International Core Journal of Engineering Volume 7 Issue 4, 2021
ISSN: 2414-1895 DOI: 10.6919/ICJE.202104_7(4).0013

Strassen Algorithm and Its Performance

Ke Zhao

Faculty of Engineering, University of Bristol, Bristol BS8 ITH, United Kingdom.

Abstract

There are many algorithms developed over time for computation. Where some tend to be limited to some ranges, some tend to handle bigger problems. With the advancements in processing world, a lot of platforms have been created for computing, CUDA being one of them. CUDA is a well-known parallel computing platform developed by NVIDIA that enables developers to harness the power of GPU's in solving intensive problems. We are using CUDA to compute different mathematical matrices. Strassen's algorithm was used on CUDA to implement matrix multiplication. As we all know the phrase 'divide and conquer'. This is exactly what Strassen's algorithm does. Hence, it is a much better way of computing matrices than the standard method. The computation can be done on both CPU and GPU. On practical analysis, it was noticed that the computation of matrices of bigger sizes was much faster in GPU implementation than that of CPU implementation. Small matrices also came into account and it was found out that algorithms vary depending upon the size of the matrix. For example, when computing a 2*2 matrix multiplication, the classic method is a better method to implement. Because of this, recursions were put in the code that chooses which algorithm to implement depending upon the sizes of the matrices. Matrix multiplication using Strassen's algorithm on the CPU has a smaller recursion limit than doing the same matrix multiplication on a GPU.

Keywords

Strassen Algorithm; CPU; GPU; Recursive Programming.

1. Introduction

Linear Algebra being a very crucial field in mathematics. It implements matrix multiplication as one of the most needed components or a very crucial part since it is used in many different scientific fields. Thus, there is a need for us to reduce the computation time that is needed to compute such operations. It is necessary to note that the computation of time differs from which to which algorithm to use and hence the importance of choosing the most sufficient algorithm that will be able to use the minimum time possible. For instance, when we use the classic method which uses three (3) for loops, the complexity of such is $O(n^3)$. And therefore, the need to know that other algorithms are less complex than others when it comes to calculating operations on the matrices and others even take more time than others depending on how recursions are implemented.

The coppersmith Winograd is the one that is known to have the lowest known exponent which has an asymptotic complexity of $O(n^{2.372869})$. However, the Big O notation hides it is a constant coefficient and it is so large that it is not convenient to implement this algorithm. This makes the Strassen algorithm to be the most convenient algorithm to implement in these modern computers have it has a complexity of $O(n^{2.807})$. This is achieved since it implements multiplication complexity of 7 instead of 8 as compared to the 3-loop matrix multiplication.

ISSN: 2414-1895 DOI: 10.6919/ICJE.202104_7(4).0013

2. Strassens algorithm

This first implementation of this algorithm was done in the year 1969 by Volker Strassen and it proofed that the 3-loop method that was used then to calculate the multiplication between matrices was not the most optimal [1].

The following example shows how the Strassen algorithm is implemented:

Example

$$B=\begin{bmatrix} A1,1 & A1,2 \\ A2,1 & A2,2 \end{bmatrix}, C=\begin{bmatrix} B1,1 & B1,2 \\ B2,1 & B2,2 \end{bmatrix}, D=\begin{bmatrix} C1,1 & C1,2 \\ C2,1 & C2,2 \end{bmatrix}$$

Let B and C be matrices whose product is matrix D which is equal to the product of B and C [1].

What we do is that we first partition the B, C and D into equal block matrices.

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

It is still very clear that we have not reduced the number of multiplications since need 8 multiplications so as to obtain our matrix C. To reduce the number of multiplications then we need to define a set of new matrices as shown below:

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

As we can see now, we get 7 matrices and therefore we reduce the number of multiplications by 1. We can now be able to get the multiplication of matrix B and C, which is matrix D by doing the computations that we can see below:

$$D_{1,1}=M_1+M_4-M_5+M_6$$
$$D_{1,2}=M_3+M_5$$
$$D_{2,1}=M_2+M_4$$
$$D_{2,2}=M_1-M_2+M_3+M_6 \quad |9|$$

And so, we keep on dividing the matrices until they attain the smallest size possible which is a 2*2 matrix [1]. Using the formulae that is shown below we can always get the time complexity:

$$T(N)=7*T(N/2)+O(N^2).$$

Using the master's theorem, now we can be able to calculate the time complexity which is $O(N^{\log 7})$ which is approximately $O(N^{2.8074})$ |8|.

