The utilisation of parallelised code on a GPU over a CPU through OpenCL and the significance of decreased processing times

Simeon, J.
Griffith School of Engineering
Griffith University
Cannon Hill, Australia
joseph.simeon@griffithuni.edu.au

***Abstract*—The advantage of OpenCL for processing information/data with highly parallelable code across a multitude of compute units within multicore devices such as GPU (integrated or otherwise).**

Keywords—computers, parallel, GPU, CPU, processing.

I. Introduction

Processing information is a necessity in all aspects through-out the world, this includes a hidden tug-of-war that is unseen which is a balance of Resources (Memory, Processing power) Vs. Time (How long to process). Graphics processing units (GPUs) within this modern day have become increasingly more powerful over the last decade with GPUs outclassing Computer processing units (CPUs) in the running of processes such as computationally heavy algorithms, running a process through a CPU will perform this as quick as the CPUs cores will allow it. When processes are parallel processed the CPU is limited to how many cores are available at one time to process each thread being used whereas current GPUs surpass the limit of the CPU by having many cores existing on one GPU board which allows the GPU is circumnavigate some of the Resource Vs. Time struggle that the CPU experiences running processes. GPUs allow the modelling of specific occurrences in nature that cannot be easily processed by geometric shapes such as coastlines, leaves and ferns. However, can be modelled using fractals, computing fractals through well-known algorithms is heavy computational and highly parallelable which is the perfect testing processing of the CPU and GPU through OpenCL methods and the relationship of Resources Vs. Time to those methods.

II. Highly Parallelable Code using Mandelbrot Fractal

Mandelbrot set is collection of complex numbers (c) which using the algorithmic function:

$$f_c(z) = z^2 + c$$

which does not diverge when iterated from specific x,y parameters inputted as the complex number (c) where c's real value is x-parameter and c's imaginary value is y-parameter. An escape time algorithm is used in conjuncture with the Mandelbrot set algorithm as that when the function does not diverge after a number of executions of the function, it is terminated and the based upon the iterations it takes to either complete to a specific value or until the max amount of iterations is allowed which then a pixel corresponding to the x,y parameter is coloured which is determined by the number of iterations that function has completed this is the highly parallelablity of the Mandelbrot set for the testing of the methods that will be used in this report. The picture produced from this method shows a Mandelbrot fractal seen below.

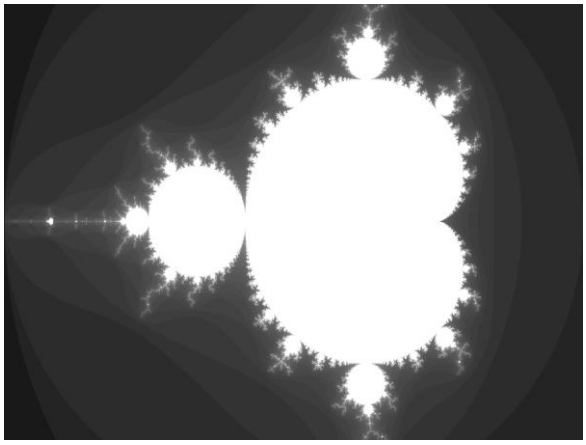

Fig 1. *Mandelbrot image processed through Open Graphics Library (OpenGL) using results generated from code that will be presented later on in the report.*

III. Computer Specifications

The specifications of the system in which the code will be processed on are below:

TABLE I. CPU Specifications

i5-4570s	
Brand	Intel
# cores	4
Clock speed	2.9 - 3.6 (Turbo) GHz
Cache	6 MB
Bus speed	5 GT/s
PCUs*	4
Local Memory Size	32 KB
Global Memory Size	2047 MB
Max Alloc Size	511 MB
Max Work-group Size	8192
Max Work-Item Dimensions	3

TABLE II. GPU Specifications

Haswell GT2	
Brand	Intel
# cores	160
Core speed	200 – 1350 (Boost) MHz
TMUs*	20
ROP*	2
Memory (Type, Size)	System shared
Bus width	System shared
PCUs*	20
Local Memory Size	64 KB
Global Memory Size	1488 MB
Max Alloc Size	372 MB
Max Work-group Size	512
Max Work-Item Dimensions	3

