PATTERN SEEK ACCELERATOR

Steven Thomi[†]
EEE4120F
University of Cape Town
South Africa
[†]THMSTE021

***Abstract*—The purpose of this paper is to elaborate on the effectiveness of a parallel algorithm implemented in Open Computing Language (OpenCL) in carrying out a search operation. The Pattern Seek Accelerator (PSA) algorithm searches for a known sequence of bytes in a particular block of memory. If a match is found, the PSA returns a success notification, start and end location of the sequence of bytes in the block of memory. In contrast, a failure alert is returned if a match is not found. The PSA algorithm applies OpenCL to parallelise the search operation by assigning each processor a unique start location to begin the search; these operations occur in parallel and result in a more stable runtime and an eventual speed-up over the gold standard implementation in C.**

I. Introduction

Pattern recognition underpins most aspects of modern computing. As a result of the increasing channel bandwidths and the rapidly increasing appetite for real-time, high resolution data streams it is imperative for us to identify similarly fast evolving alternatives to sequential pattern recognition algorithms.

A. Applications

As noted by Hancock [1], pattern recognition underpins developments in cognate fields such as:

- computer vision
- image processing
- text and document analysis
- neural networks
- biometrics
- bioinformatics
- multimedia data analysis
- data science

Computer vision and image processing algorithms are at the forefront of the development effort of the self-driving car. Autonomous vehicles rely on the quick and real-time perception of the car's systems to any impending doom by matching trained patterns to real-world situations.
Text and document analysis requires the computer to identify possible syntax, semantic, and grammatical errors in a text document. An auto-correct algorithm would be trained on a set of language rules. When something that goes against these rules is detected, an error flag is raised and the text highlighted.
Neural networks aim to mimic the operation of the human brain. The algorithm aims to identify underlying relationships and dependencies in a dataset, and build upon such dependencies in order to accomplish a task. This fetch-decode cycle is highly dependent on the patterns within the dataset and their implied meaning.
Biometrics describe physical traits which uniquely identify one person from another. The analysis of these human-specific traits requires the computer to identify patterns recognised on one human with a reference scope consisting of all other humans recorded by the system. The application of such a system would require an effective and resource-efficient pattern recognition algorithm.
Bioinformatics is a discipline aimed at using computers to understand and store biological data. The datasets stored in bioinformatic research tend to be large and complex, requiring an optimised search algorithm to traverse through them. Algorithms such as the Open Mass Spectrometry Search Algorithm (OMSSA) broadly covered in [2] require the use of efficient, sensitive and specific algorithms for peptide identification in order to attain a desired specificity in the research.
Multimedia data analysis and data science require a strong understanding of large sets of data in order to extract knowledge and insights from it and use it across a wide variety of applications.

In order to build upon these systems and improve their general performance, we first have to address the yardstick that ties them all together: pattern seek algorithms.

II. Background

In order to tackle the processing speed barriers of modern computing, the pattern search operation needs to be optimised. Linear and binary search algorithms (as elaborated through the gold standard) consist of many data independent iterations which could be carried out by independent processing elements.
Hancock [1] highlights the importance of the use of effective pattern recognition algorithms since they play a crucial role in most fields of computing.
Geer et al [2] in addition, goes on further to explain on the importance of effective pattern recognition algorithms in their field of work: bioinformatics. The paper discusses the reliance of high throughput proteomics on the analysis of thousands of peptide spectra derived from biological samples

and notes that new ways of efficiently processing the data are being sought.

This paper aims to use parallel computation, through the OpenCL framework, to deliver an algorithm that maintains a steady processing time in spite of an increasing data throughput.

III. Methodology

A. Software

The Open Computing Language (OpenCL) framework is used to develop this project.

B. Hardware

The hardware used to run the timing tests of the OpenCL and C implementations is Blue Lab PC 18. The processor and memory resources in this computer are captured in *Appendix A*.
I; however, carried out the development and initial testing of the OpenCL kernel and host on a 2.9 GHz Dual-Core Intel Core i5 MacBook Pro with 8 GB 2133 MHz of LPDDR3 memory.

