6CS005 Learning Journal - Semester 1 2020/2021

Name: Pranish Acharya

UV_ID: 2038527

Table of Contents

1 Parallel and Distributed Systems

1.1 Answer of First Question

Threads are an independent instruction stream that can be scheduled by the OS to run. Threads allow applications to simultaneously perform multiple tasks. Threads allow multiple computations to be performed in parallel.

1.2 Answer of Second Question

The two process scheduling policies are :

i. Pre-emptive: The schedular is in charge of how long a process runs for. If a process exceeds its time slice, it is stopped by the schedular.
ii. Co-operative: Each process is in-charge of how long it runs for. When a process feels like co-operating, it will surrender execution.

As every process gets its equal execution time, pre-emptive is preferable. The execution time period of the C threads which change the choice of policies..

1.3 Answer of Third Question

Centralized Systems	Distributed Systems
1. Centralized Systems uses the architecture of the client-server where one or more client nodes are connected directly to the central server.	1. Distributed networks use a peer-to-peer architecture in which each node makes its own choice. The system's final action is the sum of all nodes' decisions.
2. The system is composed of one component with non-autonomous components.	2. This system is made up of various autonomous components.
3. All the services are available in this system.	3. Resources may not be available in this system.
4. All the time, components are shared by users.	4. There is no sharing of components among all users.

1.4 Answer of Fourth Question

Transparency in Distributed System is defined to hide something. In order to realize the single system image, transparency is an essential issue that makes systems as simple to use as a single processor system.

Classification of the Transparency:

i. Access Transparency: Data and resources can be used in a consistent way.
ii. Location Transparency: A user cannot tell where resources are located.
iii. Migration Transparency: Resources can move at will without changing their names.
iv. Concurrency Transparency: Multiple users can share resource automatically.
v. Failure Transparency: A user does not notice resource failure.
vi. Performance Transparency: Systems are reconfigured to improve performance as loads vary.
vii. Scaling Transparency: Systems can expand in size without changing the system structure and the application programs.

1.5 Answer of Fifth Question

The output of the 1st statement is dependent on input of the 2nd statement, as defined in Flow Dependence. B = C+D, C = B + D. Here, B is the 1st statement's output and is used in the 2nd statement as an input.

The output of the 2nd statement is dependent on input of the 1st statement, as defined in Anti Dependency. B = C+D, C = B + D. Here, C is the input of the first statement and is the output of the second statement.

The output of the 1st statement is dependent on output of the 2nd statement, as defined in Output Dependence. B = A + C, B = C+D. Here, B is the output of the first statement and the output of the second statement is the output.

1.6 Answer of Sixth Question

The output of the given programs are 100000 and 500000 respectively. In these two programs pthread_join() function is call which terminate thread after executing target thread but in first program thread is call before reaching to the target thread. So, the result is different although the two programs are almost same.

2 Applications of Matrix Multiplication and Password Cracking using HPC-based CPU system

2.1 Single Thread Matrix Multiplication

- The analysis of the algorithm's complexity. (1 mark)

  The time complexity of the given program is O(n^3).

- Suggest at least three different ways to speed up the matrix multiplication algorithm given here. (Pay special attention to the utilisation of cache memory to achieve the intended speed up). (1 marks)

  Ans.

  There are different approaches present to speed up of the given program. Some of them are:

  i. Using multithreading.

     In multi-threading, instead of using a single processor core, all or more core is used to solve the problem. The problem is divided into different blocks.

  ii. Using cache memory

     By utilizing the cache memory, the program took reference from previous executed program which have stored result in cache memory.

  iii. By replacing a nested loop by first building a hash and then looping.

- Write your improved algorithms as pseudo-codes using any editor. Also, provide reasoning as to why you think the suggested algorithm is an improvement over the given algorithm. (1 marks)

```
Algorithm's pseudo code

int A[N][P], B[P][M], C[N][M];
  int i,j,k;

  for (i = 0; i < N; i++)
  {
     for (j = 0; j < P; j++)
     {

     }
  }


  for (i = 0; i < P; i++)
  {
     for (j = 0; j < M; j++)
     {

     }
  }

  for (i = 0; i < N; i++)
  {
     for (k = 0; k < P; k++)
     {
       C[i][j] = 0;

       for (k = 0; k < P; k++)
       {
          C[i][j] = C[i][j] + A[i][k] * B[k][j];
       }
     }
  }
```

- Write a C program that implements matrix multiplication using both the loop as given above and the improved versions that you have written. (1marks)

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

int main()
{
  struct timespec start, finish;
  long long int time_elapsed;

  clock_gettime(CLOCK_MONOTONIC, &start);
  int N,M,P;
  printf("\nEnter size of first Matrix\n");
  scanf("%d%d", &N,&P);
  printf("\nEnter the size second of matrix\n");
  scanf("%d%d",&P,&M);

  //int SizeOfArray = *Value;
  int A[N][P], B[P][M], C[N][M];
  int i,j,k;
  printf("\nEnter values for First Matrix\n");
  for (i = 0; i < N; i++)
  {
    for (j = 0; j < P; j++)
    {
    scanf("%10d",&A[i][j]);
    }
  }
   printf("\n");
   printf("Enter values for Second Matrix\n");
   for (i = 0; i < P; i++)
   {
     for (j = 0; j < M; j++)
     {
         scanf("%10d",&B[i][j]);
     }
   }

    for (i = 0; i < N; i++)
    {
```

Figure 1: Original Code From Given Pseudo-Code

```
40      for (j = 0; j < M; j++)
41      {
42        C[i][j] = 0;
43
44        for (k = 0; k < P; k++)
45        {
46        C[i][j] = C[i][j] + A[i][k] * B[k][j];
47        }
48         }
49       }
50       printf("\nThe results is: \n");
51       for (i = 0; i < N; i++)
52       {
53         for (j = 0; j < M; j++)
54      {
55         printf("%10d  ", C[i][j]);
56       }
57      printf("\n");
58       }
59       int time_difference(struct timespec *start, struct timespec *finish, long long int *difference) {
60        long long int ds =  finish->tv_sec - start->tv_sec;
61        long long int dn =  finish->tv_nsec - start->tv_nsec;
62
63        if(dn < 0 )
64        {
65          ds--;
66          dn += 1000000000;
67        }
68       *difference = ds * 1000000000 + dn;
69       return !(*difference > 0);
70       }
71
72       clock_gettime(CLOCK_MONOTONIC, &finish);
73       time_difference(&start, &finish, &time_elapsed);
74       printf("Time elapsed was %lldns or %0.9lfs\n", time_elapsed, (time_elapsed/1.0e9));
75       return 0;
76   }
77
```

Figure 2: Original Code from Given Pseudo-code

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

int main()
{
  struct timespec start, finish;
  long long int time_elapsed;

  clock_gettime(CLOCK_MONOTONIC, &start);
  int N,M,P;
  printf("\nEnter size of first Matrix\n");
  scanf("%d%d", &N,&P);
  printf("\nEnter the size second of matrix\n");
  scanf("%d%d",&P,&M);
  //int SizeOfArray = *Value;
  int A[N][P], B[P][M], C[N][M];
  int i,j,k;
  printf("\nEnter values for First Matrix\n");
  for (i = 0; i < N; i++)
  {
    for (j = 0; j < P; j++)
    {
   scanf("%10d",&A[i][j]);
    }
  }

  printf("\n");

  printf("Enter values for  Second Matrix\n");
  for (i = 0; i < P; i++)
  {
    for (j = 0; j < M; j++)
    {
   scanf("%10d",&B[i][j]);
    }
  }

