Parallel Programming Using CUDA

BE Final Year Project

By

Aman Sethi Roll No: 001211001006

Debajyoti De Roll No: 001211001002

Supervisor:

Mr. Utpal Kr. Ray

Assistant Professor

Department of Information Technology

Jadavpur University

ACKNOWLEDGEMENT

We take this opportunity to express our sincere thanks to our guide Prof. Utpal Kr. Ray for his support, encouragement and advice. His constant guidance was invaluable in the realization of this report.

Aman Sethi and Debajyoti De
Jadavpur University
May 19, 2016

ABSTRACT

This report documents our final year project, which is about parallel programming using CUDA, the NVIDIA GPU architecture with support for general purpose computing. The purpose of this report is to uncover the qualities of CUDA as a parallel computing platform, determining the performance of an algorithm when it is implemented using parallel programming approach.

We examine this by implementing an algorithm to find a minimum in an array of elements using parallel programming approach. We measure the time the algorithm takes to complete its task for different size of an array and made a time-vs.-size graph.

We also implement parallel version of a standard Matrix Inversion algorithm and study the performance of the parallel algorithm when it is executed in CPU and also when it is executed in the GPU using CUDA.

TABLE OF CONTENTS

1.	Introduction	4
2.	Parallel Programming	5
3.	GPU Programming using CUDA • What is CUDA? • How CUDA works? • GPU Memory Hierarchy	10 10 10 14
4.	Find Minimum in an Array using CUDA • Problem Statement • Approach • Results	16 16 16 19
5.	GPU based Matrix Inversion using CUDA • Problem Statement • Approach • Results	20 20 20 24
6.	Applications • Hill Cipher • Example	25 25 29

7.	Conclusion and Future Work	32
8.	References	33
9.	Appendix A – CUDA Installation Guidelines	34
10.	Appendix B – System Specifications	39
11.	Appendix C – Compilation and Execution of a CUDA program	40

1. INTRODUCTION

Graphic Processor Units

A graphic processor unit is simply a processor attached to the graphics card used in video games, PlayStations and computers. The way they are different from the CPUs, the central processing units, are that they are massively threaded and parallel in their operations. This is because of the nature of their work – they are used for fast rendering; same operation is carried out for each of the pixels in the image. Thus, they have more transistors devoted to data processing rather than flow control and data caching.

Today graphic processor units have outdated their primary purpose. They are being used and promoted for scientific calculations all over the world by the name of GPGPUs or general purpose GPUs; engineers are achieving several times speed-up by running their programs on

GPUs. Applications fields are many: image processing, general signal processing, physics simulations, computational biology, etc.

Motivation

Graphic Processor Units can speed-up the computation manifolds over the traditional CPU implementation. Image processing, being inherently parallel, can be implemented quite effectively on a GPU. Thus, many applications which are otherwise run slowly can be fastened up, and can be put to useful real-time applications. This was the motivation behind my final year project.

NVIDIA CUDA is a parallel programming model and software, which has developed specifically to address the problems of efficiently program the GPU as well as be compatible with a wide variety of GPU cores available in the market. Further, being an extension to the standard C language, it presents a low-learning curve for the programmers, as well giving them flexibility to put in their creativity in the parallel codes they write.

2. PARALLEL PROGRAMMIING

In computing, a parallel programming model is an abstraction of parallel computer architecture, with which it is convenient to express algorithms and their composition in programs. A parallel program is composed of simultaneously executing processes.

- There are two different approaches of achieving parallelism while solving a problem.
- They are CONTROL PARALLEL approach and DATA PARALLEL approach.
- In Control Parallel approach, parallelism is achieved by applying different operations to the data set simultaneously. It is about attaching computations to the data set. Sometimes it is easier to implement this approach, but speedup achieved is limited. Pipelining is a special case of Control Parallel Approach.
- In Data Parallel approach, parallelism is achieved by applying same operation to the different data set simultaneously. It is about attaching different data set to the computations. Implementing this is more complex, but more speedup can be achieved through this method.

A Sample problem

Find the ratio of the product and the sum of all the elements of a given matrix.

The Sequential Solution

To calculate Tseq; let's assume every addition takes x amount of time, every multiplication takes 2x amount of time and the division takes 3x amount of time. So, Tseq = $N^2 2X + N^2 X + 3X$.

The Control Parallel Solution

There are three different distinct computation type here - Multiplication, Addition and Division. Out of these, first two can run in parallel; and the last one has to start once the first two are complete. So a two processor machine is good enough for running this solution.

Out of this Multiplication and Addition; multiplication will take more time ($N^2 2X$) to complete than the addition ($N^2 X$). So, the processor running the addition (after his addition operation gets completed) has to wait for the other processor to complete multiplication. Once both these operation are over, either of the processor can take up the division which takes 3X amount of time to complete.

So, Tpar = $N^2 2X + 3X$

Speedup S = $(N^2 2X + N^2 X + 3X) / (N^2 2X + 3X)$

$= (3N^2 + 3) / (2N^2 + 3)$

$= 1.5$ (N being large)

And, Efficiency E = S/P = 1.5/2 = .75

The Data Parallel Solution

For the sake of simplicity, let's assume there are only two processors available. So, the total data (here, it's a matrix) need to be divided into two halves (row wise division is good). The first half goes to the processor 1 and the second half goes to the processor 2. Each of this processor will run the computations (multiplication and addition) on their own data set. Once, the partial products and sums are available; either of the processor can compute the total and then the division.

So, the time taken to compute the partial product by any of the processor is $(N^2 2x)/2$. And the time take to compute the partial sum by any of the processor is $(N^2x)/2$. So, the total time taken by any of the processor to run his own part of computation is $(N^2 2x + N^2x)/2$ or $(3N^2x)/2$.

After this one of the processor will compute the total product (takes 2x) and total sum (takes x). It is to be noted that this work cannot be parallelized. And the last piece of computation done by one of the processor is division (takes 3x).

So, $T_{par} = (3N^2x)/2 + 2x + x + 3x$

$= (3N^2x)/2 + 6x$

$= (3N^2x + 12x)/2$

Speedup $S = ((N^2 2x + N^2x + 3x) / (3N^2x + 12x)) * 2$

$= ((3N^2 + 3) / (3N^2 + 12)) * 2$

$= 2$ (N being large)

And, Efficiency $E = S/P = 2/2 = 1.0$

Multithreaded Programming – A Type of Parallel Programming

The threads model of parallel programming is one in which a single process (a single program) can spawn multiple, concurrent "threads" (sub-programs). Each thread runs independently of the others, although they can all access the same shared memory space (and hence they can communicate with each other if necessary). Threads can be spawned and killed as required, by the main program.

A challenge of using threads is the issue of collisions and race conditions, which can be addressed using synchronization. If multiple threads write to (and depend upon) a shared memory variable, then care must be taken to make sure that multiple threads don't try to write to the same location simultaneously. There are mechanisms when using threads to implement synchronization, and to implement mutual exclusivity (mutex variables) so that shared variables can be locked by one thread and then released, preventing collisions by other threads. These mechanisms ensure threads must "take turns" when accessing protected data.

POSIX Threads (Pthreads)

POSIX Threads (Pthreads for short) is a standard for programming with threads, and defines a set of C types, functions and constants.

A Sample Program

A Multi-threaded program to find the minimum from a large integer array.

```
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>
#include <pthread.h>
int NO_OF_THREADS;
#define SIZE 100000000
```

```
int arr[SIZE],glob_min;
pthread_mutex_t lock;

struct threadData{
    int threadId;
};

/* thread function */
void *findMIN(void *a)
{
    struct threadData *D = (struct threadData *)a;
    int i = D->threadId;
    int sizeOfBlock = ((SIZE + (NO_OF_THREADS - 1)) / (NO_OF_THREADS));
    int start = i*sizeOfBlock;
    int min = INT_MAX;

    /* Each thread calculate minimum of all elements present in their
respective blocks */
    for (; start < SIZE; start++)
    {
        if (start == ((i + 1)*sizeOfBlock))
            break;
        if(arr[start]<min)
            min=arr[start];
    }

    /*Lock used for Thread synchronization on shared variable glob_min */
    pthread_mutex_lock(&lock);

    if(min<glob_min)
        glob_min=min;

    pthread_mutex_unlock(&lock);
    pthread_exit(NULL);

    return NULL;
}

int main(void)
{
    int i=0,t=0,ret_val;
    glob_min=INT_MAX;
    printf("Enter the no of threads\n");
    scanf("%d",&NO_OF_THREADS);
    pthread_t threads[NO_OF_THREADS];
    struct threadData a[NO_OF_THREADS];     /* create a threadData argument
array */

    /* Assign random values to array elements */
    for (i = 0; i < SIZE; i++)
        arr[i] = (rand() % 50) + 50;

    arr[SIZE/2]=2;
    /* Initialise Mutex */
    if (pthread_mutex_init(&lock, NULL) != 0)
    {
        printf("\n mutex init failed\n");
        return 1;
    }

    /* create threads */
    for (t = 0; t < NO_OF_THREADS; t++)
    {
        a[t].threadId = t;
        ret_val = pthread_create(&threads[t], NULL, findMIN, (void *)&a[t]);
        if (ret_val)
        {
            printf("Problem creating thread !! Bye !!");
            exit(1);
        }
    }

    /* block until all threads complete */
    for (t = 0; t < NO_OF_THREADS; t++)
    pthread_join(threads[t], NULL);
    printf("Minimum: %d\n",glob_min);
    pthread_mutex_destroy(&lock);

    return 0;

}
```

Above is a multithreaded program to find the minimum from a large integer array. The program uses the Data Parallel Approach of Parallel Programming. Each thread finds the minimum from a particular set of data and updates the global minimum correspondingly. Since the global minimum is shared by all the threads, so a mutex lock has been used to avoid any race conditions.

