Computer Science Extended Essay:

Investigating the effect of using a general-purpose GPU instead of the CPU on the performance of approximate string-matching algorithms.

Research Question:

To what extent is the speed of a Levenshtein's distance-based approximate string-matching algorithm different when being processed on a CPU vs. on a GPU.

CS EE World
https://cseeworld.wixsite.com/home
May 2022
29/34
A
Submitter Info:
Email: officalquincy [at] gmail [dot] com

Word Count: 3925 Words

Content

1 Introduction 3
2 Theory & Concepts 4
2.1 Approximate String Matching (ASM) 4
2.2 Levenshtein's Distance 6
2.3 Graphical Processing Unit 7
3 Methodology & Testing 9
3.1 Preface 9
3.2 Dependent & Control Variables 10
3.3 Search Term Size - Hypothesis 11
3.4 Search Term Size – Test 12
3.5 Dictionary Size Test – Hypothesis 14
3.6 Dictionary Size Test – Test 16
3.7 Additional Graphs 18
4 Conclusion 19
4.1 Effect of varying Search Term Size 19
4.2 Effect of varying Dictionary Size 20
4.3 Comparison 21
5 Extensions 22
5.1 Multithreading 22
5.2 Server CPUs 22
5.3 Kernel Architecture 22
6 Appendix 23
6.1 References 23
6.2 General Code 25
6.3 CPU Specific Code 33
6.4 GPU-Specific Code 36
6.5 GPU Kernel 47

1 Introduction

This essay will focus on the application of a general-purpose graphics processing unit (GPGPU) on an approximate string matching (ASM) algorithm. GPGPUs are graphics processing units that can be used for general-purpose calculations as opposed to solely graphics-based calculations. Due to GPUs having thousands of processing cores, they are extremely well versed at running thousands of small tasks simultaneously. This is referred to as parallel programming and it can lead to dramatic speed increases in specific scenarios. This essay will investigate to what extent using the GPU to process a Levenshtein's Distance-based ASM algorithm can increase the processing speed of the algorithm, leading to the research question: To what extent is the speed of a Levenshtein's distance approximate string-matching algorithm different when being processed on a CPU vs. on a GPU.

2 Theory & Concepts

2.1 Approximate String Matching (ASM)

Approximate String Matching is the process of finding the closest, or n-closest matches of a given Search Term in a dictionary. To do so, given an array of strings which serves as our dictionary, named 'D', and a Search Term 'S', then for each string in D, where the current element index is 'X', we must find the number of transformations required to morph D_X into S. There are a few valid types of transformations for Levenshtein's Distance, these are[1]:

Insertion, adding a character into the query string: "Wrld" → "W**o**rld"

Deletion, removing a character from the query string: "Wo**a**rld" → "World"

Substitution, replacing a character from the query string: "W**u**rld" → "W**o**rld"

The number of transformations between two strings is known as the Edit Distance, and there are several algorithms available to determine this distance. A couple examples include: The Longest Common Subsequence, The Hamming distance, The Jaro Distance, and the Levenshtein distance. The difference between each algorithm is which transformations they count as valid. For example, the Hamming distance only allows substitution, thus it only applies to strings of the same length. This essay will focus on an ASM algorithm built on the Levenshtein's distance paradigm, whose valid transformations are shown above.

The process of finding the n-closest strings of a given search string in a brute-force manner can be summarized in the following 3-steps:

Step 1: For each string in the algorithm's dictionary:

Step 2: Compute the Levenshtein Distance, store the score and the current dictionary string within a binary search tree.

Step 3: Once each string has been computed, traverse the tree in order and output up to n strings.

While ASM is a niche field, it has had a large influence in not only computer science, but also external fields, such as biology, among others. It plays a crucial role in several real-world problems. For example, detecting plagiarism, bioinformatics, digital forensics, spell checkers, spam filters, and search engines[2]. In certain cases, such as search engines, the dictionary of strings to search from can become massive. There are existing optimizations to speed up the time it takes to complete an ASM query, such as indexing, which reduces the total number of strings we must iterate through using some indexing method, such as the first couple characters of each string. Existing ASM optimization methods are all software-based, but unless we find a software-based optimization with an O(1) runtime – meaning it would take the same amount of time to run regardless of the size of the input, which as far as we know is impossible – software-based optimizations can only speed up our query up to a point. This is where the massively parallel nature of the GPU comes in. Being able to take advantage of a GPU's immense parallel computing capabilities can theoretically dramatically increase the speed of ASM on large databases. Combined with the possibility for a data centre to possess dozens, to hundreds of computers each with GPUs installed within, having a parallelized version of ASM could allow for blazing fast ASM queries even for massive dictionaries.

2.2 Levenshtein's Distance

Levenshtein's Distance (LD) is a method of calculating the Edit Distance between 2 strings that considers the previously discussed Insertion, Deletion, and Substitution operations. Unlike some other methods, it does not incorporate Transposition (Swapping the positions of two characters).

This essay focuses on the LD algorithm as opposed to other Edit Distance algorithms due to its ease of implementation as well as it being able to consider three Edit Operations. Calculating the Levenshtein Distance of two strings will be done using the following matrix, shown in *Figure 2.2.1* below:

$$\begin{bmatrix} 0 & \cdots & len(S)+1 \\ \vdots & \ddots & \vdots \\ len(D_X)+1 & \cdots & LD \end{bmatrix}$$

Figure 2.2.1: Matrix used to determine Levenshtein's Distance

Where 'S' is the Search Term, 'D_X' is the current Dictionary Term, and the function 'len()' returns the number of characters in the inputted term. Initially, this matrix is empty. To fill it, there are a couple of possible methods. This essay will use an Iterative method with a full matrix, I chose this method as GPUs are known to be able to accelerate matrix-based calculations. The algorithm used in an iterative full matrix approach to finding LD involves traversing through the matrix in row-major order with two for-loops, then setting the element at the current coordinates given by the for-loops to the result of the following piecewise function shown in *Figure 2.2.2* below:

$$M(x,y) = \begin{cases} \max(\mathrm{x},\mathrm{y}), & if\ \min(\mathrm{x},\mathrm{y}) = 0, \\ min \begin{cases} M(x-1,y) \\ M(x,y-1) \\ M(x-1,y-1) + 1_{(D_{X_x} \neq S_y)} \end{cases} & else \end{cases}$$

Figure 2.2.2: Function used to determine the LD in matrix 'M' at indices x, y

2.3 Graphical Processing Unit

The Graphical Processing Unit (GPU) is a piece of hardware that is most commonly connected to a computer via a serial expansion bus, such as PCIe, a peripheral connection interface which allows for the highspeed transfer of dozens of gigabytes per second. This amount of speed is required to have reasonable interoperability speeds between the CPU and GPU; For example, one of the most common uses of a GPU is real-time computer graphics (Hence the name). To achieve real-time speeds, potentially several gigabytes of data stating what and where to draw things on the screen must be transferred between the computer's Main Memory to the GPU's onboard memory via a PCIe expansion bus. The transfer of data from Main Memory to GPU Memory poses an overhead. There are 3 major sources of overhead when programming on the GPU[3]:

- CPU Wrapper Overhead: This is the overhead created by the wrappers around GPU API (i.e., OpenCL / CUDA) functions, which are called from the CPU.
- Memory Overhead: This is the overhead created by moving data back and forth between Main Memory and the GPU's memory.
- GPU Launch Overhead: This is the overhead created by the time it takes for the GPU to retrieve the command given to it and begin executing it.

GPUs are structured differently from CPUs. The main difference is that whereas modern CPUs have between two and 64 cores or so, with most consumer processors containing four to 16 cores[4]; modern GPUs have two to three thousand specialized cores. Additionally, whereas according to Flynn's Taxonomy, multi-core CPUs operate using Multiple Instruction, Multiple Data (MIMD) techniques, GPUs operate with Single Instruction, Multiple Data (SIMD) techniques.

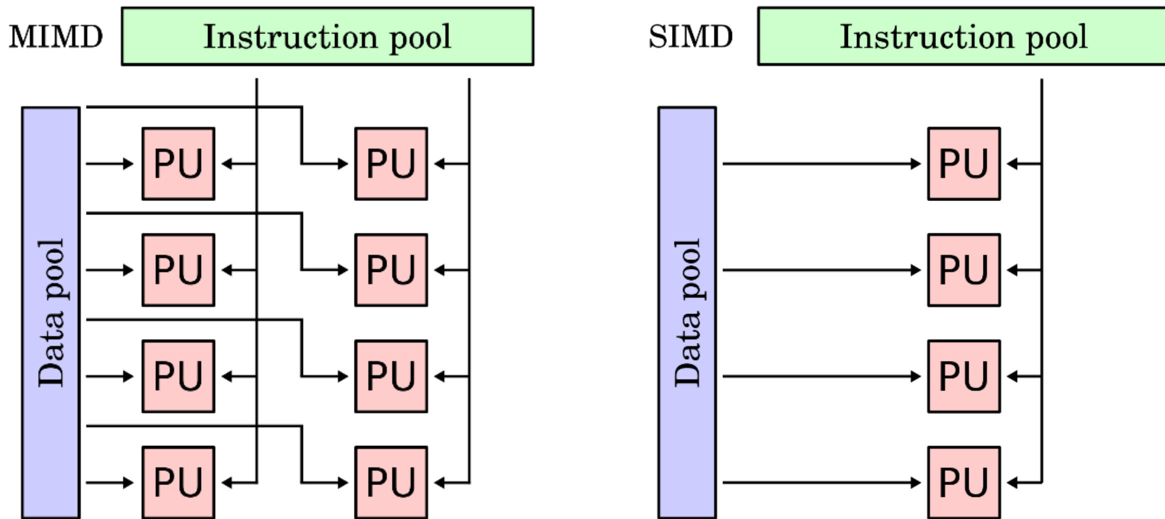

Figure 2.3.1 (Left): A Diagram showing MIMD architecture

Figure 2.3.2 (Right): A Diagram showing SIMD architecture

This difference in processing architectures creates notable differences between how things are processed on each device. For example, the GPU's or SIMD's main advantage is that it can compute mathematical operations on large sets of data-points extremely quickly relative to other architectures, with minimal memory costs, as the instruction and data pool is loaded into the SIMD device's memory once and is then shared with the totality of the processing units. The main disadvantage of SIMD architecture is that not every algorithm can be efficiently applied to it. It also takes a considerable amount of extra human interaction to create SIMD / parallel programs.[5] In contrast to SIMD, MIMD's main advantage is that it is trivial to program, as there is no explicit need for communication between processing units[6], since every processing unit has its own memory.