C. Development Procedure

The steps I carried out in writing the parallel program are summarised below:

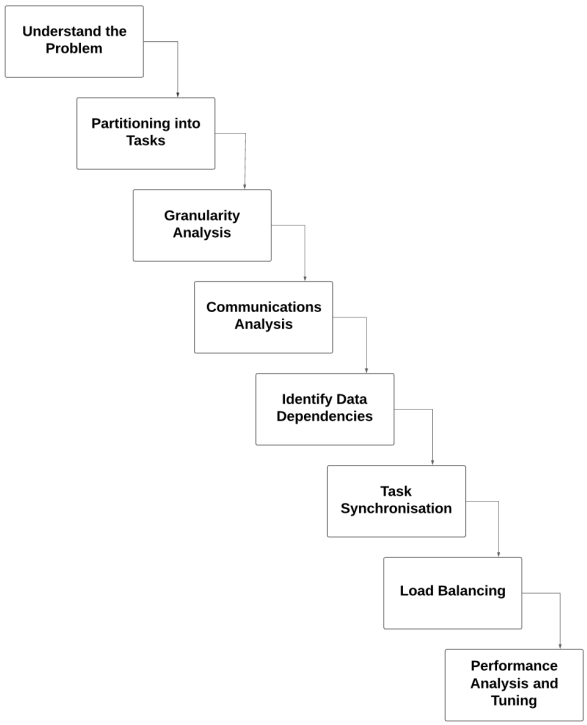

Fig. 1: Development Procedure

D. OpenCL Program Lifecycle

The OpenCL program lifecycle can be broken down into the following steps:

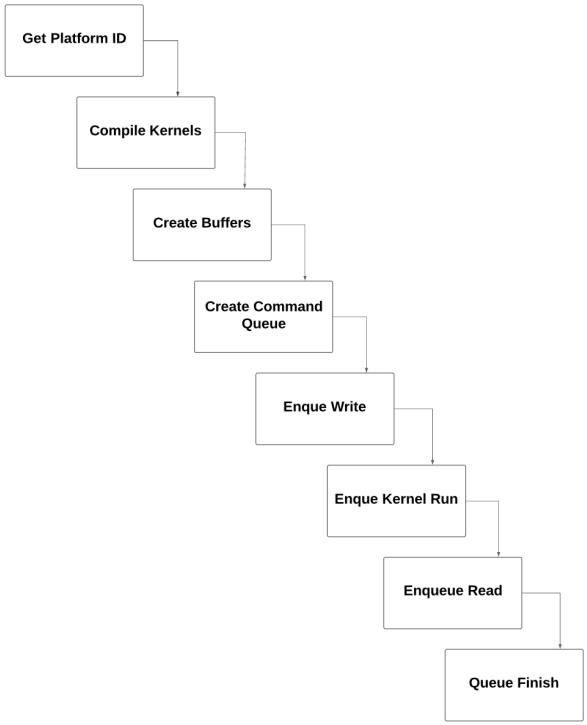

Fig. 2: OpenCL Program Lifecycle

The development effort takes caution to ensure each process executed by the kernel is carried out independently of the next and all heap memory allocations are freed at the *queue finish* point.

E. Results Interpretation

The OpenCL process execution time will be measured. The process execution time is measured across the *enque kernel run* stage discussed in *subsection D*.

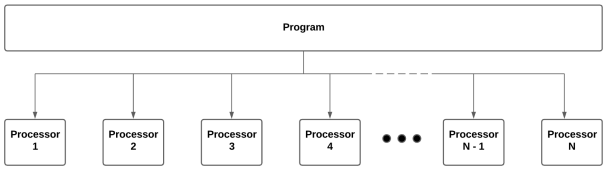

Fig. 3: OpenCL Program Execution

The golden measure's execution time will also be measured. The execution time of the C program will be measured across the main iterative loop controlling the search effort.
The measurement metric used to gain a greater understanding of the benefits of parallel programming over serial programming is the program **speed-up**. The program speed-up of the parallel program's process execution time

over the serial programs execution time is calculated and recorded for each list length. The speed-up metric informs us of how fast (or slow) the parallel program is over the serial alternative.