```

Figure 3: Modified Code

```
38
39      for (i = 0; i < N; i++)
40      {
41        for (k = 0; k < P; k++)
42        {
43       C[i][j] = 0;
44
45       for (k = 0; k < P; k++)
46       {
47         C[i][j] = C[i][j] + A[i][k] * B[k][j];
48       }
49        }
50      }
51      printf("\nThe results is: \n");
52      for (i = 0; i < N; i++)
53      {
54        for (j = 0; j < M; j++)
55        {
56       printf("%10d  ", C[i][j]);
57        }
58        printf("\n");//
59      }
60       int time_difference(struct timespec *start,struct timespec *finish,long long int *difference)
61       {
62         long long int ds = finish->tv_sec - start->tv_sec;
63         long long int dn = finish->tv_nsec - start->tv_nsec;
64
65         if (dn < 0)
66         {
67        ds--;
68        dn += 1000000000;
69         }
70         *difference = ds * 1000000000 + dn;
71         return !(*difference > 0);
72        }
73
74         clock_gettime(CLOCK_MONOTONIC, &finish);
75         time_difference(&start, &finish, &time_elapsed);
76         printf("Time elapsed was %lldns or %0.9lfs\n", time_elapsed, (time_elapsed / 1.0e9));
77         return 0;
78  }
```

Figure 4: Modified Code

- Measure the timing performance of these implemented algorithms. Record your observations. (Remember to use large values of N, M and P – the matrix dimensions when doing this task). (1 marks)
  Insert a paragraph that hypothesises how long it would take to run the original and improved algorithms. Include your calculations. Explain your results of running time.

  Original code implemented using given pseudo-code have produces 4.8 seconds in 3*3 matrix while improved program produces 2.9 seconds using same 3*3 matrix. So, the improved program produces result in less time as compared to the original program. The time complexity of the program has decreased from previous one.

2.2 Multithreaded Matrix Multiplication

Multithread Matrix Multiplication

```
#include <pthread.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <time.h>


#define dim 1024

int thread_count;

struct param {
  int start_line, end_line;
};

int a[dim][dim], b[dim][dim], c[dim][dim];

void init_matrix(int m[dim][dim]) {
   int i, j;
   for (i = 0; i < dim; i++)
     for (j = 0; j < dim; j++)
       m[i][j] = rand() % 5;
}

void* thread_function(void *v) {
  struct param *p = (struct param *) v;

  int i, j, k;
  for (i = p->start_line; i < p->end_line; i++)
    for (j = 0; j < dim; j++) {
      int s = 0;
      for (k = 0; k < dim; k++)
        s += a[i][k] * b[k][j];
      c[i][j] = s;
    }
}
```

```
int time_difference(struct timespec *start, struct timespec *finish,
                    long long int *difference) {
  long long int ds =  finish->tv_sec - start->tv_sec;
  long long int dn =  finish->tv_nsec - start->tv_nsec;
  if(dn < 0 ) {
    ds--;
    dn += 1000000000;
  }
  *difference = ds * 1000000000 + dn;
  return !(*difference > 0);
}
int main() {
  struct timespec start, finish;
  long long int time_elapsed;
  int i;
  int start_time, end_time;

  printf("Enter the number of thread: ");
  scanf("%d",&thread_count);
  struct param params[thread_count];
  pthread_t t[thread_count];


  clock_gettime(CLOCK_MONOTONIC, &start);
  init_matrix(a);
  init_matrix(b);



  for (i = 0; i < thread_count; i++) {
    int code;
    params[i].start_line = i * (dim / thread_count);
    params[i].end_line = (i + 1) * (dim / thread_count);

    code = pthread_create(&t[i], NULL, thread_function, &params[i]);
    if (code != 0)
      printf("Error starting thread %d\n", i);
  }
```

```
for (i = 0; i < thread_count; i++)
    pthread_join(t[i], NULL);

       printf("\nThe results is…\n");
       for(int i=0; i<dim; i++) {
        for(int j=0; j<dim; j++) {
          printf("%d  ", c[i][j]);
         }
       printf("\n");
       }
  clock_gettime(CLOCK_MONOTONIC, &finish);
  time_difference(&start, &finish, &time_elapsed);
  printf("Time taken was %lldns or %0.9lfs\n", time_elapsed,
         (time_elapsed/1.0e9));

}
```

- Insert a table that has columns containing running times for the original program and your multithread version. Mean running times should be included at the bottom of the columns.

Number of Threads	Time Elapsed in Minutes
1 Thread	4.97
2 Thread	2.32
3 Thread	3.07
4 Thread	1.48
5 Thread	1.71
6 Thread	1.56
7 Thread	1.77
8 Thread	1.64
9 Thread	1.83
10 Thread	1.69
Mean Running Time:	**2.204**

