# SOFT354 Report

Adam Edwards

## 1 Introduction

In this report I will be investigating the use of parallel programming in Matrix-Matrix multiplication, how the efficient it is to do so and how scalable it is.

## 2 Matrix-Matrix multiplication explained

A matrix is a rectangular arrangement of numbers into rows and columns, where each number inside it is referred to an element or entry. Each entry can be referenced with a coordinate-like system, for example in figure 1, the first element in the matrix P is P[0,0] the second is P[1,0] and so on. When multiplying two matrices M and N, each element of the output matrix P is the dot product of a row in matrix M and a column of matrix N.

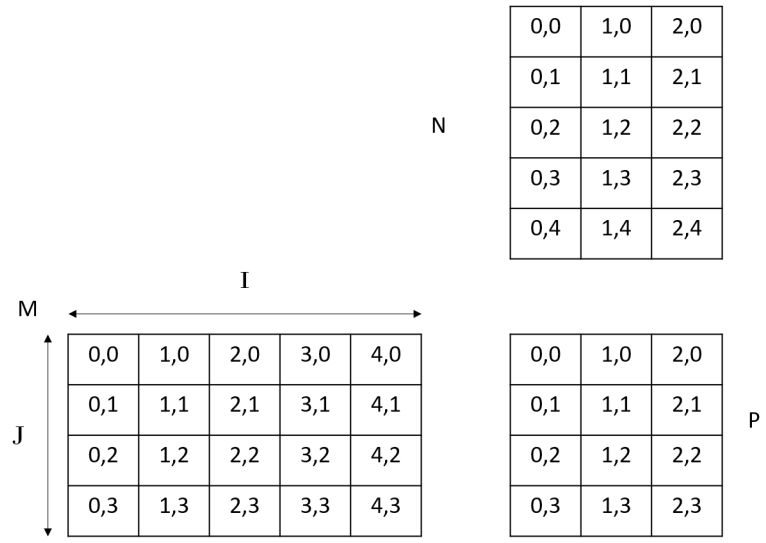



Figure 1: Matrices represented to show how a multiplication is formed

To work out the value of the element P[0,0], we use the formula P[0,0] = M[:,0] N[0,:], where : implies all values. Expanding this we get $P[0,0] = M[0,0] \times N[0,0] + M[1,0] \times N[0,1] + \ldots M[k,0] \times N[0,h]$. leading to the general formula of the figure below.

$$P_{i,j} = \sum_{k=0}^{\text{width}(M)-1} \sum_{h=0}^{\text{height}(N)-1} M_{k,j} N_{i,h}$$

Figure 2: Formula for matrix multiplication *Gianni* (2019)

# 3 Matrix multiplication uses

Matrices are an essential tool for any mathematical program, and are used frequently to represent graphical data, this means implementing matrix operations like matrix multiplication is essential to image processing and other similar programs. Another use of matrices is in machine learning, Neural networks often use matrices to represent the data and in order to update weight matrices matrix matrix multiplication is required in the calculations. As this is such a basic tool for programmers to use its essential that matrix multiplication can be done efficiently and quickly, which is where the use of parallel computation becomes increasingly more valuable.

# 4 CUDA and parallel computation

A CPU deals with instructions in sequential order, the concept of parallel computation is the instructions that do not need to be completed in sequential order can be done at the same time to decrease the time taken to perform calculations.