IV. Design

A. OpenCL Host

```c
/** \file main.c
 *  Host Application - OpenCL Host
 *  The purpose of this file is to set up the OpenCL
 *  environment, inputs and outputs, and kernel.
 */
int main(void) {
    int i;

    //! Define input array length variables
    const int block_size = pow(4, 1);
    const int pattern_size = 4;

    //! Create the two input arrays
    int *Pattern = (int*)malloc(sizeof(int)*pattern_size);
    int *Block = (int*)malloc(sizeof(int)*block_size);

    //! Populate the Block (block memory) array
    for(i = 0; i < block_size; i++){
      Block[i] = (block_size-1)-i;
    }

    //! Define the pattern to be searched for
    Pattern[0] = 3;
    Pattern[1] = 2;
    Pattern[2] = 1;
    Pattern[3] = 0;
```

Listing 1: OpenCL Host Application Definitions

The code snippet provided in Listing 1 illustrates the list size definitions (i.e., pattern size, block size) and the array heap memory allocation and population.
These variables are used in the definition of memory objects and memory buffers passed to the kernel.
Block[] will be divided into subsections of which Pattern[] will be compared.
The block size must be either greater than or equal to the pattern size.

```c
//! Create the OpenCL kernel
cl_kernel kernel = clCreateKernel(program, "search", &ret);

//! Set the arguments of the kernel
ret = clSetKernelArg(kernel, 0, sizeof(cl_mem), (void *)&pattern_mem_obj);
ret = clSetKernelArg(kernel, 1, sizeof(cl_mem), (void *)&block_mem_obj);
ret = clSetKernelArg(kernel, 2, sizeof(cl_mem), (void *)&pattern_size_mem_obj);
ret = clSetKernelArg(kernel, 3, sizeof(cl_mem), (void *)&output_index_mem_obj);

//! Execute the OpenCL kernel on the list
size_t global_item_size = block_size; //!< Process the entire lists
size_t local_item_size = 1; //!< Divide work items into groups
tic();
ret = clEnqueueNDRangeKernel(command_queue, kernel, 1, NULL,
       &global_item_size, &local_item_size, 0, NULL, NULL);
printf("\nOpenCL Implementation: %.5f ms\n\n", toc()/1e-3);

//! Read the memory buffer output_index on the device to the local variable output_index
int output_index;
ret = clEnqueueReadBuffer(command_queue, output_index_mem_obj, CL_TRUE, 0,
       1 * sizeof(int), &output_index, 0, NULL, NULL);
```

Listing 2: OpenCL Host Application Kernel Definition

The code snippet provided in Listing 2 demonstrates the kernel's is creation and parameter loading. The kernel takes in four parameters:

- **5** : the memory object holding Pattern[]
- **6** : the memory object holding Block[]
- **7** : the memory object holding pattern size
- **7** : the memory object returning the output index

We pass the entire list size to be processed (**11**) and assign each work item a single value to process.

B. OpenCL Kernel

```c
/** \file search_kernel.cl
 *  search - OpenCL kernel
 *  The aim of this kernel is to search for a pattern of
 *  bytes by traversing through i:i+p memory addresses.
 */
__kernel void search(__global const int *Pattern,   //!< in: 1-D int array
                     __global const int *Block,     //!< in: 1-D int array
                     __global const int *pattern_size,  //!< in: int pointer
                     __global int *output_index     //!< out: int pointer
                     ) {
    //! <h3>Search Kernel</h3>
    int i = get_global_id(0); //!< index of the current element to be processed
    int j = 0; //!< sentinel index of the pattern

    *output_index = -1; //0xFFFFFFFF //!< output if not found

    // Do the search operation
    while(Pattern[j] == Block[i+j]){
        //! If a match is made, check for a match in the whole pattern
        j = j + 1; //!< traverse pattern

        if(j == *pattern_size){
          //! If the whole pattern matches, output memory location
          *output_index = i; //!< output if found
        }
    }
}
```

Listing 3: OpenCL Search Kernel

The kernel performs the following actions:

- **12** : the kernel fetches the global work-item ID using the instructions provided in Listing 2 (i.e, 0, 1, 2, ...)
- **15** : the default state of the output index is set to 'not found' (-1)
- **18** : the pattern is checked for a match
- **22** : if consecutive matches for all values of the pattern are found (**24**) output index returns the location of the match (process id)

The operation of the kernel is illustrated below:

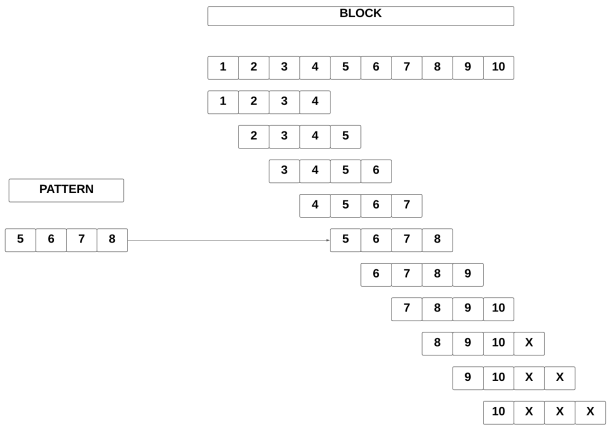

Fig. 4: OpenCL Kernel

Using Fig. 4, it is observed that each block division is run on an an independent process. The pattern is checked to establish a match between itself and the block being held by the process.
If a match is established, the process number is set to the output index.

C. Gold Standard

```
//! Do the operation
tic();
for(i = 0; i < block_size; i++){
    j = 0;
    while(Pattern[j] == Block[i+j]){
        j = j + 1;

        if(j == pattern_size){
        //! return memory position
          output_index = i;
        }
    }
}
printf("\nC Implementation: %.5f ms\n\n", toc()/1e-3);
```

Listing 4: C Search Implementation

The C implementation carries out a serial algorithm implementation of the program described in *subsection B*.
The C program aims to give us a measure of serial program performance and its shortcomings.

V. Proposed Development Strategy

The PSA algorithm is optimised for cases in which it is fed with large chunks of data under a high throughput channel and is faced with hard deadlines to meet. A potential application for this kind of research is in the field of autonomous vehicles.
Autonomous vehicles rely on a set of sensors to collect data within a given angle of sight, process these data points and respond to the output received all within the blink of an eye. System unresponsiveness (i.e. as a result of execution time lag) would likely result in a fatal collision. In order to eliminate (or at least minimise) unresponsive time, a parallelised pattern seek accelerator (PSA) algorithm should be deployed to filter through incoming data, recognise trained patterns and make quick decisions.

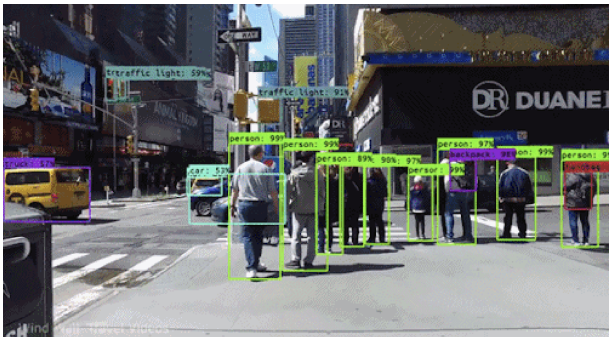

Fig. 5: Computer Vision [3]

Fig. 5 illustrates the different types of obstacles likely to be processed by a computer vision system found in an autonomous vehicle. The self-driving vehicle has to accurately identify the obstacles within its scope to a particular label and monitor its environment for any changes to these labels.

Another fast developing area of research is the field of biometrics. In order to guarantee the authenticity of a bio password, it is necessary to take numerous measurements of the user's characteristics. If the user's characteristics perfectly match those of an authenticated personnel, they are granted access to the service.
Voice recognition, heart-rate sensors, iris recognition, fingerprint scanning and facial recognition technologies are at the forefront of biometric study. Advanced adaptations of facial recognition technologies (i.e., True Depth system) transform the depth map of your face (via an infrared image) into a mathematical representation and compares that representation to the enrolled facial data [4].