Figure 5: Time Elapsed of Multithread Program

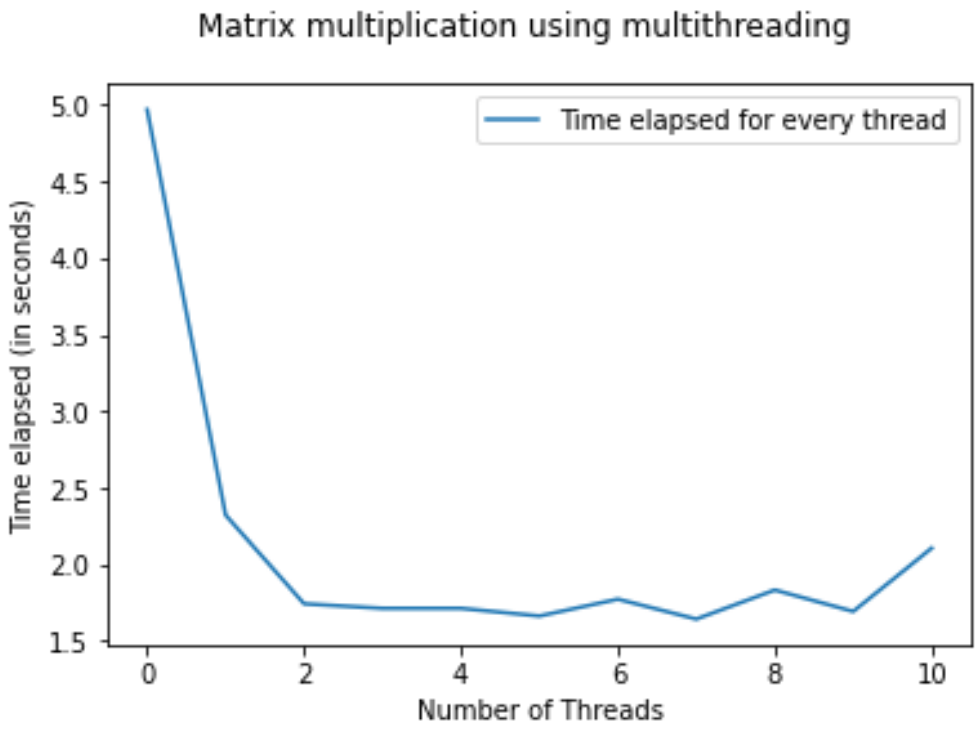

Figure 6: Graph of Executed Program

- Insert an explanation of the results presented in the above table.

  In the above program the time was decreasing by increasing the number of threads. At certain point matrix takes same time to execute the program. The minimal time to execute the thread is 1.48 second which is considered as sweet point.

2.3 Password cracking using POSIX Threads

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <crypt.h>
#include <unistd.h>
#include <time.h>


int counter=0;


void substr(char *dest, char *src, int start, int length){
  memcpy(dest, src + start, length);
  *(dest + length) = '\0';
}

void crackPassword(char *salt_and_encrypted){
  int p, r, a;
  char salt[7];
  char plain[7];
  char *enc;

  substr(salt, salt_and_encrypted, 0, 6);


  for(p='A'; p<='Z'; p++){
    for(r='A'; r<='Z'; r++){
      for(a=0; a<=99; a++){
        sprintf(plain, "%c%c%02d",p, r, a);
        enc = (char *) crypt(plain, salt);
        counter++;
        if(strcmp(salt_and_encrypted, enc) == 0){
        printf("#%-8d%s %s\n", counter, plain, enc);

        }else{
        printf("%-8d%s %s\n", counter, plain, enc);
        }
      }
    }
  }
}

```

Figure 7: Original Password Cracking Code

```
43
44  int time_difference(struct timespec *start, struct timespec *finish,
45                      long long int *difference) {
46    long long int ds =  finish->tv_sec - start->tv_sec;
47    long long int dn =  finish->tv_nsec - start->tv_nsec;
48    if(dn < 0 ) {
49      ds--;
50      dn += 1000000000;
51    }
52    *difference = ds * 1000000000 + dn;
53    return !(*difference > 0);
54  }
55
56  int main(int argc, char *argv[]){
57    struct timespec start, finish;
58    long long int time_elapsed;
59    clock_gettime(CLOCK_MONOTONIC, &start);
60    crack("$6$AS$gnjzTs539N0Ly5zdQrpW3a9QdXCfWVJ/FtSdR/9/OkV9YPVPLg4H2k/J8lUGEALdxikWnCRz1sj21rqGe7SY51");
61
62    printf("%d solutions explored\n", counter);
63    clock_gettime(CLOCK_MONOTONIC, &finish);
64    time_difference(&start, &finish, &time_elapsed);
65    printf("Time taken was %lldns or %0.9lfs\n", time_elapsed,
66           (time_elapsed/1.0e9));
67    return 0;
68  }
69
70
```

Figure 8: Original Password Cracking Code

- Insert a table of 10 running times and the mean running time.

A	B	C	D	E	F	G	H
Time elapsed was	168176060002	ns	or	168.176060002	s		
Time elapsed was	166343892919	ns	or	166.343892919	s		
Time elapsed was	159343291992	ns	or	159.343291992	s		
Time elapsed was	158821618138	ns	or	158.821618138	s		
Time elapsed was	167578174213	ns	or	167.578174213	s		
Time elapsed was	164281395115	ns	or	164.281395115	s		
Time elapsed was	163876228477	ns	or	163.876228477	s		
Time elapsed was	165271504488	ns	or	165.271504488	s		
Time elapsed was	165333620671	ns	or	165.333620671	s		
Time elapsed was	165173742484	ns	or	165.173742484	s		
Mean Running Time =	164419952849.9	NanoSeconds		164.4199528499	Seconds	2.74033255	Minutes

Figure 9: Time Elapsed of Original Password Crack

- Insert a paragraph that hypothesises how long it would take to run if the number of initials were to be increased to 3. Include your calculations.

Original password cracking program took 164.42 seconds (2.74 minutes) mean runtime, if we add 1 more initial i.e. one more for-loop starting from A-Z in program, it will take 4274.92 seconds (71.25 minutes) to run and create all three possible passwords of three alphabets and two numbers for a whole program. This is because it would take an additional 26 more time than the original program due to extra for loop. Therefore, the cost of time would rise by 26times.

Password Crack Using three Initial in Original Program

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <crypt.h>
#include <unistd.h>
#include <time.h>

int counter=0;

void substr(char *dest, char *src, int start, int length){
  memcpy(dest, src + start, length);
  *(dest + length) = '\0';
}

void crack(char *salt_and_encrypted){
  int p, r, a, n;
  char salt[7];
  char plain[7];
  char *enc;

  substr(salt, salt_and_encrypted, 0, 6);

for(p='A'; p<='Z'; p++){
  for(r='A'; r<='Z'; r++){
    for(a='A'; a<='Z'; a++){
      for(n=0; n<=99; n++){
        sprintf(plain, "%c%c%c%02d",p, r, a, n);
        enc = (char *) crypt(plain, salt);
        counter++;
        if(strcmp(salt_and_encrypted, enc) == 0){
           printf("#%-8d%s %s\n", counter, plain, enc);

        }else{
           printf("%-8d%s %s\n", counter, plain, enc);
        }
      }
    }
  }
}
}
```

```c
int time_difference(struct timespec *start, struct timespec *finish,
                    long long int *difference) {
  long long int ds =  finish->tv_sec - start->tv_sec;
  long long int dn =  finish->tv_nsec - start->tv_nsec;
  if(dn < 0 ) {
    ds--;
    dn += 1000000000;
  }
  *difference = ds * 1000000000 + dn;
  return !(*difference > 0);
}

int main(int argc, char *argv[]){
  struct timespec start, finish;
  long long int time_elapsed;
  clock_gettime(CLOCK_MONOTONIC, &start);
  crack("$6$AS$1mYI1AoWnumQr1TU5VoPouGGT0IGe4jKnKYY6SS7o1pbKWAJz.19AXQqRQHgH9Hwp3Zgy.MsRuZj/bHdvcNS41");

  printf("%d solutions explored\n", counter);
  clock_gettime(CLOCK_MONOTONIC, &finish);
  time_difference(&start, &finish, &time_elapsed);
  printf("Time taken was %lldns or %0.9lfs\n", time_elapsed,
         (time_elapsed/1.0e9));
  return 0;
}
```