CUDA is a parallel computational architecture developed by NVIDIA which makes it possible to use the many computing cores in a graphics processor to perform general-purpose mathematical calculations, achieving dramatic speedups in computing performance. In these GPU-accelerated applications the essential sequential parts of the workload is done by the CPU, while compute intensive portions are run on the thousands of GPU cores in parallel. NVIDIA Developer (2020)

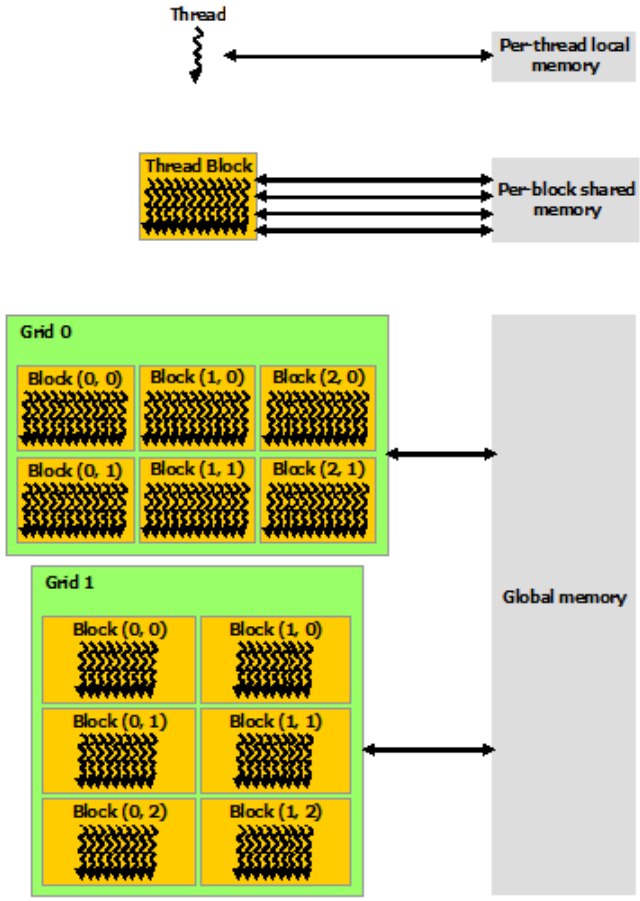



Figure 3: CUDA program architecture *Docs.nvidia.com.* (2020)

A CUDA program uses several GPU cores ordered into Grids, these grids are then split up into blocks, and blocks are split into threads, where each thread executes one part of the parallel program individually. The general program flow of a CUDA program involves a serial portion of the program which sets up memory for the devices, the execution of a kernel, To allow communication between grids blocks and threads, there are several layers of memory, a thread has its own local memory, above that is the shared memory which is shared between all threads in a single block, and above that is the global memory which is visible to all blocks and all grids.

In matrix-matrix multiplication each of the elements in an output matrix (P) can be calculated without the knowledge of any other elements in P. This means that we can apply parallel techniques as a similar process is being applied to each of the elements in M and N to achieve significant speedup as the program size increases.

# 5 Implementation

## 5.1 CPU

Firstly we look at the implementation in a serial solution, given Matrices M and N, and output P. This algorithm is of complexity of approximately $O(n^3)$ meaning that as the size of the program increases the time taken to calculate it increases significantly. *Stothers* (2010)

**Algorithm 1** Calculate $M \times N = P$

**Require:** $width(M) = height(N)$
  Let $width(P) = height(M)$
  Let $height(P) = width(N)$
  **for** $i = 0 : height(P)$ **do**
    **for** $j = 0 : width(P)$ **do**
      $P[i, j] = 0$
      **for** $k = 0 : width(M)$ **do**
        $P[i, j] += M[i, k] \times N[k, j]$
      **end for**
    **end for**
  **end for**
  **return** P

## 5.2 GPU

In a GPU implementation A given thread will handle 1 entry for the P matrix. Using CUDA threads are grouped into blocks, and each block can hold only 1024 threads *Docs.nvidia.com.* (2020). So we use blocks of size $32 \times 32$ to maximise thread usage, and create enough blocks to cover the entire product matrix. If the product matrix is not a perfect division of the blocks it will lead to some threads being inactive on certain iterations, so this also needs to be taken into account. The code below would be run by each thread concurrently.