*TMUs – Texture Mapping Units

*ROP – Raster Operation Pipeline

*PCUs – Parallel Computing Units

IV. Baseline Process using CPU

The baseline program computes the Mandelbrot set using the CPU's processing speed as in steps through the Mandelbrot algorithm computing positions of x,y values that correspond to pixels on a screen, some of the code is shown below:

```
        a.re = xmin;
        a.im = ymin;

        FILE *fp = fopen("file.txt", "w+");

        while(a.im > ymax){
            while(a.re < xmax){
                data = MandelBrot(a);
                fprintf(fp, "%d ", data);
                a.re = a.re + stepx;
                counter++;
            }
            fprintf(fp, "%c", newline);
            a.im = a.im - stepy;
            a.re = xmin;
        }

        fclose(fp);
        after = clock();
        time_spent = (double)(after - before) / CLOCKS_PER_SEC;

        fp = fopen("time_baseline.txt", "a");
        fprintf(fp, "%g ", time_spent);
    }
```

Fig 2. *Snippet code of the main function*

The user is prompted with a message to ask for the range at which the Mandelbrot set will be calculated, for the purpose of timing, the range has been set from -2 < x < 1 and 1 > y > -1, these ranges will make sure that the image is the exact same being produced each time for the different methods that will be discussed later on in this report. The ranges that have been entered are being increased and decreased respectively by a step value which is given by these two equations:

$$stepx = \frac{\max x - \min x}{width}$$

$$stepy = \frac{\max y - miny}{height}$$

where the width and height of the screen to which the pixels will be shaded with OpenGL is fixed at a width of 1024 and a height of 768. These pixels are given to the Mandelbrot function as a structure which has only two variables, the x parameter is given to the structure's re variable which is the complex real and manipulated via the stepx variable and the y parameter is given to the structure's im variable which is the complex imaginary and manipulated via the stepy variable.

```
/* PROCESSES COMPLEX NUMBERS USING MANDELBROT ALGORITHM */
int MandelBrot(ComplexNum c){
    int counter = 0;
    struct ComplexNum z;
    z.re = 0;
    z.re = 0;

    while((complexmag(z)<=2) && (counter < MAX_ITER)){
        counter++;
        // z = z^2 + c, where c = cr + ci
        z = structadd(complexsq(z), c);
    }
    return counter;
}
```

Fig 3. *Snippet code of the Mandelbrot function*

The Mandelbrot function is the same as the Mandelbrot set with a threshold of two for the absolute value of the complex number z and a max iteration threshold of 100, this will ensure that the while loop with exit when the Mandelbrot set does not converge, the returned counter will show that all 100 values are non-converging points of the Mandelbrot. Other functions executed for the Mandelbrot set are complex number mathematic functions which calculate the magnitude, squaring a complex number and adding complex numbers together, this is the same program that generated the Mandelbrot image from Fig 1. of this report.

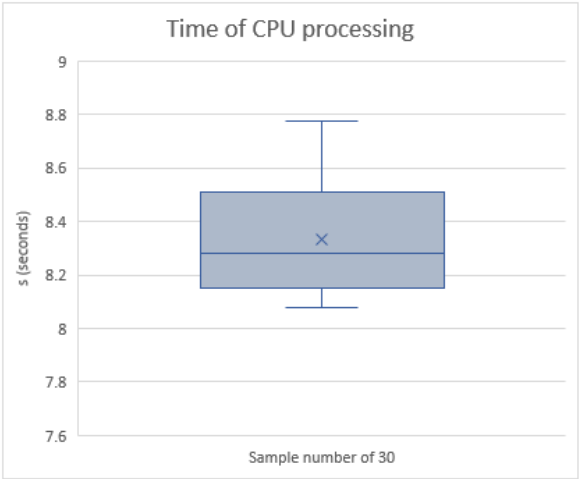

Fig 4. *Box and whisker plot of the time taken to process the Mandelbrot set using 30 samples.*

The results of the CPU in calculating the Mandelbrot set took an average of 8.333 seconds.