3. GPU PROGRAMMING USING CUDA

GPU computing is possible because today's GPU does much more than render graphics: It sizzles with a teraflop of floating point performance and crunches application tasks designed for anything from finance to medicine.

3.1 What is CUDA?

CUDA is a parallel computing platform and application programming interface (API) model created by NVIDIA. It allows software developers to use a CUDA-enabled graphics processing unit (GPU) for general purpose processing – an approach known as GPGPU. The CUDA platform is a software layer that gives direct access to the GPU's virtual instruction set and parallel computational elements, for the execution of compute kernels.

The CUDA platform is designed to work with programming languages such as C, C++ and Fortran. This accessibility makes it easier for specialists in parallel programming to utilize GPU resources, as opposed to previous API solutions like Direct3D and OpenGL, which required advanced skills in graphics programming.

3.2 How CUDA works?

The GPU programming model is very different from a CPU programming model as it inherently supports parallel programming. A usual template followed to program a GPU is as below:

- Transfer the data to the GPU memory.
- Launch the GPU kernel from the host program.
- Wait till GPU finishes off the calculations.
- Transfer the results back to CPU memory.

Thus, a CUDA program consists of two parts in its code: a) host program which runs on CPU and b) kernel which does the calculations on GPU. A brief description of the various programming features is as below:

- The execution of the kernels is done in threads and blocks. The structure of the grid of threads and blocks is passed in the host code while calling the kernel using a unique <<<…>>> syntax. A typical kernel call looks like:

  ```
  funcAdd<<<Grid,Blocks,1>>>(A, B, C)
  ```

- These threads and blocks are executed on the various multiprocessors of the GPU. The threads of each block are executed concurrently in one multi-processor; each of them may process some maximum number of blocks depending on the resources used by each thread. As the blocks get executed, new blocks are launched on the vacated processors.

- The threads are processed in groups of 32 called warps. The splitting into warps is done according to increasing thread IDs.

- Each thread and block has a unique ID, which is defined by a 3 component vector namely threadIdx and blockIdx respectively. The thread IDs are with respect to their position in the block. A typical grid of threads and blocks is shown in figure 1.

- Threads within a block can synchronize using a barrier synchronization function __syncthreads(). This function stops the execution of threads until all the threads have reached that point.

- The threads are executed in parallel unless their paths diverge or they read/write from the same memory location. In the former case, the processor executes each path serially, disabling threads on other paths until the paths converge again. In the latter case too, the access is serialized, but in the case of non-atomic write, the outcome of the write is not guaranteed. Also the order in which access occurs is undefined.

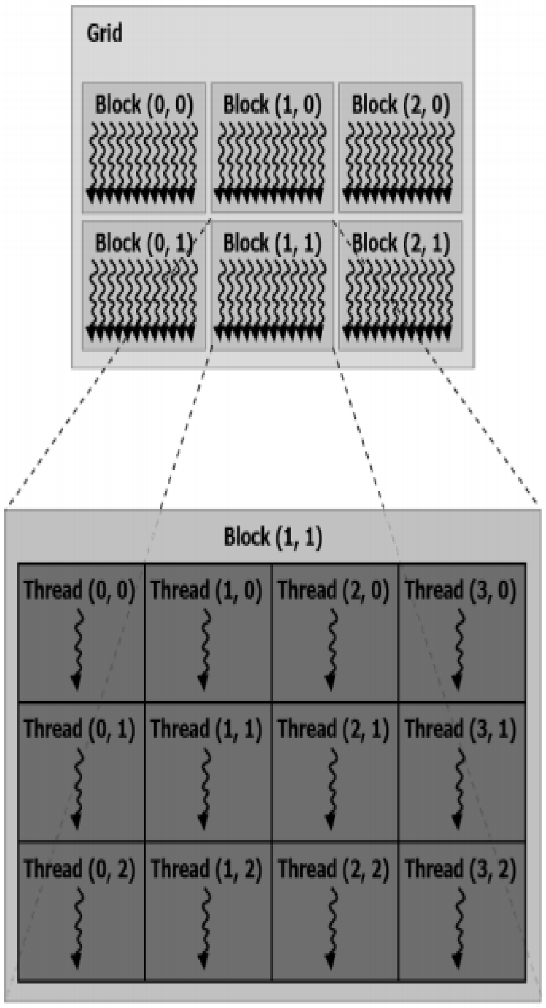


Figure 1: Grid and Thread Blocks

GPU derives its computing from an array of Streaming Multiprocessors (SMs). Each multiprocessor consists of eight Scalar Processor cores and a multithreaded instruction unit. The multiprocessor handles multiples threads using the architecture called SIMT, which maps each thread to a scalar processor. Further each multiprocessor has on-chip memory of different types: a set of 32-bit registers, an on-chip shared memory shared by all the scalar processors, a constant cache to speed up reading from constant memory and a texture cache which speeds up read from the texture memory space. A schematic of the GPU hardware is shown in figure 3.

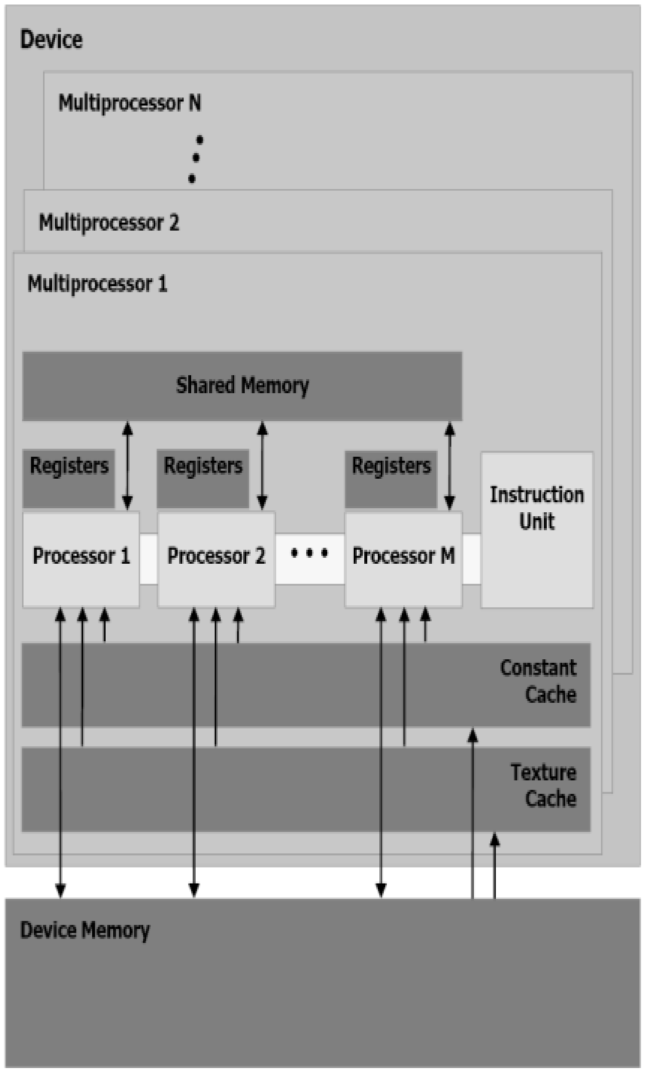


Figure 3: Multiprocessors on GPU

3.3 GPU Memory Hierarchy

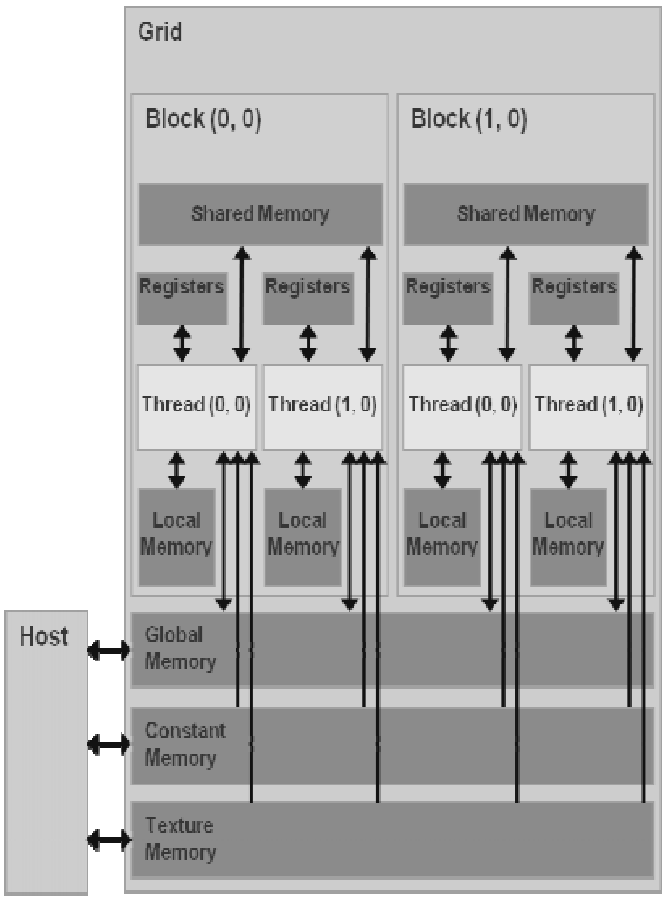


Figure 2: GPU Memory Model

GPU has different levels of memory - global

memory, shared memory, constant memory, texture memory and local memory and registers.

