Strassen-Winograd’s Matrix Multiplication

# Introduction

Strassen is the first algorithm for matrices multiplication with asymptotic complexity O(N2.81) compare to the standard three-loop algorithm O(N3). Even though there’s algorithm invented with lower asymptotic complexity (such as Coppersmith-Winograd with O(N2.38) ), they only provide improvement for matrices which are so large hence not very practical. Strassen’s algorithm still remains the most practical fast matrix multiplication algorithm.

My approach is to apply Strassen’s algorithm in matrix multiplication and at the same time, utilize the parallelism of GPU in implementing core kernel computations (such as Addition, Subtraction) in order to speed up the process.

# General high level design

The pseudo code below shows how the approach implemented:

If (cutoff == true){

Perform standard three-loop algorithm //On device

} else {

Apply Strassen’s algorithm recursively//On host

When Addition or Subtraction is needed,

Transfer participant matrices to device then

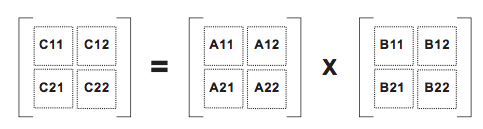
Execute kernel functions //On device

}

# Strassen’s Algorithm

## General Idea

The basic idea of Strassen’s algorithm is to decompose each matrix into 4 equal parts. Given matrices with sizes are powers of 2 (m,k,n).



Instead of computing m x n dot products in order to achieve outcome matrix C, Strassen computes C by 14 additions (subtractions) and 7 multiplications of smaller matrices. Each multiplication then, is recursively computed using Strassen until the matrices become small enough to be computed using the traditional method.

In order to speed up the process, each core computations (Addition, Subtraction and Standard Multiplication) will be executed on GPU and in parallel. Instead of computing each cell of the outcome matrix one by one, each cell of the same matrix will be computed in parallel with each other in different threads. Blocks and Threads are allocated according to each particular matrix and may be varied between different hardware.

Where:

#BLOCK : Actual number of block allocated for a kernel

#THREAD: Actual number of thread per block allocated for a kernel

THREADS\_PER\_BLOCK: A constant represents the maximum number of thread that a block can handle.

m, n: Size of the outcome matrix.

Example: Subtraction kernel

\_\_global\_\_ void kernel\_subtract(float \*a, float \*b, float \*c, int m, int n){

int idx = blockIdx.x \* THREADS\_PER\_BLOCK + threadIdx.x;

if (idx < m\*n){

int x = idx / n;

int y = idx % n;

c[x\*n+y] = a[x\*n+y] - b[x\*n+y];

}

}

Top of Form

Bottom of Form

## Cutoff criterion

Intuitively, we know that it’s not necessary to apply Strassen’s algorithm all the way to scalar level since the traditional three-loop algorithm can perform better on particular small-sized matrices. We use a criterion to determine the cutoff :

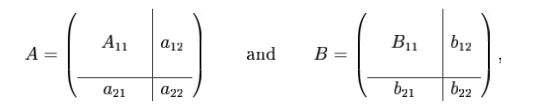
Where:

T is a coefficient derived empirically. In this case, we use T = 12.

m, k and n are respectively sizes of 2 participant matrices (m x k and k x n)

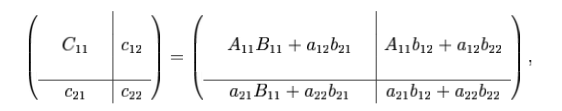
## Dealing with Odd-sized matrices

There’re a few methods that can be used for dealing with odd-sized matrices. In this case, we use Dynamic Peeling which is proved overcomes other methods in terms of processing time.



In the figure above, given A(m,k) and B(k,n) are odd-sized matrices; the figure shows how to solve the odd-sized problem by decomposing matrices’ elements into blocks, which a12 is a (m-1)\*1 matrix, a21 is a 1\*(k-1) matrix, b12 is a (k-1)\*1 matrix, b21 is a 1\*(k-1) matrix and a22 and b22 are scalars.

Then we have matrix C computed by the following formula:



By applying Dynamic Peeling, now we have A11 and B11 are even-sized matrices, hence the product A11B11 can be computed by using Strassen’s algorithm.

The Strassen’s algorithm will be recursively called every time we need to perform matrices multiplication until the cutoff criterion satisfied, then the standard three-loop algorithm will take place to finish the process.

# Conclusion

Our major idea of implementing multiplication is to use Strassen’s algorithm, which is arithmetical the fastest algorithm for multiplication combined with GPU parallelism capability. Applying Strassen’s algorithm decomposes the original multiplication into numerous kernel computations with smaller-sized matrices. It’s not only faster in term of arithmetic but also enables a chance to utilize the parallelism capability of modern GPU .