3 Methodology & Testing

3.1 Preface

Primary experimental data is the main source of data for this paper, for which two logically identical programs were created. The first implemented in Java, which is to be ran on the CPU. The second will be a kernel implemented in a variation of the C programming language specialized for OpenCL. Both programs will be provided in this paper's appendix.

An experimental approach – where I conduct an experiment to create primary data – was chosen to answer this paper's research question due to a lack of broad secondary data to answer the question. While some papers provided information on parallelized string-matching algorithms, most used a specific API or algorithm. For example, the paper "Using GPUs to Speed-Up Levenshtein Edit Distance Computation" used exclusively CUDA as its GPU API[7]. CUDA is only available on and is highly optimized for Nvidia GPUs, possibly resulting in higher speeds than we would expect for similarly powerful but differently branded GPUs. To avoid this issue, this paper aims to use cross-platform software to answer the research question at the most general scope possible, removing the performance enhancements specific hardware manufacturers may be able to give to their own hardware. To that end, both the Java Virtual Machine and the OpenCL API are cross-platform.

3.2 Dependent & Control Variables

The variable I will be using to compare the CPU and GPU's performance is the average time taken to complete a single ASM query in milliseconds. While there are other factors that can be considered, such as memory usage or power draw, time will give us the most quantifiable measure of how performant ASM is on both devices.

The average time taken was acquired by adding the time taken to complete each individual query, then dividing it by 30 (Number of repeats). This value was then converted to milliseconds by dividing it by 1,000,000.

The time taken to complete an individual query was acquired using 2 calls to Java's *System.nanoTime()* function, surrounding the function that completes the ASM Query on a given device, as shown in the pseudocode below:

```
long IndividualTime = System.nanoTime();
QueryCPULev(SearchTerm);
TotalTime += System.nanoTime() - IndividualTime;
```

Figure 3.2.1: Individual ASM Query Time Measurement Psuedocode

System.nanoTime() was chosen due to its high-resolution source and because time was returned in nanoseconds. As such, I could be sure that the time values gathered were accurate and precise.

Control Variables:

- The amount of ASM Queries done to find the average (repeats).
  This will be set at 30 for every test. Because I am calculating averages, the more repeats, the better. However, the more repeats, the longer the execution time. 30 stroke a good balance between precision and time taken.
- The hardware used for each test. This must be controlled as using different hardware will evidently cause changes in performance.

3.3 Search Term Size - Hypothesis

This first experiment conducted investigated the effect of different Search Term Sizes on the average time taken to complete an ASM Query.

Search Term Size

Current Dictionary Term Size

$$\begin{bmatrix} 0 & \cdots & len(S)+1 \\ \vdots & \ddots & \vdots \\ len(D_X)+1 & \cdots & LD \end{bmatrix}$$

Looking back at *figure 2.2.1* above, we can see that the size of the matrix increases as the size of the search term increases. This matrix must be filled for each term in the dictionary, meaning that if the search term size increases, more work must be done per dictionary term.

Both the CPU and GPU will have to complete the same amount of work per matrix; However, the difference appears when we consider that the CPU only has one processing unit available to iterate through the dictionary and compute the matrix for each term. Here, we see the GPU's massive advantage: it can assign each dictionary term to one of its thousands of processing units.

The total work done by the CPU can be summarized as: $O(S \cdot D \cdot \overline{D_x})$, while the total work done by each GPU processing unit can be summarized as: $O(S \cdot \overline{D_x})$; Where 'S' is the length of the Search Term, 'D' is the length of the Dictionary, and '$\overline{D_x}$' is the average length of a Dictionary Term. Looking at both equations, we can expect linear growth for the processing time taken for both the CPU and the GPU, however we can expect a much steeper gradient from the CPU due to the extra factor 'D'. While each processing unit is doing the same amount of work to fill one matrix, because the work is divided across so many processors, I hypothesize that the effect of increasing the search time size will be much greater on the CPU than on the GPU.

3.4 Search Term Size – Test

The independent variable for this test was the Search Term Size, measured in character count. For example: “Hello” would have a size of 5.

Testing was conducted with a dictionary of random integers up to but not restricted to 15 digits long. The dictionary consisted of 10,000 elements. The experiment had 30 repeats. Search terms with the following sizes were tested:

1,5,10,25,50,75,100,250,500,750,1000,2000,3000…,8000,9000,10,000.

Smaller values (From 1 to 1000) were used to investigate if there was a noticeable difference in performance at very small search term lengths, the thought being that due to the GPU’s initial overhead, small search term lengths might perform better on the CPU.

Larger values (From 1000 to 10,000) were used to more clearly highlight the relationship between the dependent and independent variables. Small values do not highlight the relationship as well due to the very small differences in processing time between two search term sizes with a small difference between them.

The test simply consists of:

- Generating a random string of current Search Term Size
- Conducting a ASM Query on that string 30 times
- Measuring the Overall time taken for all 30 queries, and the Average time taken for 1 query (By dividing Overall time taken by 30).
- Repeat for each Search Term Size tested

Search Term Size (# of Chars)	GPU Average (ms)	CPU Average (ms)	GPU Overall (ms)	CPU Overall (ms)	Percentage Difference of Average Time Taken	Percentage Difference of Overall Time Taken
1	**170**	**246**	*5122*	*7387*	144.71%	144.22%
5	**168**	**259**	*5040*	*7778*	154.17%	154.33%
10	**166**	**287**	*4994*	*8615*	172.89%	172.51%
25	**173**	**315**	*5213*	*9470*	182.08%	181.66%
50	**170**	**359**	*5112*	*10798*	211.18%	211.23%
75	**175**	**675**	*5269*	*20280*	385.71%	384.89%
100	**172**	**719**	*5172*	*21576*	418.02%	417.17%
250	**179**	**765**	*5390*	*22977*	427.37%	426.29%
500	**192**	**818**	*5764*	*24564*	426.04%	426.16%
750	**206**	**962**	*6200*	*28870*	466.99%	465.65%
1000	**223**	**1124**	*6719*	*33722*	504.04%	501.89%
2000	**279**	**1919**	*8382*	*57578*	687.81%	686.92%
3000	**340**	**2730**	*10229*	*81901*	802.94%	800.67%
4000	**404**	**3614**	*12130*	*108440*	894.55%	893.98%
5000	**464**	**4497**	*13929*	*134920*	969.18%	968.63%
6000	**537**	**5366**	*16130*	*160995*	999.26%	998.11%
7000	**641**	**6211**	*19240*	*186358*	968.95%	968.60%
8000	**681**	**6942**	*20434*	*208284*	1019.38%	1019.30%
9000	**741**	**7908**	*22231*	*237243*	1067.21%	1067.17%
10000	**796**	**8705**	*23904*	*261165*	1093.59%	1092.56%

Figure 3.4.1: Results of the test in tabular form

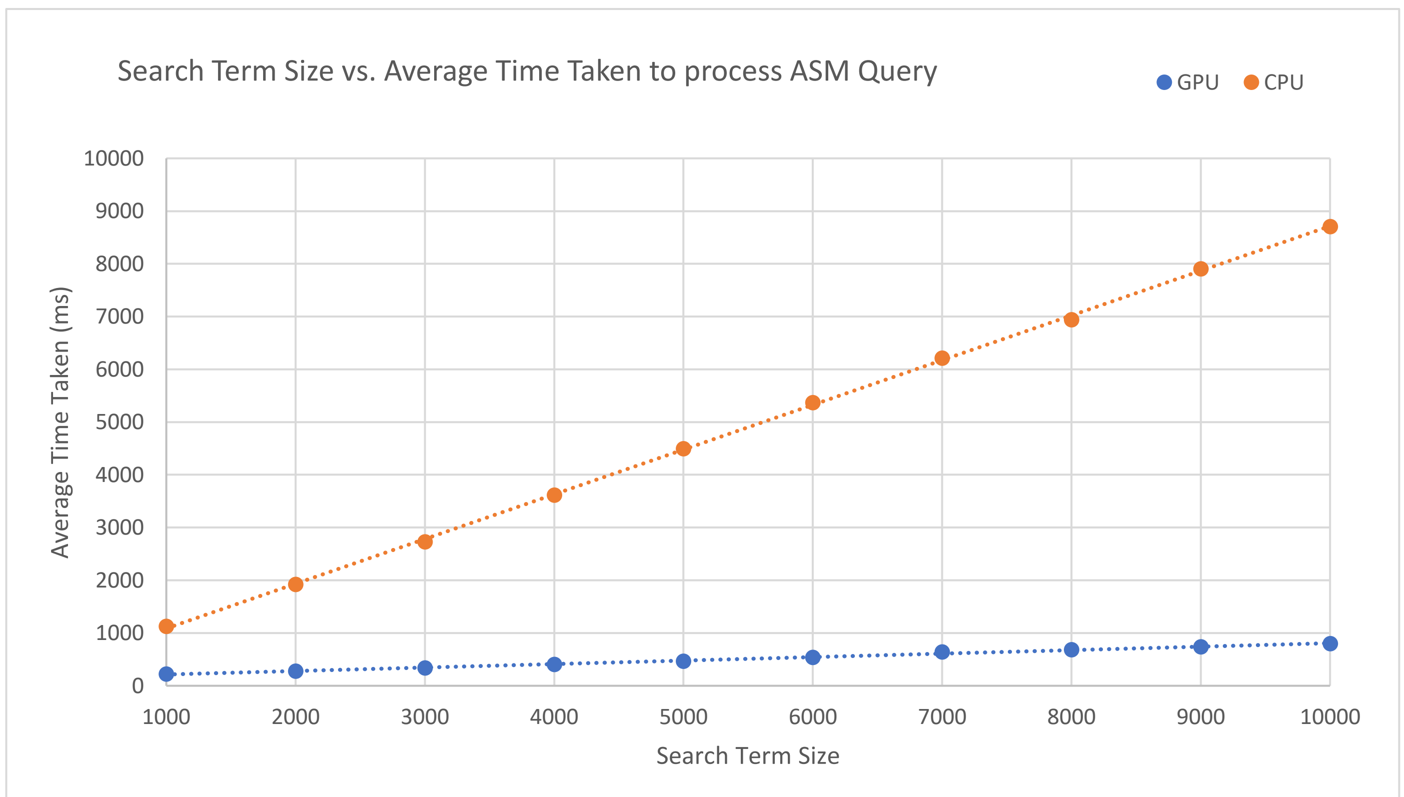

Figure 3.4.2: Results of the test in graphical form

3.5 Dictionary Size Test – Hypothesis

Unlike in the test in section 3.4, here we are not measuring the effect of changing the size of the matrix in which we evaluate Levenshtein's Distance. Instead, as can be seen in *figure 3.5.1* below, we measure the effect of increasing the number of matrices – the number of terms for which we calculate a Levenshtein's Distance for.

