IJCSE International Journal of Computer Sciences and Engineering Open Access
Research Paper Volume-4, Issue-10 E-ISSN: 2347-2693

Matrix Multiplication using Strassen's Algorithm on CPU & GPU

Utsab Ray[1]*, Tapan Kumar Hazra[2] and Utpal Kumar Ray[3]

[1]*Department of Information Technology, Institute of Engineering & Management, Kolkata, India*
[2] *Departmentof Information Technology, Institute of Engineering & Management, Kolkata, India*
[3]*Department of Information Technology, Jadavpur University, Kolkata, India*

Available online at: www.ijcseonline.org

Received: 23/Sep/2016 Revised: 02/Oct/2016 Accepted: 17/Oct/2016 Published: 31/Oct/2016

***Abstract*—** In this paper we have successfully implemented Matrix Multiplication using Strassen's Algorithm on a NVIDIA GPU using CUDA. We have used the multiple cores of the GPU to reduce the computation time drastically. We have also compared the time taken by matrix multiplication using Strassen's algorithm on both CPU and GPU. We have found that the GPU implementation was much faster, but only when the recursion was performed till a certain limit. Beyond that limit, the computation took much more time than expected. Also, we found that implementing Matrix Multiplication using Strassen's algorithm on the CPU yielded some very positive results. By conducting experiments, we came to the conclusion that the recursion limit can be comparatively smaller for matrix multiplication using Strassen's algorithm on CPU than for matrix multiplication using Strassen's algorithm on GPU.

***Keywords*—** GPU, CUDA, Matrix Multiplication, Strassen's Algorithm, Cache, Speedup

I. INTRODUCTION

Matrix multiplication is one of the most basic and crucial linear algebra operations. Matrix multiplication is used for a variety of scientific calculations in a variety of fields, and any decrease in the computation time will be extremely beneficial. The complexity of matrix multiplication using the classic method(using 3 for loops) is $O(n^3)$. But there are other algorithms which have a lower complexity than $O(n^3)$. The current $O(n^k)$ algorithm with the lowest known exponent k is a generalization of the Coppersmith–Winograd algorithm that has an asymptotic complexity of $O(n^{2.3728639})$, by François Le Gall[1]. However, the constant coefficient hidden by the Big O notation is so large that it is not feasible to implement these algorithms [2]. Thus Strassen's algorithm is the most feasible algorithm to implement on modern day computers, as it has a complexity of $O(n^{2.807})$. Strassen's algorithm achieves a lower complexity by using 7 multiplications, instead of 8, as is used in the traditional 3 loop matrix multiplication. We have discussed further about Strassen's algorithm later on in the paper.

In Section 2 we have discussed about some of the work which has been done in this field. Section 3 has some general information about Strassen's Algorithm. A general overview of GPU and CUDA has been given in section 4. Section 5 contains an explanation of the CUDA kernel. The desgin and implementation of Strassen's Algorithm has been described in section 6. In section 7, the experimental setup has been described. The experimental results have been documented in section 8. Ultimately we have concluded in section 9.

Corresponding Author: *Utsab Ray,* utsab_ray@yahoo.co.in
Department of Information Technology, Institute of Engineering & Management, Kolkata, India

II. RELATED WORK

There has been some, although not much work related to implementing Strassen's algorithm on the GPU. Li, Ranka and Sahni [3] have implemented Strassen's algorithm on GPU, but they have not talked about the recursion limit. Arafat, Elango and Sadayappan[4] have implemented Strassen-Winograd's algorithm for matrix multiplication on the GPU. They've also discussed about a cutoff point, where the algorithm switches from Strassen's algorithm to the classic method. Yugopuspito, Sutrisno, and Hudi[5] have talked about managing the memory required for implementing Strassen's algorithm. Since it is a recursive algorithm, multiple matrices are declared at each step, and hence the memory required is quite large. Khan, Al-Mouhamed and Fatayer[6] have further developed a method to optimize Strassen's algorithm on GPU.

