Project Report

CSE4001 Parallel and Distributed Computing

GPU Programming using CUDA and cuBLAS in cryptography

Faculty: Professor Kumar R.

Kartikay Kaul 16BCE1057
Shubham K. Gupta 16BCE1146

VELLORE INSTITUTE OF TECHNOLOGY CHENNAI CAMPUS

Fall Semester 2018-19

Acknowledgements

We would like to thank my teacher Professor Kumar R. for being there in providing us unconditional support and guidance in conducting this endeavor. We would like to express our gratitude towards our friends who have always helped augment motivation in us to not give up on this creation of ours that we value the most. We also value this prestigious institute of Vellore Institute of Technology Chennai to give us the opportunity to pursue this field and achieve what we have always wanted to in our life. We are grateful to each and every one who walked us through this path.

Abstract

This project endeavours to understand and learn CUDA and cuBLAS API for GPU programming. We start off with trying out a few sample programs given in documentation to learn CUDA and cuBLAS methods. We apply the CUDA code to different cryptographic algorithms in the current standard.

Introduction

The programmable Graphic Processor Unit (GPU) has evolved into extensively parallel and multithreaded, manycore processor comprising of huge computational power and memory bandwidth due to the increasing market demands for high definition graphics. According to a study, there is a huge discrepancy in the figures of FLOP capability between the CPU and GPU. This is because GPU is specialized for compute-intensive, highly parallel computations. More transistors are devoted to data processing rather than data caching and flow control. CPU has given more space to control and cache but GPU processor has given more space to ALU and less space to control and cache. GPU is well-suited to address problems that can be expressed as data-parallel computations – the same program is executed on many data elements in parallel – with a huge ratio of arithmetic operations to memory operations. We want to make use of such high computational power of GPU to run our programs in parallel rather than a sequential execution of a program on many data elements. This reduces a lot of time of computation. Here comes CUDA. In November 2006, NVIDIA introduced CUDA. CUDA is a general purpose parallel computing programming model that makes GPUs solve complex computational problems in quite efficient way than a CPU can. CUDA environment helps developers make use of C as a high-level programming language.

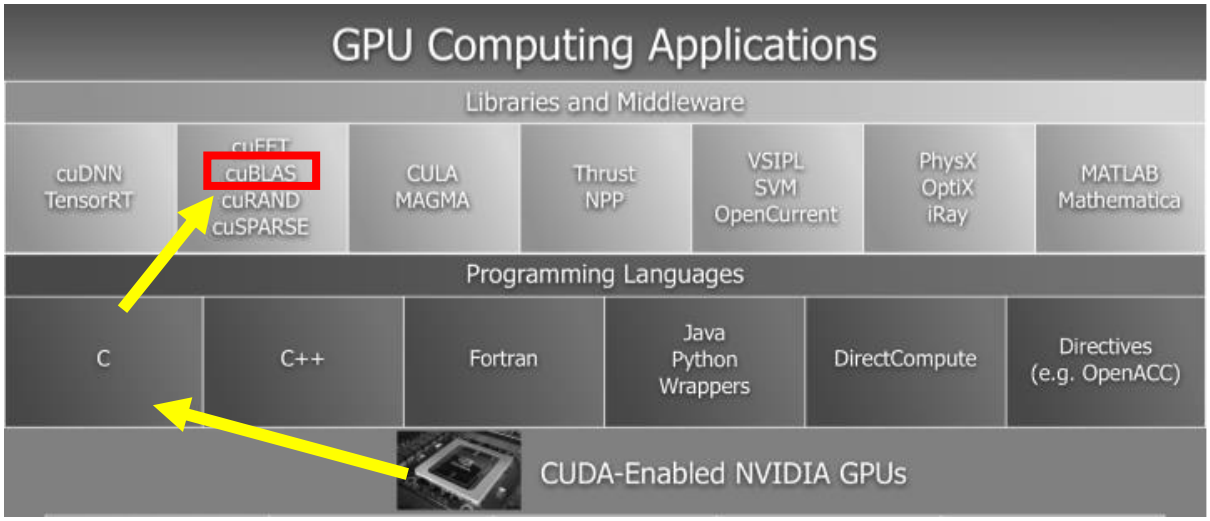


Figure 1 source:CUDA DOCUMENTATION

In CUDA C, programmer can define C functions which are called as kernels that when called are executed N times in parallel by N different CUDA threads.

A kernel is defined using the `__global__` declaration specifier and the number of CUDA threads that execute that kernel for a given kernel call is specified

using a `<<< … >>>` . Each thread that executes the kernel is given a unique thread ID accessed using `threadIdx` variable.