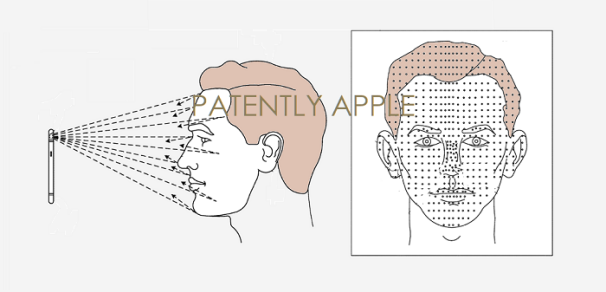

Fig. 6: Facial Recognition [4]

Fig. 6 illustrates the use of infrared imaging to create a depth map of the user's face. The infrared rays are shone onto the user's face multiple times to generate a comprehensive depth map of the users face. This map is compared to the pattern stored on the chip; if a match is found, the user is granted access.

The large quantity of data being processed requires the application of a parallel pattern seek accelerator to comb through the data and produce a match or no match condition in a fraction of a second.

VI. Planned Experimentation

A. Spatial Accuracy

The sequence of bytes within the block holding the pattern is shifted around: from the start of the block, to the middle of the block, and finally to the end of the block. The PSA is run on each of these cases to ensure the correct start and end locations are returned. Moreover, the pattern is removed from the block to test the ability of the algorithm to detect this and output a 'not found' criteria.
If the algorithm functions on all four of these bases, it is robust and meets user requirements.

B. Run-time

The run time of the PSA and gold standard algorithm is investigated.
For both these algorithms, the block size is incremented from 4^1 to 4^{10} and the time to carry out the operation is recorded. Any further increments beyond these points would risk memory or communication deadlock contributing

significantly to the observed speed-up (or speed-down).

$$block\ size = 4^N\ where\ N = 1:10 \tag{1}$$

VII. Results

A. Spatial Accuracy

1) Case 0: Pattern[] not in Block[]: The following illustration depicts the scenario in which the pattern is not in the block:

```
(base) Stevens-MacBook-Pro:OpenCL Implementation steventhomi$ gcc-
Clock resolution: 1000 ns

OpenCL Implementation: 46.25800 ms

The pattern to be searched for is:
1       2       4       8
The block to be searched is:
Index: 0         Block Element: 1
Index: 1         Block Element: 2
Index: 2         Block Element: 4
Index: 3         Block Element: 16
Index: 4         Block Element: 16
Index: 5         Block Element: 32
Index: 6         Block Element: 64
Index: 7         Block Element: 128
Index: 8         Block Element: 256
Index: 9         Block Element: 512
Index: 10        Block Element: 1024
Index: 11        Block Element: 2048
Index: 12        Block Element: 4096
Index: 13        Block Element: 8192
Index: 14        Block Element: 16384
Index: 15        Block Element: 32768

The pattern was not found in the block.
(base) Stevens-MacBook-Pro:OpenCL Implementation steventhomi$
```

Fig. 7: Case 0: Pattern[] not in Block[]

As evident in Fig. 7, the PSA correctly identifies the pattern is not in the memory block.

2) Case 1: Pattern[] at the start of Block[]: The following illustration depicts the scenario in which the pattern is at the start of the block:

```
(base) Stevens-MacBook-Pro:OpenCL Implementation steventhomi$ g
Clock resolution: 1000 ns

OpenCL Implementation: 54.10600 ms

The pattern to be searched for is:
1       2       4       8
The block to be searched is:
Index: 0         Block Element: 1
Index: 1         Block Element: 2
Index: 2         Block Element: 4
Index: 3         Block Element: 8
Index: 4         Block Element: 16
Index: 5         Block Element: 32
Index: 6         Block Element: 64
Index: 7         Block Element: 128
Index: 8         Block Element: 256
Index: 9         Block Element: 512
Index: 10        Block Element: 1024
Index: 11        Block Element: 2048
Index: 12        Block Element: 4096
Index: 13        Block Element: 8192
Index: 14        Block Element: 16384
Index: 15        Block Element: 32768

The pattern was found in the block.
 Start Index: 0
 End Index: 3
(base) Stevens-MacBook-Pro:OpenCL Implementation steventhomi$
```

Fig. 8: Case 1: Pattern[] at the start of Block[]

As evident in Fig. 8, the PSA correctly identifies the pattern is at the start of the memory block; starting on the first memory address (index 0) in the block and ending on the fourth memory address (index 3) of the block.