III. STRASSEN'S ALGORITHM

Volker Strassen [7] first published this algorithm in 1969 and proved that the three loop method to multiply two matrices was not optimal.
Let A, B be two matrices. We want to calculate the matrix product C, where C = AB
We partition A, B and C into equally sized block matrices,

$$\mathbf{A} = \begin{bmatrix} \mathbf{A}_{1,1} & \mathbf{A}_{1,2} \\ \mathbf{A}_{2,1} & \mathbf{A}_{2,2} \end{bmatrix}, \mathbf{B} = \begin{bmatrix} \mathbf{B}_{1,1} & \mathbf{B}_{1,2} \\ \mathbf{B}_{2,1} & \mathbf{B}_{2,2} \end{bmatrix}, \mathbf{C} = \begin{bmatrix} \mathbf{C}_{1,1} & \mathbf{C}_{1,2} \\ \mathbf{C}_{2,1} & \mathbf{C}_{2,2} \end{bmatrix}$$

Fig 1. Partitioning of matrices

$C_{1,1} = A_{1,1}B_{1,1} + A_{1,2}B_{2,1}$
$C_{1,2} = A_{1,1}B_{1,2} + A_{1,2}B_{2,2}$
$C_{2,1} = A_{2,1}B_{1,1} + A_{2,2}B_{2,1}$
$C_{2,2} = A_{2,1}B_{1,2} + A_{2,2}B_{2,2}$

With this construction we have not reduced the number of multiplications. We still need 8 multiplications to obtain the result matrix C. To reduce the number of multiplications, we define new matrices as follows

$M_1 = (A_{1,1} + A_{2,2})(B_{1,1} + B_{2,2})$
$M_2 = (A_{2,1} + A_{2,2})B_{1,1}$
$M_3 = A_{1,1}(B_{1,2} - B_{2,2})$
$M_4 = A_{2,2}(B_{2,1} - B_{1,1})$
$M_5 = (A_{1,1} + A_{1,2})B_{2,2}$
$M_6 = (A_{2,1} - A_{1,1})(B_{1,1} + B_{1,2})$
$M_7 = (A_{1,2} - A_{2,2})(B_{2,1} + B_{2,2})$

We can see for the 7 matrices M_i, where $1<=i<=7$, 7 multiplications are performed, thus reducing the number of multiplications by 1. We obtain the result matrix C by performing the computations listed below

$C_{1,1} = M_1 + M_4 - M_5 + M_7$
$C_{1,2} = M_3 + M_5$
$C_{2,1} = M_2 + M_4$
$C_{2,2} = M_1 - M_2 + M_3 + M_6$ [9]

We recursively keep on dividing the matrices into smaller matrices until we get matrices of size 2x2. The time complexity can be written as $T(n) = 7*T(n/2) + O(n^2)$. From Master's Theorem we can calculate the complexity to be $O(n^{\log 7})$ which is approximately $O(n^{2.8074})$ [8].

Originally the algorithm was meant to be performed on two matrices of dimension $2^n x 2^n$, as the matrices have to be divided into 4 equal parts recursively. The division stops once we reach a matrix of size 2x2. But practically the time taken to perform Strassen's algorithm on matrices below a certain size is more than the time taken to do normal matrix multiplication on those matrices. Thus there should be a recursion limit. Below the recursion limit, the program should switchover from Strassen's algorithm to normal matrix multiplication. This will ensure an optimal solution. This also means that we do need matrices that are of the order $2^n x 2^n$. As long as the dimensions of the matrices are perfectly divisible by the recursion limit, we should not face any problem. We have discussed further about the recursion limit later on in this paper.

IV. GPU & CUDA

Graphics Processing Units or GPUs have multiple streaming processors(SM), which have multiple cores which can be used for computational purposes. CUDA allows developers to access these cores and use them for their own computations, which is known as GPU computing. CUDA is a parallel computing platform and application programming interface(API) model created by Nvidia. It is a massively multi-threaded parallel computing platform. Using high-level languages, GPU-accelerated applications run the sequential part of their workload on the CPU – which is optimized for single-threaded performance – while accelerating parallel processing on the GPU.

The Nvidia GPU we have used while conducting experiments is Quadro K620. This GPU has 1985 MB of memory available for computation. The GeForce 820m has a CUDA Capability of 5.0. It has 3 streaming multiprocessors(SM), each of which have 128 CUDA cores. Thus, in total it has 384 CUDA cores. The maximum number of threads per streaming multiprocessor(SM) is 1536 and the maximum number of threads per block is 1024.

