Parallel Programming Lab

NAINA CHABRA

210968234

B2

PARALLEL PROGRAMMING ASSIGNMENT 2

WEEK 6: (MPI) Message Passing Interface

Q1. Write a simple MPI program to find out pow (x,rank) for all the processes where 'x' is the integer constant and 'rank' is the rank of the process.

CODE:

```
#include <mpi.h>

#include <stdio.h>

#include <cmath>


using namespace std;


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

   MPI_Init(&argc, &argv);

   int x=10;

   int rank;

   MPI_Comm_rank(MPI_COMM_WORLD, &rank);

   printf("My rank is %d and pow(%d,%d) is %f", rank,x,rank,pow(x,rank));

   MPI_Finalize();

}
```

OUTPUT:

```
My rank is 4 and pow(10,4) is 10000.000000

My rank is 3 and pow(10,3) is 1000.000000

My rank is 0 and pow(10,0) is 1.000000

My rank is 1 and pow(10,1) is 10.000000

My rank is 2 and pow(10,2) is 100.000000
```

Q2) Write a simple MPI program where each even ranked process prints "hello" and odd ranked process "world".

CODE:

```
#include <mpi.h>
#include <stdio.h>
#include <cmath>

using namespace std;

int main(int argc, char** argv) {
        MPI_Init(&argc, &argv);
        int rank;
        MPI_Comm_rank(MPI_COMM_WORLD, &rank);
        if (rank % 2 == 0) {
                printf("My rank is %d and I print: Hello",rank);
        }
        else {
                printf("My rank is %d and I print: World", rank);
        }
        MPI_Finalize();
}
```

OUTPUT:

```
My rank is 4 and I print: Hello
My rank is 3 and I print: World
My rank is 0 and I print: Hello
My rank is 1 and I print: World
My rank is 2 and I print: Hello
```

Q3) Write a program in MPI to simulate simple calculator. Perform each operation using different processes in parallel.

CODE:

```
#include <mpi.h>
#include <stdio.h>
#include <cmath>

using namespace std;

int main(int argc, char** argv) {
    int a = 12;
    int b = 32;
    printf("a=%d b=%d\n", a, b);
    MPI_Init(&argc, &argv);
    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    if (rank == 0) {
            printf("My rank is %d I perform addition a+b=%d", rank, a + b);
    }
    else if (rank == 1) {
            printf("My rank is %d I perform subtraction a-b=%d", rank, a - b);
    }
    else if (rank == 2) {
            printf("My rank is %d I perform multiplication a*b=%d", rank, a * b);
    }
    else {
            printf("My rank is %d I perform division a/b=%d", rank, a / b);
    }
    MPI_Finalize();
}
```

OUTPUT:

```
a=12 b=32

My rank is 0 I perform addition a+b=44

a=12 b=32

My rank is 1 I perform subtraction a-b=-20

a=12 b=32

My rank is 2 I perform multiplication a*b=384
```

```
a=12 b=32
My rank is 3 I perform division a/b=0
```

Q4) Write a program in MPI to toggle characters of a string by the respective rank.

CODE:

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

using namespace std;

int main(int argc, char** argv) {
  int rank;
        MPI_Init(&argc, &argv);
        MPI_Comm_rank(MPI_COMM_WORLD, &rank);
        char s[] = "Hello";
        for(int i=0;i<5;i++){
     if (s[i] >= 65 && s[i] <= 90) {
        if (i == rank) {
           printf(" %c toggled to %c by process %d ", s[i], s[i] + 32, rank);
           i++;
        }
     }
     else {
        if (i == rank) {
           printf(" %c toggled to %c by process %d ", s[i], s[i] - 32, rank);
           i++;
        }
        }
            }
            MPI_Finalize();
    return 0;
}
```

OUTPUT:

```
H toggled to h by process 0
e toggled to E by process 1
l toggled to L by process 2
l toggled to L by process 3
o toggled to O by process 4
```

WEEK 7: (MPI) Message Passing Interface

Q1) MPI program using synchronous send, sender sends a message to second process, the second process will toggle each character and then send it back to the sender.

CODE:

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

using namespace std;

int main(int argc, char** argv) {
   int rank;
        MPI_Init(&argc, &argv);
        MPI_Comm_rank(MPI_COMM_WORLD, &rank);
        char message[50];
        sprintf_s(message, "hello");
        MPI_Status status;
        if (rank == 0) {
                printf("sending message to 1 , message=%s\n",message);
                MPI_Ssend(message, 6, MPI_CHAR, 1, 0, MPI_COMM_WORLD);
        }
        if (rank == 1) {
                MPI_Recv(message, 6, MPI_CHAR, 0, 0, MPI_COMM_WORLD,status);
                printf("\nmessage recieved at process %d rank", rank);
                printf("\nmessage is %s", message);
                for (int i = 0; message[i] != '\0'; i++) {
                    if (message[i] >= 65 && message[i] <= 90) {
                            message[i] = message[i] + 32;
                    }
                    else {
                            message[i] = message[i] - 32;
                    }
            }
            MPI_Ssend(message, 6,MPI_CHAR,0,0,MPI_COMM_WORLD);
    }
    if (rank == 0) {
            MPI_Recv(message, 6, MPI_CHAR, 1, 0, MPI_COMM_WORLD, status);
            printf("\nmessage received at 0 after toggling message = %s",message);
    }
    MPI_Finalize();
  return 0;
}
```

OUTPUT:

```
sending message to 1 , message=hello
message received at 0 after toggling message = HELLO
message received at process 1 rank
message is hello
```

Q2) Write a program where the master process sends a message to all slave process and slave process receive the message and print it out.

CODE:

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

```
#include <cmath>
#include <string>

using namespace std;

int main(int argc, char** argv) {
   int rank;
         MPI_Init(&argc, &argv);
         MPI_Comm_rank(MPI_COMM_WORLD, &rank);
         char message[50];
         sprintf_s(message, "hello");
         MPI_Status status;

         if (rank == 0) {
                  printf("sending message to all, message=%s\n",message);
                  MPI_Send(message, 6, MPI_CHAR, 1, 0, MPI_COMM_WORLD);
                  MPI_Send(message, 6, MPI_CHAR, 2, 0, MPI_COMM_WORLD);
                  MPI_Send(message, 6, MPI_CHAR, 3, 0, MPI_COMM_WORLD);
                  MPI_Send(message, 6, MPI_CHAR, 4, 0, MPI_COMM_WORLD);
                  MPI_Send(message, 6, MPI_CHAR, 5, 0, MPI_COMM_WORLD);
         }
         if (rank == 1) {
                  char message1[50];
                  MPI_Recv(message1,    6,MPI_CHAR,    0,    0,    MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
                  printf("message recieved by rank %d message = %s", rank,message1);
         }
         if (rank == 2) {
                  char message1[50];
                  MPI_Recv(message1,    6,    MPI_CHAR,    0,    0,    MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
```

```
			printf("message recieved by rank %d message = %s", rank, message1);
		}
		if (rank == 3) {
			char message1[50];
			MPI_Recv(message1,	6,	MPI_CHAR,	0,	0,	MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
			printf("message recieved by rank %d message = %s", rank, message1);
		}
		if (rank == 4) {
			char message1[50];
			MPI_Recv(message1,	6,	MPI_CHAR,	0,	0,	MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
			printf("message recieved by rank %d message = %s", rank, message1);
		}
		if (rank == 5) {
			char message1[50];
			MPI_Recv(message1, 6, MPI_CHAR, 0, 0, MPI_COMM_WORLD, status);
			printf("message recieved by rank %d message = %s", rank, message1);
		}
		MPI_Finalize();
   return 0;
}
```

OUTPUT:

```
sending message to all, message=hello
message recieved by rank 5 message = hello
message recieved by rank 3 message = hello
message recieved by rank 4 message = hello
message recieved by rank 2 message = hello
message recieved by rank 1 message = hello
```

Q3) Write an MPI Program to read N numbers of the array in the root process where N is equal to the total number of process. The root process sends one value to each of the slaves. Let even number process find the square of it and odd one to find the cube of the same.

CODE:

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

using namespace std;

int main(int argc, char** argv) {
   int rank;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Status status;
    int value;
    int arr[8];
    if (rank == 0) {
            for (int i = 0; i < 8; i++) {
                    scanf_s("%d", &arr[i]);
            }
            int buffer_size = 8 * sizeof(int) + MPI_BSEND_OVERHEAD;
            MPI_Buffer_attach(malloc(buffer_size), buffer_size);
            for (int i = 1; i < 8; ++i) {
                    MPI_Bsend(&arr[i - 1], 1, MPI_INT, i, 0, MPI_COMM_WORLD);
                    printf("Master sent value %d to process %d\n", arr[i - 1], i);
            }
    }
    else {
            MPI_Recv(&value, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);
            printf("Process %d received value %d\n", rank, value);
            if(rank%2==0){
                    int square = value*value;
                    printf("The square of the value is %d",square");
            }
            else{
                    int cube = value*value*value;
                    printf("The cube of the value is %d",cube);
}