Registers and local memory

Each multiprocessor has certain number of 32-bit registers. They are allocated per thread and only that thread can access the contents of the register. These are the fastest accessible memories, and are allocated at the compile time depending on the requirement of a thread. If the threads in a block need more registers than provided in a multiprocessor, the kernel fails to launch. Local Memory is also available per thread and the compiler automatically allocates certain variables on the local memory, like a big array. Local memory is not cached and hence its access is as costly as global memory.

Shared Memory

Shared memory is an on-chip memory and hence, it is faster than local or global memory. Every multiprocessor has 16kB of shared memory, and is accessible by all the threads in a block. The shared memory of one block cannot be accessed by another block even if they are running on the same multiprocessor. To achieve high bandwidth, the shared memory is divided into memory banks, which can be accessed simultaneously. If threads in the same half-warp access the same bank address, then there is a bank conflict and the access is serialized, thus reducing the effective bandwidth.

Global Memory

It is an off-chip memory and is accessible to all threads. The data transferred from CPU is stored in the global memory initially. The memory is connected to all the processors through a wide bus, which can be up to 512 bits wide. Since the memory is not cached, access to the global memory is expensive and should be avoided as far as possible. The memory bandwidth is effectively utilized when the threads in a half-warp access the memory in a sequence, resulting in coalescing of the memory transactions.

Constant Memory

Used for storing constant values that does not change over time. Constant memory resides in the global memory. Constant memory accesses are cached in constant cache. A separate constant cache exists for each multiprocessor and it is a read only cache.
For example the constant memory on any architecture is 64KB
Cache size per multiprocessor on Fermi and Kepler are 8KB

Texture Memory

CUDA also supports a part of the texturing hardware which is used for graphics processing. This allows the texture memory to be read using device functions called texture fetches. As the texture memory is cached, it is much faster than global memory access. It also supports various attributes like texture coordinates, which can be used to access the texture elements, interpolation options, normalization, etc.

4. FIND MINIMUM IN AN ARRAY USING CUDA

The following is an insight into the effectiveness of CUDA in solving a general problem, like finding minimum in an array.

4.1 The Problem Statement

The aim of this problem is to implement the problem of finding the minimum element in an array using the CUDA framework by solving the problem using the computation power of the highly parallel structure of the GPU. The aim of this task is to observe how the time required by the program to complete its task varies as the size of the array is changed.

4.2 Approach

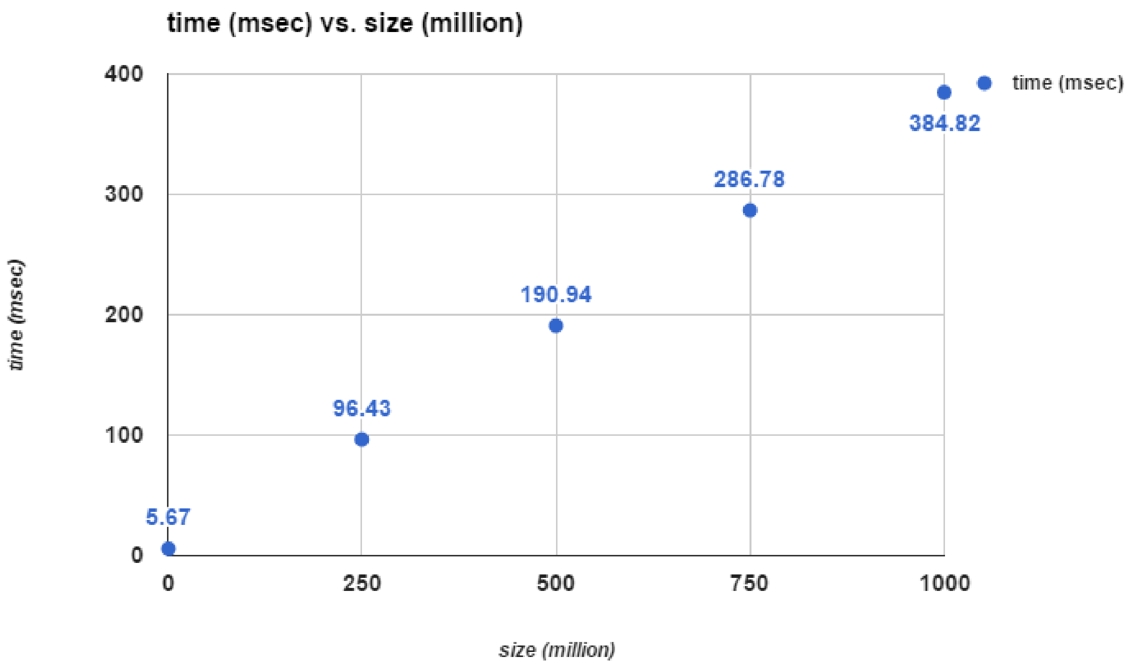
The domain of GPU Programming using the CUDA framework is relatively new, so extensive materials for study were not available. The approach towards solving the problem was hence, somewhat intuitive.

```
/* CUDA 6.5 */
/* Finding minimum element from a character array. All elements in the array
are random values ranging from 15 to 31. Only at one random position in the
array the value is 2 to ensure correct result is obtained */

#include<stdlib.h>
```

```
#include<stdio.h>
#include<time.h>
#include<limits.h>
#include<cuda.h>
#include<cuda_runtime.h>
#include<device_launch_parameters.h>
#include<sm_11_atomic_functions.h>
#define SIZE (1000000)          // Array Size : 1 million.
#define NO_OF_THREADS 128       // Number of threads per block
#define NO_OF_BLOCKS 64         // Number of blocks

// The function to find the minimum element which has been parallelised.
__global__ void FindMin(char *a,char *b, int n)
{
    __shared__ char cache[NO_OF_THREADS];   // Cache array is shared by threads
in a thread block.

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

    int cacheIndex = threadIdx.x;
    cache[cacheIndex] = CHAR_MAX;

    for (int j = i; j < n;)
    {
        if (a[j] < cache[cacheIndex])
            cache[cacheIndex] = a[j];

        j += blockDim.x * gridDim.x;
    }

    __syncthreads();        // A thread cannot proceed further unless all other
threads reaches this point.

    char ans = CHAR_MAX;
    /* Finding the minimum element from an array (cache) where each thread in a
thread block has calculated
       its own minimum element */
    if (threadIdx.x == 0)
    {
        for (int j = 0; j < NO_OF_THREADS; j++)
        {
            if (ans>cache[j])
                ans = cache[j];
        }
        // Saving the minimum element calculated by the thread block.
        b[blockIdx.x] = ans;
    }
}

int main()
{
    char *a;
    char *d_a;
    char *minarr;
    char *d_minarr;
    char min;
    int i;
    cudaEvent_t start, stop;
    float milliseconds = 0;
    cudaEventCreate(&start);
    cudaEventCreate(&stop);
    char hash[5] = { '5', '6', '7', '8', '9' };

    // Allocate memory for array a and minarr
    a = (char*)malloc(SIZE*sizeof(char));
    minarr = (char*)malloc(NO_OF_BLOCKS*sizeof(char));

    // Allocate memory on the GPU for array a and minarr
    cudaMalloc(&d_a, SIZE*sizeof(char));
    cudaMalloc(&d_minarr, NO_OF_BLOCKS*sizeof(char));

    // Randomly assigning values to the array
    for (int i = 0; i < SIZE; i++)
        a[i] = hash[((rand()%5))];

    a[50] = '2';

    // Copy the values in array a and minarr to the memory space allocated in
the GPU
    cudaMemcpy(d_a, a, SIZE*sizeof(char), cudaMemcpyHostToDevice);
    cudaMemcpy(d_minarr, minarr, NO_OF_BLOCKS*sizeof(char),
cudaMemcpyHostToDevice);

    cudaEventRecord(start);
    // Launching the kernel
    FindMin <<< NO_OF_BLOCKS, NO_OF_THREADS >>>(d_a,d_minarr, SIZE);
    cudaThreadSynchronize();
    cudaEventRecord(stop);

    // The result array d_minarr is copied to minarr
    cudaMemcpy(minarr, d_minarr, NO_OF_BLOCKS *sizeof(char),
cudaMemcpyDeviceToHost);

    min = minarr[0];
    // Finding the global minimum form minarr array.
    for (i = 1; i < NO_OF_BLOCKS; i++)
    {
        if (minarr[i] < min)
            min = minarr[i];
    }

    cudaEventSynchronize(stop);
    cudaEventElapsedTime(&milliseconds, start, stop);

    printf("MIN= %c \n", (min));
    printf("TIME = %f\n", (milliseconds));
    // Free the memory which was allocated for a and minarr in the host's
memory
    free(a);
    free(minarr);

    // Free the memory which was allocated for a and minarr in the GPU
    cudaFree(d_a);
    cudaFree(d_minarr);

    return 0;
}
```

4.3 Results

The execution time was recorded for different size of the array and the following graph was obtained.