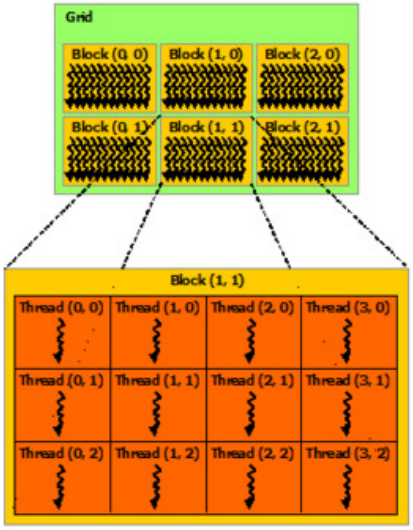

Fig 2. Grid of Thread Blocks

Threads and blocks are the main aspects of the CUDA Programming Model. CUDA C extends C by allowing the programmer to define C functions, called kernels, that, when called, are executed N times in parallel by N different CUDA threads, as opposed to only once like regular C functions [9]. Threads can be identified using a one-dimensional, two-dimensional, or three-dimensional thread index, forming a one-dimensional, two-dimensional, or three-dimensional block of threads, called a thread block. This provides a natural way to invoke computation across the elements in a domain such as a vector, matrix, or volume. There is a limit to the number of threads per block, since all threads of a block are expected to reside on the same processor core and must share the limited memory resources of that core. On current GPUs, a thread block may contain up to 1024 threads. However, a kernel can be executed by multiple equally-shaped thread blocks, so that the total number of threads is equal to the number of threads per block times the number of blocks. Blocks are organized into a one-dimensional, two-dimensional, or three-dimensional grid of thread blocks. The number of thread

blocks in a grid is usually dictated by the size of the data being processed or the number of processors in the system, which it can greatly exceed [10].

CUDA threads may access data from multiple memory spaces during their execution. Each thread has private local memory. Each thread block has shared memory visible to all threads of the block and with the same lifetime as the block. All threads have access to the same global memory.
The CUDA programming model assumes that the CUDA threads execute on a physically separate device that operates as a coprocessor to the host running the C program. This is the case, for example, when the kernels execute on a GPU and the rest of the C program executes on a CPU. This is illustrated by the figure below.

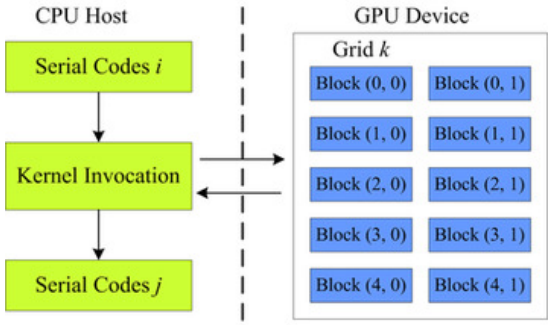

Fig 3. CUDA Programming Model

V. CUDA KERNEL

As stated in the earlier section, Kernels are C functions, that, when called, are executed by different CUDA threads in parallel. A kernel is defined using the __global__ declaration specifier. Each thread that executes the kernel is given a unique thread ID that is accessible within the kernel through the built-in threadIdx variable [9].
As an example, the following code adds two matrices *A* and *B* of size NxN and stores the result into matrix *C*:

```
__global__ void MatAdd(char *A, char *B, char *C, int N){
   int i = blockIdx.x * blockDim.x + threadIdx.x;
   int j = blockIdx.y * blockDim.y + threadIdx.y;
   if (i < N && j < N)
        *(C + i*N + j) = *(A + i*N + j) + *(B + i*N + j);
}
int main(){
   ...
   dim3 threadsPerBlock(48, 4);
   dim3
   numOfBlocks(N/threadsPerBlock.x,N/threadsPerBlock.y);
   //Invoking the kernel
   MatAdd << <numOfBlocks, threadsPerBlock >> >(pA,
pB, pC, N);
   ...
}
```

In the above example we have performed matrix addition in the kernel "MatAdd." We have defined the number of threads per block using the variable "threadsPerBlock," which is of type dim3. In this particular example there are 48x4 threads per block. The total number of blocks have been defined using the variable "numOfBlocks" which is of type dim3. In this particular example, "threadsPerBlock.x" is 48, and "threadsPerBlock.y" is 4. Therefore the total number of blocks is N/48 times N/4. Here we have used dim3 which is an integer vector type that is used to specify dimensions. The syntax for kernel launch is, "function name" << <"number of blocks", "number of threads per block" >> >("arguments to be sent to the kernel").

