Contents

OpenCL exercise

Fotis Bistas
AM: 3190135
uni_email: p3190135@aueb.gr
personal_email: fotis.bistas@gmail.com

May 2023

1 Introduction

The second set of exercises in this course focuses on parallel programming using OpenCL. Specifically, I will be parallelizing a program that performs a Gaussian blur on an image.

1.1 Setup

Metrics were conducted on the following system:

Computer	Processor	Cores	Compiler
DESKTOP-SIL3NU4	AMD Ryzen 5 3600	6	gcc 6.3.0

Instead of running them on WSL like the first exercise I run them locally on windows for **OpenCL** to be able to work properly. That means different result for performance metrics are expected. Also my PCIe is gen3 which supports speeds up to 8GT/s (roughly translates to 1GB/s per PCIe lane).

1.2 Modifications

In order to make metric gathering more easy I added the extra command line argument threads:
gaussian-blur.exe radius filename threads
Which specifies how many threads are going to be used in the openMP functions.

Additionally concerning openCL I added these extra functions:

```c
/* Helper function to read kernel from file*/
size_t read_kernel_from_file(char** source_str, char* filename) {
    FILE* fp = fopen(filename, "rb");
    if (!fp) {
    fprintf(stderr, "Failed to open kernel file: %s\n", filename);
        return 0;
    }

    fseek(fp, 0, SEEK_END);
    size_t program_size = ftell(fp);
    rewind(fp);

    *source_str = (char*)malloc(program_size + 1);
    if (!(*source_str)) {
        fprintf(stderr, "Failed to allocate memory for kernel\n");
        fclose(fp);
        return 0;
    }

    size_t read_size = fread(*source_str, sizeof(char),
        program_size, fp);
    if (read_size != program_size) {
        fprintf(stderr, "Failed to read kernel from file: %s\n",
            filename);
        free(*source_str);
        fclose(fp);
        return 0;
    }

    (*source_str)[program_size] = '\0';

    fclose(fp);
    return program_size;
}
```

Listing 1: Reads the kernel code exectued by OpenCL from a file

```
 /*Release open cl memory after program execution or an error
     occurs*/
void release_cl_memory(cl_kernel* kernel,
cl_program* program,
cl_context* context,
cl_command_queue* command_queue)
{
    if(kernel){
        printf("Releasing kernel\n");
        clReleaseKernel(*kernel);
    }
    if(program){
        printf("Releasing program\n");
        clReleaseProgram(*program);
    }
    if(command_queue){
        printf("Releasing command_queue\n");
        clReleaseCommandQueue(*command_queue);
    }
    if(context){
        printf("Releasing context\n");
        clReleaseContext(*context);
    }
}
```

Listing 2: Frees openCL objects

Furthermore I modified **bmp_write_data_to_file** to write the files to the producedImages folder:

```
static
void bmp_write_data_to_file(char *fname, img_t *img)
{
        FILE *file;
        bmpheader_t *bmph = &(img->header);
        //custom modification for image folder
        //all images created from the algorithm are written to the
            folder
        char* name_with_folder = malloc(strlen(fname) + strlen(
            IMG_FOLDER) + 1);
        strcpy(name_with_folder, IMG_FOLDER);
        strcat(name_with_folder, fname);
        //changed fname to name with folder
        file = fopen(name_with_folder, "wb");
        fwrite(bmph, sizeof(bmpheader_t)+1, 1, file);
        fseek(file, bmph->data, SEEK_SET);
        fwrite(img->imgdata, bmph->arraywidth, 1, file);
        fclose(file);
        free(name_with_folder);
}
```

Listing 3: Saves results from the imgout to a .bmp file inside the producedImages folder