Motivation

We want to make use of such high computational power of GPU to run our programs in parallel rather than a sequential execution of a program on many data elements. This reduces a lot of time of computation. Cryptography is a very advanced concept in the field of Cybersecurity. Organisations such as NSA (National Security Agency) based in USA have put their primary focus on cryptography to the point of even deploying an entirely different block for the cryptography related technology and innovations. New cryptographic algorithms are released each day by many programmers and security organisations for maintaining the integrity and confidentiality of their documents and softwares. The algorithms have their lines code ranging from a mere two-digit number to almost a million lines of code. With the help of CUDA some of the kLOCs of code can be reduced and also reduce the computation time with the help of the GPUs parallel architecture.

Implementation details

We have written many samples of codes in CUDA before trying our safe passage into the cryptographic programs. All the codes have been written by us with help of already existing algorithms.

Squaring numbers of a 1-dimensional Array

This example is very simple and concise to demonstrate CUDA to beginner students.

Code

```
#include<stdio.h>

//our kernel functions that will run on DEVICE and be called from HOST
__global__ void squarenum(float *d_in, float *d_out)
 {
  int idx = threadIdx.x; //Applying the threadID to idx variable to traverse array
  float num = d_in[idx];
  d_out[idx] = num * num;
 }

int main()
 {
   int i;
   const int ARRAY_SIZE = 64;
   const int ARRAY_BYTES = ARRAY_SIZE * sizeof(float);
```

```
   float h_in[ARRAY_SIZE], h_out[ARRAY_SIZE];

   //ALLOCATING DEVICE VARIABLES
   float *d_in;
   cudaMalloc(&d_in, ARRAY_BYTES);
   float *d_out;
   cudaMalloc(&d_out, ARRAY_BYTES);

   for(i=0; i<ARRAY_SIZE; i++)
     h_in[i] = float(i);

   //COPYING DEVICE VARIABLES FROM HOST TO DEVICE
   cudaMemcpy(d_in, h_in, ARRAY_BYTES, cudaMemcpyHostToDevice);

   squarenum<<<1, ARRAY_SIZE>>>(d_in, d_out);

   //BRINGING BACK THE VALUES INTO HOST VARIABLES
   cudaMemcpy(h_out, d_out, ARRAY_BYTES, cudaMemcpyDeviceToHost);

   for(i=0; i<ARRAY_SIZE; i++)
        printf("%0.2f\n", h_out[i]);

   //necessary to free device variables
   cudaFree(d_in); cudaFree(d_out);
   printf("\n");
   return 0;
 }
```

Output

```
D:\zPrograms\cuda>nvcc squarenums.cu -o squarenum
squarenums.cu
   Creating library squarenum.lib and object squarenum.exp

D:\zPrograms\cuda>squarenum
0.00
1.00
4.00
9.00
16.00
25.00
36.00
49.00
64.00
81.00
100.00
121.00
144.00
169.00
196.00
225.00
256.00
289.00
324.00
361.00
400.00
441.00
```

```
484.00
529.00
576.00
625.00
676.00
729.00
784.00
841.00
900.00
961.00
1024.00
1089.00
1156.00
1225.00
1296.00
1369.00
1444.00
1521.00
1600.00
1681.00
1764.00
1849.00
1936.00
2025.00
2116.00
2209.00
2304.00
2401.00
2500.00
2601.00
2704.00
2809.00
2916.00
3025.00
3136.00
3249.00
3364.00
3481.00
3600.00
3721.00
3844.00
3969.00
D:\zPrograms\cuda>
```

Matrix addition code

This code is a bit more complex than previous one.

Code

```
#include<stdio.h>
#include<stdlib.h>
#define N 100
#define r 10
#define c 10

__global__ void matAdd(int *d_1, int *d_2, int *d_res)
```

```
{
  int i = threadIdx.x;
  int j = threadIdx.y;
  //d_res[i][j] = d_1[i][j] + d_2[i][j];
  *(d_res + i)= *(d_1 + i) + *(d_2 + i);
}

int main()
 {
   int i, j, count=0;
   int h_res[r][c], h_1[r][c], h_2[r][c];

   for(i=0; i<r; i++)
        for(j=0; j<c; j++)
              h_1[i][j] = h_2[i][j] = ++count;


   int MATRIX_SIZE = r*c*sizeof(int);
   int *d_1;
   cudaMalloc((void **)&d_1, MATRIX_SIZE);
   int *d_2;
   cudaMalloc((void **)&d_2, MATRIX_SIZE);
   int *d_res;
   cudaMalloc((void **)&d_res, MATRIX_SIZE);

   cudaMemcpy(d_1, h_1, MATRIX_SIZE, cudaMemcpyHostToDevice);
   cudaMemcpy(d_2, h_2, MATRIX_SIZE, cudaMemcpyHostToDevice);

   matAdd<<<1, r*c>>>(d_1, d_2, d_res);

   cudaMemcpy(h_res, d_res, MATRIX_SIZE, cudaMemcpyDeviceToHost);

   printf("\nThe resultant matrix is:\n");
   for(i=0; i<r; i++)
   {
        for(j=0; j<c; j++)
             printf("%d ", h_res[i][j]);
        printf("\n");
   }
   cudaFree(d_1); cudaFree(d_2); cudaFree(d_res);
   return 0;
 }
```