    }
    MPI_Finalize();
```

```
        return 0;
    }
```

OUTPUT:

```
1 2 3 4 5 6 7 8
Process 5 received value 5
Cube of the value is 125
Process 3 received value 3
Cube of the value is 27
Master sent value 1 to process 1
Master sent value 2 to process 2
Master sent value 3 to process 3
Master sent value 4 to process 4
Master sent value 5 to process 5
Master sent value 6 to process 6
Master sent value 7 to process 7
Process 6 received value 6
Square of the value is 36
Process 1 received value 1
Cube of the value is 1
Process 4 received value 4
Square of the value is 16
Process 7 received value 7
Cube of the value is 343
Process 2 received value 2
Square of the value is 4
```

Q4) Write a MPI program to read an integer value in the root process. Root process sends this value to process1, process1 sends this value to process2 and so on. Last process sends the value back to root process. when sending the value each process will first increment the received value by 1. Write a program using point to point communication routines

CODE:

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

int main(int argc, char** argv) {
    int rank, size;
    int value = 0;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    if (rank == 0) {
        printf("Enter an integer value: ");
        fflush(stdout);
        scanf_s("%d", &value);
        printf("Process 0 sending value %d\n", value);
        value++;
        MPI_Send(&value, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
    }
    else if (rank == size - 1) {
        MPI_Recv(&value, 1, MPI_INT, rank - 1, 0, MPI_COMM_WORLD, 
MPI_STATUS_IGNORE);
        printf("Process %d received value %d\n", rank, value);
        value++;
        printf("Process %d sending value %d\n", rank, value);
        MPI_Send(&value, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
    }
    else {
        MPI_Recv(&value, 1, MPI_INT, rank - 1, 0, MPI_COMM_WORLD, 
MPI_STATUS_IGNORE);
```

```
        printf("Process %d received value %d\n", rank, value);
        value++;
        printf("Process %d sending value %d\n", rank, value);
        MPI_Send(&value, 1, MPI_INT, rank + 1, 0, MPI_COMM_WORLD);
    }
    if (rank == 0) {
        MPI_Recv(&value, 1, MPI_INT, size - 1, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        printf("Process 0 received value %d\n", value);
    }
    MPI_Finalize();
    return 0;
}
```

OUTPUT:

```
Enter an integer value: 2
Process 6 received value 8
Process 6 sending value 9
Process 1 received value 3
Process 1 sending value 4
Process 4 received value 6
Process 4 sending value 7
Process 2 received value 4
Process 2 sending value 5
Process 0 sending value 2
Process 0 received value 10
Process 5 received value 7
Process 5 sending value 8
Process 7 received value 9
Process 7 sending value 10
Process 3 received value 5
Process 3 sending value 6
```

Q5) Write an MPI program to read N elements of an array in the master process. Let N processes including master process check the array values are prime or not.

CODE:

```
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
#include <cmath>
#define ARRAY_SIZE 8

int main(int argc, char** argv) {
    int rank, size;
    int data[ARRAY_SIZE];
    int recv_value;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    if (size < ARRAY_SIZE) {
        if (rank == 0)
            fprintf(stderr, "Error: This program requires %d processes.\n", ARRAY_SIZE);
        MPI_Finalize();
        return 1;
    }
    if (rank == 0) {
        printf("Enter %d elements:\n", ARRAY_SIZE);
        for (int i = 0; i < ARRAY_SIZE; ++i)
            scanf_s("%d", &data[i]);
    }

    MPI_Scatter(data, 1, MPI_INT, &recv_value, 1, MPI_INT, 0, MPI_COMM_WORLD);

    printf("Process %d received value %d\n", rank, recv_value);
    if (recv_value <= 1) {
        printf("%d is not prime, recieved by rank %d\n", recv_value, rank);
    }
    else {
        bool flag = true;
        for (int i = 2; i <= sqrt(recv_value); ++i) {
            if (recv_value % i == 0) {
                flag = false;
                break;
            }
        }
        if (flag) {
            printf("%d is prime, recieved by rank %d\n", recv_value, rank);
        }
        else {
            printf("%d is not prime, recieved by rank %d\n", recv_value, rank);
        }
    }
    MPI_Finalize();
    return 0;
}
```