Plus this function which prints the error message based on the code received by openCL:

```
const char *getErrorString(cl_int error)
{
switch(error){
    // run-time and JIT compiler errors
    case 0: return "CL_SUCCESS";
    case -1: return "CL_DEVICE_NOT_FOUND";
    case -2: return "CL_DEVICE_NOT_AVAILABLE";
    case -3: return "CL_COMPILER_NOT_AVAILABLE";
    case -4: return "CL_MEM_OBJECT_ALLOCATION_FAILURE";
    case -5: return "CL_OUT_OF_RESOURCES";
    case -6: return "CL_OUT_OF_HOST_MEMORY";
    case -7: return "CL_PROFILING_INFO_NOT_AVAILABLE";
    case -8: return "CL_MEM_COPY_OVERLAP";
    case -9: return "CL_IMAGE_FORMAT_MISMATCH";
    case -10: return "CL_IMAGE_FORMAT_NOT_SUPPORTED";
    case -11: return "CL_BUILD_PROGRAM_FAILURE";
    case -12: return "CL_MAP_FAILURE";
    case -13: return "CL_MISALIGNED_SUB_BUFFER_OFFSET";
    case -14: return "CL_EXEC_STATUS_ERROR_FOR_EVENTS_IN_WAIT_LIST"
        ;
    case -15: return "CL_COMPILE_PROGRAM_FAILURE";
    case -16: return "CL_LINKER_NOT_AVAILABLE";
    case -17: return "CL_LINK_PROGRAM_FAILURE";
    case -18: return "CL_DEVICE_PARTITION_FAILED";
    case -19: return "CL_KERNEL_ARG_INFO_NOT_AVAILABLE";

    // compile-time errors
    case -30: return "CL_INVALID_VALUE";
    case -31: return "CL_INVALID_DEVICE_TYPE";
    case -32: return "CL_INVALID_PLATFORM";
    case -33: return "CL_INVALID_DEVICE";
    case -34: return "CL_INVALID_CONTEXT";
    case -35: return "CL_INVALID_QUEUE_PROPERTIES";
    case -36: return "CL_INVALID_COMMAND_QUEUE";
    case -37: return "CL_INVALID_HOST_PTR";
    case -38: return "CL_INVALID_MEM_OBJECT";
    case -39: return "CL_INVALID_IMAGE_FORMAT_DESCRIPTOR";
    case -40: return "CL_INVALID_IMAGE_SIZE";
    case -41: return "CL_INVALID_SAMPLER";
    case -42: return "CL_INVALID_BINARY";
    case -43: return "CL_INVALID_BUILD_OPTIONS";
    case -44: return "CL_INVALID_PROGRAM";
    case -45: return "CL_INVALID_PROGRAM_EXECUTABLE";
    case -46: return "CL_INVALID_KERNEL_NAME";
    case -47: return "CL_INVALID_KERNEL_DEFINITION";
    case -48: return "CL_INVALID_KERNEL";
    case -49: return "CL_INVALID_ARG_INDEX";
    case -50: return "CL_INVALID_ARG_VALUE";
    case -51: return "CL_INVALID_ARG_SIZE";
    case -52: return "CL_INVALID_KERNEL_ARGS";
    case -53: return "CL_INVALID_WORK_DIMENSION";
    case -54: return "CL_INVALID_WORK_GROUP_SIZE";
    case -55: return "CL_INVALID_WORK_ITEM_SIZE";
    case -56: return "CL_INVALID_GLOBAL_OFFSET";
    case -57: return "CL_INVALID_EVENT_WAIT_LIST";
    case -58: return "CL_INVALID_EVENT";
    case -59: return "CL_INVALID_OPERATION";
```

```
    case -60: return "CL_INVALID_GL_OBJECT";
    case -61: return "CL_INVALID_BUFFER_SIZE";
    case -62: return "CL_INVALID_MIP_LEVEL";
    case -63: return "CL_INVALID_GLOBAL_WORK_SIZE";
    case -64: return "CL_INVALID_PROPERTY";
    case -65: return "CL_INVALID_IMAGE_DESCRIPTOR";
    case -66: return "CL_INVALID_COMPILER_OPTIONS";
    case -67: return "CL_INVALID_LINKER_OPTIONS";
    case -68: return "CL_INVALID_DEVICE_PARTITION_COUNT";

    // extension errors
    case -1000: return "CL_INVALID_GL_SHAREGROUP_REFERENCE_KHR";
    case -1001: return "CL_PLATFORM_NOT_FOUND_KHR";
    case -1002: return "CL_INVALID_D3D10_DEVICE_KHR";
    case -1003: return "CL_INVALID_D3D10_RESOURCE_KHR";
    case -1004: return "CL_D3D10_RESOURCE_ALREADY_ACQUIRED_KHR";
    case -1005: return "CL_D3D10_RESOURCE_NOT_ACQUIRED_KHR";
    default: return "Unknown OpenCL error";
    }
}
```

Listing 4: Getting error string from openCL error code

2 Gaussian blur

2.1 The problem

The program applies Gaussian blur filtering to a BMP image using OpenMP for parallel processing. It reads in the BMP file and separates the RGB channels using utility functions. The Gaussian blur filter is applied using either a serial or parallel function depending on the mode of operation. The output image is created by cloning an empty image structure and calculating each pixel using a weighted average of its surrounding pixels based on the Gaussian kernel. Finally, the RGB channels are combined and written to a BMP file.