5. GPU BASED MATRIX INVERSION USING CUDA

This section further explores the effectiveness of CUDA in solving a challenging problem, like finding the inverse of a large matrix. In linear

algebra, an *n*-by-*n* square matrix A is called invertible (also nonsingular or no degenerate) if there exists an *n*-by-*n* square matrix B such that

$$AB = BA = I_n$$

Where I_n denotes the *n*-by-*n* identity matrix.

5.1 The Problem Statement

The aim of this problem is to implement the problem of finding the inverse of a matrix using the CUDA framework by solving the problem using the computation power of the highly parallel structure of the GPU. The aim of this task is to observe how the time required by the program to complete its task varies as the size of the matrix is changed.

5.2 Approach

A variant of Gaussian elimination called Gauss–Jordan elimination can be used for finding the inverse of a matrix, if it exists. If A is a n by n square matrix, then one can use row reduction to compute its inverse matrix, if it exists. First, the n by n identity matrix is augmented to the right of A, forming a n by 2n block matrix [A | I]. Now through application of elementary row operations, find the reduced echelon form of this n by 2n matrix. The matrix A is invertible if and only if the left block can be reduced to the identity matrix I; in this case the right block of the final matrix is A^{-1}. If the algorithm is unable to reduce the left block to I, then A is not invertible.

```
/* CUDA 6.5 */
/* Inverse of a Matrix using Gauss-Jordan Elimination Method. */
#include<stdlib.h>
#include<stdio.h>
#include<time.h>
#include<limits.h>
```

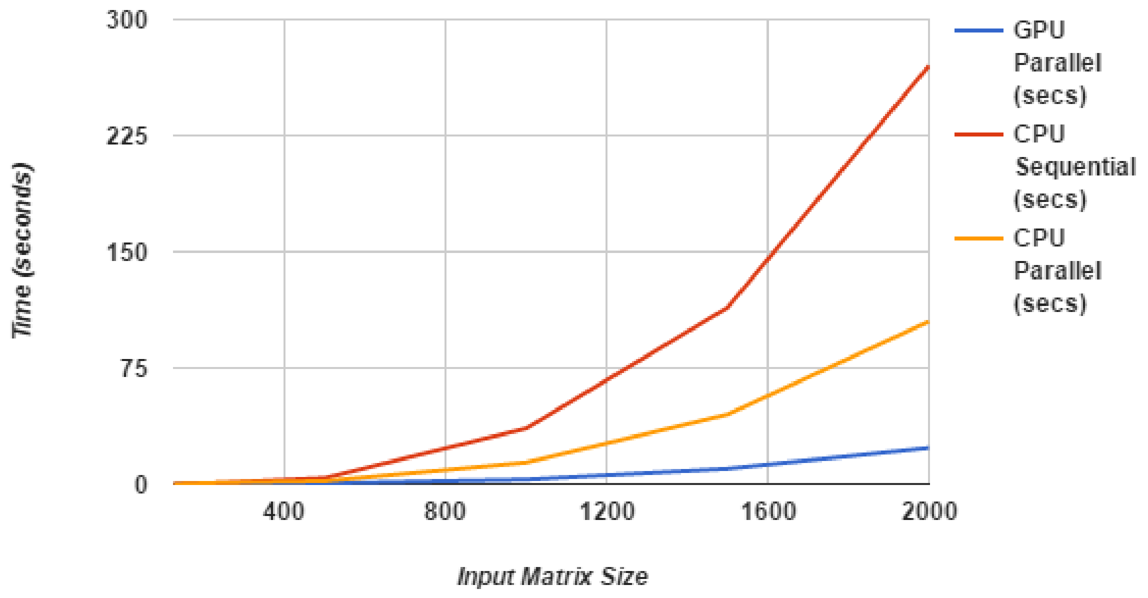
```
#include<cuda.h>
#include<cuda_runtime.h>
#include<device_launch_parameters.h>
#define ORDER 2000
#define Index(i,j,k) ((i*k)+j)  // Index Translation from 2D to 1D

// The initializeMatrix function assigns random values to matrix elements
void initializeMatrix(double *matrix, int row, int column)
{
    srand(time(NULL));
    for (int i = 0; i < row; i++)
    {
        for (int j = 0; j < ORDER; j++)
        {
            matrix[Index(i, j, column)] = (rand() % 50) + 1;
        }
        for (int j = ORDER; j < column; j++)
        {
            if (i == (j - ORDER))
                matrix[Index(i, j, column)] = 1.0;
            else
                matrix[Index(i, j, column)] = 0.0;

        }
    }
}

// The findRow function finds a row with non-zero pivot element when a pivot
element value is zero
int findRow(double *matrix, int row, int pivotRowNumber)
{
    int i = pivotRowNumber + 1;
    for (i = pivotRowNumber + 1; i < row; i++)
    {
        if (matrix[i] != 0.0)
            return i;
    }
    return -1;
}

// The parallelised function switches the values of two rows
__global__ void switchRows(double *d_matrix, int i, int rowNumber, int n)
{
    int id = blockIdx.x*blockDim.x + threadIdx.x;
    if (id < n)
    {
        double temp = d_matrix[Index(i, id, n)];
        d_matrix[Index(i, id, n)] = d_matrix[Index(rowNumber, id, n)];
        d_matrix[Index(rowNumber, id, n)] = temp;
    }
}

// The parallelised function makes the pivot element value equal to one
```

```
__global__ void normalizePivot(double *d_matrix, int n, int rowNumber, double
pivotValue)
{
    int id = blockIdx.x*blockDim.x + threadIdx.x;
    if (id < n)
    {
        d_matrix[Index(rowNumber, id, n)] = (d_matrix[Index(rowNumber, id, n)]
/ pivotValue) ;
    }
}

// The parallelised function updates values of other rows so that elements
above and below pivot element are zero
__global__ void updateOtherRows(double *d_matrix, int R, int C, int i, double
*d_factor)
{
    int row = blockIdx.y*blockDim.y + threadIdx.y;
    int column = blockIdx.x*blockDim.x + threadIdx.x;

    if (row < R && row != i)
    {
        if (column < C)
        {
            d_matrix[Index(row, column, C)] = (d_matrix[Index(row, column, C)]
- (d_factor[row] * d_matrix[Index(i, column, C)]));
        }
    }
}


int main()
{
    int row = ORDER, column = 2 * ORDER;
    int NO_OF_THREADS_X = 32, NO_OF_THREADS_Y = 12;
    int NO_OF_BLOCKS_X = ((column + NO_OF_THREADS_X - 1) / NO_OF_THREADS_X);
    int NO_OF_BLOCKS_Y = ((row + NO_OF_THREADS_Y - 1) / NO_OF_THREADS_Y);
    int threadsPerBlock = 384;
    int numberOfBlocks = ((column + threadsPerBlock - 1) / (threadsPerBlock));
    double *factor, *matrix, *d_matrix, *testColumnMatrix, *d_factor;
    dim3 threads(NO_OF_THREADS_X, NO_OF_THREADS_Y);
    dim3 blocks(NO_OF_BLOCKS_X, NO_OF_BLOCKS_Y);
    float total = 0;

    testColumnMatrix = new double[row];
    factor = new double[row];

    //Allocate memory in host for matrix
    matrix = new double[(row*column)];

    //Allocate memory in device for matrix
    cudaMalloc(&d_matrix, (row*column)*sizeof(double));
    cudaMalloc(&d_factor, (row)*sizeof(double));

    //Initialize 2D Matrix
    initializeMatrix(matrix, row, column);

    cudaEvent_t start, stop;
    cudaEventCreate(&start);
    cudaEventCreate(&stop);
    cudaEventRecord(start);
    cudaMemcpy(d_matrix, matrix, (row*column)*sizeof(double),
cudaMemcpyHostToDevice);
    for (int i = 0; i < row; i++)
    {
        // Copy ith column from matrix stored in device to host to check
whether Pivot element (ith element of ith row) is zero
        cudaMemcpy2D(testColumnMatrix, sizeof(double), &d_matrix[i],
column*sizeof(double), sizeof(double), row, cudaMemcpyDeviceToHost);

        // If pivot element is zero find a row with non-zero pivot element and
interchange row elements.
        if (testColumnMatrix[i] == 0.0)
        {
            int rowNumber = findRow(testColumnMatrix, row, i);
            if (rowNumber == -1)
            {
                printf("Given Matrix Impossible to invert. %d\n",i);
                return 0;
            }
            else
            {
                // Interchange rows if required
                switchRows << <numberOfBlocks, threadsPerBlock >> >(d_matrix,
i, rowNumber, column);
                cudaDeviceSynchronize();
            }
        }

        // Copy ith column from matrix stored in device to host
        cudaMemcpy2D(testColumnMatrix, sizeof(double), &d_matrix[i],
column*sizeof(double), sizeof(double), row, cudaMemcpyDeviceToHost);

        // Normalize Pivot ( Make pivot element 1 )
        double pivotValue = testColumnMatrix[i];
        normalizePivot << <numberOfBlocks, threadsPerBlock >> >(d_matrix,
column, i, pivotValue);
        cudaDeviceSynchronize();

        // Update other rows.
        for (int k = 0; k < row; k++)
            factor[k] = testColumnMatrix[k];

        cudaMalloc(&d_factor, (row)*sizeof(double));
        cudaMemcpy(d_factor, factor, row*sizeof(double),
cudaMemcpyHostToDevice);
        updateOtherRows << <blocks, threads >> >(d_matrix, row, column, i,
d_factor);
        cudaDeviceSynchronize();
    }
    cudaMemcpy(matrix, d_matrix, (row*column)*sizeof(double),
cudaMemcpyDeviceToHost);
    cudaEventRecord(stop);
    cudaEventSynchronize(stop);
    cudaEventElapsedTime(&total, start, stop);

    // Free the memory which was allocated for d_matrix and d_factor in GPU
    cudaFree(d_matrix);
    cudaFree(d_factor);

    printf("Time = %f msecs\n", total);
    return 0;
}
```

5.3 Results

The execution time for both sequential program in CPU and parallel program in GPU using CUDA for different matrix size are recorded.