3) Case 2: Pattern[] in the middle of Block[]: The following illustration depicts the scenario in which the pattern is in the middle of the block:

```
(base) Stevens-MacBook-Pro:OpenCL Implementation steventhomi$ g
Clock resolution: 1000 ns

OpenCL Implementation: 41.86200 ms

The pattern to be searched for is:
1       2       4       8
The block to be searched is:
Index: 0         Block Element: 1
Index: 1         Block Element: 2
Index: 2         Block Element: 4
Index: 3         Block Element: 16
Index: 4         Block Element: 16
Index: 5         Block Element: 32
Index: 6         Block Element: 1
Index: 7         Block Element: 2
Index: 8         Block Element: 4
Index: 9         Block Element: 8
Index: 10        Block Element: 1024
Index: 11        Block Element: 2048
Index: 12        Block Element: 4096
Index: 13        Block Element: 8192
Index: 14        Block Element: 16384
Index: 15        Block Element: 32768

The pattern was found in the block.
 Start Index: 6
 End Index: 9
(base) Stevens-MacBook-Pro:OpenCL Implementation steventhomi$
```

Fig. 9: Case 2: Pattern[] in the middle of Block[]

As evident in Fig. 9, the PSA correctly identifies the pattern is in the middle of the memory block; starting on the seventh memory address (index 6) in the block and ending on the tenth memory address (index 9) of the block.

4) Case 3: Pattern[] at the end of Block[]: The following illustration depicts the scenario in which the pattern is at the end of the block:

```
(base) Stevens-MacBook-Pro:OpenCL Implementation steventhomi$ gc
Clock resolution: 1000 ns

OpenCL Implementation: 53.90300 ms

The pattern to be searched for is:
1       2       4       8
The block to be searched is:
Index: 0         Block Element: 1
Index: 1         Block Element: 2
Index: 2         Block Element: 4
Index: 3         Block Element: 16
Index: 4         Block Element: 16
Index: 5         Block Element: 32
Index: 6         Block Element: 64
Index: 7         Block Element: 128
Index: 8         Block Element: 256
Index: 9         Block Element: 512
Index: 10        Block Element: 1024
Index: 11        Block Element: 2048
Index: 12        Block Element: 1
Index: 13        Block Element: 2
Index: 14        Block Element: 4
Index: 15        Block Element: 8

The pattern was found in the block.
 Start Index: 12
 End Index: 15
(base) Stevens-MacBook-Pro:OpenCL Implementation steventhomi$
```

Fig. 10: Case 3: Pattern[] at the end of Block[]

As evident in Fig. 10, the PSA correctly identifies the pattern is at the end of the memory block; starting on the thirteenth memory address (index 12) in the block and ending

on the sixteenth memory address (index 15) of the block.

B. Run-time

The parallel and serial PSA algorithms were evaluated on a variable length block of memory ($block\ size = 4^N$) and their run-times recorded in Table I.

TABLE I: Parallel and Serial Run-time

N	C [ms]	OPENCL [ms]
4^1	0.000190	0.019680
4^2	0.000220	0.019390
4^3	0.000350	0.034030
4^4	0.000720	0.019210
4^6	0.011140	0.019150
4^8	0.013872	0.035020
2×4^8	0.27996	0.03633
3×4^8	0.42321	0.03807
4^9	0.56176	0.03945
2×4^9	1.03655	0.03916
3×4^9	1.59432	0.05858
4^{10}	2.167240	0.063160

The parallel and serial run-time are illustrated below:

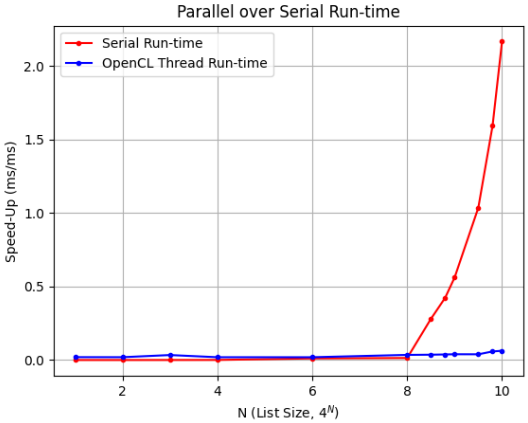

Fig. 11: Parallel and Serial Run-time

Using Fig. 11 it is evident that parallel computation runs at a much faster rate than serial computation when it comes to processing large data sets. At a list size of 4^8, the performance of the serial algorithm matches that of the parallel algorithm; increasing the list size beyond this point sees the run-time of the serial algorithm increase exponentially from 0.0139 ms to a recorded maximum of 2.167 ms (at 4^{10}) while the the parallel algorithm's run-time increases only minimally (mainly due to memory and bandwidth limitations) from 0.0350 ms to 0.0632 ms (at 4^{10}).