Code Description:

There is a code of 3 alphabets and 2 numbers in the program above. One additional for-loop was added in the original programs to make the third for-loop, starting from A-Z, for the alphabet. The first six digits are referred to as salt in the one encrypted password given. 6KB$ is the salt shown in the password that has been encrypted. In the main function with the data type int, the variable "I" was declared, the start time and the end time were declared in order to calculate the elapsed time of the program.

After clock gettime (CLOCK MONOTONIC, &start) is executed, the start time will be noted from when the password began to crack. For-loop, which runs from 0 to n<passwords, was used. Nested for-loop has been used in the crack function in which the first three for-loops measure the alphaets and the last for-loop calculates numbers. It will be called at last clock-gettime (CLOCK-MANOTONIC, &finish) to mention the time when the program has ended.

time_difference () function has been called to calculate the total time taken by the whole program to generate all possible passwords.

Crypt () helps to convert salt and plan passwords into encrypted password.

Substr () extracts salt from encrypted password.

Strcmp () has been used to compare whether the encrypted password and password generated by the combination matches or not. If the password matches, # will be used at the front of the password to identify easily.

- Explain your results of running your 3 initial password cracker with relation to your earlier hypothesis.

After executing three initials, the program took 4247.02 seconds (70.78 minutes). Comparing the exact time after the code with above hypothesises, the exact time the program took was 4247.023895071 seconds where estimated time was 4274.92 seconds. As we can see that the difference between two is just 27.894879026 seconds. This distinction may have occurred due to the running of several processes in the background.

- Write a paragraph that compares the original results with those of your multithread password cracker.

Password Crack Using multithread

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <crypt.h>
#include <time.h>
#include <pthread.h>


int n_passwords = 1;
int counter = 0;


char *encrypted_password[]={"$6$AS$5MUX6oD4KOSmVBi5yCmu.8FTv1EgYaX09sO49BVt7CutV9Y1KWJVmjqSHoeub/AnfQU4eEV6NeIaDUf.OG7XK/"
};


void substr(char *dest, char *src, int start, int length){
  memcpy(dest, src + start, length);
  *(dest + length) = '\0';
}

void runThread(){
       int i;
       pthread_t PA1, PA2;

       void *kernel_function_1();
       void *kernel_function_2();
       for(i=0; i<n_passwords;i<i++){

       pthread_create(&PA1, NULL, kernel_function_1, encrypted_password[0]);
       pthread_create(&PA2, NULL, kernel_function_2, encrypted_password[0]);

       pthread_join(PA1, NULL);
```

```
void *kernel_function_1(char *salt_and_encrypted){
  int p, r, a;
  char salt[7];
  char plain[7];
  char *enc;
  int counter=0;
  substr(salt, salt_and_encrypted, 0, 6);

  for(p='A'; p<='M' ; p++){
    for(r='A'; r<='Z'; r++){
       for(a=0; a<=99; a++){


        sprintf(plain, "%c%c%02d", p, r, a);
        enc = (char *) crypt(plain, salt);
        counter++;
        if(strcmp(salt_and_encrypted, enc) == 0){
           printf("#%-8d%s %s\n", counter, plain, enc);

        }
        else{
            printf("#%-8d%s %s\n", counter, plain, enc);

        }
       }
      }
    }
    printf("%d solutions explored\n", counter);
}
```

```
void *kernel_function_2(char *salt_and_encrypted){
  int p, r, a;
  char salt[7];
  char plain[7];
  char *enc;
  int counter=0;
  substr(salt, salt_and_encrypted, 0, 6);

  for(p='N'; p<='Z' ; p++){
    for(r='A'; r<='Z'; r++){
       for(a=0; a<=99; a++){


        sprintf(plain, "%c%c%02d", p, r, a);
        enc = (char *) crypt(plain, salt);
        counter++;
        if(strcmp(salt_and_encrypted, enc) == 0){
           printf("#%-8d%s %s\n", counter, plain, enc);

        }
        else{
           printf("#%-8d%s %s\n", counter, plain, enc);

        }
       }
      }
    }
    printf("%d solutions explored\n", counter);
}
```

```
int time_difference(struct timespec *start,
                    struct timespec *finish,
                    long long int *difference) {
  long long int ds =  finish->tv_sec - start->tv_sec;
  long long int dn =  finish->tv_nsec - start->tv_nsec;

  if(dn < 0 ) {
    ds--;
    dn += 1000000000;
  }
  *difference = ds * 1000000000 + dn;
  return !(*difference > 0);
}

int main(){
int i;
  struct timespec start, finish;
  long long int time_elapsed;

  clock_gettime(CLOCK_MONOTONIC, &start);

  runThread();
  clock_gettime(CLOCK_MONOTONIC, &finish);

  printf("%d solutions explored\n", counter);

