# Parallelizing matrix multiplication using openCL

Cameron F Clark[†] and Kian S Frassek[‡]
EEE4120F Class of 2024
University of Cape Town
South Africa
[†]CLRCAM007 [‡]KIAFRS001

## I. Introduction

Parallel computing is a power tool which enables code to run at phenomenal rates compared to single thread. There are many different ways to implement parallel programs on many different problems. The implementation that this report focuses on is using OpenCL and C++ to write a parallel program to efficiently multiply square matrices. Importantly, it is futile to design parallel programs that do not actually increase the efficiency and/or speed of the non parallelized programs, thus, this report will additionally evaluate the performance of the parallel program versus the single thread program with different matrix sizes and number of matrix multiplications.

## II. Methodology

### A. Hardware

The hardware used to conduct these experiments is a MacBook Pro 13" 2019 model. All of these parallel programs were conducted on the GPU which is a Intel Iris Plus Graphics 1536 MB. All of the single thread programs were conducted on the CPU which is 2 GHz Quad-Core Intel Core i5.

### B. Implementation

*1) Single Thread Program:* The single thread program is a relative simple implementation in C code. It requires two for loops which iterate over the row and column of the result matrix and a third which calculates the dot product of the input matrices to find the value of the cell of the output matrix. There is one additional for loop which allows for multiplying a given number of matrices together

```
int matrix_output[matrix_size], matrixA[matrix_size], cell;
for (int c = 0; c < matrix_size; c++) matrixA[c] = matrices[0][c];

//Matrix multiplication
for (int m = 1; m < MATRIX_COUNT; m++) {
 for (int row = 0; row < Size; row++) {
  for (int col = 0; col < Size; col++) {
   cell = 0;
   for (int dot = 0; dot < Size; dot++) {
    cell += matrixA[row * Size + dot] * matrices[m][col + dot * Size];
   }
   matrix_output[row * Size + col] = cell;
  }
 }
 //rewrite matrix A with output so loop can re-iterate
 for (int c = 0; c < matrix_size; c++) matrixA[c] = matrix_output[c];
}
```

*2) Parallel Program:* The parallel program involves setting up OpenCL to run instances of a kernel program. The kernel program itself runs a very similar instance of the cell calculation for the single thread program above where the row is equivalent to the workGroupNumber and the column is equivalent to the localGroupID. A for loop is used to calculate the dot product as in the single thread program. Lastly, there is an additional for loop in the OpenCL setup code which allows for successive matrix multiplication.

```
__kernel void matrixMultiplication(
 __global int* matrixA_buffer,
 __global int* matrixB_buffer,
 __global int* output_buffer
){
 int workItemNum = get_global_id(0); //Work item ID
 int workGroupNum = get_group_id(0); //Work group ID
 int localGroupID = get_local_id(0); //Work items ID within each work group
 int localSize = get_local_size(0); //Get the work number of items per group

 //Row <=> workGroupNum; Column <=> localGroupID
 //workItemNum <=> workGroup * localSize + localGroupID
 int cell = 0;
 for (int i = 0; i < localSize; i++) {
  cell = cell + *(matrixA_buffer + workGroupNum * localSize + i) *
   *(matrixB_buffer + localGroupID + i * localSize);
 }
 *(output_buffer + workItemNum) = cell;
}
```

*3) Code Verification:* Both the single thread and parallel program will undergo tests to ensure their correct functionality before their performances are evaluated. The two programs will carry out the following known matrix multiplication as well as a similar version with matrix sizes of 5:

$$\begin{bmatrix}1 & 2 & 3\\1 & 2 & 3\\1 & 2 & 3\end{bmatrix}\begin{bmatrix}2 & 4 & 6\\2 & 4 & 6\\2 & 4 & 6\end{bmatrix}\begin{bmatrix}1 & 2 & 3\\1 & 2 & 3\\1 & 2 & 3\end{bmatrix}=\begin{bmatrix}72 & 144 & 216\\72 & 144 & 216\\72 & 144 & 216\end{bmatrix}$$

### C. Experiment Procedure

To evaluate the performance of the single thread versus parallel program two experiments will be conducted and evaluate the difference in time taken and the speed up of the parallel program. The first will be multiplying two matrices with different sizes, and the second will be to multiply different numbers of successive matrices of the same size. All matrices of any size will follow the following format for a matrix of size $n \times n$:

$$\begin{bmatrix}1 & 2 & \dots & n\\1 & 2 & \dots & n\\\vdots & \vdots & \ddots & \vdots\\1 & 2 & \dots & n\end{bmatrix}$$

Importantly, both tests will take an average of several tests in order to maximise accuracy and the parallel program will be run additional times before starting its data collection to warm up the cache. Both methods of testing have different amounts of parallelization and different amounts of overhead and should give good understanding of the effectiveness of parallelizing matrix multiplication code.