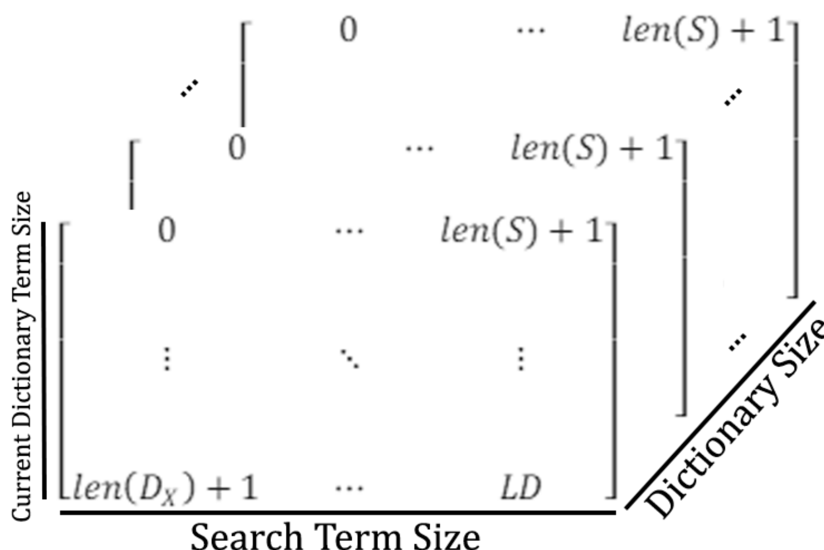

Figure 3.5.1: 3D Visualization of the Matrices used to evaluate the LD of every term in a dictionary

The reason for conducting this test is that the GPU operates with 'work-groups'. These work-groups represent a portion of data that the GPU must execute the current kernel[1] with. As the GPU does not have an infinite amount of processing units, the GPU maintains a pool of work-groups, which the processing units of the GPU then retrieve one-by-one until the pool is emptied[8], at which point the GPU has no more work to do and the task is complete.

My kernel was designed in such a fashion that each dictionary term represented a single work-group. (Kernel Architecture is a point which will be discussed later)

[1] A kernel is a set of user-defined instructions to be executed specifically on the GPU. It can be thought of as a function within regular CPU programming.

This test will therefore compare the impact of the GPU having to swap out an increasing amount of work-groups as the number of dictionary terms outnumber the number of processing units available on the GPU, with the CPU who can simply process the dictionary sequentially with no need to retrieve data from a pool.

I hypothesized that despite the potential added overhead of having to retrieve data from a work-group pool, the divide-and-conquer advantage that the GPU has thanks to its thousands of cores will still allow it to vastly outperform the CPU. This is exacerbated by the fact that modern GPUs possess exceedingly fast memory – the GPU used for every test's memory has a bandwidth of 448GB/s[9] compared to the RAM's average read/write speed of ~7GB/s. This means that while the GPU does have a pool of work-groups to retrieve work from, it can do it extremely quickly.

Even if my hypothesis is correct and the GPU is still faster than the CPU, the results of this experiment can be compared with the others to conclude which variable has the greatest impact on the time taken to execute an ASM query. That conclusion can be applied by developers to identify where to optimize their software to get the fastest implementation of ASM possible. Additionally, there may be a difference in which variable causes the greatest impact depending on the device used.

3.6 Dictionary Size Test – Test

The independent variable for this test was the Size of the Dictionary, measured in the number of terms within it.

Testing was conducted with dictionaries of random integers up to but not restricted to 15 digits long. The dictionaries had the following sizes:

1,10,50,100,500,1000,2500,5000,7500,10000,20000,30000,…,100000

The experiment had 30 repeats.

As with the previous experiment, smaller values from 1 to 10,000 were used to investigate if there was a noticeable difference in performance with very small dictionary sizes.

While larger values (From 10,000 to 100,000) were used to more clearly highlight the relationship between the dependent and independent variables.

Methodology:

- Generate a random string with length 15.
- Conducting a ASM Query on that string 30 times with the current Dictionary Size
- Measuring the Overall time taken for all 30 queries, and the Average time taken for 1 query (By dividing Overall time taken by 30).
- Repeat for each Dictionary Size to test

Dictionary Size (Terms)	GPU Average (ms)	CPU Average (ms)	GPU Overall (ms)	CPU Overall (ms)	Percentage Difference of Average Time Taken	Percentage Difference of Overall Time Taken
1	**<1**	**<1**	*27*	*2*	0.00%	7.41%
10	**<1**	**<1**	*20*	*8*	0.00%	40.00%
50	**<1**	**<1**	*22*	*17*	0.00%	77.27%
100	**<1**	**<1**	*27*	*28*	0.00%	103.70%
500	**1**	**4**	*54*	*121*	400.00%	224.07%
1000	**3**	**6**	*103*	*194*	200.00%	188.35%
2500	**13**	**18**	*391*	*567*	138.46%	145.01%
5000	**44**	**72**	*1332*	*2175*	163.64%	163.29%
7500	**94**	**168**	*2833*	*5057*	178.72%	178.50%
10000	**165**	**295**	*4959*	*8852*	178.79%	178.50%
15000	**371**	**692**	*11148*	*20775*	186.52%	186.36%
20000	**658**	**1225**	*19755*	*36763*	186.17%	186.09%
25000	**1021**	**2011**	*30633*	*60351*	196.96%	197.01%
30000	**1489**	**3366**	*44672*	*100987*	226.06%	226.06%
35000	**1989**	**5105**	*59698*	*153177*	256.66%	256.59%
40000	**2655**	**8966**	*79656*	*268991*	337.70%	337.69%
45000	**3437**	**12433**	*103113*	*373002*	361.74%	361.74%
50000	**4210**	**16571**	*126320*	*497146*	393.61%	393.56%