  time_difference(&start, &finish, &time_elapsed);
  printf("Time elapsed was %lldns or %0.9lfs\n", time_elapsed,(time_elapsed/1.0e9));
  return 0;
}
```

Code Description: #include <pthread.h> must be included every time, in order to execute a program in posix multithread. In main function, int i, start and finish and time_elapsed has been done. After that runThread () function has been called. Inside runThread (), two thread PA1, PA2 and two kernel functions kernel_function_1 () and kernel_function_2() has been declared. pthread_create () helps to create a new thread and makes it executable. It has 4 parameters.

First Parameter	&PA1/ &PA2	Identifier for new thread.
Second parameter	NULL	Thread attribute object /argument
Third parameter	Kernel_function_1 /2	start_routine / function name
Fourth parameter	encrypted_password[ ]	argument that may be passed to start_routine or function.

pthread_join () helps to join the two thread i.e. PA1 and PA2.

time_difference () function has been called to calculate the total time taken by the whole program to generate all possible passwords.

Crypt () helps to convert salt and plan passwords into encrypted password.

Substr () extracts salt from encrypted passwords.

Strcmp () has been used to compare whether the encrypted password and password generated by the combination matches or not. If the password matches, # will be used at the front of the password to identify easily.

A	B	C	D	E	F	G	H
Time elapsed was	130308825622	ns	or	130.308825622	s		
Time elapsed was	130517757745	ns	or	130.517757745	s		
Time elapsed was	129036211641	ns	or	129.036211641	s		
Time elapsed was	128134080836	ns	or	128.134080836	s		
Time elapsed was	129909025408	ns	or	129.909025408	s		
Time elapsed was	128.396789037	ns	or	128.396789037	s		
Time elapsed was	129.244345951	ns	or	129.244345951	s		
Time elapsed was	133040944449	ns	or	130.040944449	s		
Time elapsed was	132117440840	ns	or	131.11744084	s		
Time elapsed was	131986504096	ns	or	131.986504096	s		
Mean Running Tim	104505079089.464	NanoSeconds		129.8691925625	Seconds	2.16448654270833	Minutes

Figure 10: Time Elapsed of Password Cracking Using Multithread Program

Original password cracking program took 164.4199528499 seconds (2.74033254749833 minutes) mean running time to run while it took 129.8691925625 seconds (2.16448654270833 minutes) mean running time to run after inserting POSIX multithread in the same code. Using POSIX multithread, we can see that the cost of time from its initial program has been reduced, which is more effective. This is because it runs asynchronously as a thread.

3 Applications of Password Cracking and Image Blurring using HPC-based CUDA System

3.1 Password Cracking using CUDA

Password Crack Using CUDA version.

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

__device__ char* CudaCrypt(char* rawPassword){


        char * newPassword = (char *) malloc(sizeof(char) * 11);

       newPassword[0] = rawPassword[0] + 2;
       newPassword[1] = rawPassword[0] - 2;
       newPassword[2] = rawPassword[0] + 1;
       newPassword[3] = rawPassword[1] + 3;
       newPassword[4] = rawPassword[1] - 3;
       newPassword[5] = rawPassword[1] - 1;
       newPassword[6] = rawPassword[2] + 2;
       newPassword[7] = rawPassword[2] - 2;
       newPassword[8] = rawPassword[3] + 4;
       newPassword[9] = rawPassword[3] - 4;
       newPassword[10] = '\0';

       for(int i =0; i<10; i++){
              if(i >= 0 && i < 6){
                     if(newPassword[i] > 122){
                            newPassword[i] = (newPassword[i] - 122) + 97;
                     }else if(newPassword[i] < 97){
                            newPassword[i] = (97 - newPassword[i]) + 97;
                     }
              }else{
                     if(newPassword[i] > 57){
                            newPassword[i] = (newPassword[i] - 57) + 48;
                     }else if(newPassword[i] < 48){
                            newPassword[i] = (48 - newPassword[i]) + 48;
                     }
              }

       }
```

```
return newPassword;
}

__device__ int is_password(char* Encrypted){

        char Password[]="pa15";

        char *a=Encrypted;

        char *p=CudaCrypt(Password);

        while (*a == *p){

        if (*a == '\0')
        {
        printf("Encrypted Password: %s\n",Encrypted);
        printf("Password Found: %s\n",Password);
        break;
        }
        a++;
        p++;
        }
        return 0;
}

__global__ void crack(char * alphabet, char * numbers){

   char genRawPass[4];

   genRawPass[0] = alphabet[blockIdx.x];
   genRawPass[1] = alphabet[blockIdx.y];

   genRawPass[2] = numbers[threadIdx.x];
   genRawPass[3] = numbers[threadIdx.y];

   char *generated=CudaCrypt(genRawPass);

   is_password(generated);

}
```

```
int time_difference(struct timespec *start,
                    struct timespec *finish,
                    long long int *difference) {
  long long int ds =  finish->tv_sec - start->tv_sec;
  long long int dn =  finish->tv_nsec - start->tv_nsec;

  if(dn < 0 ) {
    ds--;
    dn += 1000000000;
  }
  *difference = ds * 1000000000 + dn;
  return !(*difference > 0);
}

int main(int argc, char ** argv){
  struct timespec start, finish;
  long long int time_elapsed;
  clock_gettime(CLOCK_MONOTONIC, &start);

  char cpuAlphabet[26] =  {'a','b','c','d','e','f','g','h','i','j','k','l','m','n','o','p','q','r','s','t','u','v','w','x','y','z'};
  char cpuNumbers[26] = {'0','1','2','3','4','5','6','7','8','9'};

  char * gpuAlphabet;
  cudaMalloc( (void**) &gpuAlphabet, sizeof(char) * 26);
  cudaMemcpy(gpuAlphabet, cpuAlphabet, sizeof(char) * 26, cudaMemcpyHostToDevice);

  char * gpuNumbers;
  cudaMalloc( (void**) &gpuNumbers, sizeof(char) * 26);
  cudaMemcpy(gpuNumbers, cpuNumbers, sizeof(char) * 26, cudaMemcpyHostToDevice);

  crack<<< dim3(26,26,1), dim3(10,10,1) >>>( gpuAlphabet, gpuNumbers);

  cudaDeviceSynchronize();

  clock_gettime(CLOCK_MONOTONIC, &finish);
  time_difference(&start, &finish, &time_elapsed);
  printf("Time elapsed was %lldns or %0.9lfs\n", time_elapsed, (time_elapsed/1.0e9));
  return 0;
}
```

Code description:

Firstly, it is important to add #include <cuda runtime api.h> on top of the file. This will help to improve connectivity for the CPU and GPU. In is_a_match function (), one password, which we have set with 2 alphabets and 2 numeric will be stored in plain_password1/. Password generated by kernel function will get stored in attempts and while loop will compare each password and if the password has '\0' at its last index, it will indicate that the password has matched and printf statement will be called and break statement will be executed and while loop will get terminated. This program will give output of the encrypted password and plain password along with time it takes to crack the password.

- Insert a table that shows running times for the original and CUDA versions.