Output

```
D:\zPrograms\cuda>nvcc matrixaddition.cu -o matadd
matrixaddition.cu
matrixaddition.cu(10): warning: variable "j" was decla

   Creating library matadd.lib and object matadd.exp

D:\zPrograms\cuda>matadd

The resultant matrix is:
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
```

Matrix coalescence code

This code is quite complex and demonstrates the effects of memory coalescing.

```
Code

//matrix coalescing program

#include <stdio.h>

#include <stdlib.h>

#include <math.h>

#include <cuda.h>


#define BlockSize 16                          // Size of blocks, 32 x 32 threads,
fixed, used globally


__global__ void gpu_Comput (int *h, int N, int T) {


// Array loaded with global thread ID that acesses that location


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

       int row = threadIdx.y + blockDim.y * blockIdx.y;


       int threadID = col + row * N;

       int index = row + col * N;       // sequentially down each row


       for (int t = 0; t < T; t++)             // loop to repeat to reduce other
time effects

          h[index] = threadID;          // load array with flattened global thread
ID

}


void printArray(int *h, int N) {


       printf("Results of computation, every N/8 numbers, eight numbers\n");


       for (int row = 0; row < N; row += N/8) {

         for (int col = 0; col < N; col += N/8)

              printf("%6d  ", h[col + row * N]);
```

```
        printf("\n");
      }
}

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

      int T = 100;                          // number of iterations, entered at
keyboard
      int B = 1;                            // number of blocks, entered at
keyboard
      char key;

      int *h, *dev_h;                       // ptr to array holding numbers on
host and device

      cudaEvent_t start, stop;              // cuda events to measure time
      float elapsed_time_ms1;
      cudaEventCreate( &start );
      cudaEventCreate( &stop );

/* ------------------------ Keyboard input ----------------------------------*/

do {  // loop to repeat complete program

      printf("Grid Structure 2-D grid, 2-D blocks\n");
      printf("Blocks fixed at 16 x 16 threads, 512 threads, max for compute cap.
1.x\n");
      printf("Enter number of blocks in grid, each dimension, currently %d\n",B);
      scanf("%d",&B);
      printf("Enter number of iterations, currently %d\n",T);
      scanf("%d",&T);

      int N = B * BlockSize;                      // size of data array, given
input data

      printf("Array size (and total grid-block size) %d x %d\n", N, N);
```

```
      dim3 Block(BlockSize, BlockSize);                   //Block structure, 32 x
32 max
      dim3 Grid(B, B);                              //Grid structure, B x B

/* ------------------------ Allocate Memory----------------------------------*/

      int size = N * N * sizeof(int);        // number of bytes in total in array
      h = (int*) malloc(size);               // Array on host

      cudaMalloc((void**)&dev_h, size);             // allocate device memory

/* ------------------------ GPU Computation ----------------------------------*/

      cudaEventRecord( start, 0 );

      gpu_Comput<<< Grid, Block >>>(dev_h, N, T);

      cudaEventRecord( stop, 0 );            // instrument code to measue end
time
      cudaEventSynchronize( stop );          // wait for all work done by threads
      cudaEventElapsedTime( &elapsed_time_ms1, start, stop );

      cudaMemcpy(h,dev_h, size ,cudaMemcpyDeviceToHost);  //Get results to check

      printArray(h,N);
      printf("\nTime to calculate results on GPU: %f ms.\n", elapsed_time_ms1);

/* ------------------------REPEAT PROGRAM INPUT---------------------------------
-*/

      printf("\nEnter c to repeat, return to terminate\n");

      scanf("%c",&key);
      scanf("%c",&key);
```

```
} while (key == 'c'); // loop of complete program


/* --------------  clean up  ---------------------------------------*/


	free(h);

	cudaFree(dev_h);


	cudaEventDestroy(start);

	cudaEventDestroy(stop);


	return 0;

}
```

Output

```
D:\zPrograms\cuda>nvcc matrix_coal.cu -o matrix_coal
matrix_coal.cu
   Creating library matrix_coal.lib and object matrix_coal.exp

D:\zPrograms\cuda>matrix_coal
Grid Structure 2-D grid, 2-D blocks
Blocks fixed at 16 x 16 threads, 512 threads, max for compute cap. 1.x
Enter number of blocks in grid, each dimension, currently 1
2
Enter number of iterations, currently 100
50
Array size (and total grid-block size) 32 x 32
Results of computation, every N/8 numbers, eight numbers
     0     128     256     384     512     640     768     896
     4     132     260     388     516     644     772     900
     8     136     264     392     520     648     776     904
    12     140     268     396     524     652     780     908
    16     144     272     400     528     656     784     912
    20     148     276     404     532     660     788     916
    24     152     280     408     536     664     792     920
    28     156     284     412     540     668     796     924

Time to calculate results on GPU: 0.017792 ms.

Enter c to repeat, return to terminate


D:\zPrograms\cuda>
```

Computing Pi

We created a CUDA program to calculate the value of pi. We also calculated the timings of both the CPU and GPU calculated pi value with the help of clock() function from time.h library.

