EEE4120F Practical 02

Rory Schram[†] and Natasha Soldin[‡]
EEE4120F Class of 2023
University of Cape Town
South Africa
[†]SCHROR002 [‡]SLDNAT001

***Abstract*—This paper explores the use of OpenCL and by extension parallel programming on a matrix multiplication program implementation to test the advantages afforded over a serial implementation.**

I. Introduction

A. Aim

To determine the potential performance benefit of using parallel programming on a matrix multiplication program.

B. Theory

This practical uses Open Computing Language (OpenCL), which is a heterogeneous programming tool that implements cross-platform parallel programming. Therefore, it is possible to complete program computations using multiple processors.

OpenCL improves the overall performance of a program's execution by 'passing data' from the central processing unit (CPU) to the graphics processing unit (GPU) for processing executed by a kernel, in actuality this is done by passing the GPU a pointer to a memory buffer stored in the Random Access Memory (RAM) utilised by the CPU. The result of which is returned to the CPU for output.

This practical aims to compare and contrast the performance of the Golden Standard (serial) program implementation and OpenCL program implementations. Specifically, in terms of speed-up and data transfer overhead.

The program used for comparison will be a matrix multiplication program which has been implemented using serial and parallel programming methods. The parallel implementation utilises the GPU by using OpenCL and a kernel which we create to run the required code.

II. Methodology

A. Hardware and Software

The software used in this practical is predominantly C++ in combination with OpenCL and the hardware involved is the hardware that is installed on our designated PC in Blue Lab. These PCs have the necessary software packages installed. The CPU and GPU specifications of the PC are summarised in table I below:

TABLE I: CPU and GPU Specifications

	CPU	**GPU**
Type	Intel(R) Core(TM) i5-10500	GP108[GeForce GT 1030]
Vendor	Genuine Intel	NVIDIA Corporation
Clock Speed	3.10GHz	1.228 GHz
No. of Cores	12	384

B. Implementation and Experimental Procedure

The following programs were utilised in practical and can be found in a shared GIT repository:

- `multiplication.cpp`: C++ program containing all the necessary files for OpenCL programs. It contains two functions (`createKnownSquareMatrix` and `createRandomSquareMatrix`) which output either a known or random square matrix in the form of a 1D array. The other functions aid in the kernel applications operation.
- `kernel.cl`: this program details exactly how the matrix multiplication works in parallel. This is done by splitting the matrix multiplication into separate tasks that are able to execute simultaneously.
- `multiplicationGoldenStandard.cpp`: C++ program that implements the practical's golden standard. This program simply performs the required matrix multiplication with a serial implementation.

Program execution time throughout the practical will be measured using the following code:

Listing 1: Time Measurement Function

```
start = clock(); //start running clock
\\ critical section of code needing to be timed
end = clock();
printf ("Run␣Time:␣%0.8f␣sec␣\n",((float) end - start)/CLOCKS_PER_SEC);
```

Each time measurement recorded is an average of 10 individual time measurements to ensure minimal discrepancies. Running tests multiple times also produces more accurate data results as it allows for the program to 'warm up' (i.e. for cache to load) and is a standard of good testing practices.

1) Speed-Up: Speed-up is a statistical function that compares the runtime of an un-optimised program T_{p1} (i.e. CPU execution time) to that of an optimised program T_{p2} (i.e. GPU execution time).

$$Speedup = \frac{T_{p1}}{T_{p2}} = \frac{Golden\ Standard}{OpenCL} \tag{1}$$

The serial (CPU) and parallel (GPU) matrix multiplication programs will be applied to matrices of varying sizes and their execution times will be measured. The measurement for serial/ golden standard execution time will be taken using the following code:

Listing 2: Golden Standard Time Measurement

```
start = clock();
    for (int row = 1; row <= Size; row++) {
            for (int col = 1; col <= Size; col++) {
                    index = (row*Size + col) - (Size + 1);
                    for (int i = 1; i <= Size; i++) {
                            result += matrixA[(row*Size + i) -
            (Size + 1)]*matrixB[(i*Size + col) - (Size + 1)];
                    }
                    output[index] = result;
                    result = 0;
            }
    }

    long long outputFinal[countA];
    result = 0;
    index = 0;

    for (int row = 1; row <= Size; row++) {
            for (int col = 1; col <= Size; col++) {
                    index = (row*Size + col) - (Size + 1);
                    for (int i = 1; i <= Size; i++) {
                            result += matrixA[(row*Size + i) -
            (Size + 1)]*output[(i*Size + col) - (Size + 1)];
                    }
                    outputFinal[index] = result;
                    result = 0;
            }
    }
end = clock();
printf("\nCompute␣Time:␣%0.8f␣sec\n",((float) end - start)/CLOCKS_PER_SEC);
```

Whereas the measurement for the parallel/OpenCL execution time, as shown by the code below, includes both data transfer overhead time and computation time as described by the following equation:

$$OpenCL\ Exececution\ Time = Data\ Transfer\ Overhead + Computation\ Time \tag{2}$$