The first thing to notice about the kernel is the __global__ keyword. This simply indicates that this function may be called by either the CPU or the GPU. Another interesting thing to notice is, how each thread figures out exactly which data element it is supposed to operate on. Each thread runs the same code, so the only way to differentiate themselves from the other threads is to use the threadIdx, and the blockIdx variables.

VI. DESIGN AND IMPLEMENTATION

As stated in the second section, Strassen's Algorithm is a recursive algorithm. A step by step implementation of Strassen's Algorithm is given below:
Strassen(A, B, N)

1. Compute A11, B11, . . ., A22, B22 by splitting A and B into 4 equal parts
2. If N>Recursion_Limit
   M1 ← Strassen((A11 + A22), (B11 + B22), N/2)
   Else
   Multiply((A11 + A22), (B11 + B22), N)
3. If N>Recursion_Limit
   M2 ← Strassen((A21 + A22), B11 , N/2)
   Else
   Multiply((A21 + A22), B11, N)
4. If N>Recursion_Limit
   M3 ← Strassen(A11, (B12 - B22), N/2)
   Else
   Multiply(A11, (B12 – B22), N)
5. If N>Recursion_Limit
   M4 ← Strassen(A22, (B21 – B11), N/2)
   Else
   Multiply(A22, (B21 – B11), N)
6. If N>Recursion_Limit
   M5 ← Strassen((A11 + A12), B22, N/2)
   Else

Multiply((A11 + A12), B22, N)

7. If N>Recursion_Limit
   M6 ← Strassen((A21 − A11), (B11 + B12), N/2)
   Else
   Multiply((A21 − A11), (B11 + B12), N)
8. If N>Recursion_Limit
   M7 ← Strassen((A12 − A22), (B21 + B22), N/2)
   Else
   Multiply((A12 − A22), (B21 + B22), N)
9. C11 ← M1 + M4 − M5 + M7
10. C12 ← M3 + M5
11. C21 ← M2 + M4
12. C22 ← M1 - M2 + M3 + M6
13. Output C

In the above algorithm we have multiplied matrices using Strassen's algorithm, but only to a certain limit. Once the size of the matrices becomes less than "Recursion_Limit," we use the classic method of matrix multiplication to multiply them. This is because, we have found from our experiments that below a certain size, there is negligible difference between the classic method of matrix multiplication and matrix multiplication using Strassen's algorithm.

For executing the program on the GPU, we had to write CUDA kernels for addition, subtraction and multiplication. The addition and subtraction kernels are based on the example given in the previous section, "CUDA Kernel." The multiplication kernel is based on the CUDA sample of matrix multiplication provided in the CUDA Toolkit.

VII. EXPERIMENTAL SETUP

While conducting experiments, we have used a workstation equipped with Intel Xeon processor. The workstation (HP Z440) is equipped with 16 GB RAM. The specifications of the GPU have already been stated in the section, "GPU and CUDA."

We have used Fedora 24 and CUDA Toolkit 8.0. The CUDA Toolkit includes a compiler for Nvidia GPUs, math libraries, and tools for debugging and optimizing the performance of applications. It also has programming guides, user manuals, and other relevant documentation. The programs containing CUDA kernels are compiled with the help of the nvcc. CUDA codes run on both CPU and GPU. Nvcc separates these two parts and sends the host code (the part of the code which is to be run on the CPU) to a C compiler. In our experimental setup, the C compiler is gcc 5.4.0. The device code (the part of the code which is to be run on the GPU) is sent to the GPU. The device code is further compiled by nvcc.

