
    /*temp[i] = i % (0xlllu << (sizeof(type) * 8));*/ \
    printf("temp[%d]: %10.6f\n",i, temp[i]);                  \
  }                                                           \
  fclose(file);                                                \
  temp;                                                  \
})
#endif
```

The same way I included function *__ALLOC_INIT_MATRIX_DATA_FILE_B* to read the values of matrix B from the text file and *__ALLOC_INIT_MATRIX_DATA_FILE_R* to read the multiplication result from text file.

To take matrix dimension as input I had already included the argument parameters. Where -r or --R is for M, -c or --C is for N and lastly –p or –P is for P dimension variables.

```
/* Matrix Row Size size */
if (strcmp(argv[i], "-r") == 0 || strcmp(argv[i], "--R") == 0) {
  assert (++i < argc);
  M = atoi(argv[i]);
  printf("M: %d\n", M);
  continue;
}
if (strcmp(argv[i], "-c") == 0 || strcmp(argv[i], "--C") == 0) {
  assert (++i < argc);
  N = atoi(argv[i]);
  printf("N: %d\n", N);
  continue;
}
if (strcmp(argv[i], "-p") == 0 || strcmp(argv[i], "--P") == 0) {
  assert (++i < argc);
  P = atoi(argv[i]);
  printf("P: %d\n", P);
  continue;
}
```

Then read the values from the text files

```
float* A    = __ALLOC_INIT_MATRIX_DATA_FILE_A(float, M*N);


float* B    = __ALLOC_INIT_MATRIX_DATA_FILE_B(float, N*P);
float* R    = __ALLOC_INIT_MATRIX_DATA_FILE_R(float, M*P);
float* GRef= __ALLOC_INIT_MATRIX_DATA_FILE_R(float, M*P);
```

In the *naive.c* file I just changed the matrices type to *float* which were *int* in my *mmult*.

```
/* Get the argument struct */
  args_t* parsed_args = (args_t*)args;

  /* Get all the arguments */
  register       int*   dest = (      int*)(parsed_args->output);


  register       float*   R = (      float*)(parsed_args->R);
  register const float*   A = (const float*)(parsed_args->A);
  register const float*   B = (const float*)(parsed_args->B);

  register       int*   M = (      int*)(parsed_args->M);
  register const int*   N = (const int*)(parsed_args->N);
  register const int*   P = (const int*)(parsed_args->P);

  int m = *M;
  int n = *N;
  int p = *P;

  printf("M: %d\n", m);
  printf("N: %d\n", n);
  printf("P: %d\n", p);

  //decalre three matrix here
  float matA[m][n];
  float matB[n][p];
  float matR[m][p];
```

```
  int value = 1;
  int index = 0;
  //printf("printing A: \n");
  for (register int i = 0; i < m; i++) {
            for (register int j = 0; j < n; j++) {
                        matA[i][j] = A[index];
                        //printf("%10.6f ", matA[i][j]);
                        index++;
            }
    //printf("\n");
  }
  index = 0;
  //printf("printing B: \n");
  for (register int i = 0; i < n; i++) {
            for (register int j = 0; j < p; j++) {
                        matB[i][j] = B[index];
                        //printf("%10.6f ", matB[i][j]);
                        index++;
            }
    //printf("\n");
  }

    /*
//Do the matrix multiplication here */

// Perform matrix multiplication: C = A * B
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < p; j++) {
                                    matR[i][j] = 0;
            for (int k = 0; k < n; k++) {
               matR[i][j] += matA[i][k] * matB[k][j];

            }
        }
    }
  //printf("printing multiplcation result store on R: \n");
  index = 0;
  for (int i = 0; i < m; i++) {
            for (int j = 0; j < p; j++) {
                        R[index] = matR[i][j];
                        //printf("%10.6f ", R[index]);
                        index++;
            }
    //printf("\n");
  }

 printf("Multiplication Finished\n");
```

4. Evaluation

After including necessary functionalities to the benchmark. I build the benchmark and ran it for the testing dimension. The benchmark loads the values from the matrix_A_float.txt for matrix A and matrix_B_float.txt file for matrix B and matrix_C_float.txt for golden reference to match the matrix multiplication result derived from A and B.

When the benchmark *mmult* was implemented for integer type values it worked well and successfully matched with golden references. But the issue happened when I implemented the floating point number multiplication it

didn't match with the golden references. The golden reference multiplication result and the benchmark generated multiplication result varies after the decimal point. Then I added the matching threshold *delta = 0.05* and it matched. The code segment showing the threshold applied to the matching

```
// Check with the golden reference values using a for loop
printf("Values read from the file:\n");
matchFlag = true;
float delta = 0.05;

for (int i = 0; (i < M*P) && matchFlag; i++) {
    /*
    if(R[i] == GRef[i]){
        matchFlag = true;
    }else{
        matchFlag = false;
    }
    */
    matchFlag = matchFlag && (fabs(R[i] - GRef[i]) < delta);
    printf("%10.6f ", GRef[i]);
}
printf("\n");
if(matchFlag){
    printf("matched\n");
}else{
    printf("Not matched\n");
}
printf("\n");
```

The simulation results have shown below

Table 1: Simulation result for testing dataset

Verifying results .... Success								
Starting statistics run number	1	2	3	4	5	6	7	8
Standard deviation	277467	219381	206478	201806	199652	198983	198653	198407
Average	2173505	2159093	2151004	2147709	2146130	2145881	2145635	2145450
Number of active elements	10000	9893	9785	9735	9710	9702	9698	9695
Number of masked-off	107	108	50	25	8	4	3	0
Runtimes (MATCHING): 2145450 ns								

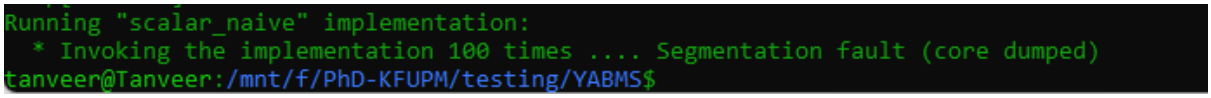
Table 2: Simulation result for small dataset

Verifying results .... Success						
Starting statistics run number	1	2	3	4	5	6
Standard deviation	1373666	1103949	1050053	1042261	1039657	1038799
Average	5301069	5205638	5172457	5166827	5164891	5164249
Number of active elements	10000	9855	9769	9752	9746	9744
Number of masked-off	145	86	17	6	2	0
Runtimes (MATCHING): 5164249 ns						

Table 3: Simulation result for medium dataset (nruns=100)

Verifying results .... Success			
Starting statistics run number	1	2	3
Standard deviation	46326659	40261862	37981600
Average	330583612	327217180	325806119
Number of active elements	100	98	97
Number of masked-off	2	1	0
Runtimes (MATCHING): 325806119 ns			

For the large and naïve dataset the benchmark showed error



5. Conclusion

After implementing the benchmark *mmult* and *mmultfloat* to perform the matrix multiplication for integer and floating point number respectively and matching golden references. I have gather much idea about benchmarking. I hope that this experience will help us to work on computer architecture in future research.

6. References:

[1] Khalid Al-Hawaj. *YABMS: Yet Another Benchmark Suites*. https://github.com/hawajkm/YABMS

[2] WSL (Windows Subsystem for Linux). https://ubuntu.com/desktop/wsl

[3] IEEE754 floating point number. https://en.wikipedia.org/wiki/IEEE_754

[4] Row-and-column-major order https://en.wikipedia.org/wiki/Row-_and_column-major_order