2.2 OpenCL implementation

2.2.1 System info

To determine appropriate global work sizes, local work group sizes etc. I runned **cl_info** in my windows cmd:

```
Number of platforms:                                1
Number of devices:                                  1
Device Type:                                        CL_DEVICE_TYPE_GPU
Max compute units:                                  10
Max work items dimensions:                          3
Max work items[0]:                                  1024
Max work items[1]:                                  1024
Max work items[2]:                                  1024
Max work group size:                                256
Minimum alignment (bytes) for any datatype:         128
Kernel Preferred work group size multiple:          64
```

Listing 5: clinfo command line output

Out of the many these are the most important. The clinfo tells us the **OpenCL** recognizes only the GPU. The GPU has 10 maximum compute units (which house work items and groups). It also tells us that the global work items in each of the three dimensions is 1024.

The max work group size is 256. That means I can't set the work group to have more than 256 work items inside it: 16X16,1X256 are correct but 16X17,2X256 are incorrect. The work group size preferably should be set to a multiple of 64, which is unfortunate for us since we can't divide the square images to ideal work groups.

Minimum alignment for bytes tells us that in each memory transaction 128 bytes are accessed.

In turn this is the code that runs the kernels:

```
// Spawn threads
const size_t globalWorkSize[2] = {imgin->header.width, imgin->
    header.height};

//modify local size
//preffered multiple of local work group 64
//max work group size is 256
const size_t localWorkSize[2] = {2,128};
cl_event kernelEvent;
err = clEnqueueNDRangeKernel(commandQueue, kernel, 2, NULL,
    globalWorkSize, localWorkSize, 0, NULL, &kernelEvent);
if (err != CL_SUCCESS) {
    fprintf(stderr, "Failed to enqueue OpenCL kernel: %s\n",
        getErrorString(err));
    goto FAIL;
}
```

Listing 6: Code that runs the kernels inside the GPU

The code essentially tries to spawn as many threads as there are pixels in the image grouping the threads into groups of 256.

2.2.2 Work group size

Different work group sizes produce different results:

work group size	first run	second run	third run	fourth run	average
16X16	0.642810	0.640408	0.641021	0.638460	0.64067475
1X256	0.657297	0.651275	0.646660	0.645799	0.65025775
10X10	0.833264	0.825065	0.831456	0.830828	0.83015325
5X10	0.721304	0.717784	0.715129	0.714942	0.71728975
2X128	0.640727	0.629585	0.641869	0.630160	0.63558525
3X64	0.655406	0.655158	0.648398	0.649557	0.65212975
2X64	0.642746	0.642406	0.637836	0.637447	0.64010875

It seems performance is maximized when: the work group size is a multiple of 64(as **clinfo** mentioned), the work group size uses the maximum available work group size and the rows are a small number. The differences are minuscule but it gives us a good lead to continue from.

2X128 work group size seems to work best for a couple of reasons. When threads try to execute kernel codes it is essential to minimize memory transactions. A work group that accesses memory locations that are neighbouring to each other will make 1 memory transaction. My guess here is that, since data is accessed this way $output\_red[i * width + j]$ which is essentially $output\_red[get\_global\_id(0) * get\_global\_size(0) + get\_global\_id(1)]$ the small row number 2, which in turn reduces non contiguous memory transactions.

On top of that since the preferred work group size multiple is 64 that probably means that column number 128 doesn't leave any threads to idle. It also uses the maximum size of the work group, probably dividing the work better.

We would expect some undefined behavior when the local work group size is not a multiple of the image size. This doesn't happen in my case so continuing with the preferred work group size multiple and the max work group size is okay.

2.2.3 Kernels

2.2.3.1 Main kernel

The main kernel implementation I found was this:

```
__kernel void gaussian_blur(__private int radius,
__constant unsigned char* input_red,
__constant unsigned char* input_green,
__constant unsigned char* input_blue,
__global unsigned char* output_red,
__global unsigned char* output_green,
__global unsigned char* output_blue)
{

   int i = get_global_id(0);
   int j = get_global_id(1);

   int width = get_global_size(0);
   int height = get_global_size(1);

   double row,col;
        double weightSum = 0.0, redSum = 0.0, greenSum = 0.0,
            blueSum = 0.0;

   for (row = i-radius; row <= i + radius; row++)
        {
      for (col = j-radius; col <= j + radius; col++)
      {
         int x = clamp((int) col, 0, (int) width-1);
         int y = clamp((int) row, 0, (int) height-1);
         int tempPos = y * width + x;
         double square = (col-j)*(col-j)+(row-i)*(row-i);
         double sigma = radius*radius;
         double weight = exp(-square / (2*sigma)) / (3.14*2*sigma);

         redSum += input_red[tempPos] * weight;
         greenSum += input_green[tempPos] * weight;
         blueSum += input_blue[tempPos] * weight;
         weightSum += weight;
      }
   }
   output_red[i*width+j] = round(redSum/weightSum);
   output_green[i*width+j] = round(greenSum/weightSum);
   output_blue[i*width+j] = round(blueSum/weightSum);

}
```

Listing 7: Main kernel my GPU executes

Where the kernel is called setting the global size to: **const size_t globalSize[2] = image_width,image_height**. This tries to spawn 1 thread for each pixel (spawns less than that).

