A Comparative Study on the Implementation of Matrix Addition in Sequential and Parallel Computing Paradigms

M. V. Rajesh
Assoc. Prof & HOD in IT
Pragati Engineering College
Andhra Pradesh, INDIA

Ch. Venkata Ramana
Asst. Prof in IT
Pragati Engineering College
Andhra Pradesh, INDIA

B. Preethi Devi
Asst. Prof in IT
Pragati Engineering College
Andhra Pradesh, INDIA

ABSTRACT
Operations on matrices are very basic and common in many fields of computer science and information technology, like Image Processing, Graph Algorithms, etc. This paper presents a comparative analysis of the implementation of additions of two matrices with large dimensions both in sequential and parallel computing paradigms. It provides a case study on the implementation of addition of two matrices with large dimensions in C language, Java Language and CUDA C Language implementations.

General Terms
Matrix operations, Java Programming, CUDA Programming, GPU Computing.

Keywords
C Implementation, CUDA C Implementation, GPGPU Computing, Java Implementation, Matrix Addition, Parallel Implementation, Sequential Implementation.

1. INTRODUCTION
This paper analyzes the implementation of basic addition operation of two matrices on various programming platforms like C programming, Java programming and CUDA C programming.

The essential condition for the addition of two matrices is that they must have an equal number of rows and columns. The addition of two matrices A and B will be a matrix which has the same number of rows and columns as same as A and B. The addition operation on matrices say A and B, denoted A + B, is computed by adding corresponding elements of A and B [1]. Matrices, being the organization of data into columns and rows, can have many applications in representing demographic data, in computer and scientific applications, among others.

In the field of computing, matrices are used in various applications like message encryption, to create three-dimensional graphic images, realistic looking motion on a two-dimensional computer screen and also in the algorithms for the calculation of Google page rankings [2] .

A comparative study is performed to analyze the performance of matrix addition using C language, java language implementation under sequential computing paradigm and CUDA C implementation under parallel computing paradigm. CUDA C implementation using the GPGPU Computing exhibits very much improved performance with respect to the time spent for performing the operations i.e., core logic for addition of two matrices.

2. C LANGUAGE IMPLEMENTATION
C programming language is an imperative and procedural language. It was designed to provide low-level access to memory; language constructs that map efficiently to machine instructions, and to require minimal run-time support. C is useful for many software development areas, for example in system programming which was formerly coded in assembly language [3].

C language supports dynamic memory allocation using which blocks of memory of any custom defined size can be requested at run-time using library functions such as malloc from a region of memory called the heap; these blocks can be subsequently released for reuse by calling the library function realloc or free[3].

In the below implementation memory for the matrices i.e., two-dimensional arrays is allocated dynamically as the dimensions of the matrices is large i.e., 10000 by 10000. Arrays are initialized with certain specific default values. Finally using a nested for loop the two input matrices are added to compute the sum of two matrices.

clock_t and clock ( ) are defined in time.h, which are useful to compute the number of clock ticks since the start of execution of the program.

CLOCKS_PER_SEC is constant defined in time.h, which is useful to find number of clocks makes one second there by to covert the number of clocks to time in seconds.

Source Code:

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

void Initialize_matrix(float A[], int m, int n) {
  int i, j;

  for (i = 0; i < m; i++)
    for (j = 0; j < n; j++)
      A[i*n+j] = i+j;
}  /* Read_matrix */

float       * Add_matrix(float A[], float B[],float C[],int m, int
n) {
  int i, j;

  for (i = 0; i < m; i++)
    for (j = 0; j < n; j++)
        C[i*n+j] = A[i*n+j]+B[i*n+j];
   return C;
}
```