Take note that previously the multiplication of that matrix was meant to happen on matrices of sizes $2^n * 2^n$ but the division of the matrices just stops once we reach matrices of size 2*2. The time taken by the Strassen algorithm to perform multiplication on matrices of a certain size of lower sizes such as 2*2 is more than the time taken when calculating the multiplication of the matrices practically. Therefore, the need to have a recursion limit. It is essential to implement a situation that when below the recursion limit the multiplication switches to the normal multiplication of matrices [2]. This is to ensure that every time we have the most optimal solutions to making this multiplication of matrices. We can attain this using loop in codes which will implement the switching of which algorithm to use.

ISSN: 2414-1895 DOI: 10.6919/ICJE.202104_7(4).0013

3. GPU versus CUDA

The graphic processing unit which is also the GPU has got multiple streaming processors that are embedded with multiple cores that are normally used for computational purposes. Through CUDA which is a computing platform and a program model which developers can be able to access these cores and then use them for their computations. This is known as called Graphic Processing Computing. NVIDIA created CUDA as a parallel computing platform and as an Application Programming Interface (API). It is a massive and trending multi-threading parallel computing platform. All this is achieved using high-level programming languages that are used to develop GPU applications which reduce the workload of the GPU optimized for single-threaded performance and then accelerate the processing of the GPU [3].

The NVIDIA GPU that we have used in this case for the test is the QUADRO K620 which has a total amount memory of 1985MB which is used for computation. The GeForce 820M has a CUDA capability of 5. There exist three streaming multiprocessors with each of the multiprocessors having 128 CUDA cores thus in total it has 384 cores and the maximum number of threads per each streaming Multiprocessor is 1536 and the maximum number of threads per block is 1024.

The CUDA programming model's main aspects are the threads and the blocks. There exists a CUDA C which extends C by allowing a programmer to define what we call C programming language functions which are also called Kernels [3]. There exist N different CUDA threads that when called the execute the kernels N times in parallel as opposed to only the C functions that are executed just once. The threads of blocks come in different dimensional perceptive being the one dimensional, two dimensional, and three-dimensional threads of blocks. Through this way, we have a natural way to call computations across elements in a domain such as matrix computations, volumes, and even computation on vectors. It is good to note that the Kernel can be executed by several equally shaped multiple threads of the block [3].

On the other hand, the threads can be organized into grids. There exist three types of grids namely: Dimensional, two-dimensional, and three-dimensional grids. The number of thread blocks that exist in a grid is determined by the actual size of data that is being processed or by the number of processors in the system which sometimes can be more than 10.

During execution time, CUDA threads can obtain data from the multiple memory spaces available. There exists a private local memory for each thread and each thread has visible memory to the other block of threads and with the same lifetime as the block. There is a global memory that all the threads have access to.

4. CUDA kernels

There exist C functions called kernels which can be executed or rather called upon by the different CUDA threads in parallel. There is a _global_ declaration that is normally used to define the kernels. There is a unique Id that is given to each thread that is used to execute the kernel that is the kernel can be able to access through a built-in threadIdx variable |9|.

The following example is a C code that adds two matrices B and C which have a size of N*N and stores the result that we get in a matrix B.

```
_global_void MatAdd (char B, char C, char D, int N) {
Int i= blockIdx.x*blockdimx+threadIdx.x;
Int j= blockIdx.y*blockDim.y+threadIdx.y;
If(i<N&&j<N) {
*(D+i*N+j) =*(B+i*N+j0+*(C+i*N+j);
}
}
int main () {
```

ISSN: 2414-1895 DOI: 10.6919/ICJE.202104_7(4).0013

…

```
dim3 threadsPerBlock (48,4);
dim3
numOfBlocks (N/threadsPerBlock.x, N, N, N, N/threadsPerBlock);
MatAdd<< <numOfBlocks, threadsPerBlock>> > (pB, pC, pC, N);
}
```

The kernel MatAdd has been used in the above example for the addition of the matrices. The "threads per block" which is a variable of type dim3 has been used to define the number of blocks needed. In the above example, there are 48*4 threads per block. The "numOfBlocks" defines the total number of blocks that have been defined using the type dim3. The number of threadsPerBlock from the example above is 48 and the threadsPerBlock is 4. From the information, we can therefore find the total number of blocks which is N/48 multiplied by N/4. We store our values in the integer format and here we have used dim3 which is a vector to hold the value of the addition.