Listing 3: OpenCL Time Measurement

```
    start = clock();

    // Create data buffers for memory management between the host and the target device
    size_t global_size = Size*Size; //total number of work items
    size_t local_size = Size; //Size of each work group
    cl_int num_groups = global_size/local_size; //number of work groups needed

    // Initialize output array
    long output[global_size]; //output array

    // Create matrixA_buffer, matrixB_buffer and output_buffer, with clCreateBuffer()

    matrixA_buffer = clCreateBuffer(context,CL_MEM_READ_ONLY |
CL_MEM_COPY_HOST_PTR, countA*sizeof(long), &matrixA, &err);
    matrixB_buffer = clCreateBuffer(context,CL_MEM_READ_ONLY |
CL_MEM_COPY_HOST_PTR, countB*sizeof(long), &matrixB, &err);
    output_buffer = clCreateBuffer(context,CL_MEM_READ_WRITE |
CL_MEM_COPY_HOST_PTR, global_size*sizeof(long), &output, &err);
    size_buffer = clCreateBuffer(context,CL_MEM_READ_ONLY |
CL_MEM_COPY_HOST_PTR,sizeof(int), &Size, &err);


    // Create the arguments for the kernel (link these to the buffers set
above, using the pointers for the respective buffers)
    clSetKernelArg(kernel, 0, sizeof(cl_mem), &matrixA_buffer);
    clSetKernelArg(kernel, 1, sizeof(cl_mem), &matrixB_buffer);
    clSetKernelArg(kernel, 2, sizeof(cl_mem), &output_buffer);
    clSetKernelArg(kernel, 3, sizeof(cl_mem), &size_buffer);


    end = clock();
    float dataTransferTime = ((float) end - start)/CLOCKS_PER_SEC;

    start = clock();
    // Enqueue kernel, deploys the kernels and determines the number of work-
items that should be generated to execute the kernel (global_size) and the
number of work-items in each work-group (local_size).
    cl_int err4 = clEnqueueNDRangeKernel(queue, kernel, 1, NULL, &global_size,
&local_size, 0, NULL, NULL);
    printf("Kernel␣check:␣%i␣\n",err4);

    //Allow the host to read from the buffer object
    err = clEnqueueReadBuffer(queue, output_buffer, CL_TRUE, 0, sizeof(output),
output, 0, NULL, NULL);

    //This command stops the program here until everything in the queue has been run
    clFinish(queue);

    end = clock();

    float computeTime = ((float) end - start)/CLOCKS_PER_SEC;
    float totalTime = dataTransferTime+computeTime;
```

It is hypothesised that at small matrix sizes the CPU time will be the same or even faster than the GPU time due to parallel and Data Transfer Overhead. However, at large matrix sizes, the GPU time will be significantly faster than that of the CPU's time and the CPU's time will increase in a manner described as order $O(n^3)$ due to the serial algorithm utilisation of three nested for loops. Thus the speed-up is hypothesised to increase drastically as the matrix size increases.

2) Data Transfer Overhead: OpenCL's method of program parallelisation is to transmit a buffer pointer variable to the GPU that allows it to access the data stored at a memory location. This custom memory space creation results in a time overhead, broadly coined "data transfer overhead". This overhead is hypothesised increase linearly with the matrix size as a larger matrix will contain a greater amount of data needing to be transferred and thus a larger data transfer overhead.

III. Results and Discussion

The serial (i.e. Golden Standard) and parallel (i.e. OpenCL) programs were run over varying matrix sizes and their execution times were measured. The speed-up was then calculated for each matrix size using the method shown in equation 1 above. This data is tabulated in table II below:

TABLE II: Time Data Collected from Experiments and Speed-Up Calculated

Matriz Size	**Golden Standard Execution Time (s)**	**OpenCL Execution Time (s)**	**Speed-Up**
10	0,000027	0,00019	0,142105263
20	0,000196	0,000172	1,139534884
40	0,001532	0,000235	6,519148936
60	0,00277	0,000292	9,48630137
80	0,003685	0,000401	9,189526185
100	0,005516	0,000546	10,1025641
120	0,008827	0,000773	11,41914618
140	0,013706	0,001056	12,97916667
160	0,019231	0,001265	15,20237154
180	0,022317	0,001788	12,48154362
200	0,038944	0,002715	14,34401473
220	0,051203	0,003223	15,88675147
240	0,068857	0,003658	18,82367414
260	0,086687	0,004711	18,40097644
280	0,104274	0,005631	18,51784763
300	0,13314199	0,007139	18,64994957
320	0,190387	0,008143	23,38044947
340	0,19427	0,009395	20,67802022
360	0,222109	0,01065	20,85530516
380	0,270475	0,011832	22,85961799
400	0,336054	0,0157604	21,32268217
420	0,36698899	0,017042	21,53438505
440	0,409991	0,018808	21,79875585
460	0,49077499	0,020728	23,67690998
480	0,64441299	0,022647	28,45467347
500	0,66327202	0,02488	26,65884325

A. Execution Time

From table II above, the execution time of the golden standard and OpenCL are plotted together against matrix size as shown in figure 1 below.