```
int main() {
  clock_t CPU_time_1 = clock();
  int m, n;
  float *h_A, *h_B, *h_C;
  float *d_A, *d_B, *d_C;
  size_t size;

  m = 10000; //number of rows
  n = 10000; //number of columns

  printf("m = %d, n = %d\n", m, n);
  size = m*n*sizeof(float);

  h_A = (float*) malloc(size);
  h_B = (float*) malloc(size);
  h_C = (float*) malloc(size);

  printf("Enter the matrices A and B\n");
  Initialize_matrix(h_A, m, n);
  Initialize_matrix(h_B, m, n);

   printf("CPU start time is : %ld \n", CPU_time_1);
   clock_t CPU_time_3 = clock();
   printf("AddMatrix start time is : %ld \n", CPU_time_3);
   Add_matrix(h_A, h_B,h_C,m,n);
   clock_t CPU_time_4 = clock();
   printf("AddMatrix end time is : %ld \n", CPU_time_4);

  free(h_A);
  free(h_B);
  free(h_C);
  clock_t CPU_time_2 = clock();
   printf("CPU end time is : %ld", CPU_time_2);
   printf("\n CLOCKS_PER_SEC is %ld seconds",
(CLOCKS_PER_SEC));
   printf("\n Core Execution TIme is %ld milli seconds",
((CPU_time_4-CPU_time_3)*1000/CLOCKS_PER_SEC));
   printf("\n Total Execution TIme is %ld milli seconds",
((CPU_time_2-CPU_time_1)*1000/CLOCKS_PER_SEC));
  printf("\n MATRIC ADDITION SUCCESSFULLY
COMPLETED ");
  return 0;
```

3. JAVA LANGUAGE IMPLEMENTATION

Java is a an object oriented programming language for general-purpose computing that is concurrent, class-based, object-oriented, and specifically designed to be platform independent. Java achieves the concept of "write once, run anywhere" (WORA) philosophy, meaning that compiled Java code can run on all platforms that support Java without the need for recompilation. Java applications are normally compiled into byte code, which is an intermediate representation that can run on any Java virtual machine (JVM) regardless of computer architecture[4].

Java provides dynamic memory allocation using which blocks of memory of arbitrary size can be requested at run-time using the new operator.

In the below implementation memory for the matrices i.e., two-dimensional arrays is allocated dynamically as the dimensions of the matrices is large i.e., 10000 by 10000. Arrays are initialized with certain specific default values. Finally using a nested for loop the two input matrices are added to compute the sum of two matrices.

System.currentTimeMillis ( ) method is used to obtain the current system time in milli seconds, so as to compute the accurate time spent for the execution of core logic of the program along with total time spent.

Source Code:

```
import java.lang.*;

public class AddMatrices

{

        public static void main(String args[])

        {

        long lStartTime = System.currentTimeMillis();

        int m=10000, n=10000;

        int h_A[][], h_B[][], h_C[][];

        h_A = new int[m][n];

        h_B = new int[m][n];

        h_C = new int[m][n];

        System.out.println("Number of rows = " + m + "
Number of Columns = " + n);

        Initialize_matrix(h_A, m, n);

        Initialize_matrix(h_B, m, n);

        long lStartTime1 = System.currentTimeMillis();

        Add_matrix(h_A, h_B,h_C,m,n);

        long lEndTime1 = System.currentTimeMillis();

  long lEndTime = System.currentTimeMillis();

  long output = lEndTime - lStartTime;

        long output1 = lEndTime1 - lStartTime1;

  System.out.println("Elapsed time in milliseconds: " +
output);

        System.out.println("### Core Elapsed time in
milliseconds: " + output1);

        System.out.println("\n MATRIC ADDITION
SUCCESSFULLY COMPLETED ");

        }

        static void Initialize_matrix(int A[][], int m, int n)

        {

        int i, j;

  for (i = 0; i < m; i++)

  for (j = 0; j < n; j++)

        A[i][j] = i+j;

        }

        static int [][] Add_matrix(int A[][], int B[][],int
C[][],int m, int n)