Matrix Size	CPU Sequential (seconds)	CPU Parallel (seconds)	GPU Parallel (seconds)
125 X 125	0	0	0.044
250 X 250	1	1	0.118
500 X 500	4	2	0.494
1000 X 1000	36	14	2.971
1500 X 1500	114	45	10.024
2000 X 2000	270	105	23.390



6. APPLICATIONS

The matrix inversion is required during image reconstruction process and in Hill Cipher to obtain the inverse of a key matrix during the decryption process to get back the original plaintext.

6.1 Hill Cipher

In classical cryptography, the Hill cipher is a polygraphic substitution cipher based on linear algebra. Invented by Lester S. Hill in 1929, it was the first polygraphic cipher in which it was practical (though barely) to operate on more than three symbols at once. To encrypt a message, each block of n letters (considered as an n-component vector) is multiplied by an invertible n × n matrix, again modulus 26. To decrypt the message, each block is multiplied by the inverse of the matrix used for encryption.

In order to decrypt, we turn the cipher text back into a vector, then simply multiply by the inverse matrix of the key matrix. The inverse of the key matrix in Hill Cipher can be obtained by using Gauss-Jordan elimination method but all operations are performed modulo n where n is the number of different characters possible in a plaintext.

The code to find inverse of a key matrix in Hill Cipher is given below:

```
/* CUDA 6.5 */
/* Inverse of a key Matrix for Hill Cipher using Gauss-Jordan Elimination
Method. */
#include<stdlib.h>
#include<stdio.h>
#include<time.h>
#include<limits.h>
#include<cuda.h>
#include<cuda_runtime.h>
#include<device_launch_parameters.h>
#define ORDER 3
#define Index(i,j,k) ((i*k)+j)  // Index Translation from 2D to 1D
#define MOD 29

// This function will find the multiplicative modular inverse.
int modInverse(int a, int m)
{
    int m0 = m, t, q;
    int x0 = 0, x1 = 1;

    if (m == 1)
        return 0;

    while (a > 1)
    {
        // q is quotient
        q = a / m;

        t = m;

        // m is remainder now, process same as
        // Euclid's algo
        m = a % m, a = t;

        t = x0;
        x0 = x1 - q * x0;

        x1 = t;
    }

    // Make x1 positive
    if (x1 < 0)
        x1 += m0;

    return x1;
}

// The initializeMatrix function assigns random values to matrix elements
void initializeMatrix(int *matrix, int row, int column)
{
    srand(time(NULL));
    for (int i = 0; i < row; i++)
    {
        for (int j = 0; j < ORDER; j++)
        {
            matrix[Index(i, j, column)] = (rand() % MOD);
            //scanf("%d", &matrix[Index(i, j, column)]);
        }
        for (int j = ORDER; j < column; j++)
        {
            if (i == (j - ORDER))
                matrix[Index(i, j, column)] = 1;
            else
                matrix[Index(i, j, column)] = 0;

        }
    }
}

// The findRow function finds a row with non-zero pivot element when a pivot
element value is zero
int findRow(int *matrix, int row, int pivotRowNumber)
{
    int i = pivotRowNumber + 1;
    for (i = pivotRowNumber + 1; i < row; i++)
    {
        if (matrix[i] != 0)
            return i;
    }
    return -1;
}

// The parallelised function switches the values of two rows
__global__ void switchRows(int *d_matrix, int i, int rowNumber, int n)
{
    int id = blockIdx.x*blockDim.x + threadIdx.x;
    if (id < n)
    {
        int temp = d_matrix[Index(i, id, n)];
        d_matrix[Index(i, id, n)] = d_matrix[Index(rowNumber, id, n)];
        d_matrix[Index(rowNumber, id, n)] = temp;
    }
}

// The parallelised function makes the pivot element value equal to one
__global__ void normalizePivot(int *d_matrix, int n, int rowNumber, int
pivotValue)
{
    int id = blockIdx.x*blockDim.x + threadIdx.x;
    if (id < n)
    {
        d_matrix[Index(rowNumber, id, n)] = (d_matrix[Index(rowNumber, id, n)]
* pivotValue)%MOD;
    }
}

// The parallelised function updates values of other rows so that elements
above and below pivot element are zero
__global__ void updateOtherRows(int *d_matrix, int R, int C, int i, int
*d_factor)
{
    int row = blockIdx.y*blockDim.y + threadIdx.y;
    int column = blockIdx.x*blockDim.x + threadIdx.x;

    if (row < R && row != i)
    {
        if (column < C)
        {
            d_matrix[Index(row, column, C)] = (d_matrix[Index(row, column, C)]
- (d_factor[row] * d_matrix[Index(i, column, C)])%MOD)%MOD;
            if (d_matrix[Index(row, column, C)] < 0)
                d_matrix[Index(row, column, C)] += MOD;
        }
    }
}

int main()
{
    int row = ORDER, column = 2 * ORDER;
    int NO_OF_THREADS_X = 32, NO_OF_THREADS_Y = 12;
    int NO_OF_BLOCKS_X = ((column + NO_OF_THREADS_X - 1) / NO_OF_THREADS_X);
    int NO_OF_BLOCKS_Y = ((row + NO_OF_THREADS_Y - 1) / NO_OF_THREADS_Y);
    int threadsPerBlock = 384;
    int numberOfBlocks = ((column + threadsPerBlock - 1) / (threadsPerBlock));
    int *factor, *matrix, *d_matrix, *testColumnMatrix, *d_factor,
*inverse_matrix;
    dim3 threads(NO_OF_THREADS_X, NO_OF_THREADS_Y);
    dim3 blocks(NO_OF_BLOCKS_X, NO_OF_BLOCKS_Y);
    float total = 0;

    testColumnMatrix = new int[row];
    factor = new int[row];
    //Allocate memory in host for matrix
    matrix = new int[(row*column)];
    inverse_matrix = new int[(row*column)];

    //Allocate memory in device for matrix
    cudaMalloc(&d_matrix, (row*column)*sizeof(int));
    cudaMalloc(&d_factor, (row)*sizeof(int));

    //Initialize 2D Matrix
    initializeMatrix(matrix, row, column);

    cudaEvent_t start, stop;
    cudaEventCreate(&start);
    cudaEventCreate(&stop);
    cudaEventRecord(start);
    cudaMemcpy(d_matrix, matrix, (row*column)*sizeof(int),
cudaMemcpyHostToDevice);
    for (int i = 0; i < row; i++)
    {
        // Copy ith column from matrix stored in device to host to check
whether Pivot element (ith element of ith row) is zero
        cudaMemcpy2D(testColumnMatrix, sizeof(int), &d_matrix[i],
column*sizeof(int), sizeof(int), row, cudaMemcpyDeviceToHost);

        // If pivot element is zero find a row with non-zero pivot element and
interchange row elements.
        if (testColumnMatrix[i] == 0)
        {
            int rowNumber = findRow(testColumnMatrix, row, i);
            if (rowNumber == -1)
            {
                printf("Given Matrix Impossible to invert. %d\n", i);
                return 0;
            }
            else
            {
                // Interchange rows if required
                switchRows << <numberOfBlocks, threadsPerBlock >> >(d_matrix,
i, rowNumber, column);
                cudaDeviceSynchronize();
            }
        }

        // Copy ith column from matrix stored in device to host
        cudaMemcpy2D(testColumnMatrix, sizeof(int), &d_matrix[i],
column*sizeof(int), sizeof(int), row, cudaMemcpyDeviceToHost);

        // Normalize Pivot ( Make pivot element 1 )
        int pivotValue = modInverse(testColumnMatrix[i],MOD);
        normalizePivot << <numberOfBlocks, threadsPerBlock >> >(d_matrix,
column, i, pivotValue);
        cudaDeviceSynchronize();
        // Update other rows.
        for (int k = 0; k < row; k++)
            factor[k] = testColumnMatrix[k];

        cudaMalloc(&d_factor, (row)*sizeof(int));
        cudaMemcpy(d_factor, factor, row*sizeof(int), cudaMemcpyHostToDevice);
        updateOtherRows << <blocks, threads >> >(d_matrix, row, column, i,
d_factor);
        cudaDeviceSynchronize();
    }
    cudaMemcpy(inverse_matrix, d_matrix, (row*column)*sizeof(int),
cudaMemcpyDeviceToHost);
    cudaEventRecord(stop);
    cudaEventSynchronize(stop);
    cudaEventElapsedTime(&total, start, stop);

    // Free the memory which was allocated for d_matrix and d_factor in GPU
    cudaFree(d_matrix);
    cudaFree(d_factor);

    printf("Time = %f msecs\n", total);
    return 0;
}
```

6.2 Example

Some key matrices and their inverse are given below, assuming that plaintext can have twenty nine different characters.

