MONASH University

[Sample Solutions]

PARALLEL SORT & SEARCH USING MPI

OBJECTIVES

- The purpose of this lab is to apply data parallelism for sort and search operations

INSTRUCTIONS

- Download and set up the Linux environment on Docker [Refer to Week 0]
- Lab materials of Week 11 will be assessed in the form of Moodle Quiz on Week 12. Please review the materials and prepare well for the Lab Quiz 4.

TASK

DESCRIPTION:

- Analysing a serial sorting algorithm.
- Analysing a parallel sorting algorithm using MPI.
- Design and implement a parallel search algorithm using MPI.
- Design and implement SSE code with Intel's Advanced Vector Extensions (AVX) intrinsic functions.
- Design and implement CUDA C on Google Collab.

MONASH University

LAB ACTIVITIES

Task 1 – Merge Sort Serial Code – Worked example

Merge sort is a comparison-based sorting algorithm. In most implementations it is stable, meaning that it preserves the input order of equal elements in the sorted output. It is an example of the divide and conquer algorithmic paradigm. It was invented by John von Neumann in 1945.

Write a serial-based merge sort code in C.

Sample solution:

```
//-----------------------------------------------------------------
// Merge Sort Code in serial implementation
//
// Author: http://www.c.happycodings.com/Sorting_Searching/code11.html
//         - Initial version
//-----------------------------------------------------------------
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

// Data
#define MAXARRAY 200

// Function prototype
void mergeSort(int[], int, int);

// Main program
int main(void)
{
    int data[MAXARRAY]; int i = 0;

    // Load random data into the array
    // Note: srand() function is not used here.
    // Hence, random number generated will be the same every time the
    application is executed.
    // This makes it easier to view the sorted results.
    for(i = 0; i < MAXARRAY; i++)
    {
```

```
        data[i] = rand() % 100;
    }

    // Print data before sorting
    printf("Before Sorting:\n");
    for(i = 0; i < MAXARRAY; i++)
    {
        printf(" %d", data[i]);
    }
    printf("\n");

    // Call the merge sort function
    mergeSort(data, 0, MAXARRAY - 1);

    // Print data after sorting
    printf("\n");
    printf("After sorting using Mergesort:\n");
    for(i = 0; i < MAXARRAY; i++)
    {
        printf(" %d", data[i]);
    }
    printf("\n");

    return 0;
}

// Function definition
void mergeSort(int inputData[], int startPoint, int endPoint)
{
    int i = 0;
    int length = endPoint - startPoint + 1;
    int pivot = 0;
    int merge1 = 0;
    int merge2 = 0;
    int working[MAXARRAY] = {0};

    if(startPoint == endPoint)
    {
        return;
    }
    pivot = (startPoint + endPoint) / 2;

    // Recursive function call
    mergeSort(inputData, startPoint, pivot);
    mergeSort(inputData, pivot + 1, endPoint);

    for(i = 0; i < length; i++)
    {
        working[i] = inputData[startPoint + i];
```

```
    }
    merge1 = 0;

    merge2 = pivot - startPoint + 1;

    for(i = 0; i < length; i++)
    {
        if(merge2 <= endPoint - startPoint)
            if(merge1 <= pivot - startPoint)
                if(working[merge1] > working[merge2])
                {
                   inputData[i + startPoint] = working[merge2++];
                }
                else
                {
                   inputData[i + startPoint] = working[merge1++];
                }
            else
            {
                inputData[i + startPoint] = working[merge2++];
            }
        else
        {
            inputData[i + startPoint] = working[merge1++];
        }
    }
}
```

MONASH University

Task 2 – Merge Sort Parallel Code using MPI – Worked example

Modify the serial code in Task 1 into a parallel code using Message Passing Interface (MPI) in C. For this question, you are free to choose the type of MPI implementation. However, only the process with rank 0 should print out the data before and after the sorting process.