Figure 3.6.1: Results of the test in tabular form

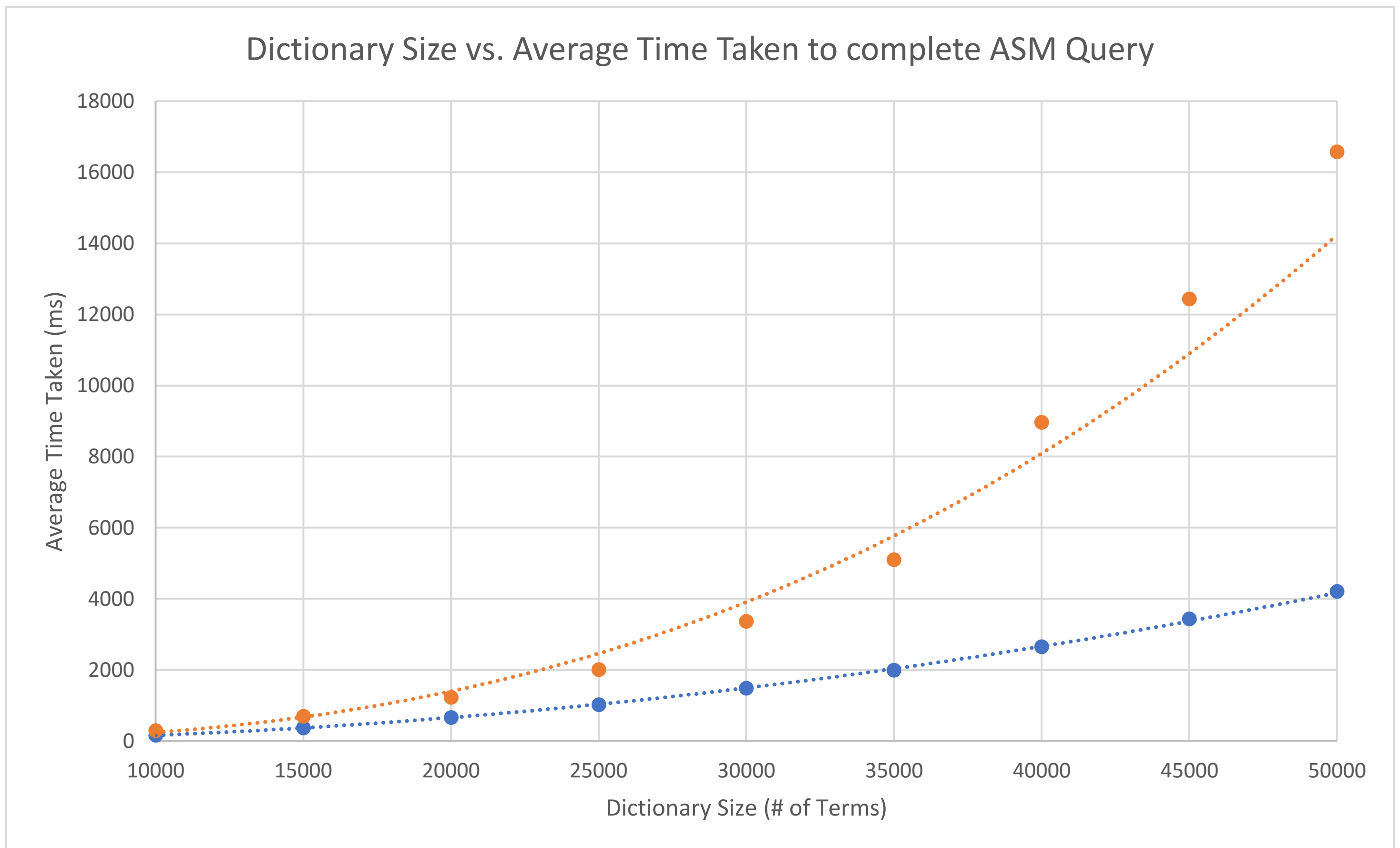

Figure 3.6.2: Results of the test in graphical form

3.7 Additional Graphs

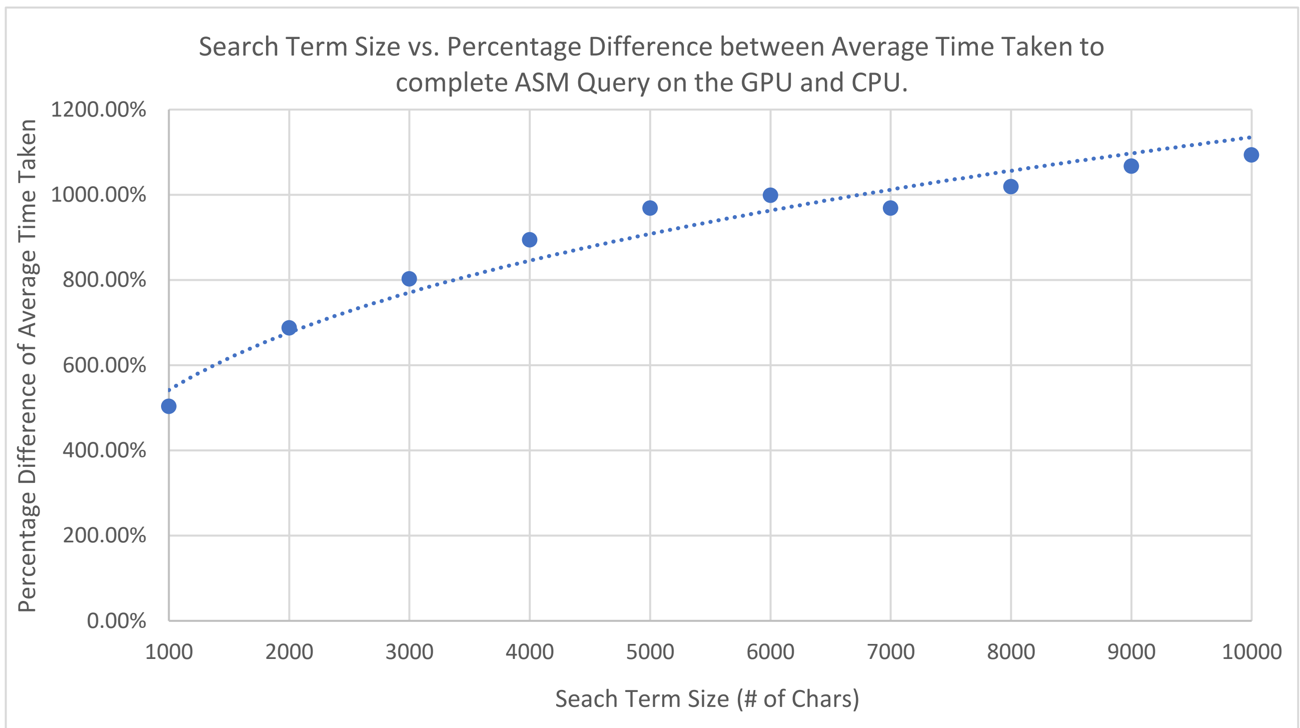

Figure 3.7.1: Graph highlighting the percentage difference between the average time taken per query on both devices when search term size is varied.

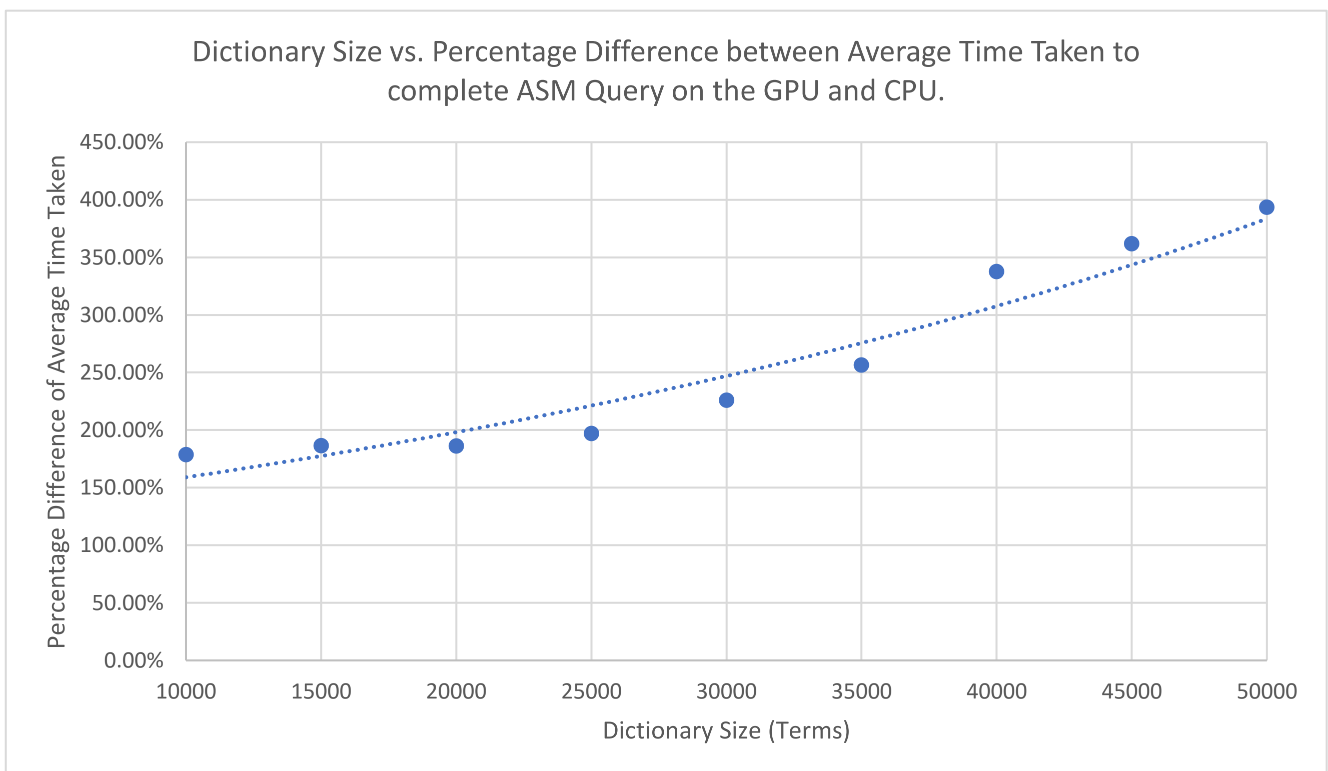

Figure 3.7.2: Graph highlighting the percentage difference between the average time taken per query on both devices when dictionary size is varied.

4 Conclusion

4.1 Effect of varying Search Term Size

To begin, it should be stated that due to how both Search Term Size (STS) and Average Dictionary Term Size shape the Levenshtein's Distance matrix, as seen in *figure 2.2.1*, it can be concluded that both variables have very similar effects on the performance of an ASM query.

From the experiment conducted in section 3.4, we can conclude that STS is linearly correlated with the average time taken to complete an ASM query. More importantly, it is clear that the GPU caused a massive performance increase over the CPU. We know this as we consistently saw the GPU completing ASM queries over 900% faster than the CPU from 8000 to 10000 search term characters.

However, as can be seen in *figure 3.7.1*, it appears as though the percentage difference between the time taken for each device is logarithmically corelated. This means that while the performance increase achieved by using the GPU increases rapidly for smaller STSs, it begins to stagnate for larger STSs. In my case, the stagnation occurred at around 5000 to 8000 search term characters with a percentage difference of around 950% to 1100%.

However, in real-world applications, STS values realistically never reach the larger experimental values I used. For example, according to WolframAlpha, the average length of an English word is 5.1 characters. Hence, for an English Spellchecker that uses ASM, you would expect the Average STS to be 5.1 as well. We saw in *figure 3.4.1* that with an STS of 5, we can expect a percentage difference of 154% or a 76ms decrease in time taken from CPU to GPU. While a performance increase is achieved, it is almost negligible due to its magnitude.

One final consideration is that because the Search Term and the Dictionary's Terms are often interrelated (If the Search Term is an English word, then the Dictionary is likely to be a Dictionary of English words, for example.), we may also conclude that both variables (STS and Average Dictionary Term Size) will vary the performance of an ASM query similarly. Nevertheless, while there was no space to do so in this paper, it might be worth conducting an experiment to see the performance impact of increasing both variables or solely Average Dictionary Term Size.

4.2 Effect of varying Dictionary Size

As seen in the experiment conducted and graph in sections *3.6* and *3.6.2*, the relationship between Dictionary Size and average time taken to complete an AQM query is exponentially corelated. This means that a lot more time is required per ASM query as Dictionary Size increases.

Up to around 50-100 terms, we can see that the GPU is actually taking more average time to complete a query than the CPU. This is likely to be the result of GPU overhead.

Looking at *figure 3.7.2*, the corelation is somewhat indeterminate. It would be advisable to gather more data past 50000 terms to verify a corelation. Based on the data gathered, it appears as though an exponential corelation is most fitting. If this is correct, then it means that the GPU results in an increasing performance increase from the CPU as the Dictionary Size increases. While this is a very positive realization, it is cancelled out by the exponential growth of the time taken per query which we've already seen in *figure 3.6.2* – while an increasing amount of time is saved from using the CPU, an also increasing amount of time is taken to process a query.

I believe a developer wanting to optimize their ASM implementation should

prioritise reducing the size of the dictionary used in ASM. This can be done with methods such as Indexing and Suffix Trees, methods discussed in *section 2.1*. Further reading on those methods can be found in a paper by Dekel Tsur entitled 'Fast index for approximate string matching'.[10]

4.3 Comparison

Comparing the two variables used in this paper, Search Term Size (STS) and Dictionary Size (DS), we see that on the GPU, the DS generally had a very minimal effect on the time taken to complete a query up until about 20000 terms. Even at 20000 terms, a query took less than a second to complete, and took less time to complete than a query with an STS of 10000. Due to the theorized exponential growth of the percentage difference between the time taken on the CPU and GPU when Dictionary Size is varied, there was actually a quite small difference in performance between the CPU and GPU for the first couple tens of thousands of terms. However, after 20000 terms, the time required per query increased drastically. In contrast to the DS, the STS required a lot more time per query even at much smaller values.

In conclusion, the GPU consistently provided a performance increase over the CPU. It is clear that a Levenshtein's Distance based ASM algorithm does gain value and is faster when ran on the GPU. Looking at the algorithm itself, varying the Search Term Size (and likely the Average Dictionary Term Size as postulated at the end of section 4.1) for the most part had a lesser effect on the time taken to complete an ASM query than changing the Dictionary Size; Therefore, it's advisable for a developer to focus on optimizing the Dictionary Size first and foremost to improve the performance of their ASM application.