## III. Results

### A. Multiplying two matrices with different sizes

This section discusses the time taken both for parallelized and single thread multiplying two matrices of the same size. This test repeated the multiplication with the size of the matrix increasing each time from a size of 2 to a size of 200. Both the single thread and parallel programs were run 20 times per size and an average was taken for maximum accuracy The results of the multiplication are shown in figure 1.

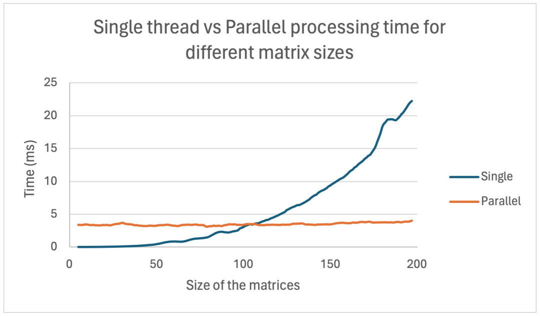

Fig. 1: Graph of single thread vs Parallel processing time for different matrix sizes

The results in 1 clearly show that the time taken for single threaded matrix multiplication is initially faster than parallelized multiplication up to size of the matrix but after a certain threshold the parallelized program becomes faster. With the hardware used in this report, the threshold was roughly at size 100.

It can be seen that both graphs appear to follow an exponential or power trend, with the parallel program having a much lower gain than the single thread program. Additionally, the single thread program times appears to start at close to 0 ms for the smallest matrix size while the parallel program times appear to start at around 4 ms. This is due to the time required to set up the parallel program is 4 ms which is longer than that of the single thread program.

However, as the matrix size increases, the time taken for single thread program increases at a faster exponential rate compared to that of the parallel program, resulting in the parallel program eventually overtaking the single thread program.

### B. Multiplying different numbers of matrices of the same size

This section discusses the time taken for parallelized verses single thread multiplying different numbers of matrices of the same size. Each test was conducted on a different matrix size (5, 10, 20, 50 or 100) and was run with a matrix count of 5 to 45. Both the single thread and parallel programs were run 20 times per count and an average was taken for maximum accuracy

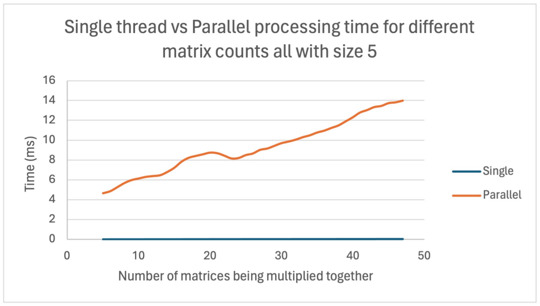

Fig. 2: Single thread vs Parallel processing time for different matrix counts all with size 10

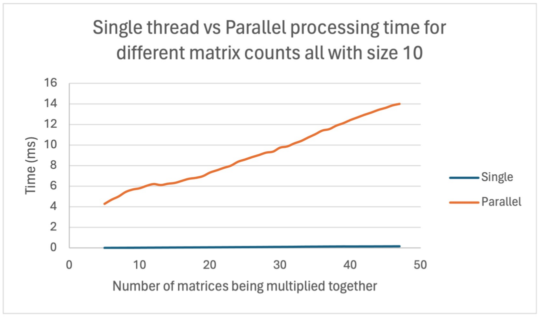

Fig. 3: Single thread vs Parallel processing time for different matrix counts all with size 10

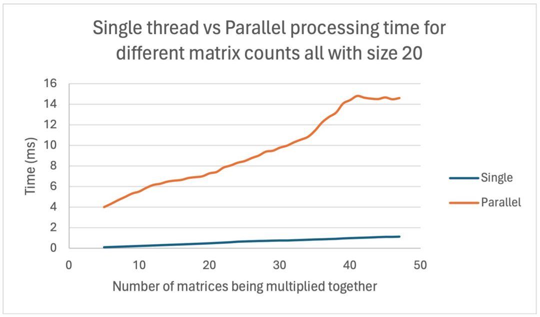

Fig. 4: Single thread vs Parallel processing time for different matrix counts all with size 20

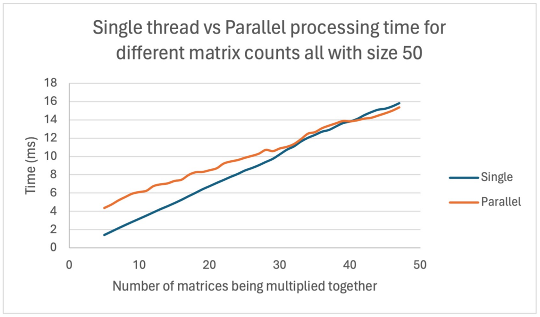

Fig. 5: Single thread vs Parallel processing time for different matrix counts all with size 50