What we see first in the above C code of the kernel is the _global_ declaration which is a keyword. The main purpose of that global declaration is it tells us that the program may either be called by the CPU or the GPU. Each thread tells what type of data element it is supposed to work on or execute. There is this same code that each thread runs on and therefore the only way we can differentiate one thread from another is by using the threadIdx, and the other blockIdx variables.

5. Design and implementation

Strassen algorithm is a recursive algorithm by default. This is a step by step implementation on how the Strassen algorithm works in this context. We have used the matrices that we used in the Strassen algorithm example that had matrices B, C, and D to show how it is implemented below:

1. Compute B11, C11, . . ., A22, C22 by splitting B and C into 4 equal parts
2. If N>Recursion_Limit
   M1 ← Strassen ((B11 + B22), (C11 + C22), N/2)
   Else
   Multiply ((B11 + B22), (C11 + C22), N)

3. If N>Recursion_Limit
   M2 ← Strassen ((B21 + B22), C11, N/2)
   Else
   Multiply ((B21 + B22), C11, N)
4. If N>Recursion_Limit
   M3 ← Strassen (B11, (C12 - C22), N/2)
   Else
   Multiply (B11, (C12 – C22), N)
5. If N>Recursion_Limit
   M4 ← Strassen (B22, (C21 − C11), N/2)

Else
   Multiply (B22, (C21 − C11), N)

6. If N>Recursion_Limit
   M5 ← Strassen ((B11 + B12), C22, N/2)
   Else

ISSN: 2414-1895 DOI: 10.6919/ICJE.202104_7(4).0013

..0. Multiply ((B11 + B12), C22, N)

7. If N>Recursion_Limit

M6 ← Strassen ((B21 − B11), (C11 + C12), N/2)

Else

Multiply ((B21 − B11), (C11 + C12), N)

8. If N>Recursion_Limit

M7 ← Strassen ((B12 − B22), (C21 + C22), N/2)

Else

Multiply ((C12 − B22), (C21 + C22), N)

9. D11 ← M1 + M4 − M5 + M7

10. D12 ← M3 + M5

11. D21 ← M2 + M4

12. D22 ← M1 - M2 + M3 + M6

13. Output

The final output of the matrices is D. This is because this is the matrix we find after the multiplication of matrices B and C. The Strassen algorithm has been used to do the computation of the multiplication up to where the recursion limit is set. Remember has we had said earlier there is a certain size of matrices that are too small and when using other alternative methods to do their computation, it is much faster than when implementing this Strassen algorithm in such scenarios. Simple mathematical operations and even some logical operations have got to be written on the CUDA kernels. This operation includes decision making and even operations like addition, subtraction, and even division operations which the developers use to give instructions to the Graphics Processing Unit of such NVIDIA graphics cards. They are processed in the GPU.

6. Experimental setup

We have used a very powerful workstation which has got one of the strongest generations of processors which is the Xeon processor running on a random-access memory of 16 GB to test this. The type of graphics we have used as we mentioned earlier is that we have used the NVIDIA QUADRO K620 graphics which has a total memory of 1985MB (approximately 2 GB) to test how this Strassen algorithm performs in accordance to the other alternative algorithms. But here we have specifically tested the Strassen Algorithm and the results we have are for the algorithm. The environments that have been used include the Fedora 24 and the CUDA toolkit 8. The NVIDIA compiler has been included in the CUDA toolkit 8 which has got a lot of features such as math libraries, a debugging environment, and many more tools for optimizing the performance of many other applications. It is got also other relevant documentation for the user such as programming guides and the manuals necessary. Most importantly CUDA codes run on both the CPU and the GPU. There exists the nvcc which is responsible for running the CUDA kernels that contain the programs. The nvcc is responsible for the separation of the codes that run in the CPU and the GPU. The host code is the one that is run by the C compiler and it is mainly executed in the CPU while the device code is run on the GPU which is compiled by the nvcc.

We have done gathering on four sets of information being matrix multiplication on the CPU, Strassen algorithm on the CPU, matrix multiplication on the CPU, and Strassen algorithm on the GPU. These are the different sets of data that we want to compare and therefore get our conclusions from them.