Screen Shot of Password Cracking Using Original Code of 2 Initials

A	B	C	D	E	F	G	H
Time elapsed was	168176060002	ns	or	168.176060002	s		
Time elapsed was	166343892919	ns	or	166.343892919	s		
Time elapsed was	159343291992	ns	or	159.343291992	s		
Time elapsed was	158821618138	ns	or	158.821618138	s		
Time elapsed was	167578174213	ns	or	167.578174213	s		
Time elapsed was	164281395115	ns	or	164.281395115	s		
Time elapsed was	163876228477	ns	or	163.876228477	s		
Time elapsed was	165271504488	ns	or	165.271504488	s		
Time elapsed was	165333620671	ns	or	165.333620671	s		
Time elapsed was	165173742484	ns	or	165.173742484	s		
Mean Running Time =	164419952849.9	NanoSeconds		164.4199528499	Seconds	2.74033255	Minutes

Screen shot of Password Cracking using Thread

A	B	C	D	E	F	G	H
Time elapsed was	130308825622	ns	or	130.308825622	s		
Time elapsed was	130517757745	ns	or	130.517757745	s		
Time elapsed was	129036211641	ns	or	129.036211641	s		
Time elapsed was	128134080836	ns	or	128.134080836	s		
Time elapsed was	129909025408	ns	or	129.909025408	s		
Time elapsed was	128.396789037	ns	or	128.396789037	s		
Time elapsed was	129.244345951	ns	or	129.244345951	s		
Time elapsed was	133040944449	ns	or	130.040944449	s		
Time elapsed was	132117440840	ns	or	131.11744084	s		
Time elapsed was	131986504096	ns	or	131.986504096	s		
Mean Running Tim	104505079089.464	NanoSeconds		129.8691925625	Seconds	2.16448654270833	Minutes

Screen shot of password Cracking Using CUDA version.

A	B	C	D	E	F	G	H
Time elapsed was	158390558	ns	or	0.158390558	s		
Time elapsed was	147206114	ns	or	0.147206114	s		
Time elapsed was	165089978	ns	or	0.165089978	s		
Time elapsed was	161444831	ns	or	0.161444831	s		
Time elapsed was	155450558	ns	or	0.155450558	s		
Time elapsed was	169244358	ns	or	0.169244358	s		
Time elapsed was	149408147	ns	or	0.149408147	s		
Time elapsed was	141282989	ns	or	0.141282989	s		
Time elapsed was	166468957	ns	or	0.166468957	s		
Time elapsed was	154505347	ns	or	0.154505347	s		
Mean Running Time =	156849183.7	NanoSeconds		0.1568491837	Seconds	0.002614153	Minutes

- Write a short analysis of the results

The mean time of the program using CUDA version is 0.1568491837seconds (0.002614153 minutes) which is very less than the original one which is 164.4199528499 seconds (2.7 minutes) and thread version which is129.8691925625 second (2.1 minutes). It is because the CUDA in C programming is a more efficient package that interacts with GPC to execute the code. Compared to the Processor and GPU core, the CPU core is 42 times faster than the GPU core. Although the GPU core is slow, the task was performed simultaneously and much faster using the GPU core in the password cracker. Large numbers of cores will be split into 16 multicore units that will be able to perform eight operations in a single cycle. For this reason, if any multicore task is performed, 8 cores will run over 4 clock cycles every time. For this purpose, the GPU performs the tasks in a cost-effective manner than the CPU. CUDA is known to be a very strong core material and has the better potential to find productive results than others, such as POSIX. The CUDA processing speed is high, so that the performance is quickly taken out.