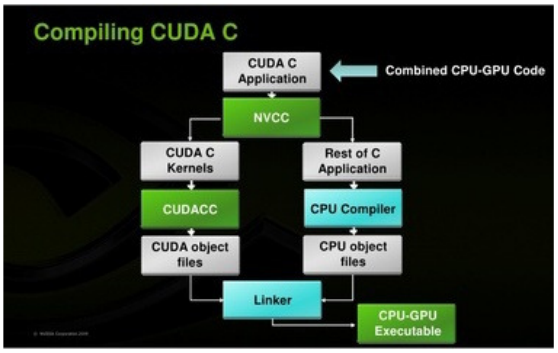

Fig 4. Flowchart illustrating the function of nvcc

VIII. EXPERIMENTAL RESULTS

We have taken 4 sets of readings for our experiment. The 4 sets being – matrix multiplication on the CPU, Strassen's algorithm on the CPU, matrix multiplication on the GPU and Strassen's algorithm on the GPU. In this section we will list those readings and compare between the different sets of data and come to a logical conclusion.

We have used matrices of size 500x500, 1000x1000, 2000x2000, 4000x4000, 8000x8000 and 16000x16000 for all 4 sets of readings. The matrices are of type char. Thus each element of the matrix occupies a space of 1 byte

A. CPU

Table 1. Classic Method of Matrix Multiplication on the CPU

Size	Time(in seconds)	Experimental Multiplier
500x500	0.45	
1000x1000	3.3	7.3
2000x2000	26.4	8.0
4000x4000	490.1	18.6
8000x8000	4178.6	8.5
16000x16000	36694.6	8.8

Experimental Multiplier = (Time taken for matrix of size nxn)/(Time taken for matrix of size (n/2)x(n/2))

The complexity of classic matrix multiplication is $O(n^3)$, where nxn is the size of the matrix. If we double the size of the matrix, then the complexity becomes $O((2n)^3)$. As a result the time taken by the matrix of size 2nx2n increases by a factor of $((2n)^3/ n^3)$. This yields a result of 8. Thus, for an ideal case, the value of the multiplier will be 8.

As we can see, for matrices of size 1000x1000 and 2000x2000, the multiplier is near to 8, if not exactly 8. Considering there are 2 input matrices and 1 output matrix, each of equal size; for 1000x1000 the total size of three

matrices is 3 MB. For 2000x2000 the total size of three matrices is 12 MB. We have calculated the size of each matrix by multiplying the space occupied by each element of the matrix(1 byte) with the number of elements of the matrix. Since 3 MB and 12 MB are both less than the total cache memory size of the CPU, the matrices can be stored in the cache. So there will always be a cache hit and data transfer will be much faster. Now for matrices of size 4000x4000, the total size of 3 matrices is 48 MB. This exceeds the capacity of the cache memory. Hence, the matrices have to be stored on the external memory, which is the RAM. So there always be a cache miss. Therefore data transfer on the external memory will always take more time. That is why the multiplier increases drastically for matrices of size 4000x4000. Thereon, all matrices have to be stored on the external memory. That is why, after 4000x4000 the multiplier once again shows values close to 8, if not exactly 8.

There is also an explanation as to why the value of the multiplier does not come exactly as 8. Since we are running the program on an OS with many applications running in the background, the CPU is unable to devote it's full power to executing the program. This is why the multiplier is varying a bit. For some cases, the applications in the background may not be consuming much resources, or in other cases it might be consuming a lot of resources. Taking multiple readings, and calculating their average is the best way to eliminate this inconsistency, and that is what we have done while taking these readings.

Table 2. Matrix Multiplication using Strassen's Algorithm on the CPU

Size	Time(in seconds)	Experimental Multiplier	Recursion Limit
500x500	0.4		125
1000x1000	2.8	7.0	125
2000x2000	19.6	7.0	125
4000x4000	137.3	7.0	125
8000x8000	963.2	7.0	125
16000x16000	6749.0	7.0	125

Experimental Multiplier = (Time taken for matrix of size nxn)/(Time taken for matrix of size (n/2)x(n/2))

The complexity of matrix multiplication using Strassen's algorithm is $O(n^{2.81})$, where nxn is the size of the matrix. If we double the size of the matrix, then the complexity becomes $O((2n)^{2.81})$. As a result the time taken by the matrix of size 2nx2n increases by a factor of $((2n)^{2.81}/ n^{2.81})$. This yields a result of approximately 7. Thus, for an ideal case, the value of the multiplier will be approximately 7.