Sample solution:

```
// MPI MergeSort
//*************************************************************************
*****************************************

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <mpi.h>

// Define the size of data
#define N 200

// Function Prototype
void mergeSort(int* data, int startPoint, int endPoint);
void merge(int *A, int sizeA, int *B, int sizeB);

// Main Function
int main(int argc, char* argv[])
{
    // Variable declaration
    int i;
    int *data = NULL;      // Initialize data pointer to null
    int scale = 0;
    int currentLevel = 0; // current level of tree
    int maxLevel = 0;      // maximum level of tree = LOG2 (number of
    Processors)

    int pivot = 0;         // middle point of data array
    int length = 0;        // length of data array
    int rightLength = 0;   // length of child node data array

    int p;                 // Number of Processors
    int myRank;            // Processor's rank
    MPI_Status status;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD,&p);
    MPI_Comm_rank(MPI_COMM_WORLD,&myRank);
```

```
    if(myRank == 0){
       // Root Node:
       maxLevel = log ((double)p) / log(2.00); // Calculate the
maximum level of binary tree
       length = N;  // Set the length of root node to N
       data = (int*)malloc(length * sizeof(int)); // Create dynamic
array buffer with size length

       // srand is not used to keep a constant set of random values at
each program execution for better debugging
       for(i = 0; i<N;i++)
           data[i] = rand()%100;

       printf("\n-----------------------------------------------------
--------------------------\n");
       printf("Unsorted Data: \n");
       for(i = 0; i<N; i++){  // Prints out unsorted data value
          if(i%10 == 0) // 10 elements in a row
              printf("\n");
          printf("%d\t",data[i]);
       }
       printf("\n-----------------------------------------------------
       --------------------------\n");
    }

    // Broadcast maxLevel to all other processors
    MPI_Bcast(&maxLevel, 1, MPI_INT, 0, MPI_COMM_WORLD);

    // Divide the data to child node
    for(currentLevel = 0; currentLevel <=maxLevel;currentLevel++){
        scale = pow(2.00, currentLevel);

        // Parent node
        if(myRank/scale <1){
            if((myRank+scale)<p){ // if child node exist (child node
rank < number of processors)
                pivot = length / 2; //Divide data length into half
                rightLength = length - pivot; // Set data length for
child node
                length = pivot; // Set new data length for parent node

                // Send child node length to the corresponding child
node

                // tag = currentLevel
                MPI_Send(&rightLength,1,MPI_INT,myRank+scale,
currentLevel,MPI_COMM_WORLD);

                // Send the right half of data array
```

```
                MPI_Send((int *) data+pivot, rightLength, MPI_INT,
myRank+scale, currentLevel, MPI_COMM_WORLD);
            }
        }
        // Child node
        else if(myRank/scale < 2){
            // Receive length from parent node
            MPI_Recv(&length, 1, MPI_INT, myRank-scale,currentLevel,
MPI_COMM_WORLD, &status); //tag = currentLevel

            // Create new dynamic data buffer with length received
from parent
            data = (int*)malloc(length * sizeof(int));

            // Receive data array from parent
            MPI_Recv(data, length, MPI_INT,myRank-scale,currentLevel,
MPI_COMM_WORLD,&status);
        }
    }

    // All processors mergeSort their own data chunk with respective
length
    mergeSort(data,0,length -1);

    // Merge the sorted data from child node to the root node starting
    // Begin the progress from the lowest level of the tree structure
    for(currentLevel = maxLevel;currentLevel>=0;currentLevel--){
        scale = pow(2.00, currentLevel);
        if(myRank/scale<1){ //Parent node receive sorted data from
child node
           if(myRank+scale<p) // If child node exist (child node rank
< number of processors)
           {
              MPI_Recv(&rightLength, 1, MPI_INT, myRank+scale,
currentLevel, MPI_COMM_WORLD,&status);
              MPI_Recv((int *) data+ length, rightLength, MPI_INT,
myRank+scale, currentLevel, MPI_COMM_WORLD, &status);

              merge(data, length, (int *)data+length, rightLength); //
Merge the data array
              length+=rightLength; // Update the length of merged data
array
           }
        }
        // Child node sends sorted data to parent node
        // tag = current level
        else if(myRank/scale<2)
        {
              // Send the length of sorted data
```

```
            MPI_Send(&length, 1, MPI_INT, myRank-scale,
currentLevel, MPI_COMM_WORLD);
            MPI_Send(data, length, MPI_INT, myRank-scale,
currentLevel, MPI_COMM_WORLD);
        }
    }

    // Root node prints out the sorted data
    if(myRank == 0){
       printf("Sorted Data: \n");
       for(i = 0; i<length;i++){
           if(i%10 == 0)
              printf("\n");
           printf("%d\t", data[i]);
       }
       printf("\n");
    }

    MPI_Finalize(); // Finalize MPI
    free(data);      // Free the data buffer

    return 0;
}
// End of main program

// Function mergeSort
void mergeSort(int* data, int startPoint, int endPoint)
{
     int pivot = (startPoint + endPoint)/2;

     // if last element then return
     if(startPoint == endPoint)
        return;
     //Recursive function call
     mergeSort(data, startPoint, pivot);
     mergeSort(data,pivot+1,endPoint);

     // Merge the sorted data from both sides into the data buffer
     merge(data+startPoint, pivot - startPoint +1, data+pivot+1,
endPoint-pivot);
}
// End of function mergeSort

// Function merge
void merge(int *A, int sizeA, int *B, int sizeB)
{
     int sizeC = sizeA + sizeB;
     int *C = (int*)malloc(sizeC * sizeof(int));
     int countA;
     int countB;
     int countC;
```

```
    // Merging the element from array A and array B into C in
ascending order
    for(countA = 0, countB = 0, countC = 0; countC< sizeC; countC++){
        if(countA>=sizeA) // If all the element from A is stored into
C
            C[countC] = B[countB++]; // store the remaining element
from B into C
        else if (countB>=sizeB) // If all the element from B is
stored into C
            C[countC] = A[countA++]; // store the remaining element
from A into C
        else
        {  // Store the element with smaller value into C then
increment the corresponding pointer array (A or B)
            if(A[countA]<=B[countB])
               C[countC] = A[countA++];
            else C[countC] = B[countB++];
        }
    }
    // Copy the merged data from C into A and B
    for(countA = 0; countA < sizeA; countA++)
        A[countA] = C[countA];

    for(countC = countA, countB = 0; countC < sizeC; countC++,
countB++)
        B[countB] = C[countC];

    // Free memory of C
    free(C);
}
// End of function merge
```