OUTPUT:

```
1 2 3 4 5 6 7 8
Enter 8 elements:
Process 0 received value 1
1 is not prime, recieved by rank 0
Process 4 received value 5
5 is prime, recieved by rank 4
Process 5 received value 6
6 is not prime, recieved by rank 5
Process 7 received value 8
8 is not prime, recieved by rank 7
Process 2 received value 3
3 is prime, recieved by rank 2
Process 6 received value 7
7 is prime, recieved by rank 6
Process 1 received value 2
2 is prime, recieved by rank 1
Process 3 received value 4
4 is not prime, recieved by rank 3
```

Week 8: CUDA

Q1) Write a program to add 2 vectors of length N using

i)Block size as N　　　ii)N threads

CODE:

i)

```
#include "cuda_runtime.h"
#include "device_launch_parameters.h"
#include <stdio.h>

__global__ void addKernel(int *a,int *b,int* c,int n){
    int i = threadIdx.x;
    if (i < n) {
        c[i] = a[i] + b[i];
    }
}

int main() {
    printf("Enter n value");
    const int n=5;
    int arr[n];
    int arr1[n];
    int res[n];
    for (int i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
    }
    for (int i = 0; i < n; i++) {
        scanf("%d", &arr1[i]);
    }
    int* a;
    int* b;
    int* c;
    cudaMalloc(&a, sizeof(int) * n);
    cudaMalloc(&b, sizeof(int) * n);
    cudaMalloc(&c, sizeof(int) * n);

    cudaMemcpy(a, arr, sizeof(int) * n, cudaMemcpyHostToDevice);
    cudaMemcpy(b, arr1, sizeof(int) * n, cudaMemcpyHostToDevice);

    addKernel << <n, 128 >> >(a,b,c,n);

    cudaMemcpy(res, c, sizeof(int) * n, cudaMemcpyDeviceToHost);

    for (int i = 0; i < n; i++) {
        printf("%d ", res[i]);
    }
}
```

ii)

```
#include "cuda_runtime.h"
#include "device_launch_parameters.h"
#include <stdio.h>

__global__ void addKernel(int *a,int *b,int* c,int n){
    int i = threadIdx.x;
    c[i] = a[i] + b[i];
}
```

```
int main() {
    printf("Enter n value");
    const int n=5;
    int arr[n];
    int arr1[n];
    int res[n];
    for (int i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
    }
    for (int i = 0; i < n; i++) {
        scanf("%d", &arr1[i]);
    }
    int* a;
    int* b;
    int* c;
    cudaMalloc(&a, sizeof(int) * n);
    cudaMalloc(&b, sizeof(int) * n);
    cudaMalloc(&c, sizeof(int) * n);

    cudaMemcpy(a, arr, sizeof(int) * n, cudaMemcpyHostToDevice);
    cudaMemcpy(b, arr1, sizeof(int) * n, cudaMemcpyHostToDevice);

    addKernel << <1, n >> >(a,b,c,n);
    cudaMemcpy(res, c, sizeof(int) * n, cudaMemcpyDeviceToHost);

    for (int i = 0; i < n; i++) {
        printf("%d ", res[i]);
    }
}
```

OUTPUT:

```
i) Arr1: 1 2 3 4 5
Arr2: 6 7 8 9 10
RESULT: 7 9 11 13 15

ii) Arr1: 1 2 3 4 5
Arr2: 6 7 8 9 10
RESULT: 7 9 11 13 15
```

Q2) Implement a CUDA program to add 2 vectors of length N by keeping the number of threads per block as 256 and vary the number of blocks to handle the N elements.

CODE:

```
#include "cuda_runtime.h"
#include "device_launch_parameters.h"
#include <stdio.h>

__global__ void addKernel(int *a,int *b,int* c,int n){
    int i = blockIdx.x*blockDim.x+threadIdx.x;
    if (i < n) {
        c[i] = a[i] + b[i];
    }
}

int main() {
    const int n=453;
    int arr[n];
    int arr1[n];
    int res[n];
    for (int i = 0; i < 453; i++) {
        arr[i]=i;
    }
    for (int i = 0; i < 453; i++) {
        arr1[i]=i;
    }
    int* a;
    int* b;
    int* c;
    cudaMalloc(&a, sizeof(int) * n);
    cudaMalloc(&b, sizeof(int) * n);
    cudaMalloc(&c, sizeof(int) * n);

    cudaMemcpy(a, arr, sizeof(int) * n, cudaMemcpyHostToDevice);
    cudaMemcpy(b, arr1, sizeof(int) * n, cudaMemcpyHostToDevice);

    addKernel << <(n/256)+1, 256 >> >(a,b,c,n);

    cudaMemcpy(res, c, sizeof(int) * n, cudaMemcpyDeviceToHost);

    for (int i = 0; i < n; i++) {
        printf("%d ", res[i]);
    }
}
```