Now for matrix multiplication using Strassen's algorithm we can see that the multiplier is uniform. There is no drastic increase in the multiplier for matrices of size 4000x4000. This is because we are using recursion. We are constantly splitting the matrices until we get matrices that reach the recursion limit. Only then do we multiply the matrices using the classic method of multiplication. Since the recursion limit is 125, that means that when we get matrices of size less than or equal to 125x125, only then do we multiply the matrices using the classic method of multiplication. Matrices of this size can easily fit inside the cache memory of the CPU, thus ensuring fast data transfer. That is why there is no drastic change in the multiplier, as matrix size increases from 500 onwards.

Now we will compare between the classic method of matrix multiplication and matrix multiplication using Strassen's algorithm. Matrix multiplication using Strassen's algorithm, as expected takes less time than the classic method of matrix multiplication. We have drawn further inferences from the following table

Table 3. Comparison between Strassen's Algorithm and Classic Method of Matrix Multiplication on the CPU

Size	Time for Strassen(in sec)	Time for Classic Method (in sec)	Strassen Improvement
500x500	0.4	0.45	1.13
1000x1000	2.8	3.3	1.18
2000x2000	19.6	26.4	1.35
4000x4000	137.3	490.1	3.57
8000x8000	963.2	4178.6	4.34
16000x16000	6749.0	36694.6	5.44

Strassen Improvement = (Time for Classic Method)/ (Time for Strassen)

Thus we can see as the matrices increase in size, the improvement in the Strassen's algorithm increases. Thus we can say that, Strassen's algorithm is effective for matrices of large size [4].

B. GPU

Table 4. Classic Method of Matrix Multiplication on the GPU

Size	Time(in seconds)	Experimental Multiplier
500x500	0.005	
1000x1000	0.040	8.0
2000x2000	0.44	11.0
4000x4000	3.32	7.5
8000x8000	26.72	8.0
16000x16000	224.77	8.4

Experimental Multiplier = (Time taken for matrix of size nxn)/(Time taken for matrix of size (n/2)x(n/2))

As we can see, for the classic method of matrix multiplication on the GPU, the multiplier is pretty much uniform. However, for smaller sizes, the computation time may not be accurate. This is because, the computation time itself is quite small. Compared to that small computation time, the kernel launch overhead is quite significant. Thus the timing for small matrices is not accurate enough. But for large matrices, since the kernel launch overhead is negligible when compared to the computation time, the timing for large matrices is quite accurate.

Table 5. Matrix Multiplication using Strassen's Algorithm on the GPU

Size	Time(in seconds)	Experimental Multiplier	Recursion Limit
500x500	0.2		500
1000x1000	0.42	2.1	1000
2000x2000	0.56	1.3	1000
4000x4000	2.83	5.0	2000
8000x8000	20.33	7.1	2000
16000x16000	147.12	7.2	2000

Experimental Multiplier = (Time taken for matrix of size nxn)/(Time taken for matrix of size (n/2)x(n/2))

As we can see from the table, as the size increases, the experimental multiplier inches closer to the theoretical multiplier. This reinforces the fact that, Strassen's algorithm is more beneficial and viable for matrices of large size. When the size of the matrices is comparatively small, the kernel launch overhead is quite significant. Also, in Strassen's algorithm we have multiple kernel launches. Every time recursion is performed, multiple kernels are launched. This increases the computation time, thus yielding inconsistent results for smaller matrices.

The recursion limit should always be less than the size of the matrix. But for matrices of size 500x500 and 1000x1000 we can see that the recursion limit is the same as the size of the matrix. This is because, for matrices of small size, the kernel launch overhead is quite large compared to the computation time. Performing recursion multiple times will result in multiple kernel launches, which will increase the computation time.

Table 6. Comparison between Strassen's Algorithm and Classic Method of Matrix Multiplication on the GPU

Size	Time for Strassen (in s)	Time for Classic Method (in s)	Strassen Improvement
500x500	0.2	0.005	0.025
1000x1000	0.42	0.040	0.09
2000x2000	0.56	0.44	0.79
4000x4000	2.83	3.32	1.17
8000x8000	20.33	26.72	1.31
16000x16000	147.12	224.77	1.53

Strassen Improvement = (Time for Classic Method)/ (Time for Strassen)

As we can see from the above table, the Strassen Improvement is quite less for matrices of small size. But as the matrix size increases, the Strassen Improvement increases. This reinforces our belief that Strassen's algorithm works best for large matrices.