V. NDRange Processing through OpenCL via GPU

OpenCL allows for parallel process of the Mandelbrot set through the use of OpenCL specification functions that prepares the GPU for use via step by step process help

guides created by many developer friendly companies such as NVIDIA. In the previous section of CPU processing the main part of the processing was being done by the Mandelbrot set algorithm, within OpenCL this part would be ran on the device called the kernel, which is primarily in C dialect of programming language.

The processing will be done using OpenCL's NDRange method in which 1DRange (using a one dimension array of information) will be given as arguments to the device called a kernel which will process the information and return a integer of a count, this method will be analysed next to the 2DRange (using a two dimension array of information) which will be given as the same kernel arguments as 1DRange but the setting up of the host data and how the kernel interacts with the 2DRange and 1DRange will be different.

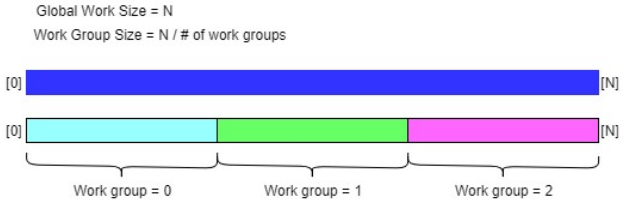

Fig 5. *NDRange visual display for 1DRange.*

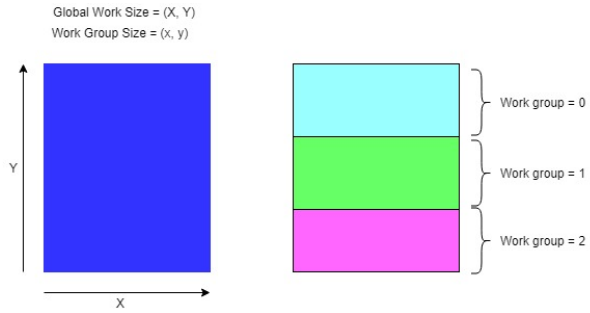

Fig 6. *NDRange visual display for 2DRange.*

Generalised steps for setting up OpenCL for NDRange.

1. Discover and initialise the platforms.
2. Discover and initialise the devices.
3. Create a context.
4. Create a command queue.
5. Create device buffers.
6. Create and compile the program.
7. Create the kernel.
8. Set the kernel arguments.
9. Configure the work-item structure.
10. Write host data to device buffers.
11. Enqueue the kernel for execution.
12. Read the output buffer back to the host.
13. Release OpenCL resources.
14. Release host resources.

The timing for the program will occur at the start and end of steps 10 and 12 respectively, the initialisation of overheads leading up to the sending and receiving of the data to and from the GPU is not included in the timing of the programs.

```
// Compute the size of the data
int elements = width*height;

// Allocate space for input/output data
size_t datasize_float = sizeof(float)*(elements);
size_t datasize_int = sizeof(int)*(elements);

// Acquire the limits for the x,y region to be calculated
printf("Please enter the min and max range value for x & y, respectively.\n");
scanf(" %i %i %i %i", &xmin, &xmax, &ymin, &ymax);
printf("Please wait while the process completes.\n");

// Calculate the steps for the variables
stepx = (float)(xmax - xmin) / width;
stepy = (float)(ymax - ymin) / height;

// Malloc array sizes
A = (float *)malloc(datasize_float);
B = (float *)malloc(datasize_float);
C = (int *)malloc(datasize_int);

//Initialise the input data
xvar = xmin;
yvar = ymax;
i = 0;
while(xvar < xmax){
    while(yvar > ymin){
        A[i] = xvar;
        B[i] = yvar;
        yvar = yvar - stepy;
        i++;
    }
    xvar = xvar + stepx;
    yvar = ymax;
}
```