5 Extensions

5.1 Multithreading

Something to consider is that the CPU code I used was not made to utilize the multiple cores a modern CPU has. By utilizing only one core of the CPU, we are wasting a lot of power. I think an interesting thing to explore in an additional research paper would be the effect of multithreading on the performance of an ASM algorithm. This is a fairly important consideration due to the next point:

5.2 Server CPUs

Server CPUs often have many more cores available to them than home or desktop CPUs. The importance of this comes when we look at the previous point. If the CPU code takes advantage of the dozens or even hundreds of cores that a Server CPU may have, could it achieve better performance than a GPU at least with a small enough Dictionary Size or other variable? This is important as ASM algorithms may be implemented in the cloud. To save costs on GPUs for the cloud servers, an owner may prefer to only use the CPU, at which point, having an efficient, multithreaded CPU ASM algorithm would be highly beneficial.

5.3 Kernel Architecture

My final consideration is that coding on the GPU is a nuanced process. Contrary to CPU programming, there is a lot of freedom with how things are processed and how memory is handled. As such, there are more ways of optimizing algorithms to maximise the use of the thousands of cores present on the GPU. I believe exploring different manners of processing ASM on the GPU, and/or exploring how to best arrange the memory passed to the GPU may an interesting avenue for future research.

6 Appendix

6.1 References

1. R. Baeza-Yates, and G. Navarro, “A faster algorithm for approximate string matching,” *Combinatorial Pattern Matching (CPM’96), Jun-1996.* [Article]. Available: https://www.researchgate.net/publication/2437209_A_Faster_Algorithm_for_Approximate_String_Matching.
2. K. Kapil, R. Soni, A. Vyas, and Dr. A. Sinhal, “Importance of String Matching in Real World Problems,” *International Journal Of Engineering And Computer Science, 6-Jul-2014.* [Article]. Available: https://www.researchgate.net/publication/304305210_Importance_of_String_Matching_in_Real_World_Problems-.
3. H. Wilper, R. Knight, and J. Cohen, “Understanding the visualization of overhead and latency in NVIDIA Nsight Systems,” *NVIDIA Developer Blog*, 22-Apr-2021. [Online]. Available: https://developer.nvidia.com/blog/understanding-the-visualization-of-overhead-and-latency-in-nsight-systems/.
4. M. Safford, “How to buy the Right CPU: A guide for 2021,” *Tom's Hardware*, 01-Feb-2020. [Online]. Available: https://www.tomshardware.com/reviews/cpu-buying-guide,5643.html.
5. “SIMD (single Instruction Multiple Data),” *Tech-FAQ*. [Online]. Available: https://www.tech-faq.com/simd.html.
6. “Multiple Instruction, Multiple Data (MIMD),” *Techopedia.com*, 11-Jul-2014. [Online]. Available: https://www.techopedia.com/definition/3479/multiple-instruction-multiple-data-mimd.
7. K. Balhaf, M. Shehab, W. Al-Sarayrah, M. Al-Ayyoub, M. Al-Saleh, and Y. Jararweh, “Using GPUs to Speed-Up Levenshtein Edit Distance Computation,” *International Conference on Information and Communication Systems (ICICS), 2016.* [Article]. Available: https://www.researchgate.net/publication/300042590_Using_GPUs_to_Speed-Up_Levenshtein_Edit_Distance_Computation.
8. “Understanding kernels, work-groups and work-items,” *TI OpenCL User's Guide*, 2018. [Online]. Available:

https://downloads.ti.com/mctools/esd/docs/opencl/execution/kernels-workgroups-workitems.html.

9. "NVIDIA GeForce RTX 3060 Ti," *TechPowerUp*. [Online]. Available: https://www.techpowerup.com/gpu-specs/geforce-rtx-3060-ti.c3681.
10. D. Tsur, "Fast index for approximate string matching," *Journal of Discrete Algorithms*, vol. 8, no. 4, pp. 339–345, 2010.

6.2 General Code

Here you will find most java classes used in gathering data for this paper.

Main.java

```
import Compute.ComputeProgram;
import Compute.GPU;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.nio.FloatBuffer;
import java.nio.IntBuffer;
import java.nio.charset.StandardCharsets;
import java.util.ArrayList;
import java.util.Scanner;

import static org.lwjgl.opencl.CL10.*;

public class Main {

    static Scanner Input;
    static String[] WordList;

    static int LongestWordLength;
    static IntBuffer DistanceMatricesBuffer;
    static IntBuffer BaseBuffer;
    static IntBuffer BaseSizeBuffer;
    static IntBuffer LongestWordLenBuffer;

    static boolean QueryGPU = true;
    static boolean QueryCPU = true;
    static boolean PrintGPU = false;
    static boolean PrintCPU = false;
    static boolean HeadrGPU = false;
    static boolean HeadrCPU = false;
    static boolean TimesGPU = false;
    static boolean TimesCPU = false;
    static boolean Manual = false;

    private static String[] LoadWordList(String List) {
        long FunctionStartTimer = System.nanoTime();
        ArrayList<String> WordList = new ArrayList<>();
        InputStream IS =
Main.class.getClassLoader().getResourceAsStream(List);
        try (BufferedReader BR = new BufferedReader( new
InputStreamReader(IS, StandardCharsets.UTF_8))) {
            String Line;
            while ((Line = BR.readLine()) != null) {
                WordList.add(Line);
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
        String[] WordListArray = new String[WordList.size()];

        for (String Word : WordList) {
            if (Word.length() > LongestWordLength) {
```

```
                LongestWordLength = Word.length();
            }
        }
        long FunctionTimeTaken1 = (System.nanoTime() - FunctionStartTimer);
        System.out.printf("Load Time (Word List + GPU Buffer): %dns
/ %dms %n", FunctionTimeTaken1, FunctionTimeTaken1 / 1000000);

        DistanceMatricesBuffer = GPU.Stack.callocInt(
                (LongestWordLength+1) * (LongestWordLength+1) //Matrix
Dimensions
                        * WordList.size() //Multiplied by the number of words
that will need a matrix
        );

        BaseBuffer = GPU.Stack.callocInt(LongestWordLength *
WordList.size());
        for (String s : WordList) {
            for (int j = 0; j < LongestWordLength; j++) {
                if (j < s.length()) {
                    BaseBuffer.put(s.charAt(j));
                } else {
                    BaseBuffer.put(0);
                }
            }
        }
        BaseBuffer.position(0);

        BaseSizeBuffer = GPU.Stack.callocInt(1);
        BaseSizeBuffer.put(0, WordList.size());

        LongestWordLenBuffer = GPU.Stack.callocInt(1);
        LongestWordLenBuffer.put(0, LongestWordLength);

        long FunctionTimeTaken2 = (System.nanoTime() - FunctionStartTimer);
        System.out.printf("Load Time (Word List + GPU Buffer): %dns
/ %dms %n", FunctionTimeTaken2, FunctionTimeTaken2 / 1000000);
        return WordList.toArray(WordListArray);
    }

    public static void QueryCPULev(String SearchTerm) {
        long FunctionStartTimer = System.nanoTime();
        SearchTree LevenshteinTree = new SearchTree(
                new LevenshteinData("#", 7.5f)
        );

        SearchTerm = "#" + SearchTerm.toLowerCase();

        for (String Base : WordList) {
            Levenshtein.Calculate(Base, SearchTerm, LevenshteinTree);
        }

        long FunctionTimeTaken1 = System.nanoTime() - FunctionStartTimer;
        if (TimesCPU) System.out.printf("%nCPU Query: %dns / %dms %n%n",
FunctionTimeTaken1, FunctionTimeTaken1 / 1000000);
        if (PrintCPU) {
            LevenshteinTree.PrintInorder(LevenshteinTree, new int[] {0,
5});
        }
    }

    public static void QueryGPULev(String SearchTerm) {
```

```
        long FunctionStartTimer = System.nanoTime();

        SearchTerm = SearchTerm.toLowerCase();

        //region Search Term
        IntBuffer SearchTermBuffer =
GPU.Stack.callocInt((SearchTerm).length());
        for (char CurrentChar : SearchTerm.toCharArray()) {
            SearchTermBuffer.put(CurrentChar);
        }
        SearchTermBuffer.position(0);
        //endregion

        //region Search Term Length
        IntBuffer SearchTermLengthBuffer = GPU.Stack.callocInt(1);
        SearchTermLengthBuffer.put(0, SearchTerm.length());
        //endregion

        FloatBuffer OutBuffer = GPU.Stack.callocFloat(WordList.length);

        ComputeProgram SolverProgram =
GPU.Programs.get("LevenshteinSolver");

        int Flags = CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR;
        SolverProgram.CreateWriteIntBuffer(0, SearchTermBuffer, Flags);
        SolverProgram.CreateWriteIntBuffer(1, SearchTermLengthBuffer,
Flags);
        SolverProgram.CreateWriteIntBuffer(2, BaseBuffer, Flags);
        SolverProgram.CreateWriteIntBuffer(3, BaseSizeBuffer, Flags);
        SolverProgram.CreateWriteIntBuffer(4, LongestWordLenBuffer, Flags);

        SolverProgram.CreateIntBuffer(5, DistanceMatricesBuffer, Flags);
        SolverProgram.CreateFloatBuffer(6, OutBuffer, Flags);

        SolverProgram.Dimensions = 1;
        SolverProgram.GlobalSize = WordList.length;

        long FunctionTimeTaken1 = System.nanoTime() - FunctionStartTimer;
        if (TimesGPU) System.out.printf("%nGPU Query (Preprocessing): %dns
/ %dms %n", FunctionTimeTaken1, FunctionTimeTaken1 / 1000000);

        SolverProgram.AutoSetKernelArgs();
        SolverProgram.AutoEnqueue1D();
        SolverProgram.ReadFloatBuffer(6, OutBuffer);
        long FunctionTimeTaken2 = System.nanoTime() - FunctionStartTimer;
        if (TimesGPU) System.out.printf("GPU Query (Processing): %dns
/ %dms %n", FunctionTimeTaken2 - FunctionTimeTaken1, (FunctionTimeTaken2 -
FunctionTimeTaken1) / 1000000);

        SearchTree LevenshteinTree = new SearchTree(
                new LevenshteinData("#", 7.5f)
        );

        int TermIndex = 0;
        while (OutBuffer.hasRemaining()) {
            LevenshteinTree.Insert(new SearchTree(
                    new LevenshteinData(WordList[TermIndex++],
OutBuffer.get())
            ));
        }
        long FunctionTimeTaken3 = System.nanoTime() - FunctionStartTimer;
```

```
        if (TimesGPU) System.out.printf("GPU Query (Postprocessing): %dns
/ %dms %n", FunctionTimeTaken3 - FunctionTimeTaken2, (FunctionTimeTaken3 -
FunctionTimeTaken2) / 1000000);
        if (TimesGPU) System.out.printf("GPU Query (Total): %dns
/ %dms %n", FunctionTimeTaken3, FunctionTimeTaken3 / 1000000);

        if (PrintGPU) {
            LevenshteinTree.PrintInorder(LevenshteinTree, new int[] {0,
5});
        }
    }

    public static void main(String[] args) {
        GPU.Init();

        GPU.AddProgram(
                "LevenshteinSolver",

Main.class.getClassLoader().getResourceAsStream("LevenshteinSolver.cl")
        );

        WordList = LoadWordList("RL/RandomNumberFileSize=10000.txt");

        if (Manual) {
            Input = new Scanner(System.in);

            while (QueryCPU && QueryGPU) {
                System.out.println("Enter Search Term: ");
                String SearchTerm = Input.nextLine();

                if (QueryGPU) {
                    if (HeadrGPU) System.out.println("GPU: ");
                    QueryGPULev(SearchTerm);
                    if (HeadrCPU) System.out.println();
                }

                if (QueryCPU) {
                    if (HeadrCPU) System.out.println("CPU: ");
                    QueryCPULev(SearchTerm);
                    System.out.println();
                }
            }
        }
        else {
            System.out.println("GPU:");
            Tests.DictionarySizeGPU();
            System.out.println();
            System.out.println("CPU:");
            Tests.DictionarySizeCPU();
        }
        GPU.Dispose();
    }

}
```