MONASH University

Task 3 – Analysing both Tasks 1 and 2

Analyse the sample solution in both Tasks 1 and 2. Complete the following:

a) Describe the Serial Merge Sort algorithm from Task 1. Include illustrations, pseudocode and/or flowcharts to describe the algorithm.
b) Modify the code in Task 1 to perform sorting of a large array of numbers. Write the sorted results to a file. Perform a theoretical speed up analysis of the serial Merge Sort Algorithm.
c) Describe the Parallel Merge Sort algorithm from Task 2. Include illustrations, pseudocode and/or flowcharts to describe the algorithm.
d) Modify the code in Task 2 to perform sorting of a large array of numbers. Write the sorted results to a file. Perform an actual speed up analysis and compare your results with the theoretical speed up.

Note: You may use CAAS to run Tasks 1 and 2.

Solution:

No specific solution provided here as long as students provide sufficient analysis for parts (a) to (d), please consult your tutor to check your answer.

MONASH University

Task 4 – Self Practice

Figure 1 represents a file (**data.txt**), which contains a set of unique product IDs and corresponding price. The first row in data.txt contains ***N*** total number of product records, and the subsequent ***N*** rows contain a list of product IDs and corresponding product price.

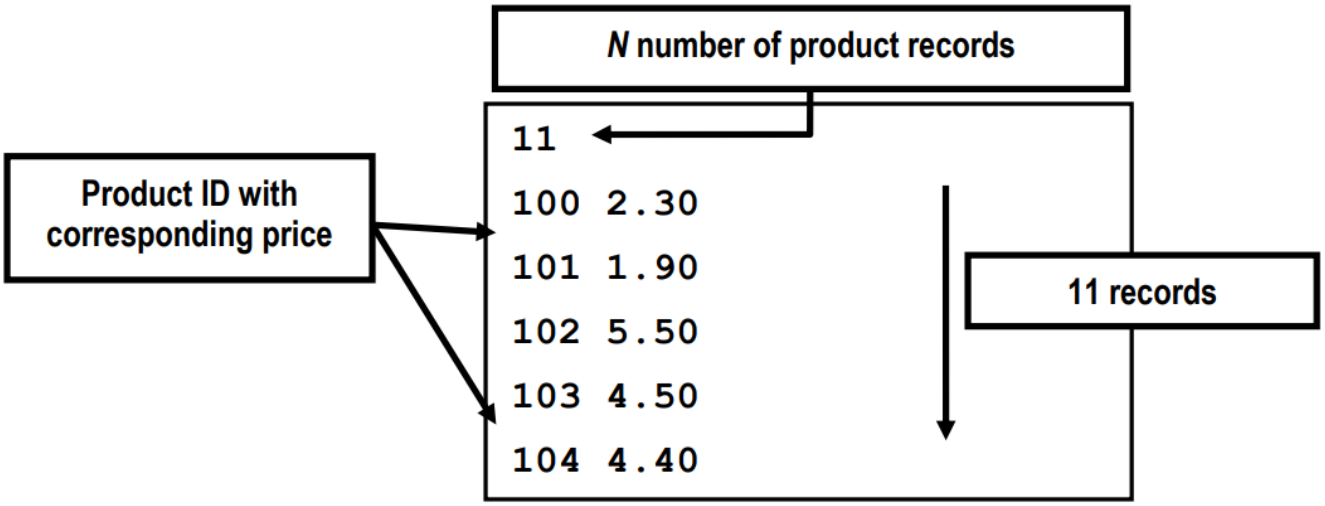
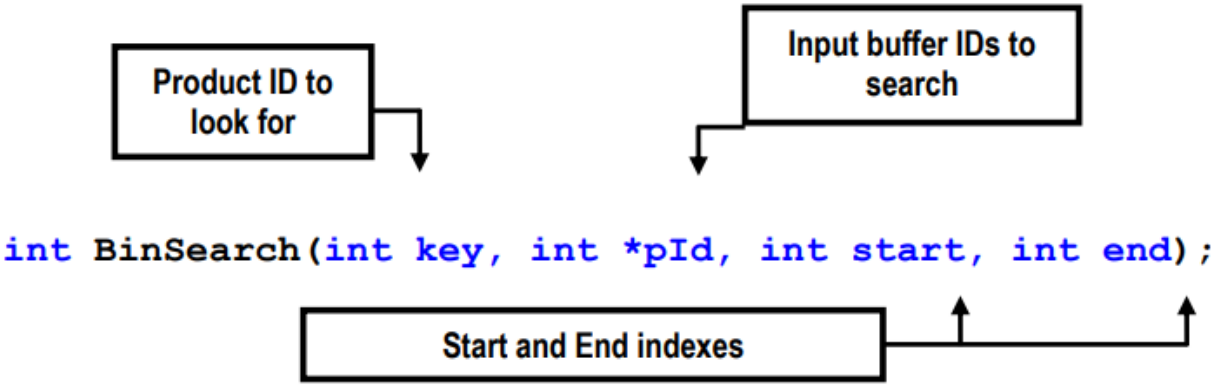


Figure 1: Content of data.txt

Note: The file, **data.txt**, is located only at the root node processor. Other processors cannot access this file. Create *your own* **data.txt** based on the sample format as per the above figure.

Using the C programming language:

(a) Write a function, `BinSearch()` to locate the index of a searched product based on an input product ID buffer using **binary search algorithm**.