C. CPU & GPU

Table 7. Comparison between classic method of matrix multiplication on CPU and GPU

Size	CPU(in seconds)	GPU(in seconds)	Speedup
500x500	0.45	0.005	90
1000x1000	3.3	0.040	82.5
2000x2000	26.4	0.44	60
4000x4000	490.1	3.32	147.6
8000x8000	4178.6	26.72	156.4
16000x16000	36694.6	224.77	163.3

Speedup = CPU/GPU

For matrix multiplication on the CPU, we noticed that for matrices of size 4000x4000 onwards, the speedup has increased a lot because of the cache miss effect in the CPU. As we can see from the table, performing matrix multiplication on the GPU yields extremely favorable results. The above table has been plotted as a graph in the figure below.

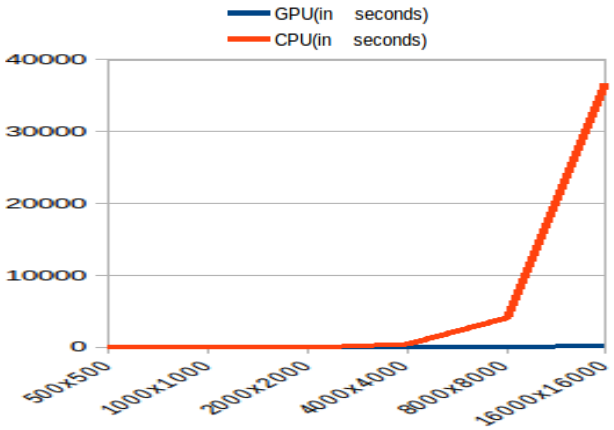

Fig 5. Comparison between classic method of matrix multiplication on CPU and GPU

Table 8. Comparison between matrix multiplication using Strassen's algorithm on CPU and GPU

Size	CPU(in seconds)	GPU(in seconds)	Speedup
500x500	0.4	0.2	2
1000x1000	2.8	0.42	6.7
2000x2000	19.6	0.56	35
4000x4000	137.3	2.83	48.5
8000x8000	963.2	20.33	47.4
16000x16000	6749.0	147.12	45.9

Speedup = CPU/GPU

For matrix multiplication using Strassen's algorithm, we can see that the speedup increases gradually, thus again proving the fact that Strassen's algorithm is more beneficial for matrices of large sizes. As to why we did not take readings beyond 16000x16000. The GPU has a total memory of 2 GB. In Strassen's algorithm there are three main matrices and multiple sub-matrices. If we go beyond 16000x16000, the cumulative space taken up by all the matrices exceeds 2 GB. The above table has been plotted as a graph in the figure below.

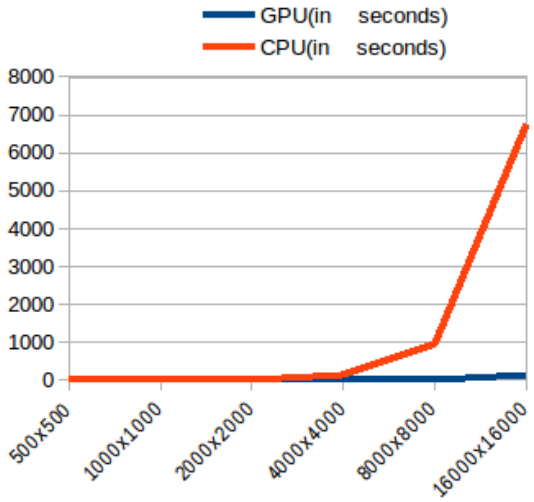

Fig 6. Comparison between matrix multiplication using Strassen's algorithm on CPU and GPU

IX. CONCLUSION

We can quite safely come to the conclusion that implementing matrix multiplication on the GPU is always faster than CPU irrespective of the algorithm[11]. We have seen that Strassen's algorithm gives better results when matrix size is larger, both on the CPU and the GPU. As long as the recursion limit is used in an intelligent manner, positive results can be garnered from implementing Strassen's algorithm.

