The assessment of 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;
}
```

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. 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. The test repeated the multiplication with the size of the matrix increasing each time. The results of the multiplication are shown in figure 1.

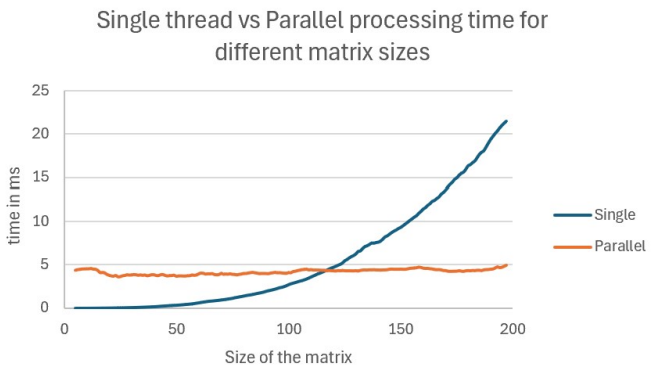

Fig. 1: Graph of single thread vs Parallel processing time for different matrix sizes

The results show that the time taken for single-thread matrix multiplication is initially faster than parallelized multiplication up to size of the matrix around 117. After size 117, parallelized multiplication becomes faster. This is indicated on the graph by time taken for the single thread processing increases with a larger exponential scale. These results occur because the time due to overhead of setting up parallel processing is 4 ms which is larger than the time taken when running the program with small sized matrices on a single thread. However, as the matrix size increases, the time taken for single-thread multiplication increases at a faster exponentially rate, resulting in longer processing time.

B. Multiplying different number of matrices with same sizes

This section discusses the time taken for parallelized verses single thread multiplying different numbers of matrices of the same size.

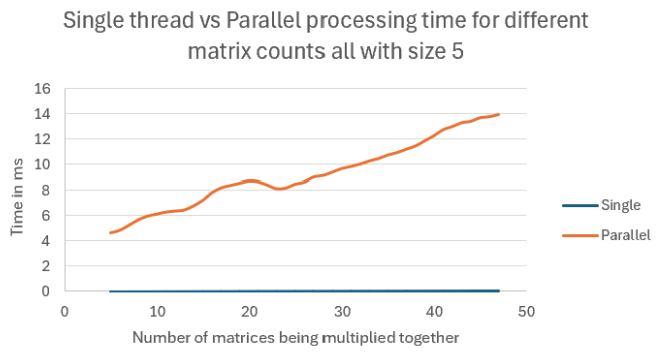

Fig. 2: Single thread vs Parallel processing time for different matrix counts all with size 10

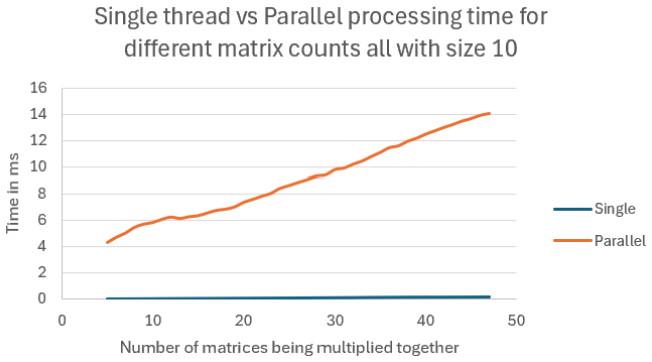

Fig. 3: Single thread vs Parallel processing time for different matrix counts all with size 10

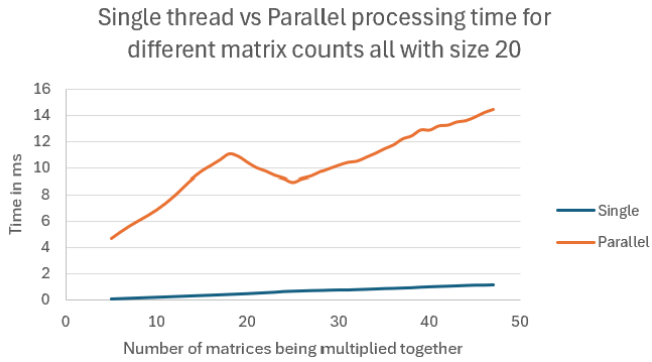

Fig. 4: Single thread vs Parallel processing time for different matrix counts all with size 20

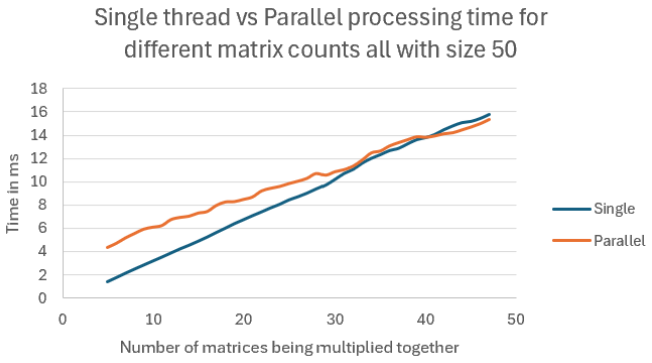

Fig. 5: Single thread vs Parallel processing time for different matrix counts all with size 50

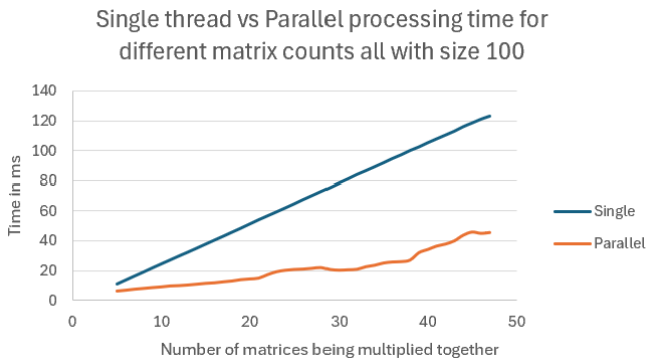

Fig. 6: Single thread vs Parallel processing time for different matrix counts all with size 100