        {

        int i, j;

        for (i = 0; i < m; i++)

        for (j = 0; j < n; j++)
```

```
        C[i][j] = A[i][j]+B[i][j];
        return C;
        }
}
```

4. CUDA C LANGUAGE IMPLEMENTATION

CUDA is an acronym for Compute Unified Device Architecture is a parallel computing platform and application programming interface (API) model created by Nvidia [1]. It allows the software development to use a CUDA-enabled graphics processing unit (GPU) for general purpose processing – which is termed as GPGPU (General-Purpose computing on Graphics Processing Units). 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 program kernels. The CUDA platform can be worked with programming languages such as C, C++, and FORTRAN [6].

In November 2006, NVIDIA introduced CUDA®, a general purpose parallel computing platform and programming model that allows and facilitates the parallel compute engine in NVIDIA GPUs to solve many complex computational problems in a more efficient way than on a CPU. CUDA provides a software environment that allows developers to use C language for developing parallel computing applications [5].

CUDA C is an extension for C which allows the programmer to define CUDA C functions, called kernels, that, when called, are executed N times in parallel by N different CUDA threads, as opposed by a single thread of execution like regular C functions. A kernel is defined using the __global__ declaration syntax and the number of CUDA threads that execute that kernel for a given kernel call is specified using a new <<<...>>> execution configuration syntax. Each thread executing the kernel is given a unique thread ID that is accessible within the kernel through the built-in threadIdx variable [5].

In the below implementation memory for the matrices i.e., two-dimensional arrays is allocated dynamically on the host memory as the dimensions of the matrices is large i.e., 10000 by 10000. Arrays are initialized with certain specific default values. Data contained in the arrays are transferred to the device memory using cudaMalloc( ) and cudaMemcpy( ).

Finally using a kernel function launched the two input matrices are added to compute the sum of two matrices.

Matrix_Addition<<<(m*n/1024), 1024>>>(d_A, d_B, d_C, m, n);

With the help of gettimeofday( ) & clock( ), time spent for core logic execution on the GPU device and as a whole program execution time is measured.

Source Code:

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
#include <cuda_runtime.h>
#include <sys/time.h>

__global__ void Matrix_Addition(int A[], int B[], int C[], int m, int n) {
  int threadIndex_ij = blockDim.x * blockIdx.x + threadIdx.x;
  /* The test shouldn't be necessary */
  if (blockIdx.x < m && threadIdx.x < n)
    C[threadIndex_ij] = A[threadIndex_ij] + B[threadIndex_ij];

}  /* Mat_add */
void Initialize_matrix(int A[], int m, int n) {
  int i, j;
  for (i = 0; i < m; i++)
    for (j = 0; j < n; j++)
      A[i*n+j] = i+j;
}

double cpuSecond() {
 struct timeval tp;
 gettimeofday(&tp,NULL);
 return ((double)tp.tv_sec + (double)tp.tv_usec*1.e-6);
}

/* Host code */
int main() {
    clock_t CPU_time_1 = clock();
   printf("CPU start time is : %ld \n", CPU_time_1);
  int m, n;
  int *h_A, *h_B, *h_C;
  int *d_A, *d_B, *d_C;
  size_t size;
  double iStart,iElaps;
  m = 10000;
  n = 10000;
  printf("Number of Rows = %d, Number of Cols = %d\n", m, n);
  size = m*n*sizeof(int);

  h_A = (int*) malloc(size);
  h_B = (int*) malloc(size);
  h_C = (int*) malloc(size);

  Initialize_matrix(h_A, m, n);
  Initialize_matrix(h_B, m, n);

  /* Allocate matrices in device memory */
  cudaMalloc(&d_A, size);
  cudaMalloc(&d_B, size);
  cudaMalloc(&d_C, size);
  /* Copy matrices from host memory to device memory */
  cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
  cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);

        clock_t CPU_time_3 = clock();
      /* Invoke kernel                    */
        iStart = cpuSecond();
```

```
        Matrix_Addition<<<(m*n/1024),      1024>>>(d_A,
d_B, d_C, m, n);

    /* Wait for the kernel to complete */
        cudaThreadSynchronize();

        iElaps = cpuSecond() - iStart;

        clock_t CPU_time_4 = clock();
  /* Copy result from device memory to host memory */
  cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);

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

  /* Free host memory */
  free(h_A);
  free(h_B);
  free(h_C);
        clock_t CPU_time_2 = clock();
   printf("CPU end time is : %ld", CPU_time_2);
   printf("\n  CLOCKS_PER_SEC  is  %ld  seconds",
(CLOCKS_PER_SEC));
        printf("\n Total Core Logic Execution TIme is %g
milli          seconds",          (double)((CPU_time_4-
CPU_time_3)*1000/CLOCKS_PER_SEC));
        printf("\n\nCore LOGIC Time elapsed %f sec\n",
iElaps);
   printf("\n Total Execution TIme is %ld milli seconds",
((CPU_time_2-CPU_time_1)*1000/CLOCKS_PER_SEC));
        printf("\n MATRIC ADDITION SUCCESSFULLY
COMPLETED ");
        return 0;
}
```