The `BinSearch()` function will **return the index of the searched product** if a matching is found, and **-1** if the name is not found. The binary (iterative) search algorithm is as follows (You can also use the recursive method):

```
while start index < end index
    find the pivot (i.e. the index of the middle element of the range of elements to be
                    searched)
    compare the element at pivot with the key
        if element at pivot is less than the key
```

```
        end index = pivot – 1
    else if element at pivot is greater than the key
        start index = pivot + 1
    else
        return pivot
return -1
```

Note: There is no need to apply any MPI design in the **`BinSearch()`** function.

(b) Using the function of part (a), write a **`main()`** function **with MPI implementation** to do the following:

(i) The root node loads the contents of **data.txt** into two separate dynamic buffers. The first buffer contains the product IDs and the second buffer contains the corresponding product prices. The size of these buffers is based on the first item as read from the **data.txt** file.

As the root node can only access the content of **data.txt** file, hence, the content of the product ID buffer is to be equally divided among the processors.

(ii) Prompt the user for an input product ID to search. Only the root node can request an input product ID from the user.

(iii) Perform the binary search operation based on the equally divided content among the processors. Use the **`BinSearch()`** function as implemented in part (a).

Note: Do not modify the implementation of **`BinSearch()`** function.

(iv) The search results are returned to the root node. If a match is found, the root node prints the price of the searched product. Otherwise, the root node prints a message indicating that the search product ID does not exist.