Looking at the results in figures: 2, 3, 4, 5 and 6, It can be seen that there are two different types of over heads the openCl setup overheads and the overheads for starting the kernel. The openCl setup overheads are the same for all the different matrix counts, this is because the openCl setup is done once and the same for all the different matrix counts. This is indicated in our graph because the time taken at matrix count of 5 remain around 5ms independent of matrix size.

The overhead of starting the kernel to run seem to result in a linear increase in time as the matrix count increases. This occurs because each time the a new test is run a the buffer need to be re initialized and the kernel need to be started. This can be seen in the graph because both the parallelized and single thread result in linear increase in time as the matrix count increases. however percentage of time spent parallelizing decreases as the matrix count increases. This is indicated in the in figure 5 where the linear graf of the time take vs matrix cont for parallelized multiplication and single thread multiplication intersect. in other words there is an increase in efficiency of parallelized multiplication as the matrix count increases.

Looking at all the graph it can be seen that as the matrix size increases the speed up for parallelized multiplication vs the single thread multiplication improves. In other words at smaller matrix sizes the time taken for parallelized multiplication is longer than the time taken for single thread multiplication and at larger matrix sizes the time taken for parallelized multiplication is shorter than the time taken for single thread multiplication.

IV. Conclusion

The experiments show tha that small matrix sizes, single-threaded matrix multiplication is faster due to over heads. As the matrix size increases, parallelized multiplication becomes quicker due to overhead becoming negligible relative to computation time. The overhead includes OpenCL setup overheads and kernel startup overheads, with the former being constant and the latter increasing linearly with matrix count. Due to the kernel startup overheads , parallelization efficiency improves slightly with larger matrix counts. Overall, parallelized multiplication offers significant speedup for larger matrices compared to the single-threaded approach.