End of Main.java

Tests.java

```
import java.util.Random;

public class Tests {
    public static String GenRandString(int Length) {
        Random R = new Random();
        StringBuilder Result = new StringBuilder();
        for (int i = 0; i < Length; i++) {
            Result.append((char)(int)(97 + R.nextFloat() * 25));
        }
        return Result.toString();
    }

    //Testing the speed of different search term sizes
    //Range: 1, 3, 5, 7, 10, 25, 50, 75, 100, 500, 1000, 2000, 5000, 10000
    //Repeats: 30?
    public static void SearchTermSize() {
        //Warmup
        Main.QueryCPULev(GenRandString(10));

        int Repeats = 30;
        int[] SearchTermSizes = new int[]
{1,5,10,25,50,75,100,250,500,750,1000,2000,3000,4000,5000,6000,7000,8000,90
00,10000};
        System.out.println("Speed of different search term sizes");
        System.out.println();
        for (int SearchTermSize : SearchTermSizes) {
            String[] TestSearchTerms = new String[Repeats];
            for (int i = 0; i < Repeats; i++) {
                TestSearchTerms[i] = GenRandString(SearchTermSize);
            }

            long SuperTotalTime = System.nanoTime();
            long TotalTime = 0;
            for (String TestSearchTerm : TestSearchTerms) {
                long IndividualTime = System.nanoTime();
                //There is only one function for this test for both devices
                //The function below was manually changed from CPU to GPU
or vice versa to test each device
                Main.QueryCPULev(TestSearchTerm);
                TotalTime += System.nanoTime() - IndividualTime;
            }

            System.out.printf("%nResults for search term size of %d%n",
SearchTermSize);
            System.out.printf("Overall Time Taken: %dns / %dms %n",
System.nanoTime() - SuperTotalTime, (System.nanoTime() - SuperTotalTime) /
1000000);
            System.out.printf("Average Time Taken: %dns / %dms %n",
(TotalTime / Repeats), (TotalTime / Repeats) / 1000000);

        }
    }

    //Simply run a query for a dataset 30 times
    public static void DictionarySizeGPU() {
        int Repeats = 30;

        long SuperTotalTime = System.nanoTime();
        long TotalTime = 0;
```

```
        for (int R = 0; R < Repeats; R++) {
            String SearchTerm = GenRandString(15);

            long ITime = System.nanoTime();
            Main.QueryGPULev(SearchTerm);
            TotalTime += System.nanoTime() - ITime;
        }
        System.out.printf("%nResults for current dataset %n");
        System.out.printf("Overall Time Taken: %dns / %dms %n",
System.nanoTime() - SuperTotalTime, (System.nanoTime() - SuperTotalTime) /
1000000);
        System.out.printf("Average Time Taken: %dns / %dms %n", (TotalTime
/ Repeats), (TotalTime / Repeats) / 1000000);
    }

    public static void DictionarySizeCPU() {
        int Repeats = 30;

        long SuperTotalTime = System.nanoTime();
        long TotalTime = 0;
        for (int R = 0; R < Repeats; R++) {
            String SearchTerm = GenRandString(15);

            long ITime = System.nanoTime();
            Main.QueryCPULev(SearchTerm);
            TotalTime += System.nanoTime() - ITime;
        }
        System.out.printf("%nResults for current dataset %n");
        System.out.printf("Overall Time Taken: %dns / %dms %n",
System.nanoTime() - SuperTotalTime, (System.nanoTime() - SuperTotalTime) /
1000000);
        System.out.printf("Average Time Taken: %dns / %dms %n", (TotalTime
/ Repeats), (TotalTime / Repeats) / 1000000);
    }

}
```

End of Tests.java

SearchTree.java

```
public class SearchTree {

    public SearchTree Less;
    public LevenshteinData Data;
    public SearchTree More;

    public SearchTree(LevenshteinData DataIn) {
        Data = DataIn;
    }

    public void Insert(SearchTree Node) {
        String Direction;

        if (Node.Data.Score > Data.Score) {
            Direction = "More";
        }
        else {
            Direction = "Less";
        }

        if (Direction.equals("More")) {
            if (More == null) {
                More = Node;
            }
            else {
                More.Insert(Node);
            }
        }
        else if (Direction.equals("Less")) {
            if (Less == null) {
                Less = Node;
            }
            else {
                Less.Insert(Node);
            }
        }
        else {
            throw new IllegalArgumentException();
        }
    }

    public void PrintInorder(SearchTree Node, int[] Limit) {
        if (Node == null) {
            return;
        }
        else {
            if (Limit[0] >= Limit[1]) {
                return;
            }
            PrintInorder(Node.Less, Limit);
            if (Limit[0] < Limit[1]) {
                if (!Node.Data.Word.equals("#")) {
                    Node.Data.Out();
                }
                else {
                    Limit[0]--;
                }
            }
            Limit[0]++;
```

```
            PrintInorder(Node.More, Limit);
        }
    }

}
```

End of SearchTree.java

6.3 CPU Specific Code

Here you will find most classes that were exclusively required by the CPU.

Levenshtein.java

```
import static java.lang.Math.max;
import static java.lang.Math.min;

public class Levenshtein {
    public static void Calculate(String BaseTerm, String SearchTerm,
SearchTree Tree) {
        String OperatingOriginalString = "#" + BaseTerm;

        int[][] Matrix = new
int[ OperatingOriginalString.length() ][ SearchTerm.length() ];

        for (int y = 0; y < Matrix.length; y++) {
            for (int x = 0; x < Matrix[0].length; x++) {

                if (min(x, y) == 0) {
                    Matrix[y][x] = max(x, y);
                }
                else {
                    int Term1 = Matrix[y - 1][x] + 1;
                    int Term2 = Matrix[y][x - 1] + 1;
                    int Term3 = Matrix[y - 1][x - 1];
                    if (OperatingOriginalString.charAt(y) !=
SearchTerm.charAt(x)) {
                        Term3++;
                    }

                    Matrix[y][x] = min(Term1, min(Term2, Term3));
                }

            }
        }

        int Distance = Matrix[OperatingOriginalString.length() -
1][SearchTerm.length() - 1];
        int TotalLen = (OperatingOriginalString.length() - 1) +
(SearchTerm.length() - 1);

        float Ratio = (float)(TotalLen - Distance) / (float)TotalLen;
        float Score = (float)Distance + (Distance == 0 ? 0 : Ratio);

        Tree.Insert(new SearchTree(
                new LevenshteinData(BaseTerm, Score)
        ));
    }

    public static void CalculateTreeless(String Original, String Search) {
        String OperatingOriginalString = "#" + Original;

        int[][] Matrix = new
int[ OperatingOriginalString.length() ][ Search.length() ];

        for (int y = 0; y < Matrix.length; y++) {
```

```
            for (int x = 0; x < Matrix[0].length; x++) {

                if (min(x, y) == 0) {
                    Matrix[y][x] = max(x, y);
                }
                else {
                    int Term1 = Matrix[y - 1][x] + 1;
                    int Term2 = Matrix[y][x - 1] + 1;
                    int Term3 = Matrix[y - 1][x - 1];
                    if (OperatingOriginalString.charAt(y) !=
Search.charAt(x)) {
                        Term3++;
                    }

                    Matrix[y][x] = min(Term1, min(Term2, Term3));
                }

            }
        }
        int Distance = Matrix[OperatingOriginalString.length() -
1][Search.length() - 1] - 1;
        System.out.println(Distance);
    }
}
```

End of Levenshtein.java

LevenshteinData.java

```
public class LevenshteinData {
    public String Word;
    public float Score;

    public LevenshteinData(String WordIn, float ScoreIn) {
        Word = WordIn;
        Score = ScoreIn;
    }

    public void Out() {
        char[] WordWithCapitalizedFirstLetter = Word.toCharArray();
        WordWithCapitalizedFirstLetter[0] = Word.toUpperCase().charAt(0);
        System.out.printf("%s - %.2f%n", new
String(WordWithCapitalizedFirstLetter), Score);
    }
}
```

End of LevenshteinData.java

6.4 GPU-Specific Code

Here you will find Java classes that connect or run GPU code.