Fig 7. *1DRange host data setup snippet.*

```
const char* programSource =
"__kernel                                                   \n"
"void mandelbrot(__global float *A,                         \n"
"                __global float *B,                         \n"
"                __global int *C){                          \n"
"                                                           \n"
"    // obtain work-item's id                               \n"
"    int idx = get_global_id(0);                            \n"
"                                                           \n"
"    // transferring host data into                         \n"
"    // real(Re) +i imaginary(Im)                           \n"
"    float Re = A[idx];                                     \n"
"    float Im = B[idx];                                     \n"
"                                                           \n"
"    // internal real(ZRe) +i imaginary(ZIm)                \n"
"    float ZRe = 0;                                         \n"
"    float ZIm = 0;                                         \n"
"    // internal temp real(TRe) +i imaginary(TIm)           \n"
"    float TRe = 0;                                         \n"
"    float TIm = 0;                                         \n"
"                                                           \n"
"    // counter                                             \n"
"    int counter = 0;                                       \n"
"                                                           \n"
"    while(((ZRe*ZRe + ZIm*ZIm) <= 4) && (counter < 100)){  \n"
"        counter++;                                         \n"
"        // z = z^2 + c, where c = Re +i Im                 \n"
"        TRe = ((ZRe*ZRe) - (ZIm*ZIm)) + Re;                \n"
"        TIm = (2*ZRe*ZIm) + Im;                            \n"
"                                                           \n"
"        ZRe = TRe;                                         \n"
"        ZIm = TIm;                                         \n"
"                                                           \n"
"    }                                                      \n"
"    C[idx] = counter;                                      \n"
"}                                                          \n"
;
```

Fig 8. *1DRange kernel snippet which utilises only the global id of 0.*

```
//--------------------------------------------------
// STEP 9: Configure the work-item structure
//--------------------------------------------------

// Define an index space (global work size) of work
// items for execution.
// A workgroup size (local work size) is not required,
// but can be used.
size_t globalWorkSize[1];
// There are 'elements' work-items
globalWorkSize[0] = elements;
```

Fig 9. *1DRange OpenCL Step 9, globalWorkSize having the same of elements which is length multiplied by the width of the Mandelbrot fractal image.*

```
//---------------------------------------------------------
// STEP 12: Read the output buffer back to the host
//---------------------------------------------------------

clEnqueueReadBuffer(
    cmdQueue,
    bufferC,
    CL_TRUE,
    0,
    datasize_int,
    C,
    0,
    NULL,
    NULL);

after = clock();
// Verify the output
fp = fopen("1Dresults.txt", "w+");

ind = 0;
for(i = 0; i < elements; i++){
    //printf("%i ", C[i]);
    fprintf(fp, "%i ", C[i]);
    ind++;
    if(ind > 1024){
        fprintf(fp, "%c", newline);
        ind = 0;
    }
}

fclose(fp);
time_spent = (double)(after - before) / CLOCKS_PER_SEC;
fp = fopen("time_1drangecl.txt", "a");
fprintf(fp, "%g\n", time_spent);
fclose(fp);
```

Fig 10. *1DRange OpenCL Step 12, the output from the kernel is returned to the host side where it is stored in a file, also time is taken between Step 10 of sending the host data to the end of Step 12 of returning the host data.*

The Fig 9. globalWorkSize is given the elements variable size of length times width of the work space (1024 x 768) which is the same malloc size of the three host arrays (A, B, & C) in Fig 7. that will be used in the kernel argument in Fig 10. The globalWorkSize corresponds to the get_global_id(0) function in the Fig 8. which will increase its value each time it is called when a core is computing the data given in the kernel, once processed through the Mandelbrot set algorithm the final counter is given back to array C at the same global id as the other two arrays in Fig 8., the saved count data is written to a text file which is processed by an external OpenGL program that will generate the Mandelbrot fractal.

```
// Compute the size of the data
int elements = width*height;

// Allocate space for input/output data
size_t datasize_float_w = sizeof(float)*(width);
size_t datasize_float_h = sizeof(float)*(height);
size_t datasize_int = sizeof(int)*(elements);

// Acquire the limits for the x,y region to be calculated
printf("Please enter the min and max range value for x & y, respectively.\n");
scanf(" %i %i %i %i", &xmin, &xmax, &ymin, &ymax);
printf("Please wait while the process completes.\n");

// Calculate the steps for the variables
stepx = (float)(xmax - xmin) / width;
stepy = (float)(ymax - ymin) / height;

// Malloc array sizes
A = (float *)malloc(datasize_float_w);
B = (float *)malloc(datasize_float_h);
C = (int *)malloc(datasize_int);

//Initialise the input data
xvar = xmin;
yvar = ymax;
for(i = 0; i < width; i++){
    A[i] = xvar;
    xvar = xvar + stepx;
}
for(i = 0; i < height; i++){
    B[i] = yvar;
    yvar = yvar - stepy;
}
```