Order: 3x3

K = [4 4 11 1 5 19 24 5 23] **K⁻¹ =** [24 2 2 15 14 9 7 10 25]

Order: 4x4

K = [4 19 2 22 22 15 23 25 10 17 19 7 20 22 28 3] **K⁻¹ =**
[17 27 10 4 24 3 15 25 18 10 16 18 26 14 22 9]

Order: 5x5

K = [13 15 28 1 13 23 20 13 24 18 24 27 17 12 28 2 14 10 21 14 1 8 17 7 3] **K⁻¹ =**
[9 26 21 2 25 26 28 10 28 27 27 28 28 7 1 15 2 6 22 6 20 24 16 27 26]

Order: 10x10

K =

[18 0 3 28 13 26 23 18 0 2 6 7 22 25 27 25 4 3 1 5 16 25 4 11 13 11 10 10 5 19 5 16 20 4 0 2 13 22 8 15 16 20 10 22

K^{-1} =

[24 8 27 18 16 16 1 15 21 23 11 15 6 3 13 27 5 11 16 13 16 18 24 25 18 5 13 16 13 10 19 22 12 3 0 19 3 7 17 19 21

Order: 20x20

Key Matrix, K =

[24 17 7 12 9 9 7 12 27 8 21 17 4 21 21 14 15 21 4 24 24 4 26 24 16 3 2 7 28 17 22 9 6 10 23 2 24 15 22 17 5 17 4 1

Inverse of Key Matrix, K^{-1} =

[10 28 25 9 23 1 22 9 3 25 4 10 22 23 2 18 18 17 1 19 26 20 3 5 25 2 11 17 1 10 20 8 28 28 18 2 0 23 28 21 15 13 1

7. CONCLUSION AND FUTURE WORK

The execution time of matrix inversion reduces to almost half when we implement the Gauss-Jordan Elimination algorithm using

multiple threads. We can further reduce the execution time if we implement and execute the algorithm in a GPU using CUDA. The execution time of the parallel program using CUDA in GPU takes less than one-tenth of the time taken by a sequential program in a CPU when the order of the input matrix size is 2000 x 2000.

As a conclusion, it can be said that this implementation is successful. The execution time is as low as expected.

FUTURE WORK

Gauss-Jordan Elimination method is one of the methods to find an inverse of a matrix. Cholesky Decomposition, Gaussian Elimination Triangular Solver can also be used to find an inverse of a matrix. These methods can be implemented in GPU using CUDA and we can study their performance.

The scope for future work in the domain of developing more optimized parallel algorithms is immense. The CUDA framework provides scope for optimization both in terms of software as well as hardware, and it will be extremely interesting to delve deeper into the possibilities that the CUDA framework has to offer.

8. REFERENCES

- https://en.wikipedia.org/wiki/Graphics_processing_unit
- https://en.wikipedia.org/wiki/General-purpose_computing_on_graphics_processing_units
- https://www.nvidia.com/object/what-is-gpu-computing.html
- http://docs.nvidia.com/cuda/cuda-getting-started-guide-for-microsoft-windows/#axzz3cRo5p3v6
- https://developer.nvidia.com/maxwell-compute-architecture
- http://www.notebookcheck.net/NVIDIA-GeForce-840M.105681.0.html
- https://en.wikipedia.org/wiki/Maxwell_(microarchitecture)
- https://en.wikipedia.org/wiki/Gaussian_elimin

ation

APPENDIX A – CUDA INSTALLATION GUIDELINES

The CUDA Framework can be installed on Linux or Windows, and used to program the GPU provided that the GPU is a CUDA Enabled GPU manufactured by NVIDIA. We have installed Visual Studio 2013, and integrated the CUDA v6.5 with it, in order to write and run CUDA code on the GPU using Windows. All the installation files have been submitted to our guide for convenience in future work, and prevention of the version conflict problem in future,

To use CUDA on your system, you will need the following installed:

- A CUDA-capable GPU
- A supported version of Microsoft Windows
- A supported version of Microsoft Visual Studio
- The NVIDIA CUDA Toolkit

STEP 1: Verify that your GPU is CUDA capable

To verify that your GPU is CUDA-capable, open the Control Panel (Start > Control Panel) and double click on System. In the System Properties window that opens, click the Hardware tab, then Device Manager. Expand the Display adapters entry. There you will find the vendor name and model of your graphics card. If it is an NVIDIA card that is listed in http://developer.nvidia .com/cuda-gpus your GPU is CUDA-capable.

STEP 2: Download the NVIDIA CUDA Toolkit

The NVIDIA CUDA Toolkit is available at http://developer.nvidia.com/cuda-downloads. Choose the platform you are using and one of the following installer formats:

1. Network Installer: A minimal installer which later downloads packages required for installation. Only the packages selected during the selection phase of the installer are downloaded. This installer is useful for users who want to minimize download time.

2. Full Installer: An installer which contains all the components of the CUDA Toolkit and does not require any further download. This installer is useful for systems which lack network access and for enterprise deployment.

The CUDA Toolkit installs the CUDA driver and tools needed to create, build and run a CUDA application as well as libraries, header files, CUDA samples source code, and other resources.

The download can be verified by comparing the MD5 checksum posted at http://developer.nvidia.com/cuda-downloads/checksums with that of the downloaded file. If either of the checksums differs, the downloaded file is corrupt and needs to be downloaded again. To calculate the MD5 checksum of the downloaded file, follow the instructions at http://support.microsoft.com/kb/889768

STEP 3: Installing CUDA

Before installing the toolkit, you should read the Release Notes, as they provide details on installation and software functionality.

Note: The driver and toolkit must be installed for CUDA to function. If you have not installed a stand - alone driver, install the driver from the NVIDIA CUDA Toolkit.

Note: The installation may fail if Windows Update starts after the installation has begun. Wait until Windows Update is complete and then try the installation again.

<u>Graphical Installation</u>

Install the CUDA Software by executing the CUDA installer and following the on-screen prompts.

Sub package Details

- Display Driver

  Required to run CUDA applications.

- CUDA Toolkit

  The CUDA Toolkit installation defaults to C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v6.5. This directory contains the following:

  - Bin\

    the compiler executable and runtime libraries

  - Include\

    the header files needed to compile CUDA programs

  - Lib\

    the library files needed to link CUDA programs

  - Doc\

    The CUDA documentation, including:

    - CUDA C Programming Guide
    - CUDA C Best Practices Guide
    - Documentation for the CUDA libraries
    - Other CUDA Toolkit-related documentation

- CUDA Visual Studio Integration

  The CUDA Visual Studio Integration registers the CUDA plugins with the supported and installed versions of Visual Studio on the system and installs Nsight Visual Studio Edition. This integration allows for CUDA development within Visual Studio.

- **CUDA Samples**

  The CUDA Samples contain source code for many example problems and templates with Microsoft Visual Studio 2010, 2012, and 2013 projects.

  The CUDA Samples installation defaults to

  C:\ProgramData\NVIDIA Corporation\CUDA Samples\v7.0.

  Note : C: \ ProgramData \ is a hidden folder. It can be made visible within Windows Explorer by enabling it through the menu options within Windows Explorer (Tools | Options).

Uninstalling the CUDA Software

All sub packages can be uninstalled through the Windows Control Panel by using the Programs and Features widget.

STEP 4: Use a Suitable Driver Model

On Windows 7 and later, the operating system provides two driver models under which the NVIDIA Driver may operate:

- The WDDM driver model is used for display devices.
- The Tesla Compute Cluster (TCC) mode of the NVIDIA Driver is available for non-display devices such as NVIDIA Tesla GPUs; it uses the

Windows WDM driver model.

The TCC driver mode provides a number of advantages for CUDA applications on GPUs that support this mode. For example:

- TCC eliminates the timeouts that can occur when running under WDDM due to the Windows Timeout Detection and Recovery mechanism for display devices.
- TCC allows the use of CUDA with Windows Remote Desktop, which is not possible for WDDM devices.
- TCC allows the use of CUDA from within processes running as Windows services, which is not possible for WDDM devices.
- TCC reduces the latency of CUDA kernel launches.

TCC is enabled by default on most recent NVIDIA Tesla GPUs. To check which driver mode is in use and/or to switch driver modes, use the nvidia-smi tool that is included with the NVIDIA Driver installation (see nvidia-smi –h for details).

Note: Keep in mind that when TCC mode is enabled for a particular GPU, that GPU cannot be used as a display device.