ComputeProgram.java

```
package Compute;

import java.io.*;
import java.nio.Buffer;
import java.nio.ByteBuffer;
import java.nio.IntBuffer;
import java.nio.charset.StandardCharsets;
import java.nio.file.Files;
import java.nio.file.Paths;

import org.lwjgl.PointerBuffer;

import java.nio.FloatBuffer;
import java.util.HashMap;

import static Compute.GPU.*;
import static Compute.InfoUtil.checkCLError;
import static org.lwjgl.opencl.CL10.*;
import static org.lwjgl.system.MemoryUtil.*;

import org.apache.commons.io.*;

public class ComputeProgram {

    //Programs must have the same filename as the kernel's name!!!!
    public long Kernel;

    private static String LoadSource(String Path) throws IOException {
        return Files.readString(Paths.get(Path));
    }

    public ComputeProgram(long Context, String SourcePath) {
        String Source = null;
        try {
            Source = LoadSource(SourcePath);
        } catch (IOException E) {
            System.err.println("Could not read Program at: " + SourcePath +
" !!");
            return;
        }
        String SourceFilename = FilenameUtils.getBaseName(SourcePath);

        long Program = clCreateProgramWithSource(Context, Source, null);

        BuildProgram(Program);

        Kernel = clCreateKernel(Program, SourceFilename, ErrorcodeReturn);
        checkCLError(ErrorcodeReturn);
        clReleaseProgram(Program);
    }

    public ComputeProgram(long Context, InputStream IS, String ProgramName)
{
        StringBuilder Source = new StringBuilder();
```

```
        try (BufferedReader BR = new BufferedReader( new
InputStreamReader(IS, StandardCharsets.UTF_8))) {
            String Line;
            while ((Line = BR.readLine()) != null) {
                Source.append(Line);
            }
        } catch (IOException e) {
            System.err.println("Could not read Program named: " +
ProgramName + " !!");
            return;
        }

        long Program = clCreateProgramWithSource(Context,
Source.toString(), null);

        BuildProgram(Program);

        Kernel = clCreateKernel(Program, ProgramName, ErrorcodeReturn);
        checkCLError(ErrorcodeReturn);
        clReleaseProgram(Program);
    }

    private void BuildProgram(long Program) {
        if (clBuildProgram(Program, GPU.Device, "", null, NULL) !=
CL_SUCCESS) {

            ByteBuffer BuildLog = ByteBuffer.allocateDirect(500000);
            PointerBuffer BuildLogSize = PointerBuffer.allocateDirect(1);

            if ( clGetProgramBuildInfo(Program, Device,
CL_PROGRAM_BUILD_LOG, BuildLog, BuildLogSize) == CL_SUCCESS ) {
                System.out.println(BuildLogSize.get(0));
                byte[] BuildLogBA = new byte[(int)BuildLogSize.get(0)];
BuildLog.get(BuildLogBA);
                try {
                    System.out.println(new String(BuildLogBA, "UTF-8"));
                } catch (UnsupportedEncodingException e) {
                    e.printStackTrace();
                }
            }

        }
    }

    public int GlobalSize = 1;
    public int LocalSize = Integer.MIN_VALUE;
    public int Dimensions = 1;

    public HashMap<Integer,Long> MemoryObjects = new
HashMap<Integer,Long>();

    public void CreateFloatBuffer(int ArgumentIndex, FloatBuffer Capacity,
int Flags) {
        long MemoryObject = clCreateBuffer(Context, Flags, Capacity,
ErrorcodeReturn);
        checkCLError(ErrorcodeReturn);
        MemoryObjects.put(ArgumentIndex, MemoryObject);
    }

    public void WriteFloatBuffer(int ArgumentIndex, FloatBuffer Data) {
        checkCLError(clEnqueueWriteBuffer(CommandQueue,
```

```
MemoryObjects.get(ArgumentIndex), true, 0, Data, null, null));
    }

    public void CreateIntBuffer(int ArgumentIndex, IntBuffer Capacity, int
Flags) {
        long MemoryObject = clCreateBuffer(Context, Flags, Capacity,
ErrorcodeReturn);
        checkCLError(ErrorcodeReturn);
        MemoryObjects.put(ArgumentIndex, MemoryObject);
    }

    public void WriteIntBuffer(int ArgumentIndex, IntBuffer Data) {
        checkCLError(clEnqueueWriteBuffer(CommandQueue,
MemoryObjects.get(ArgumentIndex), true, 0, Data, null, null));
    }

    public void CreateWriteFloatBuffer(int ArgumentIndex, FloatBuffer
InitialData, int Flags) {
        CreateFloatBuffer(ArgumentIndex, InitialData, Flags);
        WriteFloatBuffer(ArgumentIndex, InitialData);
    }

    public void CreateWriteIntBuffer(int ArgumentIndex, IntBuffer
InitialData, int Flags) {
        CreateIntBuffer(ArgumentIndex, InitialData, Flags);
        WriteIntBuffer(ArgumentIndex, InitialData);
    }

    public void AutoSetKernelArgs() {
        MemoryObjects.forEach((ArgumentIndex, MemObject) -> {
            //System.out.println(Kernel + " " + ArgumentIndex + " " +
MemObject);
            clSetKernelArg1p(Kernel, ArgumentIndex, MemObject);
        });
    }

    public void AutoEnqueue1D() {
        PointerBuffer GlobalWorksizeBuffer = GPU.Stack.callocPointer(1);
        GlobalWorksizeBuffer.put(0, GlobalSize);

        PointerBuffer LocalWorksizeBuffer = GPU.Stack.callocPointer(1);
        LocalWorksizeBuffer.put(0, LocalSize);

        PointerBuffer KernelEvent = GPU.Stack.callocPointer(1);

        clEnqueueNDRangeKernel(
                CommandQueue,
                Kernel,
                Dimensions,
                null,
                GlobalWorksizeBuffer,
                LocalSize == Integer.MIN_VALUE ? null :
LocalWorksizeBuffer,
                null,
                KernelEvent
        );

        clWaitForEvents(KernelEvent);
    }

    public void ReadIntBuffer(int OutputArgumentIndex, IntBuffer Buffer) {
```

```
        clEnqueueReadBuffer(
                CommandQueue,
                MemoryObjects.get(OutputArgumentIndex),
                true,
                0,
                Buffer,
                null,
                null
        );
    }

    public void ReadFloatBuffer(int OutputArgumentIndex, FloatBuffer
Buffer) {
        clEnqueueReadBuffer(
                CommandQueue,
                MemoryObjects.get(OutputArgumentIndex),
                true,
                0,
                Buffer,
                null,
                null
        );
    }
}
```

End of ComputeProgram.java

GPU.java

```
package Compute;

import org.lwjgl.PointerBuffer;
import org.lwjgl.opencl.CL;
import org.lwjgl.opencl.CLCapabilities;
import org.lwjgl.opencl.CLContextCallback;
import org.lwjgl.system.MemoryStack;

import java.io.InputStream;
import java.nio.IntBuffer;
import java.util.HashMap;

import static Compute.InfoUtil.checkCLError;
import static org.lwjgl.opencl.CL10.*;
import static org.lwjgl.opencl.CL11.CL_DEVICE_OPENCL_C_VERSION;
import static org.lwjgl.system.MemoryUtil.*;

public class GPU {

    public static MemoryStack Stack;
    public static IntBuffer ErrorcodeReturn;

    public static long Platform;
    public static long Device;

    private static CLCapabilities PlatformCapabilities;
    private static CLCapabilities DeviceCapabilities;

    protected static long Context;
    private static CLContextCallback ContextCallback;

    public static long CommandQueue;

    public static HashMap<String,ComputeProgram> Programs = new
HashMap<String,ComputeProgram>();

    public static void AddProgram(String Name, String Path) {
        ComputeProgram Program = new ComputeProgram(
                Context,
                Path
        );
        Programs.put(Name, Program);
    }

    public static void AddProgram(String Name, InputStream IS) {
        ComputeProgram Program = new ComputeProgram(
                Context,
                IS,
                Name
        );
        Programs.put(Name, Program);
    }


    private static void GetPlatformAndDevice(MemoryStack Stack) {
        IntBuffer PlatformCount = Stack.mallocInt(1);
        PointerBuffer AvailablePlatforms = Stack.mallocPointer(1);

        checkCLError(
```

```
                clGetPlatformIDs(AvailablePlatforms, PlatformCount) //2nd
Arg nmly (IntBuffer)null
        );
        if (PlatformCount.get(0) == 0) {
            throw new RuntimeException("No OpenCL platforms found.");
        }

        Platform = AvailablePlatforms.get(0);
        PlatformCapabilities = CL.createPlatformCapabilities(Platform);

        IntBuffer DeviceCount = Stack.mallocInt(1);
        PointerBuffer AvailableDevices = Stack.mallocPointer(1);
        checkCLError(
                clGetDeviceIDs(Platform, CL_DEVICE_TYPE_ALL,
AvailableDevices, DeviceCount)
        );
        if (DeviceCount.get(0) == 0) {
            throw new RuntimeException("No OpenCL devices found.");
        }

        Device = AvailableDevices.get(0);
        DeviceCapabilities = CL.createDeviceCapabilities(Device,
PlatformCapabilities);
    }

    private static PointerBuffer GetContextProperties(MemoryStack Stack) {
        PointerBuffer ContextProperties = Stack.mallocPointer(3);

        ContextProperties
                .put(0, CL_CONTEXT_PLATFORM)
                .put(1, Platform)
                .put(2, 0)
        ;

        return ContextProperties;
    }

    private static void GetContextCallback() {
        ContextCallback = CLContextCallback.create((errinfo, private_info,
cb, user_data) -> {
            System.err.println("[LWJGL] cl_context_callback");
            System.err.println("\tInfo: " + memUTF8(errinfo));
        });
    }

    public static void PrintInfo() {
        StringBuilder SB = new StringBuilder();

        SB.append("Compute Device successfully initialized!" + '\n' +
'\n');

        SB.append("Compute Device Information:" + '\n');

        SB.append("Name: " + InfoUtil.getDeviceInfoStringUTF8(Device,
CL_DEVICE_NAME) + '\n');

        SB.append("Type: ");
        long Type = InfoUtil.getDeviceInfoLong(Device, CL_DEVICE_TYPE);
        switch ((int)Type) {
            case CL_DEVICE_TYPE_GPU:
                SB.append("GPU" + '\n'); break;
```

```
            case CL_DEVICE_TYPE_CPU:
                SB.append("CPU" + '\n'); break;
            default:
                SB.append("Unknown" + '\n'); break;
        }

        SB.append("Device Version: " +
InfoUtil.getDeviceInfoStringUTF8(Device, CL_DEVICE_VERSION) + '\n');

        SB.append("Language Version: " +
InfoUtil.getDeviceInfoStringUTF8(Device, CL_DEVICE_OPENCL_C_VERSION) +
'\n');

        SB.append("Maximum Compute Units: " +
InfoUtil.getDeviceInfoInt(Device, CL_DEVICE_MAX_COMPUTE_UNITS) + " Units" +
'\n');

        SB.append("Maximum Workitem Dimension: " +
InfoUtil.getDeviceInfoInt(Device, CL_DEVICE_MAX_WORK_ITEM_DIMENSIONS) + 'D'
+ '\n');

        SB.append("Maximum Workgroup Size: " +
InfoUtil.getDeviceInfoLong(Device, CL_DEVICE_MAX_WORK_GROUP_SIZE) + '\n');

        SB.append("Maximum Clock Frequency: " +
InfoUtil.getDeviceInfoInt(Device, CL_DEVICE_MAX_CLOCK_FREQUENCY) + " Hz" +
'\n');

        SB.append("Allocated Stack Size: " + Stack.getSize() + " Bytes" +
'\n');

        System.out.println(SB);

    }

    public static void Init() {
        Stack = MemoryStack.create(500_000_000); //500 MB

        ErrorcodeReturn = Stack.callocInt(1);

        GetPlatformAndDevice(Stack);

        GetContextCallback();

        Context = clCreateContextFromType(GetContextProperties(Stack),
CL_DEVICE_TYPE_GPU, null, 0, ErrorcodeReturn);
        checkCLError(ErrorcodeReturn);

        CommandQueue = clCreateCommandQueue(Context, Device,
CL_QUEUE_PROFILING_ENABLE, ErrorcodeReturn);
        checkCLError(ErrorcodeReturn);

        PrintInfo();
    }

    public static void Dispose() {
        Programs.forEach((Name, Program) -> {
            Program.MemoryObjects.forEach((Key, MemObject) -> {
                clReleaseMemObject(MemObject);
            });
            Program.MemoryObjects.clear();
```

```
            clReleaseKernel(Program.Kernel);
        });
        Programs.clear();

        clReleaseCommandQueue(CommandQueue);
        clReleaseContext(Context);
        CL.destroy();

        try {
            Stack.close();
        } catch (Exception E) {}
    }
}
```