Code

```
#include <stdlib.h>
#include <stdio.h>
#include <cuda.h>
#include <math.h>
#include <time.h>
#include <curand_kernel.h>

#define TRIALS_PER_THREAD 4096
#define BLOCKS 256
#define THREADS 256
#define PI 3.1415926535  // known value of pi

__global__ void gpu_funct(float *estimate, curandState *states) {
	unsigned int tid = threadIdx.x + blockDim.x * blockIdx.x;
	int points_in_circle = 0;
	float x, y;

	curand_init(1234, tid, 0, &states[tid]);  //  Initialize CURAND


	for(int i = 0; i < TRIALS_PER_THREAD; i++) {
		x = curand_uniform (&states[tid]);
		y = curand_uniform (&states[tid]);
		points_in_circle += (x*x + y*y <= 1.0f); // count if x & y is in the
circle.
	}
	estimate[tid] = 4.0f * points_in_circle / (float) TRIALS_PER_THREAD; //
return estimate of pi
}

float host_funct(long trials) {
	float x, y;
	long points_in_circle;
	for(long i = 0; i < trials; i++) {
		x = rand() / (float) RAND_MAX;
		y = rand() / (float) RAND_MAX;
		points_in_circle += (x*x + y*y <= 1.0f);
	}
	return 4.0f * points_in_circle / trials;
}

int main (int argc, char *argv[]) {
	clock_t start, stop;
	float host[BLOCKS * THREADS];
	float *dev;
	curandState *devStates;
```

```
	printf("# of trials per thread = %d, # of blocks = %d, # of threads/block =
%d.\n", TRIALS_PER_THREAD,
BLOCKS, THREADS);

	start = clock();

	cudaMalloc((void **) &dev, BLOCKS * THREADS * sizeof(float)); // allocate
device mem. for counts

	cudaMalloc( (void **)&devStates, THREADS * BLOCKS * sizeof(curandState) );

	gpu_funct<<<BLOCKS, THREADS>>>(dev, devStates);

	cudaMemcpy(host, dev, BLOCKS * THREADS * sizeof(float),
cudaMemcpyDeviceToHost); // return results

	float pi_gpu;
	for(int i = 0; i < BLOCKS * THREADS; i++) {
		pi_gpu += host[i];
	}

	pi_gpu /= (BLOCKS * THREADS);

	stop = clock();

	printf("GPU pi calculated in %f s.\n", (stop-start)/(float)CLOCKS_PER_SEC);

	start = clock();
	float pi_cpu = host_funct(BLOCKS * THREADS * TRIALS_PER_THREAD);
	stop = clock();
	printf("CPU pi calculated in %f s.\n", (stop-start)/(float)CLOCKS_PER_SEC);

	printf("CUDA estimate of PI = %f \n", pi_gpu);
	printf("CPU estimate of PI = %f \n", pi_cpu);

	return 0;
}
```

Output

```
D:\zPrograms\cuda>nvcc calc_pi.cu -o pi
calc_pi.cu
d:\zprograms\cuda\calc_pi.cu(35) : warning C4700: uninitialized local variabl
d:\zprograms\cuda\calc_pi.cu(61) : warning C4700: uninitialized local variabl
   Creating library pi.lib and object pi.exp

D:\zPrograms\cuda>pi

GPU pi calculated in 0.286000 s.
CPU pi calculated in 10.430000 s.
CUDA estimate of PI = 3.141582
CPU estimate of PI = 3.141472
```

CuBLAS code for 1-based indexing

Code

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <cuda_runtime.h>
#include "cublas_v2.h"
#define M 6
#define N 5
#define IDX2F(i,j,ld) ((((j)-1)*(ld))+((i)-1))

static __inline__ void modify (cublasHandle_t handle, float *m, int ldm, int n,
int p, int q, float alpha, float beta){
    cublasSscal (handle, n-q+1, &alpha, &m[IDX2F(p,q,ldm)], ldm);
    cublasSscal (handle, ldm-p+1, &beta, &m[IDX2F(p,q,ldm)], 1);
}