Fig 11. *2DRange host data setup snippet.*

```
const char* programSource =
"__kernel                                                       \n"
"void mandelbrot(__global float *A,                             \n"
"                __global float *B,                             \n"
"                __global int *C){                              \n"
"                                                               \n"
"    int ind;                                                   \n"
"    // obtain work-item's id                                   \n"
"    int idx = get_global_id(0);                                \n"
"    int idy = get_global_id(1);                                \n"
"    int2 size = (int2)(get_global_size(0), get_global_size(1)); \n"
"    ind = idx + size.x*idy;                                    \n"
"                                                               \n"
"    // transferring host data into                             \n"
"    // real(Re) +i imaginary(Im)                               \n"
"    float Re = A[idx];                                         \n"
"    float Im = B[idy];                                         \n"
"                                                               \n"
"    // internal real(ZRe) +i imaginary(ZIm)                    \n"
"    float ZRe = 0;                                             \n"
"    float ZIm = 0;                                             \n"
"    // internal temp real(TRe) +i imaginary(TIm)               \n"
"    float TRe = 0;                                             \n"
"    float TIm = 0;                                             \n"
"                                                               \n"
"    // counter                                                 \n"
"    int counter = 0;                                           \n"
"                                                               \n"
"    while(((ZRe*ZRe + ZIm*ZIm) <= 4) && (counter < 100)){      \n"
"        counter++;                                             \n"
"        // z = z^2 + c, where c = Re +i Im                     \n"
"        TRe = ((ZRe*ZRe) - (ZIm*ZIm)) + Re;                    \n"
"        TIm = (2*ZRe*ZIm) + Im;                                \n"
"                                                               \n"
"        ZRe = TRe;                                             \n"
"        ZIm = TIm;                                             \n"
"                                                               \n"
"    }                                                          \n"
"    C[ind] = counter;                                          \n"
"}                                                              \n"
;
```

Fig 12. *2DRange kernel snippet which utilises the global id of 0 and global id of 1.*

```
//---------------------------------------------------------
// STEP 9: Configure the work-item structure
//---------------------------------------------------------

size_t globalWorkSize[2] = {1024, 768};
size_t localItemSize[2] = {32, 16};
// There are 'elements' work-items
```

Fig 13. *2DRange OpenCL Step 9, globalWorkSize having 2D size of length and width as well as localItemSize which could be two factors of 512, NULL could be used in which the hardware will default a size.*

The approach is significantly different in the host data setup in Fig. 11. as the A and B array have been allocated separate datasizes as width and height corresponding to the width and height of the image instead of A and B array having a datasize of the variable elements that contain width multiplied by height, the information given to each array is the steps of the x and y variables. The reason for this is because in Fig 13. the globalWorkSize is given two dimensions instead of one, allowing the GPU to know there are 1024 dimension x values and 768 dimension y values which will be used obtaining the pixel positions using the get_global_id(0) and get_global_id(1) which are the x and y index respectively in Fig 12. The kernel in Fig 12. has maintained the same algorithm as the previous 1DRange however the x and y indexes from the global id allows to access all values of A array before iterating B array by one which resets the A array allowing for reduced memory usage from two out of the three array that will be allocated the memory. The exact same output method from the 1DRange of obtaining the data and saving to a text file which will be used in the external OpenGL program to generate the image, both outputs of the generated image yielded the same image shown in Fig 1.