**Algorithm 2** Calculate $M \times N = P$ GPU *Gianni* (2019)

**Require:** $width(M) = height(N)$
  $sum = 0$
  $row = ThreadY + BlockY \times size(block)$
  $col = ThreadX + BlockX \times size(block)$
  **if** $row > Height(M)$**or**$col > width(N)$ **then**
    **return**
  **end if**
  **for** $i = 0 : width(M)$ **do**
    $sum+ = M[row \times width(M) + i] \times N[i \times width(N) + col]$
  **end for**
  $P[row \times width(P) + col] = sum$

However this algorithm is improvable, as it doesn't make use of the shared memory between threads, currently using this algorithm each thread would require the elements for calculation on local memory, but there is overlap as for example $P[0, 0]$ and $P[0, 1]$ both require $M[0, :]$ so the way to improve this is to load the elements in $M$ and $N$ on the current block into a shared array that all threads can use. This is the algorithm that i implemented as i wanted to fully utilise the parallel system to improve the speed of the program.

**Algorithm 3** Calculate $M \times N = P$ GPU with shared memory

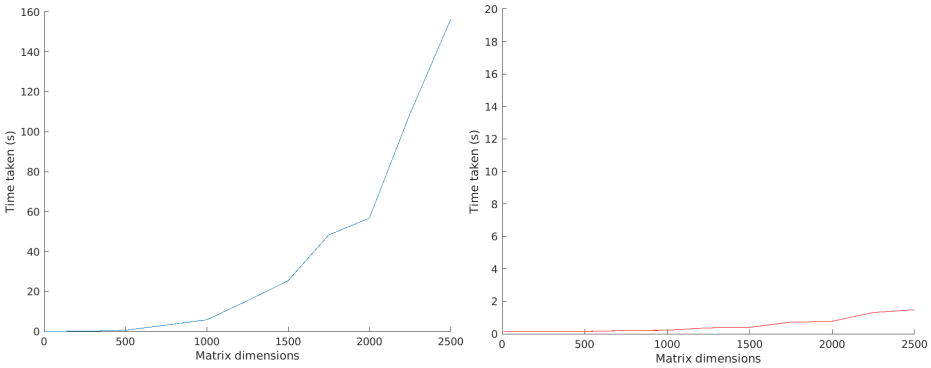
**Require:** $width(M) = height(N)$
  $Ms[size(block)][size(block)]$
  $Ns[size(block)][size(block)]$
  $row = ThreadY + BlockY \times size(block)$
  $col = ThreadX + BlockX \times size(block)$
  **for** each thread in block **do**
    **if** thread within boundaries for M **then**
      $Ms[ThreadY][ThreadX] = M[row][col]$
    **end if**
    **if** thread within boundaries for N **then**
      $Ns[ThreadY][ThreadX] = N[row][col]$
    **end if**
    $sum = 0$
    **for** $i = 0 : size(block)$ **do**
      $sum+ = Ms[ThreadY][i] \times Ns[i][threadX]$
    **end for**
  **end for**
  **if** thread within boundaries for P **then**
    $P[row][col] = sum$
  **end if**

To test the algorithm I created user input for the size of matrices and initialised

these matrices with random numbers so they could be multiplied together. I also made use of the win-timeofday.h from a practical session to get a more accurate reading of the time taken for execution.

# 6 Evaluation

Figure 4




(a) time taken against matrix size (serial) (b) taken against matrix size (parallel)

In order to properly evaluate the two implementations, i took 3 measurements at different matrix sizes from $10 \times 10$ to $2500 \times 2500$ increasing in increments of $250 \times 250$.I then averaged these values across the 3 measurements to ensure consistency. as you can see in figure 4, the benefits of parallel implementation are incredible as the problem size increases, however before matrices of size $500 \times 500$ it is actually slower to use a parallel implementation meaning for smaller programs it is unnecessary to implement this. As expected the graph for serial is exponential, confirming the approximate $O(n^3)$ complexity whereas the parallel is more linear.