2.2.3.2 Float kernel

First improvement that I thought of of this initial kernel was replacing the double values with floats. This is expected to give way better time results, since single precision is much faster than double precision in GPUs:

```
__kernel void gaussian_blur(__private int radius,
__constant unsigned char* input_red,
__constant unsigned char* input_green,
__constant unsigned char* input_blue,
__global unsigned char* output_red,
__global unsigned char* output_green,
__global unsigned char* output_blue)
{

   int i = get_global_id(0);
   int j = get_global_id(1);

   int width = get_global_size(0);
   int height = get_global_size(1);

   float row,col;
    float weightSum = 0.0, redSum = 0.0, greenSum = 0.0,
        blueSum = 0.0;

   for (row = i-radius; row <= i + radius; row++)
    {
      for (col = j-radius; col <= j + radius; col++)
      {
         int x = clamp((int) col, 0, (int) width-1);
         int y = clamp((int) row, 0, (int) height-1);
         int tempPos = y * width + x;
         float square = (col-j)*(col-j)+(row-i)*(row-i);
         float sigma = radius*radius;
         float weight = exp(-square / (2*sigma)) / (3.14*2*
             sigma);

         redSum += input_red[tempPos] * weight;
         greenSum += input_green[tempPos] * weight;
         blueSum += input_blue[tempPos] * weight;
         weightSum += weight;
      }
   }
   output_red[i*width+j] = round(redSum/weightSum);
   output_green[i*width+j] = round(greenSum/weightSum);
   output_blue[i*width+j] = round(blueSum/weightSum);

}
```

Listing 8: Main kernel my GPU executes with floats instead of doubles

This indeed produces way better time results:

precision	first run	second run	third run	fourth run	average
single	0.193269	0.195121	0.194768	0.194625	0.19444575
double	0.633313	0.637162	0.637124	0.637475	0.6362685

But loses in accuracy. This can be checked by running an image comparator online between the serial implementation and the OpenCL implementation:

Method	Absolute Error Count
All	114
Blue	42
Green	42
Red	50
Percentage Difference	0.01%

Figure 1: Difference between serial and float output.

Method	Absolute Error Count
All	0
Blue	0
Green	0
Red	0
Percentage Difference	0.00%

Figure 2: Difference between serial and double output.

We can see that double achieves better accuracy than float. The difference in accuracy is tiny, for the performance benefit, so we are going to stick with the float implementation.

2.2.3.3 Local memory kernel

Another thing that I tried is to use the GPU's local memory.

```
__kernel void gaussian_blur(__private int radius,
__constant unsigned char* input_red,
__constant unsigned char* input_green,
__constant unsigned char* input_blue,
__global unsigned char* output_red,
__global unsigned char* output_green,
__global unsigned char* output_blue,
__local unsigned char* local_red,
__local unsigned char* local_green,
__local unsigned char* local_blue)
{

int i = get_global_id(0);
int j = get_global_id(1);

//get global and local dimensions
int width = get_global_size(0);
int height = get_global_size(1);
int localSizeX = get_local_size(0);
int localSizeY = get_local_size(1);


// Load data from global memory to local memory
int localX = get_local_id(0);
int localY = get_local_id(1);
//this is the linear index that correspond to array local_color (
    because it is one dimensional)
//e.g. work group size 16X16 if localY=15 (last one) and localX=0
    localPos = 15 * 16 + 0 = 240
int localPos = localY * localSizeX + localX;
//same idea here but works for global. Inside the brackets is the
    linear global position
local_red[localPos] = input_red[j * width + i];
local_green[localPos] = input_green[j * width + i];
local_blue[localPos] = input_blue[j * width + i];

// Synchronize to ensure all data is loaded into local memory
barrier(CLK_LOCAL_MEM_FENCE);

float row,col;
float weightSum = 0.0, redSum = 0.0, greenSum = 0.0, blueSum = 0.0;

for (int row = localX - radius; row <= localX + radius; row++)
{
    for (int col = localY - radius; col <= localY + radius; col++)
    {
        int x = clamp(col, 0, localSizeX - 1);
        int y = clamp(row, 0, localSizeY - 1);
        int tempPos = y * localSizeX + x;
        float square = (col - localY) * (col - localY) + (row -
            localX) * (row - localX);
        float sigma = radius * radius;
        float weight = exp(-square / (2 * sigma)) / (3.14 * 2 *
            sigma);
```

```
            redSum += local_red[tempPos] * weight;
            greenSum += local_green[tempPos] * weight;
            blueSum += local_blue[tempPos] * weight;
            weightSum += weight;
        }
}

// Wait for all work-items to finish accessing local memory
barrier(CLK_LOCAL_MEM_FENCE);

// Store the result to the output global memory
output_red[j * width + i] = round(redSum / weightSum);
output_green[j * width + i] = round(greenSum / weightSum);
output_blue[j * width + i] = round(blueSum / weightSum);
```