Note: NVIDIA GeForce GPUs do not support TCC mode.   (Refer to Appendix B, Last point)

STEP 5: Verify the Installation

Before continuing, it is important to verify that the CUDA toolkit can find and communicate correctly with the CUDA-capable hardware. To do this, you need to compile and run some of the included sample programs.

<u>Running the Compiled Examples</u>

The version of the CUDA Toolkit can be checked by running nvcc –V in a Command Prompt window. You can display a Command Prompt window by going to Start > All Programs > Accessories > Command Prompt.

CUDA Samples include sample programs in both source and compiled form. To verify a correct configuration of the hardware and software, it is highly recommended that you run the deviceQuery program located at

C:\ProgramData\NVIDIA Corporation\CUDA Samples\v7.5\bin\win64\Release

This assumes that you used the default installation directory structure. The exact appearance and the output lines might be different on your system. The important outcomes are that a device was found, that the device(s) match what is installed in your system, and that the test passed.

If a CUDA-capable device and the CUDA Driver are installed but deviceQuery reports that no CUDA capable devices are present, ensure the device and driver are properly installed.

Running the bandwidth Test program, located in the same directory as deviceQuery above, ensures that the system and the CUDA-capable device are able to communicate correctly.

The device name and the bandwidth numbers vary from system to system. The important items are the second line, which confirms a CUDA device was found, and the second-to-last line, which confirms that all necessary tests passed.

If the tests do not pass, make sure you do have a CUDA-capable NVIDIA GPU on your system and make sure it is properly installed.

To see a graphical representation of what CUDA can do, run the sample Particles executable at

C:\ProgramData\NVIDIA Corporation\CUDA Samples\v7.5\bin\win64\Release

APPENDIX B – SYSTEM SPECIFICATIONS

NVIDIA GeForce 840M GPU

Architecture	Maxwell
Pipelines	384 - Unified
Core Speed	1029 MHz
Memory Speed	2000 MHz

Memory Bus Width	64 bit
Memory Type	DDR3
Shared Memory	No
DirectX	DirectX , Shader 5.0

Laptop Specifications

- Model : Dell Inspiron 3542 Notebook
- Intel® Core™ i5-4210 CPU@ 1.70 GHz 1.70 GHz
- RAM : 4 GB
- Operating System: Windows 8.1

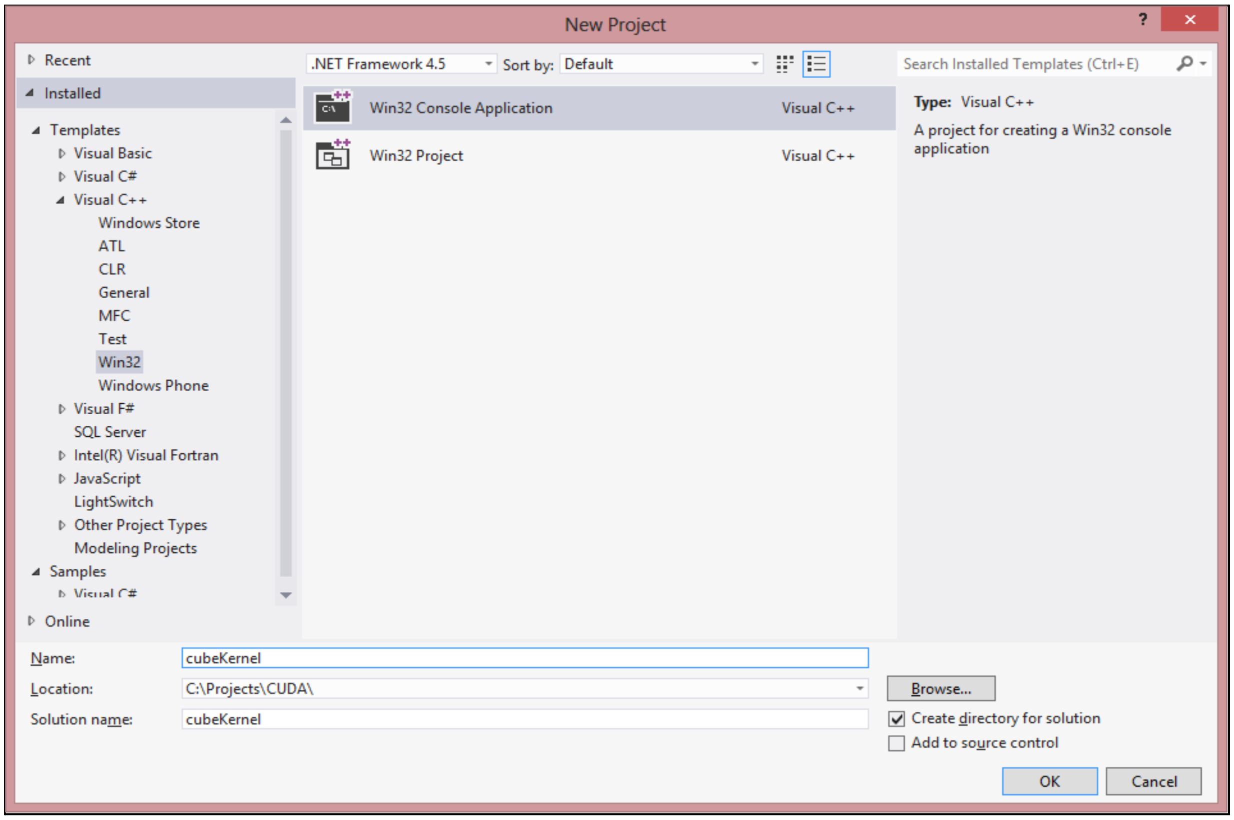
System Specifications

- CUDA GPU Computing Toolkit Version 6.5
- Microsoft Visual Studio Professional 2013

APPENDIX C – COMPILATION AND EXECUTION OF A CUDA PROGRAM

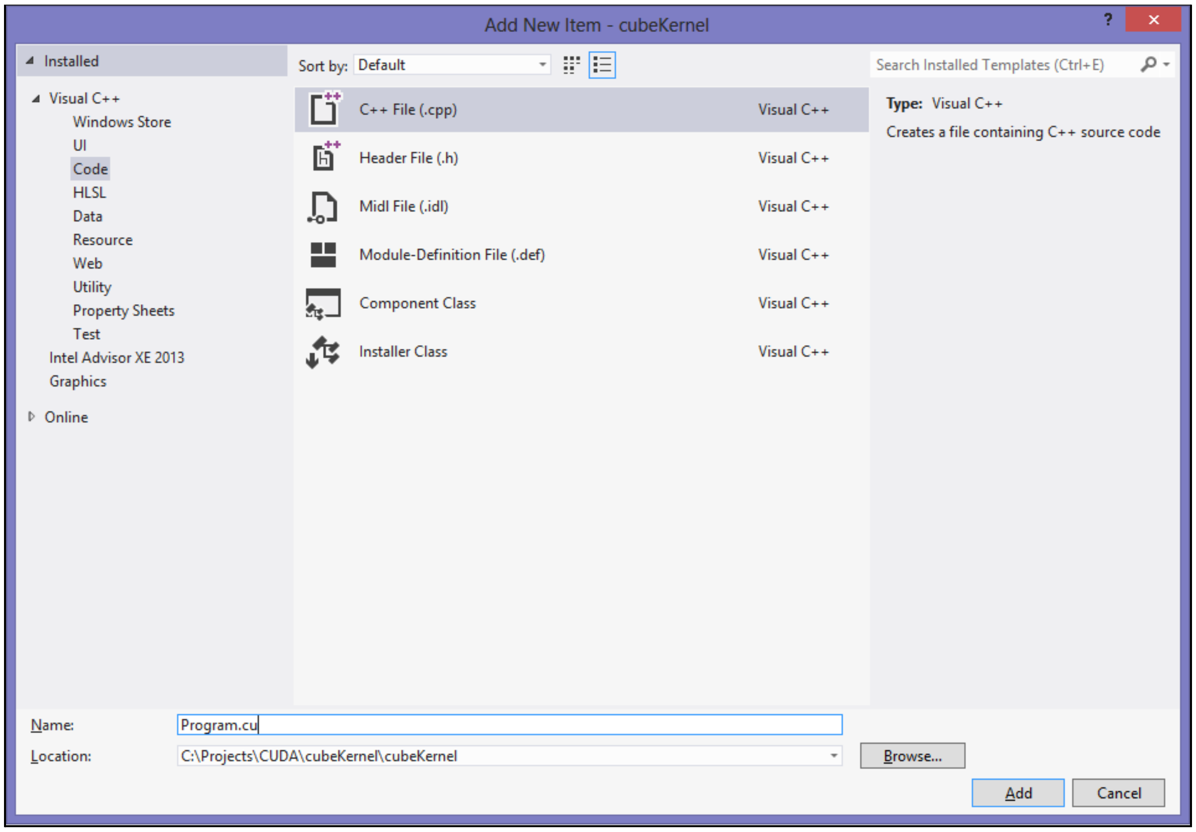
When creating a new CUDA application, the Visual Studio project file must be configured to include CUDA build customizations. To accomplish this, click File-> New | Project... NVIDIA-> CUDA->, and then select a template for your CUDA Toolkit version. For example, selecting the "CUDA 7.5 Runtime" template will configure your project for use with the CUDA 7.5

Toolkit. The new project is technically a C++ project (.vcxproj) that is preconfigured to use NVIDIA's Build Customizations. All standard capabilities of Visual Studio C++ projects will be available.