Write parts (a) and (b) above as a single program. **Figures 2 & 3** display a simple execution of this program. Test your program using **OpenMPI** with three or four processes. Assume that there are no repetitions of the product IDs in **data.txt**.

```
Total products: 11
Number of processors: 4
Product ID to search >> 104
Price of product ID: 104 is RM 4.40.
```

Figure 2: Sample Program Execution

MONASH University

```
Total products: 11
Number of processors: 4
Product ID to search >> 7788
No such product.
```

Figure 3: Another sample Program Execution

Note: Underline statements in **Figures 2 & 3** denote a user's input.

Sample Solution:

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <mpi.h>

int BinSearchV2(int key, int *pId, int start, int end);

int main(int argc, char *argv[]){

    int myRank, totalProcs, totalIds = 0, tag = 0;
    int numPerProcs, numPerProcsRemain;
    int localResult = -1, actualResult = 0;
    FILE *pFile;
    int *pId = NULL;
    float *pPrice = NULL;
    int query = 0;
    int i = 0, j = 0, offset = 0;
    char fileName[256] = {0};

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &myRank);
    MPI_Comm_size(MPI_COMM_WORLD, &totalProcs);
    MPI_Status stat;

    if(myRank == 0){
       pFile = fopen("data.txt","r");
       fscanf(pFile, "%d", &totalIds);

       pId = (int*)malloc(totalIds * sizeof(int));
       pPrice = (float*) malloc(totalIds * sizeof(float));
       for(i = 0; i < totalIds; i++){
           fscanf(pFile,"%d%f",&pId[i],&pPrice[i]);
       }
       fclose(pFile);
    }
```

```
    MPI_Bcast(&totalIds, 1, MPI_INT, 0, MPI_COMM_WORLD);

    // Workload distribution
    numPerProcs = totalIds / totalProcs;
    numPerProcsRemain = totalIds % totalProcs;

    if(myRank != 0 && myRank != totalProcs-1){
       pId = (int*)malloc(numPerProcs * sizeof(int));
    }
    if(myRank == totalProcs-1){
       pId = (int*)malloc((numPerProcs + numPerProcsRemain) *
sizeof(int));
    }

    switch(myRank){
       case 0:{
            offset += numPerProcs;
            for(i = 1; i < totalProcs; i++){
                if(i != totalProcs-1){
                   MPI_Send(pId + offset, numPerProcs, MPI_INT, i, 0,
MPI_COMM_WORLD);
                   offset += numPerProcs;
                }else{
                   MPI_Send(pId + offset, numPerProcs +
numPerProcsRemain, MPI_INT, i, 0, MPI_COMM_WORLD);
                offset += numPerProcs + numPerProcsRemain;
                }
            }
            printf("Product ID to search >> ");
            fflush(stdout);
            scanf("%d", &query);
            break;
       }
       default:{
            if(myRank != totalProcs-1){
               MPI_Recv(pId, numPerProcs, MPI_INT, 0, 0,
MPI_COMM_WORLD, &stat);
            }else{
               MPI_Recv(pId, numPerProcs + numPerProcsRemain, MPI_INT,
0, 0, MPI_COMM_WORLD, &stat);
            }
            break;
       }
    }
    MPI_Bcast(&query, 1, MPI_INT, 0, MPI_COMM_WORLD);

    // Call the binary search function.
    if(myRank != totalProcs-1){
       localResult = BinSearchV2(query, pId, 0, numPerProcs-1);
    }else{
       localResult = BinSearchV2(query, pId, 0,
```

```
numPerProcs+numPerProcsRemain-1);
    }

    if(localResult > -1){
       localResult = localResult + (myRank * numPerProcs);
    }
    MPI_Reduce(&localResult, &actualResult, 1, MPI_INT, MPI_SUM, 0,
 MPI_COMM_WORLD);

    if(myRank == 0){
       if(actualResult > totalProcs * -1){
          actualResult = actualResult + (totalProcs - 1);
          printf("Price of product ID: %d is RM %.2f\n",
 pId[actualResult], pPrice[actualResult]);
          fflush(stdout);
       }else{
          puts("No such product.\n");
          fflush(stdout);
       }
       free(pId);
       free(pPrice);
    }else{
       free(pId);
    }
    MPI_Finalize();
    return 0;
 }