Listing 9: Main kernel my GPU executes utilizing local memory

The OpenCL initialization code must be changed accordingly:

```
//create the local memory
err = clSetKernelArg(kernel, 7, sizeof(unsigned char) * (
    localWorkSize[0] * localWorkSize[1]), NULL);
if (err != CL_SUCCESS) {
    fprintf(stderr, "Failed to set kernel argument: %s\n",
        getErrorString(err));
goto FAIL;
}
err = clSetKernelArg(kernel, 8, sizeof(unsigned char) * (
    localWorkSize[0] * localWorkSize[1]), NULL);
if (err != CL_SUCCESS) {
    fprintf(stderr, "Failed to set kernel argument: %s\n",
        getErrorString(err));
goto FAIL;
}
err = clSetKernelArg(kernel, 9, sizeof(unsigned char) * (
    localWorkSize[0] * localWorkSize[1]), NULL);
if (err != CL_SUCCESS) {
    fprintf(stderr, "Failed to set kernel arguments: %s\n",
        getErrorString(err));
goto FAIL;
}
```

Listing 10: Changes in the openCL initialization code

This doesn't produce the correct image.I don't think its possible to do this without changing the algorithm, since the variables row,col,square depend on the global i,j. I tried a lot of different ways to map the local variables to the global ones and vice versa.Even if there is a way to do this correctly the speed up seems to be not that significant:

implementation	first run	second run	third run	fourth run	average
local memory	0.190829	0.190650	0.190587	0.190558	0.190656
simple floats	0.193802	0.193809	0.191883	0.193227	0.19318025

Maybe if the image sizes were larger there would be a bigger performance benefit.

2.2.3.4 Math optimization and vectorization

Next up I thought about using openCL vectors and slightly modifying the code:

```
__kernel void gaussian_blur(__private int radius,
__constant unsigned char* input_red,
__constant unsigned char* input_green,
__constant unsigned char* input_blue,
__global unsigned char* output_red,
__global unsigned char* output_green,
__global unsigned char* output_blue)
{

    const int2 global_positions = {get_global_id(0),get_global_id
        (1)};
    const int2 global_sizes = {get_global_size(0),get_global_size
        (1)};

    float row,col;
    float4 sums = {0.0, 0.0, 0.0, 0.0};

    float sigma = radius*radius;
    float expression = 1 / (3.14*2*sigma);
    for (row=global_positions[0]-radius; row <=global_positions[0]
        + radius; row++)
    {
        for (col=global_positions[1]-radius; col <=global_positions
            [1]+ radius; col++)
        {
            int x = clamp((int) col, 0, (int)global_sizes[0]-1);
            int y = clamp((int) row, 0, (int)global_sizes[1]-1);
            int tempPos = y *global_sizes[0]+ x;
            float square = (col-global_positions[1])*(col-
                global_positions[1])+(row-global_positions[0])*(row
                -global_positions[0]);
            float weight = exp(-square / (2*sigma)) * expression;

            sums[0] += input_red[tempPos] * weight;
            sums[1] += input_green[tempPos] * weight;
            sums[2] += input_blue[tempPos] * weight;
            sums[3] += weight;
        }
    }

    output_red[global_positions[0]* global_sizes[0]+
        global_positions[1]] = round(sums[0]/sums[3]);
    output_green[global_positions[0]* global_sizes[0]+
        global_positions[1]] = round(sums[1]/sums[3]);
    output_blue[global_positions[0]* global_sizes[0]+
        global_positions[1]] = round(sums[2]/sums[3]);
}
```

Listing 11: Math and vectors modifications

The difference between the simple float implementation 8 and this implementation is that I moved **float sigma = radius*radius** and **float expression = 1 / (3.14*2*sigma)** outside of the inner loop, reducing the computational load

and tried to use openCL vectors wherever I can. This increases performance:

implementation	first run	second run	third run	fourth run	average
simple float	0.193373	0.192085	0.191525	0.192057	0.19226
math and vector	0.044658	0.044834	0.044907	0.044819	0.0448045

and produces accurate results by comparing it with the image from the serial implementation:

Method	Absolute Error Count
All	115
Blue	42
Green	43
Red	50
Percentage Difference	0.01%

Figure 3: Comparison between the serial output and the math and vector improvement output.