int main (void){
    cudaError_t cudaStat;
    cublasStatus_t stat;
    cublasHandle_t handle;
    int i, j;
    float* devPtrA;
    float* a = 0;
    a = (float *)malloc (M * N * sizeof (*a));
    if (!a) {
        printf ("host memory allocation failed");
        return EXIT_FAILURE;
    }
    for (j = 1; j <= N; j++) {
        for (i = 1; i <= M; i++) {
            a[IDX2F(i,j,M)] = (float)((i-1) * M + j);
        }
    }
    cudaStat = cudaMalloc ((void**)&devPtrA, M*N*sizeof(*a));
    if (cudaStat != cudaSuccess) {
        printf ("device memory allocation failed");
        return EXIT_FAILURE;
    }
    stat = cublasCreate(&handle);
    if (stat != CUBLAS_STATUS_SUCCESS) {
        printf ("CUBLAS initialization failed\n");
        return EXIT_FAILURE;
    }
    stat = cublasSetMatrix (M, N, sizeof(*a), a, M, devPtrA, M);
    if (stat != CUBLAS_STATUS_SUCCESS) {
        printf ("data download failed");
        cudaFree (devPtrA);
        cublasDestroy(handle);
        return EXIT_FAILURE;
    }
    modify (handle, devPtrA, M, N, 2, 3, 16.0f, 12.0f);
    stat = cublasGetMatrix (M, N, sizeof(*a), devPtrA, M, a, M);
    if (stat != CUBLAS_STATUS_SUCCESS) {
```

```
        printf ("data upload failed");
        cudaFree (devPtrA);
        cublasDestroy(handle);
        return EXIT_FAILURE;
    }
    cudaFree (devPtrA);
    cublasDestroy(handle);
    for (j = 1; j <= N; j++) {
        for (i = 1; i <= M; i++) {
            printf ("%7.0f", a[IDX2F(i,j,M)]);
        }
        printf ("\n");
    }
    free(a);
    return EXIT_SUCCESS;
}
```

Output

```
D:\zPrograms\cuda>nvcc indexcublas.cu -o index1 -lcublas
indexcublas.cu
   Creating library index1.lib and object index1.exp

D:\zPrograms\cuda>index1
      1      7     13     19     25     31
      2      8     14     20     26     32
      3   1728    180    252    324    396
      4    160     16     22     28     34
      5    176     17     23     29     35

D:\zPrograms\cuda>
```

Caesar cipher program in CUDA

We created a program in C language and then created variables and methods in CUDA to run the program. We also calculated the time it took to do the computations in CPU and GPU. This will come in mostly handy when decoding huge lines of messages.

Code

```
//Author Karry Kaul//
#include<stdio.h>
#include<stdlib.h>
#include<time.h>
#include<string.h>

__global__ void encrypt(char *d_message, int key, int length)
{
      char ch;
      int i = threadIdx.x;
      if(i < length)
```

```
      {
            ch = d_message[i];
            if(ch >= 'a' && ch <= 'z'){
           ch = ch + key;

           if(ch > 'z'){
               ch = ch - 'z' + 'a' - 1;
           }

           d_message[i] = ch;
       }
       else if(ch >= 'A' && ch <= 'Z'){
           ch = ch + key;

           if(ch > 'Z'){
               ch = ch - 'Z' + 'A' - 1;
           }

           d_message[i] = ch;
       }
   }
}

int main()
{
    char message[100], ch;
    int i, key;
    //float cpu_time, gpu_time;
      clock_t start, stop;

    printf("Enter a message to encrypt: ");
    fgets(message, 100,stdin);
    printf("Enter key: ");
    scanf("%d", &key);

      int msg_length = strlen(message); char cpu_message[100];
      strcpy(cpu_message, message);

      char *d_message;
      cudaMalloc((void **)&d_message, msg_length*sizeof(char));

      cudaMemcpy(d_message, message, msg_length*sizeof(char),
cudaMemcpyHostToDevice);
      start = clock();
      encrypt<<<1, msg_length>>>(d_message, key, msg_length);
      stop = clock();
      cudaMemcpy(message, d_message, msg_length*sizeof(char),
cudaMemcpyDeviceToHost);

      printf("\nEncrypted message from the GPU is: %s and it took %0.20f s.",
message,(stop-start)/(float)CLOCKS_PER_SEC);


      start = clock();
    for(i = 0; cpu_message[i] != '\0'; ++i){
```

```
        ch = cpu_message[i];

        if(ch >= 'a' && ch <= 'z'){
            ch = ch + key;

            if(ch > 'z'){
                ch = ch - 'z' + 'a' - 1;
            }

            cpu_message[i] = ch;
        }
        else if(ch >= 'A' && ch <= 'Z'){
            ch = ch + key;

            if(ch > 'Z'){
                ch = ch - 'Z' + 'A' - 1;
            }

            cpu_message[i] = ch;
        }
    }
    stop = clock();
    printf("\n\nEncrypted message in CPU: %s and it took %0.20f s.", cpu_message,
(stop-start)/(float)CLOCKS_PER_SEC);

    return 0;
}
```

Output

```
D:\zPrograms\cuda>nvcc caesarcipher.cu -o caesarciph
caesarcipher.cu
   Creating library caesarciph.lib and object caesarciph.exp