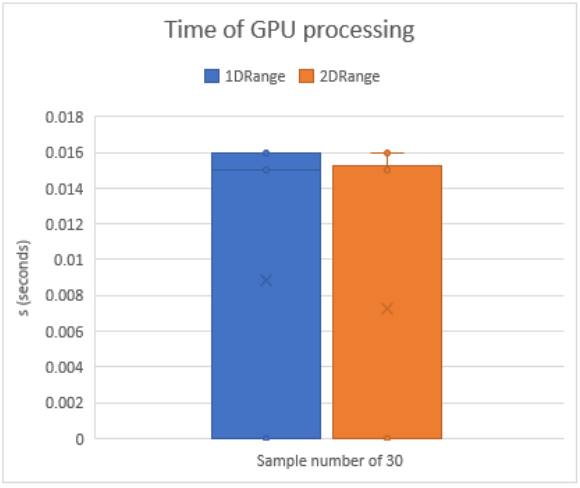

Fig 14. *The results of the NDRange box and whisker plot showing the time taken to process over 30 samples.*

The results of the NDRange via GPU show that the 1DRange took an average time of 0.008833 seconds and 2DRange took an average time of 0.007233 seconds, the results show that 2DRange could have the potential to be faster than 1DRange due to the 1DRange requiring the setting-up and sending of larger memory spaces however the integrated GPU on the CPU has shown little difference in terms of time, significant time difference would result on an older computer however due to testing benchmark performing on this computer the results yield inconclusive when assessing the difference in NDRange and conclusive that GPU processing is significantly faster than CPU processing as the Baseline program took approximately 1037 times faster than the NDRange programs.

VI. Varying Raster and Block Size via GPU

The set-up of the OpenCL for the varying raster and varying block sizes is different to the set-up of the OpenCL for the NDRange programs:

Generalised steps for setting up OpenCL for Raster/Block.

1. Discover and initialise the platforms.
2. Discover and initialise the devices.
3. Create a context.
4. Create a command queue.
5. Create device buffers.
6. Create and compile the program.
7. Create the kernel.
8. Set the kernel arguments.
9. Configure the work-item structure.
10. For-loop (Nested for-loop in Block program)
    a. Write host data to device buffers.
    b. Enqueue the kernel for execution.
    c. Read the output buffer back to the host.
11. Release OpenCL resources.
12. Release host resources.

The timing difference is still the same approach according to NDRange however the set-up for the Raster/Block include a for-loop/nested for-loop instead. The kernel that is used for the Varying Raster program is the 1DRange kernel and the kernel used for the Varying Block program is the 2DRange kernel. Whereas there are setting up of the host data and the preparation for the output is significantly different than the NDRange programs.

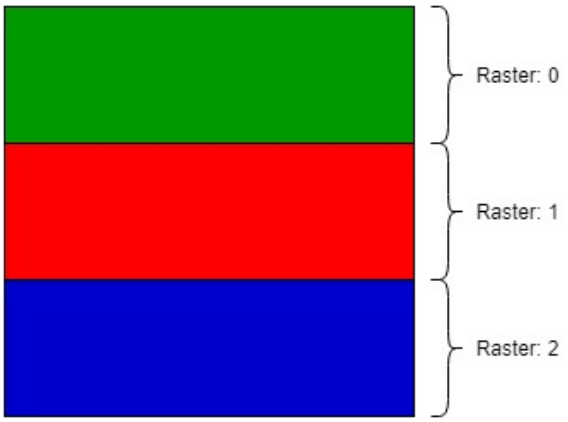

Fig 15. Visual representation of the Raster

```
// re-write A and B
xvar = xmin;
if(j==0){
    yvar = ymax;
}else{
    rasterstep = (float)j*rasterst;
    yvar = ymax - rasterstep;
}
for(i = 0; i < (elements); i++){
        A[i] = xvar;
        B[i] = yvar;
        xvar = xvar + stepx;
        if(xvar >= xmax){
            xvar = xmin;
            yvar = yvar - stepy;
        }
}
```