 // Binary search algorithm
 int BinSearchV2(int key, int *pId, int start, int end){
    int mid = (start + end) / 2;
    int result = key - pId[mid];

    if(result == 0) return(mid);
    if(start >= end) return(-1);

    if(result<0)
       return(BinSearchV2(key,pId,start,mid-1));
    else
       return(BinSearchV2(key,pId,mid+1,end));
 }
```

MONASH University

Task 5 – Matrix multiplication with AVX – Worked example

Modify the matrix multiplication codes given in Week 8's reading materials to perform SIMD (single instruction, multiple data) processing with Intel's Streaming SIMD Extensions (SSE). SSE is a set of instructions supported by Intel processors that perform high-speed operations on large chunks of data. Complete the following:

(a) Modify the MatrixGenerator.c to generate a N x M matrix with random cell float values and writes into a text file.

Sample Solution:

```
#include <stdio.h>
#include <stdlib.h>
#include <memory.h>
#include <time.h>
#include <immintrin.h>

#define MATRIX_CELL_MAXVAL 1000

int main()
{
    int row, col;
    int i,j;
    char matrixName[256] = {0};
    float randNum = 0;
    int maxVal = 100;

    printf("Welcome to the random matrix file generator!\n\n");

    printf("Specify the matrix file name (e.g. MatA.txt): ");
    scanf("%s", matrixName);

    printf("Specify the matrix row and col values (e.g. 10 10): ");
    scanf("%d%d", &row, &col);

    srand((unsigned int)time(NULL));

    FILE *pFile = fopen(matrixName, "w");
    fprintf(pFile, "%d\t%d\n", row, col);

    for(i = 0; i < row; i++){
        for(j = 0; j < col; j++){
            randNum = (float)rand()/(float)(RAND_MAX/maxVal);
            //randNum = 100 + ((unsigned int)rand() %
MATRIX_CELL_MAXVAL);
            fprintf(pFile, "%.2f\t", randNum);
        }
        fprintf(pFile, "\n");
    }

    printf("Done!\n");
```

```
    fclose(pFile);
    return 0;
}
```

(b) Using the output files from (a), complete the following:

(i) Modify the provided serial matrix multiplication code to operate on floating points.

(ii) Compile and run the code, observe the overall time taken to complete the process.

Sample Solution:

```
#include <stdio.h>
#include <stdlib.h>
#include <memory.h>
#include <time.h>

int main()
{
    // Variables
    int i = 0, j = 0, k = 0;

    /* Clock information */
    struct timespec start, end, startComp, endComp;
    double time_taken;

    // Get current clock time.
    clock_gettime(CLOCK_MONOTONIC, &start);

    // 1. Read Matrix A
    int rowA = 0, colA = 0;

    printf("Matrix Multiplication using 2-Dimension Arrays -
Start\n\n");

    printf("Reading Matrix A - Start\n");

    FILE *pFileA = fopen("MA.txt", "r");
    fscanf(pFileA, "%d%d", &rowA, &colA);

    float **ppMatrixA = (float**)malloc(rowA * sizeof(float*));
    for(i = 0; i < rowA; i++){
        ppMatrixA[i] = (float*)malloc(colA * sizeof(float));
    }

    for(i = 0; i < rowA; i++){
        for(j = 0; j < colA; j++){
            fscanf(pFileA, "%f", &ppMatrixA[i][j]);
        }

    }
    fclose(pFileA);

    printf("Reading Matrix A - Done\n");
```

```
    // 2. Read Matrix B
    int rowB = 0, colB = 0;

    printf("Reading Matrix B - Start\n");

    FILE *pFileB = fopen("MB.txt", "r");
    fscanf(pFileB, "%d%d", &rowB, &colB);

    float **ppMatrixB = (float**)malloc(rowB * sizeof(float*));
    for(i = 0; i < rowB; i++){
         ppMatrixB[i] = (float*)malloc(colB * sizeof(float));
    }

    for(i = 0; i < rowB; i++){
         for(j = 0; j < colB; j++){
              fscanf(pFileB, "%f", &ppMatrixB[i][j]);
         }

    }
    fclose(pFileB);

    printf("Reading Matrix B - Done\n");

    // 3. Perform matrix multiplication
    printf("Matrix Multiplication - Start\n");