D:\zPrograms\cuda>caesarciph
Enter a message to encrypt: hello the attack will be conducted on the night of 2056 january with the help of 20 bomb sit
es planned with help of our al qaeda brethrens
Enter key: 9

Encrypted message from the GPU is: qnuux cqn jccjlt Çruu kn lxwmdlcnm xw cqn wrpqc xo 2056 sjwdjaé Çrcq cqn qnuy xo 20 k
xvk brcnb yujwwnm Çrcq qnuy xo xda ju zjnmj kancqanwb
 and it took 0.000000 s.

Encrypted message in CPU: qnuux cqn jccjlt Çruu kn lxwmdlcnm xw cqn wrpqc xo 2056 sjwdjaé Çrcq cqn qnuy xo 20 kxvk brcnb
 yujwwnm Çrcq qnuy xo xda ju zjnmj kancqanwb
 and it took 0.000020 s.
D:\zPrograms\cuda>
```

Foursquare Cipher CUDA Program

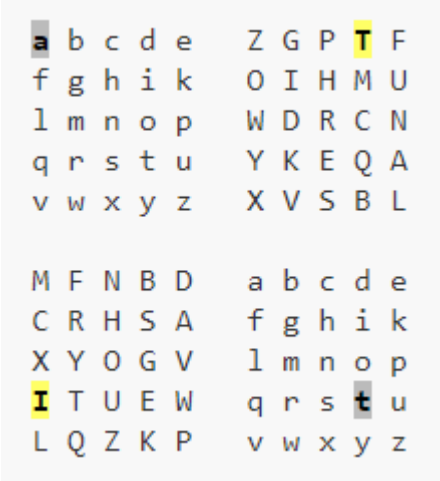
We created a sequential C program for foursquare cipher algorithm. In foursquare cipher.

Algorithm:

- Break up the plaintext into bigrams i.e. ATTACK AT DAWN --> AT TA CK AT DA WN. An 'X' (or some other character) may have to be appended to ensure the plaintext is an even length.
- Using the four 'squares', two plain alphabet squares and two cipher alphabet squares, locate the bigram to encrypt in the plain alphabet squares. The example below enciphers the bigram 'AT'. The first

letter is located from the top left square, the second letter is located in the bottom right square.

- Locate the characters in the ciphertext at the corners of the rectangle that the letters 'AT'
- Using the keys 'zgptfoihmuwdrcnykeqaxvsbl' and 'mfnbdcrhsaxyogvituewlqzkp', the bigram 'AT' is encrypted to 'TI'.
- The text 'attack at dawn', with the keys 'zgptfoihmuwdrcnykeqaxvsbl' and 'mfnbdcrhsaxyogvituewlqzkp', becomes TIYBFHTIZBSY



Code

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include<ctype.h>
#define MAX_LENGTH 200


/* SAMPLE STUFFS
	KEYS ('zgptfoihmuwdrcnykeqaxvsbl','mfnbdcrhsaxyogvituewlqzkp')

*/
const int N = 10;
//const int access_limit = 25;
const int SSI = 5; //SECOND_SQUARE_INDEX
const int TSI = 5; //THIRD_SQUARE_INDEX

typedef struct
{
	int x1, y1, x2, y2;
}index; //killemwithkindness


__global__ void return_pucca(char *d_dump, char d_matrix[][N], index *d_kwk)
{
	int i = blockIdx.y * blockDim.y + threadIdx.y;
	int j = blockIdx.x * blockDim.x + threadIdx.x;
```

```
	printf("\ni = %d\tj = %d", i,j);
	char first, second;
	first = d_dump[0], second = d_dump[1];

	if(i < SSI && j < TSI)
	{
		if(d_matrix[i][j] == first)
		{
			d_kwk->x1 = i;
			d_kwk->y1 = j;
			printf("\nx1 = %d\n", d_kwk->x1);
			printf("\nx1 = %d\n", d_kwk->x1);
		}

		if(d_matrix[i+TSI][j+SSI]  == second)
		{
		  d_kwk->x2 = i+TSI;
		  d_kwk->y2 = j+SSI;
		  printf("\nx1 = %d\n", d_kwk->x1);
		  printf("\nx1 = %d\n", d_kwk->x1);
		}
	}
}

void encrypt(char *dump, char matrix[N][N])
{
	int i, j;
	char CHECK[N][N];
	char first, second;
	index *kwk, *d_kwk; //index struct
	kwk = (index*)malloc(sizeof(index));

	//assign two different variables for checking em up
	first = dump[0], second = dump[1];

	//printf("%s\n%c\n%c", dump, first, second);

	char (*d_matrix)[N], *d_dump;
	cudaMalloc((void**)&d_matrix, N*N*sizeof(char));
	cudaMalloc((void**)&d_dump, 2*sizeof(char));
	cudaMalloc((void**)&d_kwk, sizeof(index));

	cudaMemcpy(d_matrix, matrix, N*N*sizeof(char), cudaMemcpyHostToDevice);
	cudaMemcpy(d_dump, dump, 2*sizeof(char), cudaMemcpyHostToDevice);
	cudaMemcpy(d_kwk, kwk, sizeof(index), cudaMemcpyHostToDevice);

	dim3 threadsPerBlock(N,N);
	return_pucca<<<1, threadsPerBlock>>>(dump, d_matrix, d_kwk);

	cudaMemcpy(CHECK, d_matrix, N*N*sizeof(char), cudaMemcpyDeviceToHost);
	for(i=0; i<N; i++)
	{
	 for(j=0; j<N; j++)
```

```
        printf("\t%c", CHECK[i][j]);
      }
      cudaMemcpy(kwk, d_kwk, sizeof(index), cudaMemcpyDeviceToHost);

      //printf("\nx1 = %d\n", kwk->x1);
      //printf("\ny1 = %d\n", kwk->y1);
      //printf("\nx2 = %d\n", kwk->x2);
      //printf("\ny2 = %d\n", kwk->y2);
      /*
      //find the characters from messages in squares 1 and 3
      for(i=0; i<N-SSI; i++)
       {
            for(j=0; j<N-TSI; j++)
            {
                  if(matrix[i][j] == first)
                   {
                        kwk.x1 = i;
                        kwk.y1 = j;
                   }
                   if(matrix[i+TSI][j+SSI] == second)
                   {
                        kwk.x2 = i+TSI;
                        kwk.y2 = j+SSI;
                   }
            }
       } */

      //ASSIGNING pass array with encrypted values
      for(i=0; i<TSI; i++)
      {
        for(j=SSI; j<N; j++)
          {
                  if( i==kwk->x1 && j==kwk->y2 )
                  {
                        printf("\nmatch\n");
                        dump[0] = matrix[i][j];
                        printf("\ndump[0] = %c\n", dump[0]);
                  }
          }
    }

    for(i=TSI ; i<N ; i++)
     {
       for(j=0 ; j<SSI ; j++)
        {
                  if( i == kwk->x2 && j == kwk->y1)
                  {
                        printf("\nmatch\n");
                        dump[1] = matrix[i][j];
                        printf("\ndump[1] = %c\n", dump[1]);
                  }
             }
      }