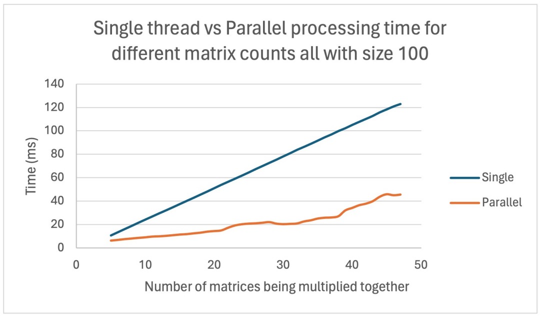

Fig. 6: Single thread vs Parallel processing time for different matrix counts all with size 100

Looking at the results in figures: 2, 3, 4, 5 and 6, it can be seen that both the single and parallel programs' time follow an approximate linear trend and that the parallel program has an initial overhead of around 5 ms with a matrix count of 5. While the graphs cannot extend to where zero matrices are multiplied together, they can be extrapolated to show that the parallel program always has an initial overhead that is significantly higher than the single thread program, even in figure 6.

It should be noted that the overhead for the parallel program can be broken up into to parts: the overhead of setting up the OpenCL device and programs and the overhead of starting each instance of the kernel program. The latter, in conjunction with the runtime of the kernel program itself, occurs once per matrix multiplication and is the reason that the parallel program's time follows a linear trend. By extension, the single threaded program also follows a linear trend for similar reasoning.

It also should be noted that the speed of the parallel program increases compared to the single thread program as the matrix count increases and as the matrix size increases. The speed up with the increase of matrix count is due to the same reasoning as in the first test, where the single thread program's time increases at a fast rate eventually allowing for the parallel program to overtake it. This is shown in figure 5 where the threshold point was around 40 matrices of size 50x50 being multiplied together.

The speed up with increasing matrix sizes, which is the more significant in this test, is due to the percentage of parallelism in the parallel program increasing as the matrix size increases. The increase in parallelism in due to the parallel program has three parts as mentioned above: the OpenCl setup overhead which stays constant, the kernel starting overhead which scales linearly with matrix count, and the kernel program which scales exponentially with matrix size but still at a slower rate than the single thread program as shown in section III-A. These factors combined cause an increase in parallelism with increasing matrix size which causes the parallel program to become faster than the single thread program as the matrix count increases.

### *C. Multiplying different numbers of matrices with different sizes*

This section is a combination of the two above which produces the graph shown in figure 7. Both the single thread and parallel programs were run only run once as running an average of multiple tests on each node reduces the total number of unique nodes that can be tested using the same amount of time and resources, and using more data points is more valuable to this report.

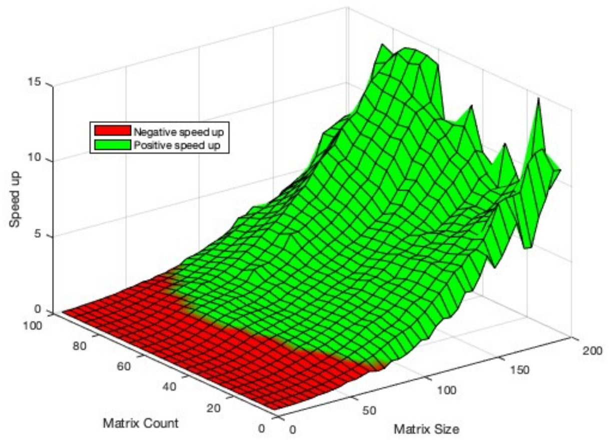

Fig. 7: A Surface plot showing the speed up from single thread to parallel matrix multiplication over different numbers of matrices of different size multiplied together

This show that there is a clear threshold line of matrix size and matrix count where is it more efficient to parallelize matrix multiplication for any increase in either parameter. With the hardware used in this report, the line extends roughly between 2 matrices of size 100x100 and 60 matrices of size 50x50 multiplied together.

The graph also indicates that for a very high number of matrices it suddenly become more efficient to use a single thread program which appears counter intuitive. This could be due to 50 matrices of size 50x50 taking up too much space to be stored on the GPU cache on the hardware used in this report, and thus additional time is taken to transfer the data to the GPU cache; however further testing will be required to confirm this hypothesis.

## IV. Conclusion

This report shows that for multiplying small matrix sizes and counts, single-threaded matrix multiplication is faster, however, when the matrix sizes and counts are large enough, parallelized matrix multiplication becomes faster due to its overhead becoming negligible relative to its computation time.

The overheads for the parallel matrix multiplication program are the OpenCL setup overhead and kernel startup overhead, with the former being constant and the latter increasing linearly with matrix count.

Lastly, there is an unexplained decrease in speed up for small matrix sizes and matrix counts over 60 which is theorised to be caused by the matrix arrays allocation and transfer between the heap, stack and cache, however this hypothesis still requires further testing to confirm.