    // Get current clock time.
    clock_gettime(CLOCK_MONOTONIC, &startComp);

    int rowC = rowA, colC = colB;
    float **ppMatrixC = (float**)calloc(rowC, sizeof(float*));
    for(i = 0; i < rowC; i++){
         ppMatrixC[i] = (float*)calloc(colC, sizeof(float));
    }

    int commonPoint = colA;
    __m256 num1;
    __m256 num2;
    __m256 result;


    for(i = 0; i < rowC; i++){
         for(j = 0; j < colC; j++){


              for(k = 0; k < commonPoint; k++){
                   ppMatrixC[i][j] += (ppMatrixA[i][k] *
ppMatrixB[k][j]);
              }
         }
    }
```

```
    // Get the clock current time again
    // Subtract end from start to get the CPU time used.
    clock_gettime(CLOCK_MONOTONIC, &endComp);
    time_taken = (endComp.tv_sec - startComp.tv_sec) * 1e9;
    time_taken = (time_taken + (endComp.tv_nsec - startComp.tv_nsec))
* 1e-9;

    printf("Matrix Multiplication - Done\n");
    printf("Matrix Multiplication - Process time (s): %lf\n",
time_taken);

    // 4. Write results to a new file
    printf("Write Resultant Matrix C to File - Start\n");

    FILE *pFileC = fopen("MC.txt", "w");
    fprintf(pFileC, "%d\t%d\n", rowC, colC);
    for(i = 0; i < rowC; i++){
         for(j = 0; j < colC; j++){
              fprintf(pFileC, "%.2f\t", ppMatrixC[i][j]);
         }
         fprintf(pFileC, "\n");
    }
    fclose(pFileC);
    printf("Write Resultant Matrix C to File - Done\n");

    // Clean up
    for(i = 0;  i < rowA; i++){
         free(ppMatrixA[i]);
    }
    free(ppMatrixA);
    for(i = 0;  i < rowB; i++){
         free(ppMatrixB[i]);
    }
    free(ppMatrixB);
    for(i = 0;  i < rowC; i++){
         free(ppMatrixC[i]);
    }
    free(ppMatrixC);

    // Get the clock current time again
    // Subtract end from start to get the CPU time used.
    clock_gettime(CLOCK_MONOTONIC, &end);
    time_taken = (end.tv_sec - start.tv_sec) * 1e9;
    time_taken = (time_taken + (end.tv_nsec - start.tv_nsec)) * 1e-9;
    printf("Matrix Multiplication using 2-Dimension Arrays - Done\n");
    printf("Overall time (Including read, multiple and write)(s):
%lf\n\n", time_taken);  // ts

    return 0;
}
```

MONASH University

(c) Replace the innermost loop of the matrix multiplication in (b) with the following to perform the operation using AVX intrinsic functions. Refer to this link for the available AVX intrinsic functions.