X. REFERENCES

[1] Francois Le Gall, "Powers of Tensors and Fast Matrix Multiplication," Cornell University Library, arXiv:1401.7714 [cs.DS], 2014.

[2] Wikipedia, Strassen Algorithm, https://en.wikipedia.org/wiki/Strassen_algorithm.

[3] Junjie Li, Sanjay Ranka, Sartaj Sahni, "Strassen's Matrix Multiplication on GPUs," 2011 IEEE 17th International Conference on Parallel and Distributed Systems(ICPADS), pp. 157-164, 2011.

[4] C. P. Patidar and Meena Sharma, "Histogram Computations on GPUs Kernel using Global and Shared Memory Atomics", ISROSET-International Journal of Scientific Research in Computer Science and Engineering, Volume-01, Issue-04, Page No (1-6), Aug 2013

[5] Pujianto Yugopuspito, Sutrisno, Robertus Hudi, "Breaking through memory limitation in GPU parallel processing using Strassen Algorithm," 2013 International Conference on Computer, Control, Informatics and Its Applications(IC3INA), pp. 201-205, 2013.

[6] Ayaz ul Hasan Khan, Mayez Al-Mouhamed, Allam Fatayer, "Optimizing strassen matrix multiply on GPUs", 2015 16th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD), pp. 1-6, 2015.

[7] V. Strassen, "Gaussian elimination is not optimal," Numerische Mathematik, Vol. 13, No. 4, pp. 354-356, August 1969.

[8] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Livest, Clifford Stein, Introduction to Algorithms, Second Edition. MIT Press and McGraw-Hill, 2001. Chapter 28: Section 28.2: Strassen's algorithm for matrix multiplication, pp. 735-741.

[9] CUDA C Programming Guide, https://docs.nvidia.com/cuda/cuda-c-programming-guide

[10] John Nickolls, "GPU parallel computing architecture

[11] Fazlul Kader Murshed Nawaz, Arnab Chattopadhyay, Kirthan G J, Girish D Mane, Rohith N Savanth, "Comparison of Open MP and CUDA", International Journal of Computer Science and Engineering E-ISSN: 2347-2693, Vol.2, Issue-12, pp.38-41, 2014.

AUTHORS' PROFILE

Utsab Ray is currently pursuing his B.Tech degree in Information Technology from Institute of Engineering & Management, Salt Lake, Kolkata. His research interests include High Performance Computing and Distributed Systems. Utsab Ray may be reached at utsab_ray@yahoo.co.in

Tapan Kumar Hazra completed his M.E degree from Jadavpur University, Kolkata, West Bengal, India. Since from 2003, he is working as a faculty member of Department of Information Technology at Institute of Engineering & Management,

Salt Lake, Kolkata, West Bengal, India.
His research interest include Design and Analysis of Algorithms, Image Processing, Natural Language Processing, Sentiment Analysis, Machine learning, Cryptography. Tapan Kumar Hazra may be reached at tapankumar.hazra@iemcal.com

Utpal Kumar Ray received the degree of B.E. in Electronics and Telecommunication Engineering in 1984 from Jadavpur University, India and the degree of M.Tech in Elecrical Engineering from Indian Institute of Technology, Kanpur in 1986
He was employed in different capacities in WIPRO INFOTECH LTD., Bangalore, India; Client: TANDEM COMPUTERS, Austin, Texas, USA; HCL America, Sunnyvale, California, USA, Client: HEWLETT PACKARD, Cupertino, California, USA; HCL Consulting, Gurgaon, India; RAVEL SOFTWARE INC., San Jose, California, USA; STRATUS COMPUTERS, San Jose, California, USA; AUSPEX SYSTEMS, Santa Clara, California, USA and SUN MICROSYSTEMS, Menlo Park, California, USA for varying periods of duration from 1986 to 2002. From 2003 he is working as Assistant Professor in the Department of Information Technology, Jadavpur University, India. He has published 17 papers in different conferences and journals. He has also published a book titled “Software Transactional Memory: An Alternative to Locks” by LAP LAMBERT ACADEMIC PUBLISHING, GERMANY in 2012 co-authored with Ryan Saptarshi Ray. Utpal Kumar Ray may be reached at utpal_ray@yahoo.com