OUTPUT:

```
0 2 4 6 8 10 12 14 16 18 20 22 24 26 28 30 32 34 36 38 40 42 44 46 48 50 52 54 56 58 60 62
64 66 68 70 72 74 76 78 80 82 84 86 88 90 92 94 96 98 100 102 104 106 108 110 112 114 116
118 120 122 124 126 128 130 132 134 136 138 140 142 144 146 148 150 152 154 156 158
160 162 164 166 168 170 172 174 176 178 180 182 184 186 188 190 192 194 196 198 200
202 204 206 208 210 212 214 216 218 220 222 224 226 228 230 232 234 236 238 240 242
244 246 248 250 252 254 256 258 260 262 264 266 268 270 272 274 276 278 280 282 284
286 288 290 292 294 296 298 300 302 304 306 308 310 312 314 316 318 320 322 324 326
328 330 332 334 336 338 340 342 344 346 348 350 352 354 356 358 360 362 364 366 368
370 372 374 376 378 380 382 384 386 388 390 392 394 396 398 400 402 404 406 408 410
```

```
412 414 416 418 420 422 424 426 428 430 432 434 436 438 440 442 444 446 448 450 452
454 456 458 460 462 464 466 468 470 472 474 476 478 480 482 484 486 488 490 492 494
496 498 500 502 504 506 508 510 512 514 516 518 520 522 524 526 528 530 532 534 536
538 540 542 544 546 548 550 552 554 556 558 560 562 564 566 568 570 572 574 576 578
580 582 584 586 588 590 592 594 596 598 600 602 604 606 608 610 612 614 616 618 620
622 624 626 628 630 632 634 636 638 640 642 644 646 648 650 652 654 656 658 660 662
664 666 668 670 672 674 676 678 680 682 684 686 688 690 692 694 696 698 700 702 704
706 708 710 712 714 716 718 720 722 724 726 728 730 732 734 736 738 740 742 744 746
748 750 752 754 756 758 760 762 764 766 768 770 772 774 776 778 780 782 784 786 788
790 792 794 796 798 800 802 804 806 808 810 812 814 816 818 820 822 824 826 828 830
832 834 836 838 840 842 844 846 848 850 852 854 856 858 860 862 864 866 868 870 872
874 876 878 880 882 884 886 888 890 892 894 896 898 900 902 904
```

Q3) Write a program in CUDA which performs convolution operation on one dimensional input array N of size width using a mask array M of size mask_width to produce the resultant one-dimensional array P of size width

CODE:

```
#include "cuda_runtime.h"
#include "device_launch_parameters.h"
#include <stdio.h>

__global__ void conv(int *a,int *b,int* c,int m,int n){
   int i = blockIdx.x*blockDim.x+threadIdx.x;
   if (i < n-m+1) {
      int prod = 0;
      int k = 0;
      while(k<m){
         prod += a[i + k] * b[k];
         k++;
      }
      c[i] = prod;
   }
}

int main() {
```

```
int n=10;
int arr[10];
for (int i = 0; i < n; i++) {
    arr[i]=i;
    printf("%d ", arr[i]);
}
printf("\n");
int m = 2;
int mask[2];
for (int i = 0; i < m; i++) {
    mask[i]=i;
    printf("%d ", mask[i]);
}
printf("\n");
int res[9];
int* a;
int* b;
int* c;

cudaMalloc(&a, sizeof(int) * n);
cudaMalloc(&b, sizeof(int) * m);
cudaMalloc(&c, sizeof(int) * (n-m+1));

cudaMemcpy(a, arr, sizeof(int) * n, cudaMemcpyHostToDevice);
cudaMemcpy(b, mask, sizeof(int) * m, cudaMemcpyHostToDevice);

conv << <1, 256 >> > (a, b, c,m,n);

cudaMemcpy(res, c, sizeof(int) * (n-m+1), cudaMemcpyDeviceToHost);

for (int i = 0; i < (n-m+1); i++) {
    printf("%d ", res[i]);
    }

}
```

OUTPUT:

```
0 1 2 3 4 5 6 7 8 9
0 1
1 2 3 4 5 6 7 8 9
```

Q4) Write a program in CUDA to process a 1D array containing angles in radians to generate sine of the angles in the output array. Use appropriate function