Fig 16. Snippet of the host data for varying raster.

```
//------------------------------------------------------
// STEP 3: Read the output buffer back to the host
//------------------------------------------------------

// Use clEnqueueReadBuffer() to read the OpenCL output
// buffer (bufferC) to the host output array (C)
clEnqueueReadBuffer(
    cmdQueue,
    bufferC,
    CL_TRUE,
    0,
    datasize_int,
    C,
    0,
    NULL,
    NULL);

// Verify the output

fp = fopen("Rresults.txt", "a");

ind = 0;
if( j != 0 ){
    fprintf(fp, "%c", newline);
}
for(i = 0; i < elements; i++){
    //printf("%i ", C[i]);
    fprintf(fp, "%i ", C[i]);
    ind++;
    if(ind > 1024){
        fprintf(fp, "%c", newline);
        ind = 0;
    }
}

fclose(fp);
```

Fig 17. Snippet for the Output for the varying raster.

# of R	Time	# of R	Time
2	0.016	32	0.061023
4	0.016	64	0.116279
8	0.030512	128	0.224558
16	0.033953	256	0.355814

Fig 18. Tabled results of the Raster division and time.

The raster program takes of similar structure of the 1DRange however the raster variable divides the elements that are held allocated to the buffers that are created like in 1DRange, the data is calculated in a for-loop that will iterate for the number of rasters that is submitted by the user, using two equations allow the tracking position of the y-variable for the new starting position at each raster position, below is the equations that utilise this.

$$raster\ index = \frac{x_{max} - x_{min}}{\#\ of\ rasters}$$

$$yvar = y_{max} + i_{forloop} * raster\ index$$

The raster cut through the vertical creating horizontal strips which are calculated, this is because the Mandelbrot Fractal generator that creates the image reads in the data via each horizontal line. The results in Fig 17. shows that time increases when the number of rasters increase, due to the tug-of-war explained earlier in the report, the memory required is decreasing however the overheads of enqueuing the buffers and kernel for execution as well as reading the output has to be performed with each interation resulting in more time thus utilisation of less resources results in longer processing time.

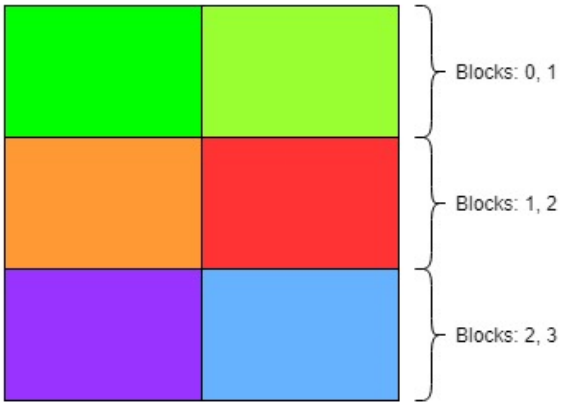

Fig 19. Snippet of the host data for varying block.

```
stepx = (float)(xmax - xmin) / width;
stepy = (float)(ymax - ymin) / height;
new_width = width / raster;
new_height = height / block;

elements = width*height / raster*block;
datasize_height = sizeof(float)*(new_height);
datasize_width = sizeof(float)*(new_width);
datasize_int = sizeof(int)*(elements);

float** A = (float **)malloc(raster*sizeof(float*));
float** B = (float **)malloc(block*sizeof(float*));
int*** D = (int ***)malloc(raster*sizeof(int**));
for(i = 0; i < raster; i++){
    D[i] = (int **)malloc(block*sizeof(int*));
    for(j = 0; j < block; j++){
        D[i][j] = (int *)malloc(elements*sizeof(int));
    }
}

//printf("Alloc data\n");
xvar = xmin;
for(i = 0; i < raster; i++){
    A[i] = (float *)malloc(datasize_width);
    for(j = 0; j < new_width; j++){
        A[i][j] = xvar;
        xvar = xvar + stepx;
        //printf("%f ", A[i][j]);
    }
}
//printf("%c", newline);
yvar = ymax;
for(i = 0; i < block; i++){
    B[i] = (float *)malloc(datasize_height);
    for(j = 0; j < new_height; j++){
        B[i][j] = yvar;
        yvar = yvar - stepy;
        //printf("%f ", B[i][j]);
    }
}
```