5. RESULTS

The results of the above three implementations to compute the addition of two matrices of dimensions 10000 by 10000, executed on the below computing environment are as below.

C Language Computing Environment:

Intel(R) Core(TM)2 Duo CPU E4600 @ 2.40GHz

Ubuntu 14.04 64 bit Operating System, 2 GB RAM

gcc version 4.8.4 – C Compiler

Java Language Computing Environment:

Intel(R) Core(TM)2 Duo CPU E4600 @ 2.40GHz

Ubuntu 14.04 64 bit Operating System, 2 GB RAM

JDK 8 – Java Compiler

CUDA C Language Computing Environment:

Intel(R) Core(TM)2 Duo CPU E4600 @ 2.40GHz

Ubuntu 14.04 64 bit Operating System, 2 GB RAM

NVIDIA GeForce GT 710 – 2 GB - GPU Card

nvcc – CUDA C Compiler

S.No	Implementation Code	Time taken for core logic execution (in milli seconds)	Time taken for complete code execution (in milli seconds)
1	C Language	1476	3516
2	Java Language	777	3657
3	CUDA C	0.004	1267

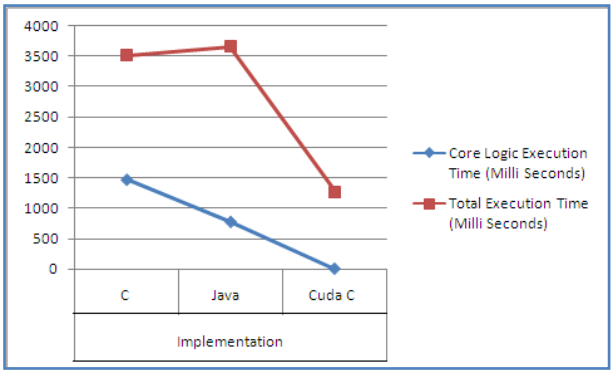

Fig.1 Graphical representation of Program Execution Time

It can be observed that the overall time taken for the execution of the complete program got decreased very much in CUDA C execution, as compared to C and Java language implementations.

It can be observed that the actual time taken for the execution of core logic of computing the addition of two matrices of each 10000 × 10000 got drastically decreased in CUDA C, as compared to C and Java implementations. The reason for this steep decrease is the execution of addition of each element of one matrix with the corresponding element of other matrix, being carried by a separate thread and the ability to execute all those GPU threads concurrently.

So in summary it can be analyzed and understood that the computation time taken in case CUDA C implementation which works on parallel computing environment is tremendously less as compared to normal C and Java implementations which work on sequential computing environments.

6. CONCLUSION

Among the above three implementations of performing matrix addition operation both under sequential and parallel computing paradigms, it can be concluded that especially for computation intensive applications like matrix operations, implementations on parallel computing environments like GPGPU Computing using CUDA C are very fast and effective. In the similar pattern many matrix operations and related algorithms can be analyzed between sequential and parallel computing environments.

In addition to the this kind of comparative analysis, under the parallel computing paradigm i.e., GPGPU Computing, it can also be compared between various parallel implementation configurations with variations in terms of number of thread blocks and the threads per block being used for cuda c kernel executions.

7. REFERENCES

[1] https://en.wikipedia.org/wiki/Matrix_addition.

[2] https://www.reference.com.

[3] https://en.wikipedia.org/wiki/C_(programming_language).

[4] https://en.wikipedia.org/wiki/Java_(programming_language).

[5] https://docs.nvidia.com/cuda/cuda-c-programming-guide/.

[6] https://en.wikipedia.org/wiki/CUDA.