3 Results

The program is compiled using the following make file:

```
IDIR="C:/Program Files (x86)/OCL_SDK_Light/include/"
CC=gcc
CFLAGS=-I $(IDIR) -L $(LDIR)

ODIR="./bin/"
LDIR="C:/Program Files (x86)/OCL_SDK_Light/lib/x86"
IMAGEDIR = "./producedImages/"

LIBS=-lm -fopenmp -lOpenCL


array-addition: array-addition.c
	$(CC) -o $(ODIR)$@ $^ $(CFLAGS) -lOpenCL

gaussian-blur: gaussian-blur.c
	$(CC) -o $(ODIR)$@ $^ $(CFLAGS)  $(LIBS)


clear-images:
	rm -f $(IMAGEDIR)*.bmp  *~ core $(INCDIR)/*~

.PHONY: clean

clean:
	rm -f $(ODIR)* *~ core $(INCDIR)/*~
```

Listing 12: Make file for compilation

In order to compile the program I run:
make gaussian-blur.
This essentially runs:
gcc -o ./bin/gaussian-blur -I IDIR -L LDIR -lm -fopenmp -lOpenCL.
The program can be run:
./bin/gaussian-blur radius filename threads

In which radius filename and threads are command line arguments. The radius can be any positive integer and the available filenames are 500.bmp, 1000.bmp, 1500.bmp. The CPU threads I choose to spawn here for the OpenMP implementation are from 1 to 15. Here I document the program's time efficiency on a radius of 8 and the 1500.bmp file.

3.1 Performance OpenCL(GPU)

Serially the algorithm performs like the following: The serial output looks like this:

serial	first run	second run	third run	fourth run	average
	108.498990	109.261769	110.026139	109.491663	109.31964025

The OpenCL output looks like this:

OpenCL	first run	second run	third run	fourth run	average
	0.044924	0.044626	0.045300	0.045078	0.044982

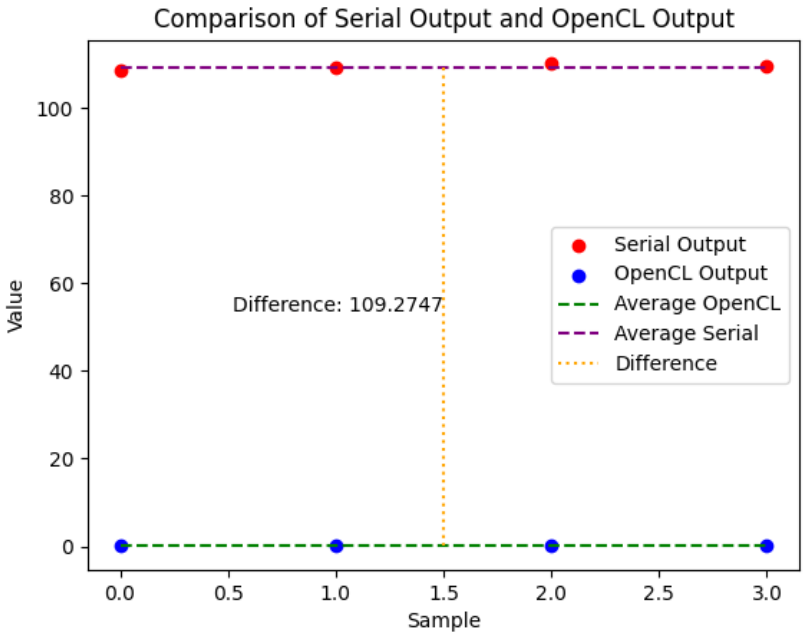

Figure 4: Comparison between the execution times between the **OpenCL** implementation and the serial implementation.

We can observe that, compared to the serial execution times, **openCL** achieves a staggering performance boost. In fact the average serial execution is 2500 times slower than openCL. Since the serial code is executed on a single thread, while the openCL code is executed on thousands of the threads this performance difference is to be expected. Still the serial execution provides a valuable baseline of evaluating the performance of the parallel code.