```
for(k = 0; k < commonPoint; k+=8){

                    num1 = _mm256_set_ps(ppMatrixA[i][k],
ppMatrixA[i][k+1], ppMatrixA[i][k+2], ppMatrixA[i][k+3],
ppMatrixA[i][k+4], ppMatrixA[i][k+5], ppMatrixA[i][k+6],
ppMatrixA[i][k+7]);
                    num2 = _mm256_set_ps(ppMatrixB[k][j],
ppMatrixB[k+1][j], ppMatrixB[k+2][j], ppMatrixB[k+3][j],
ppMatrixB[k+4][j], ppMatrixB[k+5][j], ppMatrixB[k+6][j],
ppMatrixB[k+7][j]);
                    result = _mm256_mul_ps(num1, num2);
                    for(int v = 0;  v < 8; v++)
                         ppMatrixC[i][j] += ((float*)&result)[v];
               }
```

Note: We encourage you to compile and execute your program on CAAS. Should you run your program in CAAS, please configure the job script. Observe the time difference between the local device and CAAS.

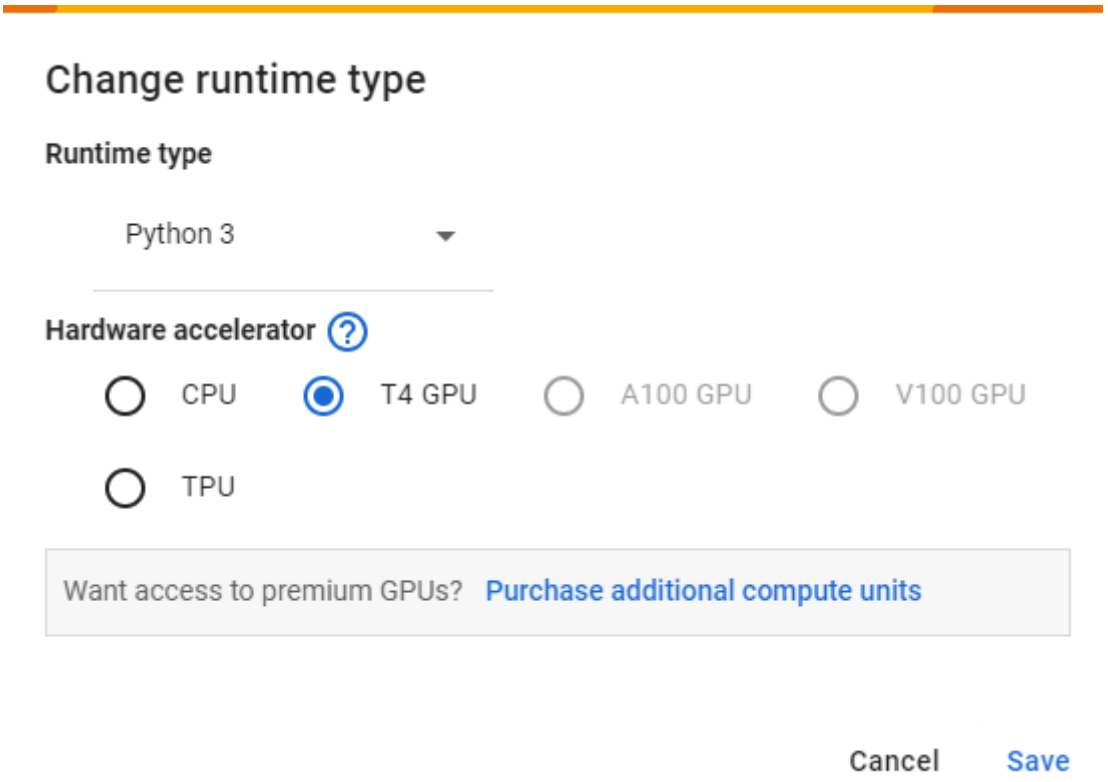
MONASH University

Task 6 – CUDA C practice on Google Collab – Worked example

Follow the instructions in this task to explore how to run CUDA C on Google Colab. Refer to this link for the instructions on C++.

(a) Go to https://colab.research.google.com in Browser and Click on New Notebook.

(b) Click to Runtime > Change > Hardware Accelerator > T4 GPU. This step switches the runtime from CPU to GPU.



(c) Check your CUDA version by running the command given below :

```
!nvcc --version
```

Output:

```
nvcc: NVIDIA (R) Cuda compiler driver Copyright (c) 2005-2022
NVIDIA Corporation Built on Wed_Sep_21_10:33:58_PDT_2022 Cuda
compilation tools, release 11.8, V11.8.89 Build
cuda_11.8.r11.8/compiler.31833905_0
```

(d) Run the command below to install a small extension to run nvcc from the Notebook cells:

```
!pip install git+https://github.com/andreinechaev/nvcc4jupyter.git
```

(e) Load the extension using the code given below:

```
%load_ext nvcc_plugin
```

(f) Run the following code:

```
%%cu
#include <stdio.h>
#include <stdlib.h>
__global__ void add(int *a, int *b, int *c) {
*c = *a + *b;
}
int main() {
int a, b, c;
// host copies of variables a, b & c
int *d_a, *d_b, *d_c;
// device copies of variables a, b & c
int size = sizeof(int);
// Allocate space for device copies of a, b, c
cudaMalloc((void **)&d_a, size);
cudaMalloc((void **)&d_b, size);
cudaMalloc((void **)&d_c, size);
// Setup input values
c = 0;
a = 3;
b = 5;
// Copy inputs to device
cudaMemcpy(d_a, &a, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_b, &b, size, cudaMemcpyHostToDevice);
// Launch add() kernel on GPU
add<<<1,1>>>(d_a, d_b, d_c);
// Copy result back to host
cudaError err = cudaMemcpy(&c, d_c, size, cudaMemcpyDeviceToHost);
if(err!=cudaSuccess) {
printf("CUDA error copying to Host: %s\n",
```

```
cudaGetErrorString(err));
}
printf("result is %d\n",c);
// Cleanup
cudaFree(d_a);
cudaFree(d_b);
cudaFree(d_c);
return 0;
}
```

Output:

```
result is 8
```

Note: You may change to a different number of blocks and threads to observe the time difference.