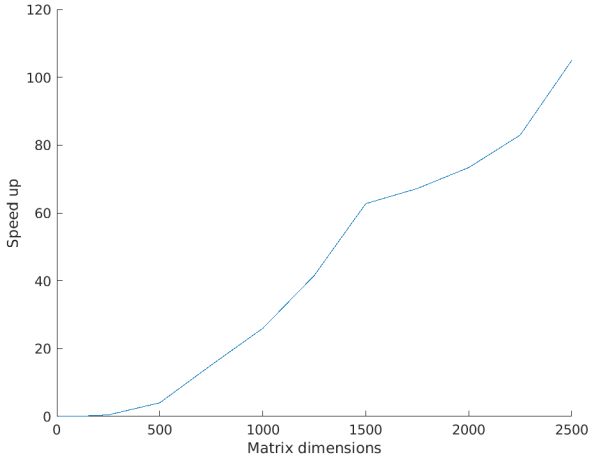



Figure 5: Speedup against matrix size

Using this data its possible to calculate the speedup of the program. The speedup is defined as $S = T_s/T_p$ where $T_s$ is the time taken for the serial solution and $T_p$ is the time taken for the parallel solution *Zhang* (2020). As seen in figure 5 as the problem size increases the speedup increases implying that it is better to use a parallel solution for large problem sizes. It is also possible to calculate the efficiency of the program, where efficiency $E = T_s/pT_p$ where $p$ is the number of processors. The GPU i used for this was a 1070ti, which has 2432 CUDA cores *NVIDIA* (2020), which leads to the graph for efficiency in figure 6. This graph shows that the current implementation is not particularly efficient but as the problem size increases it becomes more efficient.

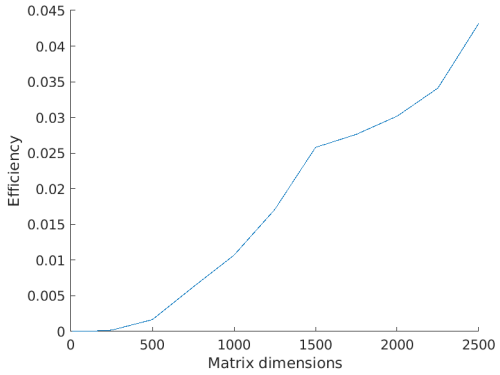



Figure 6: efficiency against matrix size

# 7 Conclusion

To summarise, in a small dataset where matrix dimensions are less than $500 \times 500$ its slower to use a parallel implementation, and until $1000 \times 1000$ there are only a few seconds difference between the two implementations meaning it may be worth it to use depending on the frequency of multiplications. And as matrix dimensions exceed $1000 \times 1000$ its extremely valuable to use parallel programming as the speedup is significant and only increases as the matrices get bigger. Further improvements could be made to this program by dynamically using parallel or serial solution depending on matrix sizes and the available cores on the GPU as this will also affect the speed of calculations.

# References


Docs.nvidia.com. (2020), 'Programming guide :: Cuda toolkit documentation'. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html , Last accessed on 12-01-2020.

Gianni, M. (2019), *Lecture 4 SOFT 356.*
**URL:** *https://dle.plymouth.ac.uk/course/view.php?id=43089*

NVIDIA (2020), 'Introducing the geforce gtx 1070ti graphics card: Gaming perfected'. https://www.nvidia.com/en-us/geforce/products/10series/geforce-gtx-1070-ti/, Last accessed on 12-01-2020.

NVIDIA Developer (2020), 'Cuda zone'. https://developer.nvidia.com/cuda-zone, Last accessed on 12-01-2020.

Stothers, A. J. (2010), *On the Complexity of Matrix Multiplication.*
**URL:** *https://www.maths.ed.ac.uk/sites/default/files/atoms/files/stothers.pdf*

Zhang, J. (2020), 'performance and scalability'. https://www.cs.uky.edu/ jzhang/CS621/chapter7.pdf , Last accessed on 12-01-2020.