3.2 Performance comparison OpenCL and OpenMP

Comparing the **OpenCL**(GPU parallelization) implementation with the **OpenMP** (CPU parallelization) implementation we can observe some major differences in performance. In order to make this a fair comparison I'll try to spawn as many OpenMP threads as possible[1]:

[1]The execution times stay around the same even if the improvements of math and vector implementation are applied.

Using OpenMP parallel loops:

threads	first run	second run	third run	fourth run	average
1	109.230622	108.760640	108.462651	108.638583	108.773124
2	54.212336	54.230278	54.899626	54.255657	54.39947425
3	36.740091	37.904157	36.865178	36.780392	37.0724545
4	27.683229	27.817559	27.907561	28.017662	27.85650275
6	21.061432	21.020122	20.988421	20.996931	21.0167265
8	15.751861	15.612627	15.686557	15.585640	15.65917125
10	12.513170	13.141648	12.480030	12.567052	12.675475
12	11.207825	11.774919	10.460732	10.460732	10.976052
14	10.589234	10.339203	10.407211	10.333202	10.4172125
15	10.409533	10.393211	10.663243	10.395263	10.4653125

Using OpenMP tasks:

threads	first run	second run	third run	fourth run	average
1	111.077615	110.271967	110.063338	111.019538	110.6081145
2	55.565283	55.653276	56.527555	55.471603	55.80442925
3	39.355126	39.097131	38.037156	37.903158	38.59814275
4	28.917542	29.009543	28.961544	29.050541	28.9847925
6	21.006496	21.407496	21.022448	19.372615	20.70226375
8	15.101260	14.848240	14.827079	14.704085	14.870166
10	12.191069	12.880073	12.054070	12.019068	12.28607
12	10.584061	10.479061	10.481059	11.165571	10.677438
14	12.707056	12.140051	11.686049	12.186050	12.1798015
15	11.885049	11.827049	11.982563	11.675046	11.84242675

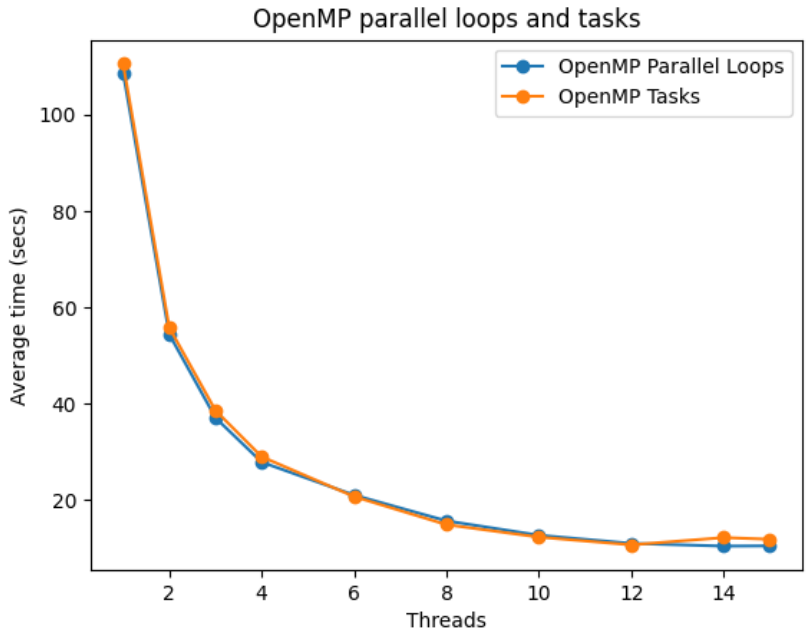

Figure 5: OpenMP loops and tasks execution time as threads increase.

The execution time of loops and tasks stagnates (and would probably degrade with more threads) after some point. The graph seems to be asymptotic to 10 seconds. In contrast the GPU is able to utilize its massive parallel processing capabilities using its cores:

OpenCL	first run	second run	third run	fourth run	average
	0.045156	0.044903	0.044802	0.044574	0.04485875

Comparing the best result of OpenMP which is the 14 thread parallel loop, we can see a huge performance difference again:

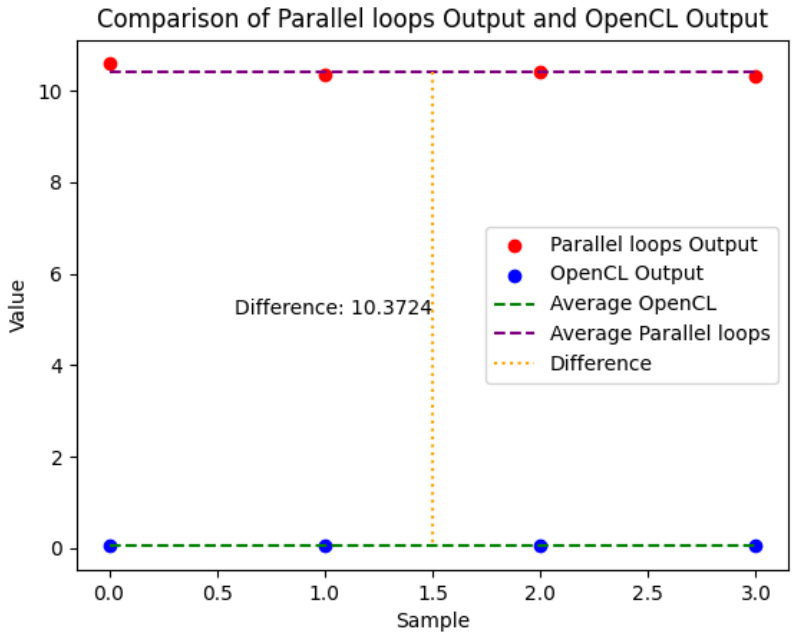

Figure 6: Comparison of the execution times between the OpenCL implementation and serial implementation.

The GPU code executes around 250 times faster than the best parallel implementation in the CPU. This is to be expected as the GPU's hardware is designed to run computanionally intensive tasks, such as the gaussian blur algorithm, while the CPU is a more general purpose processing unit.

3.3 Discussion

In conclusion, the utilization of parallel computing techniques, such as multithreading and GPU acceleration, plays a crucial role in improving computational performance. By exploiting parallelism, we can effectively distribute workloads across multiple processing units, enabling concurrent execution and reducing overall processing time. CPUs with multi-core architectures offer benefits in handling diverse workloads and optimizing serial code segments. On the other hand, GPUs excel in parallel processing scenarios by employing thousands of cores and efficiently managing massive thread workloads. Both approaches contribute to increased performance and enhanced efficiency in different contexts.

In the result section we observed an execution time stagnation when increasing the threads of the CPU. This stagnation observed after increasing the number of threads on a CPU can be attributed to several factors. Firstly, the CPU has a limited set of resources available, such as execution units, cache,

memory bandwidth, and instruction pipelines. When the number of threads exceeds the available resources, contention and resource sharing among threads can occur, leading to performance degradation. Additionally, memory access contention can arise when multiple threads simultaneously access the same memory locations, resulting in cache thrashing and increased cache misses. Synchronization overhead, introduced by locking mechanisms and thread coordination, can also impact performance as threads contend for shared resources. Lastly, managing and scheduling a larger number of threads incurs overhead in terms of thread creation, context switching, and scheduling, which can limit the potential performance gains. To mitigate these issues, careful design, optimization of multi-threaded applications, and workload balancing are essential. This leads us to the conclusion that it would be very difficult for the CPU to achieve the execution efficiency of the GPU.

In contrast to CPUs, GPUs possess the ability to efficiently spawn a large number of threads and leverage their parallel processing capabilities. GPUs consist of numerous cores that can execute thousands of threads concurrently, allowing for massive parallelism. This parallel execution model enables the GPU to better utilize its cores and handle highly parallelizable workloads, such as graphics rendering, scientific simulations, and machine learning algorithms. With the high number of threads available, the GPU can effectively hide memory access latency by switching between threads during memory operations, masking the inherent latency through thread-level parallelism. Moreover, GPUs often feature dedicated memory hierarchies and optimized memory access patterns, further enhancing their ability to handle massive thread workloads. By efficiently distributing the workload among the cores and leveraging parallel execution, GPUs can achieve significant performance gains and overcome the performance stagnation observed on CPUs when increasing the number of threads. This makes GPUs well-suited for data-parallel tasks that can be divided into numerous independent sub-tasks, harnessing the power of parallel processing to accelerate computations.

During the iterative development process of the kernel code, numerous improvements were made to fully exploit the massive processing power of GPUs. Initially, the global size was set to correspond to each pixel in the image, enabling parallel execution across GPU cores. Profiling the GPU's execution time helped determine the optimal work group size, ensuring efficient workload distribution,increased memory coalescence and resource utilization. Leveraging the superior performance of single-precision floating-point operations over double precision resulted in significant speed improvements. Although attempts to utilize the GPU's local memory introduced complexities that came with mild speed ups. By reducing the computational workload within inner loops and employing vector operations, the algorithm achieved remarkable performance gains compared to the initial float kernel implementation. These iterative refinements and GPU-specific optimizations highlight the importance of carefully harnessing the parallel processing capabilities of GPUs to achieve superior performance in highly parallel computational tasks.