Fig 20. Snippet of the host data for varying block.

```
//------------------------------------------------------
// STEP 9: Configure the work-item structure
//------------------------------------------------------

// Define an index space (global work size) of work
// items for execution.
// A workgroup size (local work size) is not required,
// but can be used.
size_t globalWorkSize[2] = {new_width, new_height};
// There are 'elements' work-items
```

Fig 21. Snippet of the globalWorkSizes.

```
//------------------------------------------------------
// STEP 3: Read the output buffer back to the host
//------------------------------------------------------

// Use clEnqueueReadBuffer() to read the OpenCL output
// buffer (bufferC) to the host output array (C)
clEnqueueReadBuffer(
    cmdQueue,
    bufferC,
    CL_TRUE,
    0,
    datasize_int,
    C,
    0,
    NULL,
    NULL);

// Verify the output
//printf("\n");
for(k = 0; k < elements; k++){
    D[i][j][k] = C[k];
    //printf("%i ", D[i][j][k]);
}
```

Fig 22. Snippet for the Output for the varying block.

```
after = clock();
time_spent = (double)(after - before) / CLOCKS_PER_SEC;

fp = fopen("time_Blocks.txt", "a");
fprintf(fp, "%g%c", time_spent, newline);
fclose(fp);

fp = fopen("Bresults.txt", "a");
for(i = 0; i < block; i++){
    for(k = 0; k < new_height; k++){
        for(j = 0; j < raster; j++){
            for(l = (k*new_width); l < (new_width + k*new_width); l++){
                fprintf(fp, "%i%c", D[j][i][l], newline);
            }
        }
    }
}
fclose(fp);
```

Fig 23. Reorganising the data for Mandelbrot Fractal generator.

Dim	Time	Dim	Time
2x2	0.016	8x8	0.292
4x3	0.047	16x12	0.635
4x4	0.068	16x16	1.135
8x6	0.166	32x24	2.557

Fig 24. Tabled results of the Block division and time.

The block program's setting up of the host data is exactly like the 2DRange datasizes however the array for A and B have been divided into arrays that will signify how much the user wants to slice the horizontal and vertical image to create blocks. Another array has been allocated as a 3D array in which the results of each block will be passed from the output of the C buffer to the 3D array where it will be processed later in Fig 24 and sent to a text file to be used to generate the Mandelbrot Fractal image. Fig 25. shows that the time taken to process is larger as the dimensions of the division of the horizontal and vertical plane increase due to the same reason as the Raster program, the overheads required to send, process and read the data are iterated more times than a raster due to a raster being split into blocks. The block function timing at 2x2 may be smaller than the raster function's 2 raster however the Block Program requires a 3D array to process the block's output correctly to a text file resulting in more memory required to run than that of the Raster Program as seen that higher raster yields less memory needed, more time to run however significantly less time than 32x24 of the Block Program. The output files for both the Raster and Block program at varying raster and block size both yield the same fractal image shown in Fig 1.

VII. Conclusion

Processing heavy computation algorithms via the GPU requires a fine balance of managing time and memory, as well as how much time delayed due to intialisation, sending and receiving via the overheads of the OpenCL functions. Through the use of a Baseline program, GPU processing of different programs such as NDRange, rasterising the image or cutting the image into blocks show significant time saving however when comparing NDRange, Raster and Block programs, the conclusive use of one of these methods approach a different outlook on why a program would be better than another. Smaller memory systems would want to utilise the Raster method to save on space that the GPU but would be penalised on the time taken to process, heavy memory GPU would want to utilise as much memory as possible to cut down on the processing time, and in-between GPU would use any of the above.

VIII. Reference

[1] BR. Gaster, L. Howes, D. Kaeli, P. Mistry and D. Schaa, "Heterogenous computing with OpenCL", 1st Ed, Waltham, MA: Morgan Kaufmann, 2012.

[2] D. Rowlands, "GPGPU Programming – GPGPU Architecture", 6305ENG: Advanced Computer Systems, Griffith Unv., Nathan, QLD, Australia, 2018.

[3] D. Rowlands, "GPGPU Programming – OpenCL" 6305ENG: Advanced Computer Systems, Griffith Unv., Nathan, QLD, Australia, 2018.