CODE:

```
#include "cuda_runtime.h"
#include "device_launch_parameters.h"
#include <stdio.h>
#include <cmath>

__global__ void conv(float *a,float* b,int n){
    int i = blockIdx.x*blockDim.x+threadIdx.x;
    if (i < n) {
        b[i] = sinf(a[i]);
    }
}

int main() {
    const int n=10;
    float arr[10];
    printf("Enter the value in radians");
    for (int i = 0; i < n; i++) {
        scanf("%f", &arr[i]);
    }
    float res[n];
    float* a;
    float* b;

    cudaMalloc(&a, sizeof(float) * n);
    cudaMalloc(&b, sizeof(float) * n);
    cudaMemcpy(a, arr, sizeof(float) * n, cudaMemcpyHostToDevice);

    conv << <1, 256 >> > (a, b, n);
    cudaMemcpy(res, b, sizeof(float) * n, cudaMemcpyDeviceToHost);

    for (int i = 0; i < n; i++) {
        printf("%f ", res[i]);
    }
}
```

OUTPUT:

```
Enter the value in radians
4
3
2
45
6
6
7
86
5
4
-0.756802 0.141120 0.909297 0.850904 -0.279415 -0.279415 0.656987 -0.923458 -0.958924
-0.75680
```

WEEK 9: CUDA

Q1) Write a program in CUDA to count the number of times a given word is repeated in a sentence. (Use atomic function)

CODE:

```
#include <cuda_runtime.h>
#include "device_launch_parameters.h"
#include <stdio.h>
#include <iostream>
#include <stdlib.h>
#include <conio.h>
using namespace std;

#define THREADS_PER_BLOCK 256

__global__ void countWordOccurrences(char* sentence, char* word, int* result, int
sentenceLength, int wordLength) {
    int tid = threadIdx.x + blockIdx.x * blockDim.x;

    if (tid < sentenceLength) {
        int count = 0;
        bool found = true;

        for (int i = 0; i < wordLength; ++i) {
            if (sentence[tid + i] != word[i]) {
                found = false;
                break;
            }
        }
        if (found)
        {
            atomicAdd(result, 1);
            //tid += wordLength;  // Jump to the next word to avoid counting overlaps
        }
        else {
            //tid++;
        }
    }
}

int main() {
    char sentence[256], word[32];
    int sentenceLength, wordLength, result = 0;
    char* d_sentence, * d_word;
    int* d_result;

    printf("Enter the sentence: ");
    fgets(sentence, 256, stdin);
    printf("Enter the word to count: ");
    fgets(word, 32, stdin);

    sentenceLength = strlen(sentence);
    wordLength = strlen(word) - 1;  // -1 to exclude the newline character
    sentence[sentenceLength - 1] = '\0'; // Removing the newline character from the sentence

    cudaMalloc((void**)&d_sentence, sentenceLength * sizeof(char));
    cudaMalloc((void**)&d_word, wordLength * sizeof(char));
    cudaMalloc((void**)&d_result, sizeof(int));
    cudaMemcpy(d_sentence, sentence, sentenceLength * sizeof(char),
cudaMemcpyHostToDevice);
    cudaMemcpy(d_word, word, wordLength * sizeof(char), cudaMemcpyHostToDevice);
    cudaMemcpy(d_result, &result, sizeof(int), cudaMemcpyHostToDevice);


    int blocks = (sentenceLength + THREADS_PER_BLOCK - 1) /
THREADS_PER_BLOCK;


    countWordOccurrences << <blocks, THREADS_PER_BLOCK >> > (d_sentence, d_word,
d_result, sentenceLength, wordLength);


    cudaMemcpy(&result, d_result, sizeof(int), cudaMemcpyDeviceToHost);


    printf("Occurrences of \"%s\" in the sentence: %d\n", word, result);


    cudaFree(d_sentence);
    cudaFree(d_word);
    cudaFree(d_result);


    return 0;
}
```

OUTPUT:

```
Enter the sentence: is is is is a a a a
Enter the word to count: a
Occurrences of "a" in the sentence: 4
```

Q2) Write a CUDA program that reads a string S and produces the string RS as follows:
Input string S: PCAP
Output string RS: PCAPPCAPCP
Note: Each work item copies required number of characters from S in RS.