3.2 Image blur using multi dimension Gaussian matrices

- Include your code using a text file in the submitted zipped file under name Task3.2

```
#include "lodepng.h"
#include <stdio.h>
#include <stdlib.h>
#include <cuda_runtime_api.h>
#include <time.h>

// nvcc -o CudaImageBlur 2038527_Task3_B.cu lodepng.cpp


__device__ unsigned int deviceWidth;

__device__ unsigned char getRed(unsigned char *image, unsigned int row, unsigned int col)
{
  unsigned int i = (row * deviceWidth * 4) + (col * 4);
  return image[i];
}

__device__ unsigned char getGreen(unsigned char *image, unsigned int row, unsigned int col)
{
  unsigned int i = (row * deviceWidth * 4) + (col * 4) + 1;
  return image[i];
}

__device__ unsigned char getBlue(unsigned char *image, unsigned int row, unsigned int col)
{
  unsigned int i = (row * deviceWidth * 4) + (col * 4) + 2;
  return image[i];
}

__device__ unsigned char getAlpha(unsigned char *image, unsigned int row, unsigned int col)
{
  unsigned int i = (row * deviceWidth * 4) + (col * 4) + 3;
  return image[i];
}

__device__ void setRed(unsigned char *image, unsigned int row, unsigned int col, unsigned char red)
{
  unsigned int i = (row * deviceWidth * 4) + (col * 4);
  image[i] = red;
}
```

```
__device__ void setGreen(unsigned char *image, unsigned int row, unsigned int col, unsigned char green)
{
  unsigned int i = (row * deviceWidth * 4) + (col * 4) + 1;
  image[i] = green;
}

__device__ void setBlue(unsigned char *image, unsigned int row, unsigned int col, unsigned char blue)
{
  unsigned int i = (row * deviceWidth * 4) + (col * 4) + 2;
  image[i] = blue;
}

__device__ void setAlpha(unsigned char *image, unsigned int row, unsigned int col, unsigned char alpha)
{
  unsigned int i = (row * deviceWidth * 4) + (col * 4) + 3;
  image[i] = alpha;
}

__global__ void changeImage(unsigned char* image, unsigned char* newImage, unsigned int *width){
  int row = blockIdx.x+1;
  int col = threadIdx.x+1;

  deviceWidth = *width;

  unsigned redTL, redTC, redTR;
  unsigned redL, redC, redR;
  unsigned redBL, redBC, redBR;
  unsigned newRed;

  unsigned greenTL, greenTC, greenTR;
  unsigned greenL, greenC, greenR;
  unsigned greenBL, greenBC, greenBR;
  unsigned newGreen;

  unsigned blueTL, blueTC, blueTR;
  unsigned blueL, blueC, blueR;
  unsigned blueBL, blueBC, blueBR;
  unsigned newBlue;
```

```
setGreen(newImage, row, col, getGreen(image, row, col));
  setBlue(newImage, row, col, getBlue(image, row, col));
  setAlpha(newImage, row, col, 255);

  redTL = getRed(image, row - 1, col - 1);
  redTC = getRed(image, row - 1, col);
  redTR = getRed(image, row - 1, col + 1);

  redL = getRed(image, row, col - 1);
  redC = getRed(image, row, col);
  redR = getRed(image, row, col + 1);

  redBL = getRed(image, row + 1, col - 1);
  redBC = getRed(image, row + 1, col);
  redBR = getRed(image, row + 1, col + 1);

  newRed = (redTL+redTC+redTR+redL+redC+redR+redBL+redBC+redBR)/9;

  setRed(newImage, row, col, newRed);

  greenTL = getGreen(image, row - 1, col - 1);
  greenTC = getGreen(image, row - 1, col);
  greenTR = getGreen(image, row - 1, col + 1);

  greenL = getGreen(image, row, col - 1);
  greenC = getGreen(image, row, col);
  greenR = getGreen(image, row, col + 1);

  greenBL = getGreen(image, row + 1, col - 1);
  greenBC = getGreen(image, row + 1, col);
  greenBR = getGreen(image, row + 1, col + 1);

  newGreen = (greenTL+greenTC+greenTR+greenL+greenC+greenR+greenBL+greenBC+greenBR)/9;



  setGreen(newImage, row, col, newGreen);

  blueTL = getBlue(image, row - 1, col - 1);
  blueTC = getBlue(image, row - 1, col);
  blueTR = getBlue(image, row - 1, col + 1);
```

```
blueL = getBlue(image, row, col - 1);
  blueC = getBlue(image, row, col);
  blueR = getBlue(image, row, col + 1);

  blueBL = getBlue(image, row + 1, col - 1);
  blueBC = getBlue(image, row + 1, col);
  blueBR = getBlue(image, row + 1, col + 1);

  newBlue = (blueTL+blueTC+blueTR+blueL+blueC+blueR+blueBL+blueBC+blueBR)/9;

  setBlue(newImage, row, col, newBlue);
}


int time_difference (struct timespec *start, struct timespec *finish, long long int *difference) {
       long long int ds = finish->tv_sec - start->tv_sec;
       long long int dn = finish->tv_nsec - start->tv_nsec;
       if (dn < 0) {
              ds--;
              dn += 1000000000;
       }
       *difference = ds * 1000000000 + dn;
       return! (*difference > 0);
}


int main(int argc, char **argv)
{
  struct timespec start, finish;
  long long int time_elapsed;
  clock_gettime(CLOCK_MONOTONIC, &start);



  unsigned char *image;
  const char *filename = argv[1];
  const char *newFileName = "filtered.png";
  unsigned char *newImage;
  unsigned int height = 0, width = 0;
```

```
lodepng_decode32_file(&image, &width, &height, filename);
  newImage = (unsigned char *)malloc(height * width * 4 * sizeof(unsigned char));

  unsigned char * gpuImage;
  cudaMalloc( (void**) &gpuImage, sizeof(char) * height*width*4);
  cudaMemcpy(gpuImage, image, sizeof(char) *  height*width*4, cudaMemcpyHostToDevice);

  unsigned char * gpuNewImage;
  cudaMalloc( (void**) &gpuNewImage, sizeof(char) * height*width*4);

  unsigned int* gpuWidth;
  cudaMalloc( (void**) &gpuWidth, sizeof(int));
  cudaMemcpy(gpuWidth, &width, sizeof(int), cudaMemcpyHostToDevice);

  printf("Image width = %d height = %d\n", width, height);

  changeImage<<<height-1,width-1>>>(gpuImage, gpuNewImage, gpuWidth);
  cudaThreadSynchronize();

  cudaMemcpy(newImage, gpuNewImage, sizeof(char) * height * width * 4, cudaMemcpyDeviceToHost);

  lodepng_encode32_file(newFileName, newImage, width, height);

  clock_gettime(CLOCK_MONOTONIC, &finish);
  time_difference(&start, &finish, &time_elapsed);
  printf("Time elapsed was %lldns or %0.9lfs\n", time_elapsed, (time_elapsed/1.0e9));
  return 0;
}
```

- Insert a table that shows running times for the original and CUDA versions.