End of GPU.java

InfoUtil.java

```
package Compute;

import org.lwjgl.*;
import org.lwjgl.system.*;

import java.nio.*;

import static org.lwjgl.opencl.CL10.*;
import static org.lwjgl.system.MemoryStack.*;
import static org.lwjgl.system.MemoryUtil.*;

/** OpenCL object info utilities. */
public final class InfoUtil {

    private InfoUtil() {
    }

    public static void PrintDeviceInfo(long Device) {
        System.out.println();
        System.out.println("Device Name: " +
InfoUtil.getDeviceInfoStringUTF8(Device, CL_DEVICE_NAME));
        //region Get Device Type String
        long DeviceType = InfoUtil.getDeviceInfoLong(Device,
CL_DEVICE_TYPE);
        String DeviceTypeString = "";
        if (DeviceType == CL_DEVICE_TYPE_GPU) {
            DeviceTypeString = "GPU";
        }
        else if (DeviceType == CL_DEVICE_TYPE_CPU) {
            DeviceTypeString = "CPU";
        }
        else {
            DeviceTypeString = "Unknown";
        }
        //endregion
        System.out.println("Device Type: " + DeviceTypeString);
        System.out.println();
        System.out.println("Compute Units: " +
InfoUtil.getDeviceInfoInt(Device, CL_DEVICE_MAX_COMPUTE_UNITS));
        System.out.println("At Frequency: " +
InfoUtil.getDeviceInfoInt(Device, CL_DEVICE_MAX_CLOCK_FREQUENCY));
        System.out.println();
        System.out.println("Local Memory: " +
InfoUtil.getDeviceInfoLong(Device, CL_DEVICE_LOCAL_MEM_SIZE));
        System.out.println("Global Memory: " +
InfoUtil.getDeviceInfoLong(Device, CL_DEVICE_GLOBAL_MEM_SIZE));
        System.out.println();
    }

    public static String getPlatformInfoStringASCII(long cl_platform_id,
int param_name) {
        try (MemoryStack stack = stackPush()) {
            PointerBuffer pp = stack.mallocPointer(1);
            checkCLError(clGetPlatformInfo(cl_platform_id, param_name,
(ByteBuffer)null, pp));
            int bytes = (int)pp.get(0);

            ByteBuffer buffer = stack.malloc(bytes);
            checkCLError(clGetPlatformInfo(cl_platform_id, param_name,
```

```
buffer, null));

            return memASCII(buffer, bytes - 1);
        }
    }

    public static String getPlatformInfoStringUTF8(long cl_platform_id, int
param_name) {
        try (MemoryStack stack = stackPush()) {
            PointerBuffer pp = stack.mallocPointer(1);
            checkCLError(clGetPlatformInfo(cl_platform_id, param_name,
(ByteBuffer)null, pp));
            int bytes = (int)pp.get(0);

            ByteBuffer buffer = stack.malloc(bytes);
            checkCLError(clGetPlatformInfo(cl_platform_id, param_name,
buffer, null));

            return memUTF8(buffer, bytes - 1);
        }
    }

    public static int getDeviceInfoInt(long cl_device_id, int param_name) {
        try (MemoryStack stack = stackPush()) {
            IntBuffer pl = stack.mallocInt(1);
            checkCLError(clGetDeviceInfo(cl_device_id, param_name, pl,
null));
            return pl.get(0);
        }
    }

    public static long getDeviceInfoLong(long cl_device_id, int param_name)
{
        try (MemoryStack stack = stackPush()) {
            LongBuffer pl = stack.mallocLong(1);
            checkCLError(clGetDeviceInfo(cl_device_id, param_name, pl,
null));
            return pl.get(0);
        }
    }

    public static long getDeviceInfoPointer(long cl_device_id, int
param_name) {
        try (MemoryStack stack = stackPush()) {
            PointerBuffer pp = stack.mallocPointer(1);
            checkCLError(clGetDeviceInfo(cl_device_id, param_name, pp,
null));
            return pp.get(0);
        }
    }

    public static String getDeviceInfoStringUTF8(long cl_device_id, int
param_name) {
        try (MemoryStack stack = stackPush()) {
            PointerBuffer pp = stack.mallocPointer(1);
            checkCLError(clGetDeviceInfo(cl_device_id, param_name,
(ByteBuffer)null, pp));
            int bytes = (int)pp.get(0);

            ByteBuffer buffer = stack.malloc(bytes);
            checkCLError(clGetDeviceInfo(cl_device_id, param_name, buffer,
```

```
null));

            return memUTF8(buffer, bytes - 1);
        }
    }

    public static int getProgramBuildInfoInt(long cl_program_id, long
cl_device_id, int param_name) {
        try (MemoryStack stack = stackPush()) {
            IntBuffer pl = stack.mallocInt(1);
            checkCLError(clGetProgramBuildInfo(cl_program_id, cl_device_id,
param_name, pl, null));
            return pl.get(0);
        }
    }

    public static String getProgramBuildInfoStringASCII(long cl_program_id,
long cl_device_id, int param_name) {
        try (MemoryStack stack = stackPush()) {
            PointerBuffer pp = stack.mallocPointer(1);
            checkCLError(clGetProgramBuildInfo(cl_program_id, cl_device_id,
param_name, (ByteBuffer)null, pp));
            int bytes = (int)pp.get(0);

            ByteBuffer buffer = stack.malloc(bytes);
            checkCLError(clGetProgramBuildInfo(cl_program_id, cl_device_id,
param_name, buffer, null));

            return memASCII(buffer, bytes - 1);
        }
    }

    public static void checkCLError(IntBuffer errcode) {
        checkCLError(errcode.get(errcode.position()));
    }

    public static void checkCLError(int errcode) {
        if (errcode != CL_SUCCESS) {
            throw new RuntimeException(String.format("OpenCL error [%d]",
errcode));
        }
    }

}
```

End of InfoUtil.java

6.5 GPU Kernel

Here you will find the GPU Kernel which solves ASM queries.

LevenshteinSolver.cl

```
int F3D(int X, int XS, int Y, int Z, int ZS) { /*Flatten 3D Coordinates*/
    return X + XS * (Y + ZS * Z);
}

kernel void LevenshteinSolver(
    global const int *Term,
    global const int *SearchTermLen,
    global const int *Base,
    global const int *BaseSize,
    global const int *LongestBaseTermLen,
    global int *DistanceMatrices,
    global float *Output
    )
{
    int GI = get_global_id(0);
    int LTL = LongestBaseTermLen[0]+1;

    int CurrentBaseTermLen = 0;
    bool CurrentBaseTermLenFound = false;

    for (int i = 0; i < LTL; i++) {
        DistanceMatrices[F3D(i, LTL, 0, GI, LTL)] = i;
    }
    for (int j = 0; j < LTL; j++) {
        DistanceMatrices[F3D(0, LTL, j, GI, LTL)] = j;
    }

    for (int x = 1; x < SearchTermLen[0] + 1; x++) { /*For each character
in the search term*/
        int CurrentSearchChar = Term[x - 1];
        for (int y = 1; y < LTL; y++) { /*For each character in the base
assigned to this work item*/
            int CurrentBaseChar = Base[((GI * LTL-1) + y) - GI];

            if (CurrentBaseChar == 0) {
                break;
            }

            CurrentBaseTermLen += 1 * (!(CurrentBaseTermLenFound^0) &&
(CurrentBaseChar^0));

            if (min(x,y) == 0) {
                DistanceMatrices[F3D(x, LTL, y, GI, LTL)] = max(x,y);
            }
            else {
                int EqTerm1 = DistanceMatrices[F3D(x - 1, LTL, y, GI, LTL)]
+ 1;
                int EqTerm2 = DistanceMatrices[F3D(x, LTL, y - 1, GI, LTL)]
+ 1;
                int EqTerm3 = DistanceMatrices[F3D(x - 1, LTL, y - 1, GI,
LTL)];

                EqTerm3 += 1 * (bool)(CurrentSearchChar^CurrentBaseChar);
```

```
                DistanceMatrices[F3D(x, LTL, y, GI, LTL)] = min(EqTerm1,
min(EqTerm2, EqTerm3));
            }
        }
        CurrentBaseTermLenFound = true;
    }

    int Distance =
DistanceMatrices[F3D(SearchTermLen[0],LTL,CurrentBaseTermLen,GI,LTL)];
    int TotalLength = SearchTermLen[0] + CurrentBaseTermLen;
    float Ratio = ((float)(TotalLength - Distance) / TotalLength) *
(bool)(Distance^0);

    Output[GI] = Distance + Ratio;
}
```

End of LevenshteinSolver.cl