CODE:

```
#include <stdio.h>

#include <stdlib.h>

#include "string.h"

#include <cuda.h>

#include "cuda_runtime.h"

#include "device_launch_parameters.h"

#include <iostream>

#include <cuda_runtime.h>


__global__ void copyStringInDecreasingPattern(char* d_RS, const char* d_S, int
stringLength, int totalLength) {

    int idx = threadIdx.x + blockIdx.x * blockDim.x;

    int k = 0;

    if (idx < stringLength) {

        int copyLength = stringLength - idx;

        for (int j = copyLength; j >= 0; j--) {

            for (int i = 0; i <= j; i++) {

                d_RS[idx + k] = d_S[i];

                k++;

            }

        }

    }

}


int main() {

    std::string S="PCAP";

    // std::cout << "Enter a string: ";

    // std::getline(std::cin, S); // Corrected to read a line with spaces

    int stringLength = S.length();

    int totalLength = stringLength * (stringLength + 1) / 2;
    char* d_S;
    char* d_RS;

    cudaMalloc(&d_S, sizeof(char) * (stringLength + 1));
    cudaMalloc(&d_RS, sizeof(char) * (totalLength + 1));

    cudaMemcpy(d_S, S.c_str(), sizeof(char) * (stringLength + 1),
cudaMemcpyHostToDevice);

    // Initialize d_RS to zero to ensure clean output
    cudaMemset(d_RS, 0, sizeof(char) * (totalLength + 1));

    int threadsPerBlock = 256;
    int blocksPerGrid = (stringLength + threadsPerBlock - 1) / threadsPerBlock;

    copyStringInDecreasingPattern << <blocksPerGrid, threadsPerBlock >> > (d_RS, d_S,
stringLength, totalLength);

    char* RS = new char[totalLength + 1];
    cudaMemcpy(RS, d_RS, sizeof(char) * (totalLength + 1), cudaMemcpyDeviceToHost);

    std::cout << "Input String: S: " << S << std::endl;
    std::cout << "Output String: RS: " << RS << std::endl;

    delete[] RS;
    cudaFree(d_S);
    cudaFree(d_RS);

    return 0;
}
```

OUTPUT:

```
Input String: S: PCAP
Output String: RS: PPPCPCAPAP
```

WEEK 10: CUDA – Matrix multiplication

Q1) Write a program in CUDA to add two matrices for the following specifications:

- *Each row of resultant matrix to be computed by one thread.*
- *Each column of resultant matrix to be computed by one thread*
- *Each element of resultant matrix to be computed by one thread*

CODE:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <cuda.h>
#include "cuda_runtime.h"
#include "device_launch_parameters.h"

// Kernel to add matrices element by element
__global__ void addMatricesElementByElement(float* A, float* B, float* C, int N) {
   int row = threadIdx.y + blockDim.y * blockIdx.y;
   int col = threadIdx.x + blockDim.x * blockIdx.x;
   if (row < N && col < N) {
      C[N * row + col] = A[N * row + col] + B[N * row + col];
   }
}

int main() {
   int N = 4; // Size of the matrices
   float* h_A = (float*)malloc(N * N * sizeof(float));
   float* h_B = (float*)malloc(N * N * sizeof(float));
   float* h_C = (float*)malloc(N * N * sizeof(float));
```

```
    memset(h_A, 0, N * N * sizeof(float));
    memset(h_B, 0, N * N * sizeof(float));
    memset(h_C, 0, N * N * sizeof(float));
    float* d_A, * d_B, * d_C; // Device matrices

    // Initialize host matrices
    for (int i = 0; i < N * N; i++) {
        h_A[i] = i;
        h_B[i] = i;
    }

    // Allocate device memory
    cudaMalloc((void**)&d_A, N * N * sizeof(float));
    cudaMalloc((void**)&d_B, N * N * sizeof(float));
    cudaMalloc((void**)&d_C, N * N * sizeof(float));

    // Copy host matrices to device
    cudaMemcpy(d_A, h_A, N * N * sizeof(float), cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, N * N * sizeof(float), cudaMemcpyHostToDevice);

    // Define block and grid sizes
    dim3 threadsPerBlock(32, 32);
    dim3 numBlocks((N + threadsPerBlock.x - 1) / threadsPerBlock.x, (N + threadsPerBlock.y
- 1) / threadsPerBlock.y);

    // Launch the kernel
    addMatricesElementByElement << <numBlocks, threadsPerBlock >> > (d_A, d_B, d_C,
N);

    // Synchronize to ensure all kernels have completed
    cudaDeviceSynchronize();
    // Copy result back to host
    cudaMemcpy(h_C, d_C, N * N * sizeof(float), cudaMemcpyDeviceToHost);

    // Print the resultant matrix
    printf("Resultant matrix:\n");
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            printf("%f ", h_C[i * N + j]);
        }
        printf("\n")
    }