}
```

```
//decryption function
void decrypt(char *dump, char matrix[N][N])
{
	int i, j;

	char first, second;
	index kwk; //index struct

	//assign two different variables for checking em up
	first = dump[0], second = dump[1];

	//trying to find the variables
	for(i=0 ; i<TSI ; i++)
	 {
		for(j=SSI ; j<N ; j++)
		 {
			if(matrix[i][j] == first)
			 {
				printf("\n%c", matrix[i][j]);
				kwk.x1 = i;
				kwk.y1 = j;
			 }

			 if(matrix[i+TSI][j-SSI] == second)
			 {
				printf("\n%c", matrix[i+TSI][j-SSI]);
				kwk.x2 = i+TSI;
				kwk.y2 = j-SSI;
			 }
		 }
	 }

	for(i=0; i<TSI; i++)
	 {
		for(j=0; j<SSI; j++)
		 {
			if( i==kwk.x1 && j==kwk.y2)
			{
				printf("\nmatch\n");
				dump[0] = matrix[i][j];
				printf("\ndump[0] = %c\n", dump[0]);
			}
		 }
	 }

	 for(i=TSI; i<N; i++)
	  {
		for(j=SSI; j<N; j++)
		 {
		  if( i == kwk.x2 && j == kwk.y1)
			{
				printf("\nmatch\n");
				dump[1] = matrix[i][j];
				printf("\ndump[1] = %c\n", dump[1]);
```

```
                    }
                }
            }
        //printf("%s\n%c\n%c", dump, first, second);
} //decrypt fnction


int main()
 {
        //array initialisation variables
        int i,j, k;

        //alphabets array
        char alphabets[26] = {'a', 'b', 'c', 'd', 'e',
                                        'f', 'g', 'h', 'i', 'k',
                                        'l', 'm', 'n', 'o', 'p',
                                        'q', 'r', 's', 't', 'u',
                                        'v', 'w', 'x', 'y', 'z',
                                        'j'
                                        };

        //The main matrix
        char foursquare[N][N];

        //just some random peasant variables :P
        char message[MAX_LENGTH], encrypted_message[MAX_LENGTH],key_enc1[30],
key_enc2[30];

        /*CODE ID:1 = Fills the square 1 and square 3 with alphabets array */
        k=0; //variable to access the alphabets array
        for(i=0; i<N-SSI; i++)
         {
                for(j=0; j<N-TSI; j++)
                {
                        foursquare[i][j] = foursquare[i+TSI][j+SSI] = alphabets[k];
                        k++;
                }
         }
        /* CODE ID 1 ends */

        //printf("DEBUG: Shit down the throat!\n\n");

        //Taking the keys inputs from user
        printf("Enter the keys:\n");
        scanf("%s", &key_enc1);
        scanf("%s", &key_enc2);

        printf("TEST:\n %s \n %s\n\n", key_enc1, key_enc2);

        /*CODE ID:2 = Fills the square 2 and 4 with the 25 digit keys */
        /*The encryption and decryption keys are filled in square 2 and 4
respectively */
        k=0;
        for(i=0; i<TSI; i++)
        {
```

```
        for(j=SSI; j<N; j++)
         {
             foursquare[i][j] = tolower(key_enc1[k]);
             foursquare[i+TSI][j-SSI] = tolower(key_enc2[k]);
             k++;
         }
      }
      /*CODE ID:2 ends */


      //Take the input message
      printf("Enter the message: ");
      scanf("%s", &message);

      //removeSpaces(message);

      printf("\nMESSAGE: %s\n", message);

      int message_length = strlen(message);

      //turning odd length message to even length
      if(message_length%2 != 0)
       {
             message[message_length] = 'x';
             message[message_length+1] = '\0';
             //printf("\nEVENED OUT MESSAGE IS : %s\n", message);
       }

       char dump_var[2];

       /*CODE ID:3 = Encryption code */
      //printf("DEBUG: Sh\n\n");
      k=0;
      while(k<message_length)
      {
       //get the first two characters from message
       dump_var[0] = message[k];
       dump_var[1] = message[k+1];

       //let the accept variable get the encrypted result
       encrypt(dump_var, foursquare);

       //insert the result in corresponding places of encrypted result
       encrypted_message[k] = dump_var[0];
       encrypted_message[k+1] = dump_var[1];

       k+=2;
      }
      encrypted_message[k] = '\0'; //adding a null to make it a string

      //printf("DEBUG: Sh!\n\n");
      /*CODE ID:3 ends */

      //if debug
      for(i=0; i<N; i++)
```

```
       {
             for(j=0; j<N; j++)
             {
                   printf("%c\t", foursquare[i][j]);
             }
             printf("\n");
       }
      //endif

       printf("\nEncrypted message: ");
       printf("%s\n", encrypted_message);

       /* CODEID:4 = Decryption code */
        //printf("DEBUG: Sh!\n\n");
      k=0;
      while(k<message_length)
      {
             //get the first two characters from message
          dump_var[0] = encrypted_message[k];
          dump_var[1] = encrypted_message[k+1];

             //let the accept variable get the encrypted result
          decrypt(dump_var, foursquare);

          //insert the result in corresponding places of encrypted result
             message[k] = dump_var[0];
             message[k+1] = dump_var[1];
             k+=2;
      }

      message[k] = '\0';
      /* CODE ID:4 ends */

       printf("Reverted from Encryption: %s", message);
       return 0;
}
```