As you can observe from the table1, we have used matrices of different sizes being 500*500, 1000*1000, 4000*4000, 8000*8000, and 16000*16000 on all the four types of sets of data. We have

ISSN: 2414-1895 DOI: 10.6919/ICJE.202104_7(4).0013

arranged them from the smallest size to the biggest and we have given the time in seconds that it takes to get implemented and the Experimental multiplier which we get by doing the following multiplication (the time taken by the matrix size n*n)/ (the time taken by the matrix size n/2*n/2). The matrices used are of the character type and therefore each matrix occupies a size of 1 byte.

Table 1. Computation Analysis

Size	Time in seconds	Experimental multiplier
500*500	0.45	
1000*1000	3.3	7.3
2000*2000	26.4	8
4000*4000	490.1	18.6
8000*8000	4178.6	8.5
16000*16000	36694.6	8.8

From what we learnt earlier, the complexity of a classic method in matrix multiplication is $O(n^3)$ where n*n is the size of the matrix. Therefore, in a classic method if we double the size of the matrix then the complexity now becomes $O(2n^3)$. Due to this, the time that is taken by this matrix of size 2n*2n becomes $2n^3/n^3$ and therefore this yields a result of 8. This is seen from the table above since the experimental multiplier is closer to 8 or can be even exactly 8 when the size is smaller that is 1000*1000 and 2000*2000. The calculation of the size of the matrix can be calculated by the multiplying the number of elements in each matrix by the space that is occupied by each element of the matrix. It is good to know that since the size is 3mb and 12 MB which is actually lesser than the cache then the result is stored in the cache. For the matrices 4000 by 4000 the size is 48 MB which is much bigger than the cache size and hence the matrices have got to be saved in the external memory. The external memory is the RAM and there will be more time that will be needed since this time it is saved in an external memory. It is the reason why the multiplier increases abruptly when the size is at 4000*4000. After this all the matrices have got to be stored in the external memory and the multiplier again after this starts to turn back close to 8 if not 8 exactly.

7. Conclusion

Strassen's algorithm, if used in the right way is a game changer for computing matrices. Its importance is unmeasurable. It can save a lot of our time. It breaks complex matrices into parts and computes them in a very less amount of time. But sometimes traditional algorithms are encouraged as they can solve them quickly. So, for this purpose recursions should be used to help us in distinguishing which algorithm to use. GPU have more cores than that of CPU. Also, parallelization capacities of GPU are much greater than that of CPU which make GPU the obvious choice form the two and make them the number one choice for computing different complex matrices. To prove this practically, we performed certain tasks. We solved different matrices of simple as well as complex size. After repeatedly performing computation, we recorded the results in a table.

All the experiments and comparisons above lead to some key conclusions:

(I) CUDA gives us a taste of all the potential GPU has, gives us processing power like never before.

(II) Matrix multiplication implemented in GPU is always faster/better than implemented in CPU whatever the algorithm may be.

(III) For matrices of bigger sizes, Strassen's algorithm computes the result faster than any other algorithm whether the implementation is done in CPU or GPU.

(IV) In the case of small size matrices, the recursive limit should be applied that should indicate which algorithm should be preferred for solving the matrices.

All these points lead to the fact that GPU should always be preferred for computation. No matter what type of computation it is, it should be implemented on GPU so that time can be saved. To unlock the

ISSN: 2414-1895 DOI: 10.6919/ICJE.202104_7(4).0013

power of GPU's, CUDA is the best platform. Then, we get to know that Strassen's algorithm is the number one option for computing bigger matrices. Also, the importance of having a good recursion technique that will give a quicker output of complex problems.

References

[1] Gary Miller (2018) Introduction and Strassen's Algorithm. http:// www. cs. cmu. edu/ ~15451-f20/ LectureNotes/ lec01-strassen. pdf Blogs Dope (2020) Strassen's Matrix Multiplication. https://www. codesdope.com/blog/article/strassens-matrix-multiplication/

[2] Sanfoundry, Manish Bhojasia, Java Program to Implement Strassen Algorithm. https://www.sanfoundry. com/java-program-strassen-algorithm/

[3] Edward Kandrot, Jason Sanders (2010) CUDA by Example, An Introduction to General-Purpose GPU Programming. NVIDIA.