    // Free device memory
    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);

    // Free host memory
    free(h_A);
    free(h_B);
    free(h_C);

    return 0;
}
```

OUTPUT:

```
Resultant matrix:
0.000000 2.000000 4.000000 6.000000
8.000000 10.000000 12.000000 14.000000
```

```
16.000000 18.000000 20.000000 22.000000
24.000000 26.000000 28.000000 30.000000
```

Q2 Write a program in CUDA to multiply two matrices for the following specifications:

- *Each row of resultant matrix to be computed by one thread.*
- *Each column of resultant matrix to be computed by one thread*
- *Each element of resultant matrix to be computed by one thread*

CODE:

```
#include <stdio.h>
#include <stdlib.h>
#include <cuda_runtime.h>

// Kernel to compute each row of the resultant matrix by one thread
__global__ void matrixMultiplyRow(float* A, float* B, float* C, int width, int P, int Q) {
    int row = blockIdx.x;
    if (row < P) {
        for (int col = 0; col < Q; col++) {
            float sum = 0;
            for (int k = 0; k < width; k++) {
                sum += A[row * width + k] * B[k * Q + col];
            }
            C[row * Q + col] = sum;
        }
    }
}

// Kernel to compute each column of the resultant matrix by one thread
__global__ void matrixMultiplyColumn(float* A, float* B, float* C, int width, int P, int Q) {
    int col = blockIdx.x;
    if (col < Q) {
        for (int row = 0; row < P; row++) {
            float sum = 0;
            for (int k = 0; k < width; k++) {
                sum += A[row * width + k] * B[k * Q + col];
            }
            C[row * Q + col] = sum;
        }
    }
}

// Kernel to compute each element of the resultant matrix by one thread
__global__ void matrixMultiplyElement(float* A, float* B, float* C, int width, int P, int Q) {
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    if (row < P && col < Q) {
        float sum = 0;
        for (int k = 0; k < width; k++) {
            sum += A[row * width + k] * B[k * Q + col];
        }
        C[row * Q + col] = sum;
    }
}

int main() {
    int N = 4; // Size of the matrices
    int SIZE = N * N;

    // Allocate memory on the host
    float* h_A = (float*)malloc(SIZE * sizeof(float));
    float* h_B = (float*)malloc(SIZE * sizeof(float));
    float* h_C = (float*)malloc(SIZE * sizeof(float));

    // Initialize matrices on the host
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            h_A[i * N + j] = i + 1;
            h_B[i * N + j] = j + 1;
        }
    }

    // Allocate memory on the device
    float* d_A, * d_B, * d_C;
    cudaMalloc((void**)&d_A, SIZE * sizeof(float));
    cudaMalloc((void**)&d_B, SIZE * sizeof(float));
    cudaMalloc((void**)&d_C, SIZE * sizeof(float));

    // Copy matrices to the device
    cudaMemcpy(d_A, h_A, SIZE * sizeof(float), cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, SIZE * sizeof(float), cudaMemcpyHostToDevice);

    // Define block and grid sizes
    dim3 threadsPerBlock(32, 32);
    dim3 numBlocks((N + threadsPerBlock.x - 1) / threadsPerBlock.x, (N + threadsPerBlock.y
- 1) / threadsPerBlock.y);

    // Launch kernels
    matrixMultiplyRow << <numBlocks, threadsPerBlock >> > (d_A, d_B, d_C, N, N, N);
    cudaDeviceSynchronize();
    matrixMultiplyColumn << <numBlocks, threadsPerBlock >> > (d_A, d_B, d_C, N, N, N);
    cudaDeviceSynchronize();
    matrixMultiplyElement << <numBlocks, threadsPerBlock >> > (d_A, d_B, d_C, N, N, N);
    cudaDeviceSynchronize();

    // Copy result back to host
    cudaMemcpy(h_C, d_C, SIZE * sizeof(float), cudaMemcpyDeviceToHost);

    // Print the resultant matrix
    printf("Resultant matrix:\n");
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            printf("%f ", h_C[i * N + j]);
        }
        printf("\n");
    }

    // Free device memory
    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);

    // Free host memory
    free(h_A);
    free(h_B);
    free(h_C);

    return 0;
}
```

OUTPUT:

```
Resultant matrix:
10.000000 20.000000 30.000000 40.000000
10.000000 20.000000 30.000000 40.000000
10.000000 20.000000 30.000000 40.000000
10.000000 20.000000 30.000000 40.000000
```