The parallel speed-up over the serial PSA algorithm, evaluated on a variable length block of memory ($block\ size = 4^N$), was calculated using data captured from Table I. The results are represented in Table II.

TABLE II: Parallel and Serial Speed-Up

N	Speed-Up
4^1	0.00965
4^2	0.01135
4^3	0.01029
4^4	0.03748
4^6	0.58172
4^8	0.39612
2×4^8	7.706
3×4^8	11.11663
4^9	14.2398
2×4^9	26.46961
3×4^9	27.21611
4^{10}	34.31349

The parallel-over-serial speed-up is illustrated below:

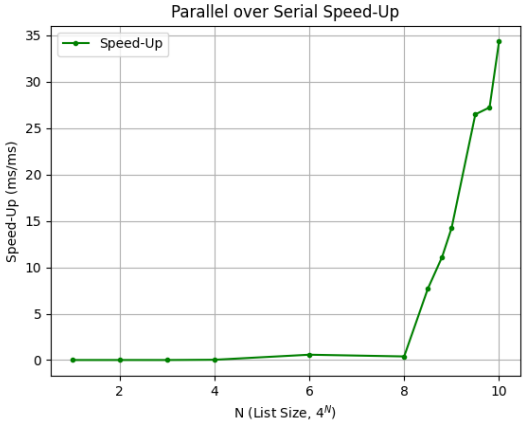

Fig. 12: Parallel over Serial Speed-Up

Using Fig. 12 it is evident that parallel computation experiences a much greater speed-up over serial computation when it comes to processing large data sets. At a list size of 4^8, the speed-up of the parallel algorithm over the serial algorithm begins to increase exponentially from a speed-up of 0.39612 ms to a maximum recorded speed-up of 34.3135 ms (at 4^{10}).

VIII. Conclusion

The Pattern Seek Accelerator (PSA) algorithm was successfully implemented in a serial C gold standard and a OpenCL parallel implementation . The serial implementation yielded shorter run-times at smaller list size values and larger run-times at larger list size values. Using these results, it can be inferred that the serial implementation would continue to record a slow-down at even larger list sizes. The serial implementation is; therefore, not suitable for use in high throughput application.

On the other hand, the OpenCL implementation yielded

consistent, acceptable run-times at both small and large list size values. The serial implementation managed to run faster and outperform the OpenCL implementation at smaller list size values; the parallel implementation ,however , surpassed the serial implementation in efficacy at larger list size values. Using these results, it can be inferred that the parallel implementation would continue to experience a speed-up at even larger list sizes. The parallel implementation is; therefore, suited for use in high throughput application.

IX. References

[1] E. Hancock, Pattern Recognition. Available at: https://www.journals.elsevier.com/pattern-recognition

[2] L. Geer, P. Markey et al, Open Mass Spectrometry Search Algorithm. Journal of Proteome Research: 2004. Available at: https://pubs.acs.org/doi/full/10.1021/pr0499491

[3] A. Lai, How do Self-Driving Cars See? Available at: https://towardsdatascience.com/how-do-self-driving-cars-see-13054aee2503

[4] J. Purcher, TrueDepth Face ID System with Special Bracket Assembly. Patently Apple: 2019. Available at: https://www.patentlyapple.com/patently-apple/2019/04/apple-won-54-patents-today-covering-their-truedepth-camera-system-for-face-id-and-future-hover-gesture-sensing.html

[5] E. Smistad, Getting started with OpenCL and GPU Computing: 2018. Available at: https://www.eriksmistad.no/getting-started-with-opencl-and-gpu-computing/

X. Appendix

A. Appendix A

Blue Lab PC 18 CPU information is captured below:

Fig. 13: Blue Lab PC 18 CPU information

B. Appendix B

The code resources used to realise this project can be found here: **GitHub Repository**