Type: Visual C++
A project for creating a Win32 console application

Name: cubeKernel
Location: C:\Projects\CUDA\
Solution name: cubeKernel
[✓] Create directory for solution
[ ] Add to source control


Add New Item - cubeKernel
Installed
Visual C++
Windows Store
UI
Code
HLSL
Data
Resource
Web
Utility
Property Sheets
Test
Intel Advisor XE 2013
Graphics
Online
Sort by: Default
Search Installed Templates (Ctrl+E)
C++ File (.cpp) Visual C++
Header File (.h) Visual C++
Midl File (.idl) Visual C++
Module-Definition File (.def) Visual C++
Component Class Visual C++
Installer Class Visual C++
Browse...
Add
Cancel

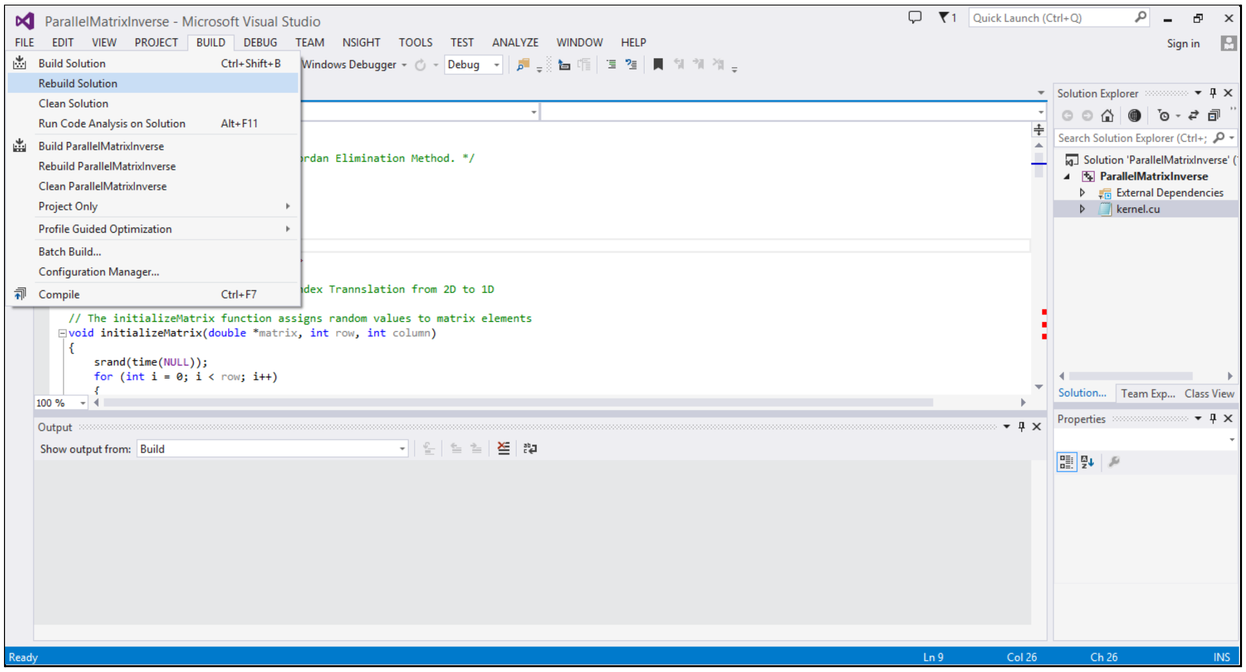
Type: Visual C++

Creates a file containing C++ source code

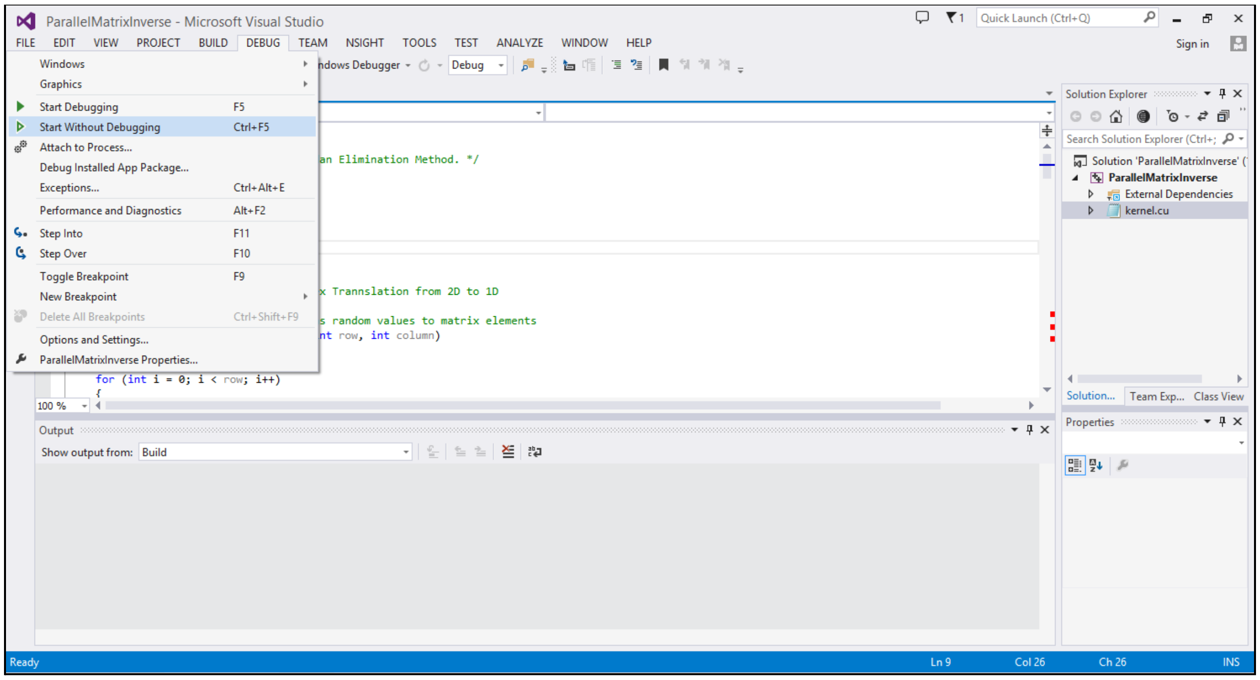
Name: Program.cu

Location: C:\Projects\CUDA\cubeKernel\cubeKernel

After creating a .cu file and editing the file, it needs to be compiled. To accomplish this click on BUILD ->Build Solution. After successful build operation, click on DEBUG -> Start without Debugging. The screenshots of the above two steps are given below.


ParallelMatrixInverse - Microsoft Visual Studio
Quick Launch (Ctrl+Q)
FILE EDIT VIEW PROJECT BUILD DEBUG TEAM NSIGHT TOOLS TEST ANALYZE WINDOW HELP
Sign in
Build Solution Ctrl+Shift+B
Rebuild Solution
Clean Solution
Run Code Analysis on Solution Alt+F11
Build ParallelMatrixInverse
Rebuild ParallelMatrixInverse
Clean ParallelMatrixInverse
Project Only
Profile Guided Optimization
Batch Build...
Configuration Manager...
Compile Ctrl+F7
Windows Debugger
Debug
Solution Explorer
Search Solution Explorer (Ctrl+;)
Solution 'ParallelMatrixInverse' (
ParallelMatrixInverse
External Dependencies
kernel.cu
Solution... Team Exp... Class View
Properties
100 %
Output
Show output from: Build
Ready Ln 9 Col 26 Ch 26 INS

```
rdan Elimination Method. */

dex Translation from 2D to 1D

// The initializeMatrix function assigns random values to matrix elements
void initializeMatrix(double *matrix, int row, int column)
{
    srand(time(NULL));
    for (int i = 0; i < row; i++)
    {
```


ParallelMatrixInverse - Microsoft Visual Studio
Quick Launch (Ctrl+Q)
FILE EDIT VIEW PROJECT BUILD DEBUG TEAM NSIGHT TOOLS TEST ANALYZE WINDOW HELP
Sign in
Windows
Graphics
Start Debugging F5
Start Without Debugging Ctrl+F5
Attach to Process...
Debug Installed App Package...
Exceptions... Ctrl+Alt+E
Performance and Diagnostics Alt+F2
Step Into F11
Step Over F10
Toggle Breakpoint F9
New Breakpoint
Delete All Breakpoints Ctrl+Shift+F9
Options and Settings...
ParallelMatrixInverse Properties...
ndows Debugger
Debug
Solution Explorer
Search Solution Explorer (Ctrl+;)
Solution 'ParallelMatrixInverse' (
ParallelMatrixInverse
External Dependencies
kernel.cu
Solution... Team Exp... Class View
Properties
100 %
Output
Show output from: Build
Ready Ln 9 Col 26 Ch 26 INS

```
an Elimination Method. */

x Translation from 2D to 1D

s random values to matrix elements
nt row, int column)

    for (int i = 0; i < row; i++)
    {
```