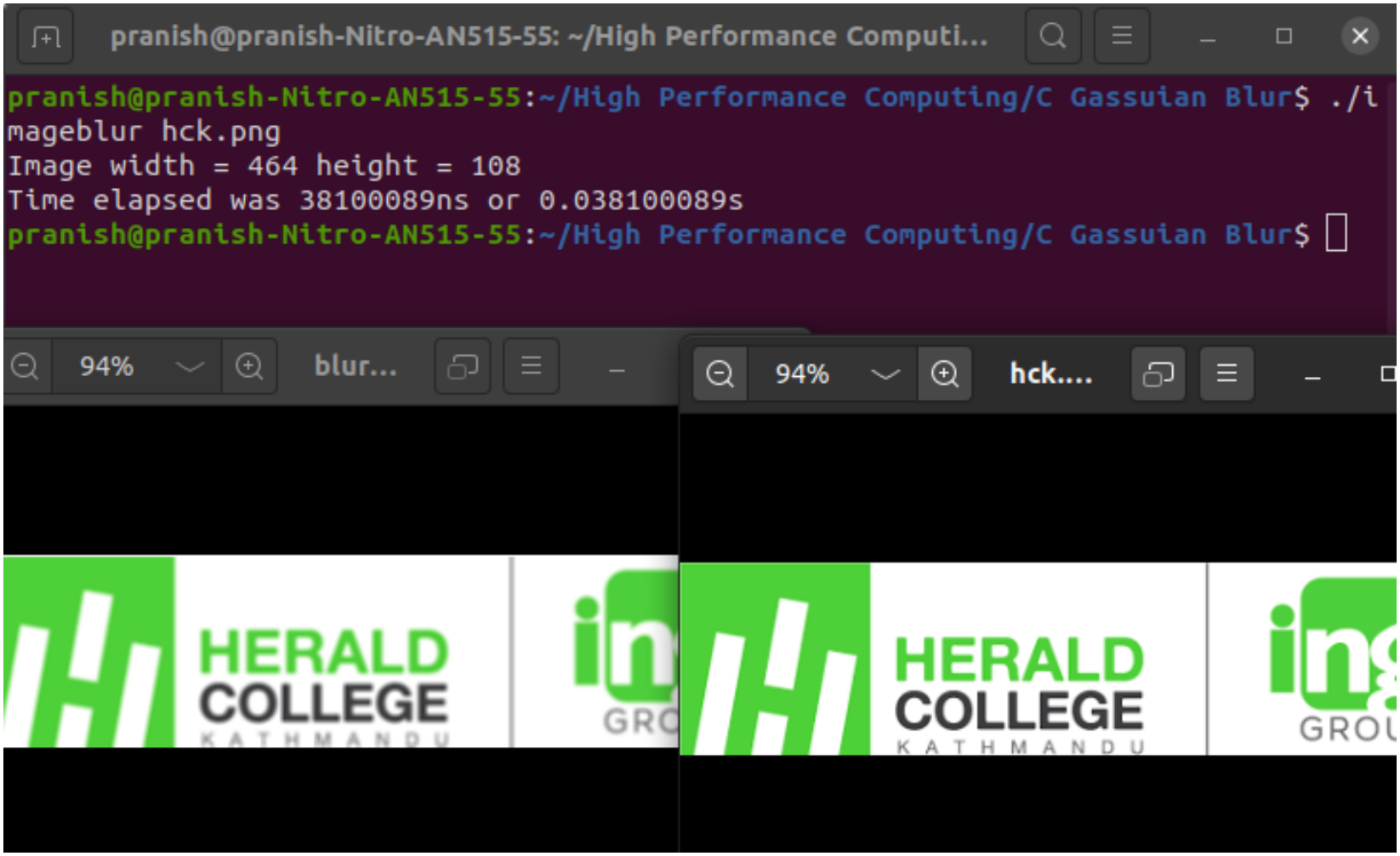

Figure 11: Image Blur using C Program

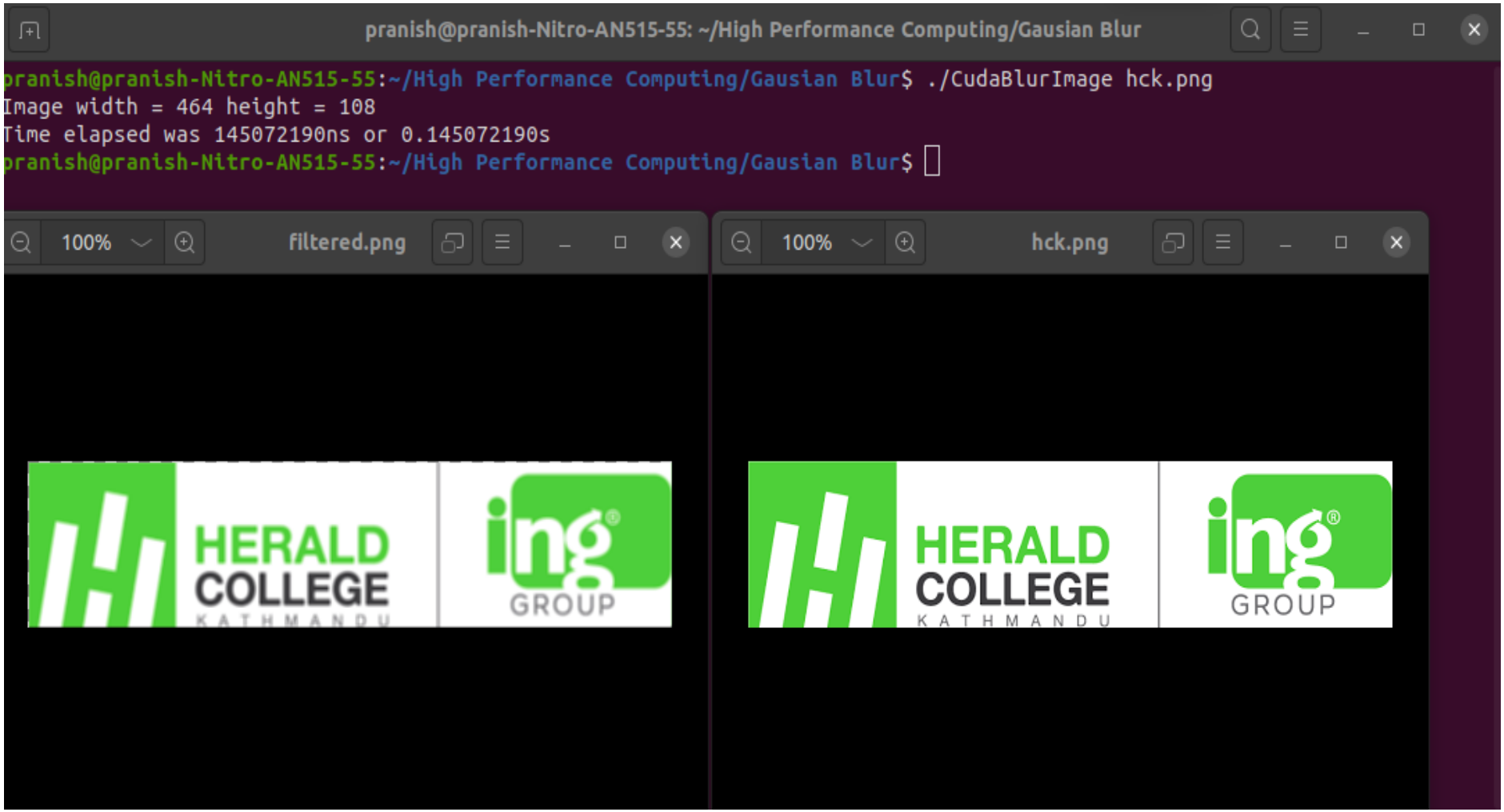

Figure 12: Image Blur using CUDA

- Write a short analysis of the results

The mechanism by which an image is blurred through a gauss function is Gaussian blurring. This technique is, in general, used to Reduce image noise, graphics software, machine information, and Applications for vision and image processing. In this process the primary concept is to change a pixel's value with the average Pixels for neighbours

Gaussian blurring process has been implemented in the logo of Herald College Kathmandu, in CUDA CORE GPU's. The same blurring algorithm was implemented and runs on CPU in order to compute the speed up based on execution times.
Following are the main steps used for calculating Gaussian blur
Step 1: Separating RGBA image to red, green and blue channels.
Step 2: Applying Gaussian blurring method for each time.
Step 3: Taking red, green and blue values into RGBA again

Applications from CUDA run faster than Applications based on conventional CPUs. Basic the basic the advantage of extreme parallelism is the logic behind this fact, supported by the architectural GPU. The GPUs do not need a Mechanism of flow control as in CPUs, because GPUs are used mainly for repeating image processing for each single image Pixels, 3D array, software for signal processing and Complicated equations. Therefore, the GPUs are designed to enhance the capability of parallel data processing rather than an Enhanced mechanism for caching and flow control, as in CPUs.