Output

(Debug statements removed)

```
Enter the keys:
zgptfoihmuwdrcnykeqaxvsbl
mfnbdcrhsaxyogvituewlqzkp
TEST:
 zgptfoihmuwdrcnykeqaxvsbl
 mfnbdcrhsaxyogvituewlqzkp

Enter the message: destroybomb

MESSAGE: destroybomb
a       b       c       d       e       z       g       p       t       f
f       g       h       i       k       o       i       h       m       u
l       m       n       o       p       w       d       r       c       n
q       r       s       t       u       y       k       e       q       a
v       w       x       y       z       x       v       s       b       l
m       f       n       b       d       a       b       c       d       e
c       r       h       s       a       f       g       h       i       k
x       y       o       g       v       l       m       n       o       p
i       t       u       e       w       q       r       s       t       u
l       q       z       k       p       v       w       x       y       z

Encrypted message: fbquqyvbdgpq

Decrypted message, destroybombx
```

The CUDA function to encrypt did not work properly in the last program due to the matrix not transferring properly. So in the modified code, the matrix is not transferred to the device instead only the message is transferred to the GPU. We could instead define foursquare matrix as a device variable and only transfer message to the gpu and do all processing there to receive encrypted message. This could be done by defining a structure in the device memory.

Conclusion

The use of GPU in encrypting and decrypting long lines of messages is quite useful and reduces time that will be taken to run a cryptography algorithm in a sequential program running over CPU. The clock times calculated during execution of some programs showed promising results that GPU could be used in utility platforms for easy and fast calculations of huge computational algorithms. In future, we could implement CUDA on high end algorithms such as md5 hash, RSA, SHA1, etc. We can even do this on plagiarism checking programs.