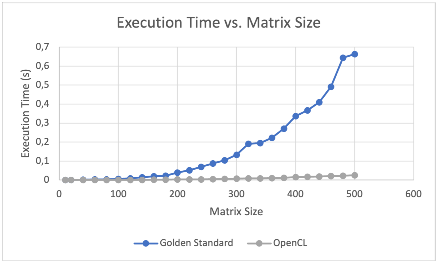

Fig. 1: Execution Time vs. Matrix Size

From this it is evident that at smaller matrix sizes (i.e. 10x10 to 100x100), the serial and parallel programs' execution times were similar. However, as matrix size increases (100x100 and above), the serial program's execution time increases alongside while the parallel program's execution time stays relatively constant emphasising the proper operation and yielded benefit of the parallel implementation.

B. Speed-Up

The Speed-Up shown in table II above is plotted in figure 2 below:

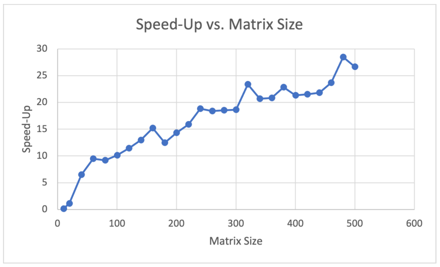

Fig. 2: Speed-Up vs. Matrix Size

From this it is evident that the speed-up increases with matrix size, this is due to the numerator (i.e. serial execution time) increasing in size and the denominator (i.e. parallel execution time) remaining relatively constant when referencing equation 1. As previously discussed, the real benefit of the OpenCL implementation is predominantly seen after the matrix size has increased past 100x100. The speed-up at matrix size 10x10 is 0.1421, at 100x100 is 10.1026 and at 500x500 is 26.6588.

C. Data Transfer Overhead

The data transfer overhead involved in the OpenCL's parallel computation is shown in figure 3 below

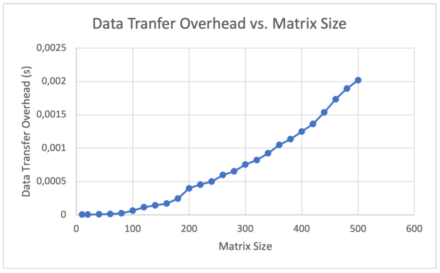

Fig. 3: Data Transfer Overhead vs. Matrix Size

This emphasises that as the matrix size increases so too does the data transfer overhead. This is due to a greater amount of data (i.e. matrix data) needing to be stored in custom memory buffers accessible to the GPU.

The OpenCL execution time is comprised of data transfer overhead time and actual computation time as expressed by equation 2 above and graphically shown in figure 4 below.

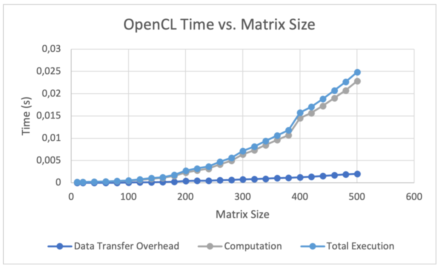

Fig. 4: OpenCL Time vs. Matrix Size

This shows that the OpenCL execution time is dominated by the computation time but that the data transfer overhead still contributes, especially as the matrix size increases. This is due to the increase of data transfer overhead proportional to an increase in matrix size as shown in figure 3 above.

D. Number of Threads

Due to the parallel implementation of the OpenCL program, the number of threads created is directly proportional to the matrix size. The relationship is described by the following equation and graphically shown in figure 5 below.

$$Number\ of\ Threads = Matrix\ Size^2 \tag{3}$$

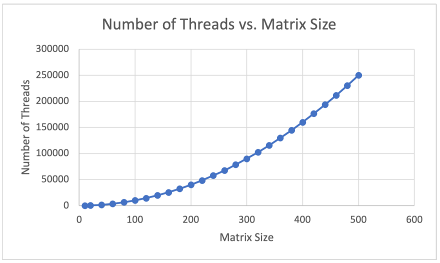

Fig. 5: Number of Threads vs. Matrix Size

IV. Conclusion

Using OpenCL's parallel programming for matrix multiplication was only significantly better than a serial implementation once the matrix size reached 100x100. At this matrix size the speed-up was 10.1026 and as the matrix size increased as too did the speed-up. The data transfer overhead also increased with matrix size as there was a greater amount of data to be transferred. However, this data transfer overhead increase is not worry some due to the little contribution made by the data transfer overhead to the parallel execution time. In conclusion, as with many parallel program implementation, there is only significant advantage derived at larger dataset values. However, at these larger values, parallel program implementation is deemed worthwhile as a result of increased program efficiency, speed and resource allocation.