Zachary Job
Prof. Mordohai
CS 677
4/28/15

Project Report

Note: Metrics and Analysis are at the end

***INTRODUCTION***

The internet is an incredibly powerful resource. It provides a means to share information required to make a stock trade, to decide on which car to buy, or even just to check out what information is trending. The only issue with the web is finding data truly relevant to a specific query that searches data across multiple web sites and documents to find similarities and new information. Sometimes digging for this information isn't feasible. If the information could be targeted turned into a frequency model so much could be done. Document comparisons could be accelerated, trending information between resources could be immediately recognized, and the lesser known things could be discovered. Another application would be machine learning. Coupled with this software, it could weight decisions based on frequency to attempt and follow trends or evolving ones. This could be used to act on a decision either while it is popular, new, or even fading in interest. The application is very broad and can be applied elsewhere.

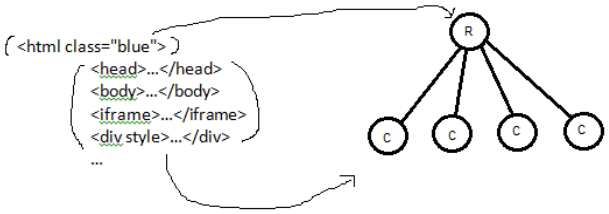
Imagine a CPU system that must count each occurrence of a word. At best, each thread can work to generate its own count into a hash map or histogram structure. Never mind the fact that the available thread count is typically small. At the end however, these structures must merge. By increasing the threads, contention can be inadvertently increased. Also, instead of fetching data the CPU is made busy with storing and organizing it. This can prove to be a serious drag on performance. A big requirement that would also lessen speed would be UTF 8 encoding. The web has migrated to this encoding and it definitely complicates the CPU end. Lastly, filtering the content would definitely kill the CPU. For example characters like ":" or "/" occur often. Worse than this, web data includes redundant information meant to indicate action by the system that stores the data. For example, the frequency of "|comment=" could be useless. It could also occur at a very high frequency. The list goes on. With the CPU already under heavy load there is no room to implement this on the CPU without killing efficiency.

Enter the GPU with thousands of threads. By using this processor of as a coprocessor contention could be eliminated on the CPU. It simply would have to feed the GPU information. For example, the CPU is reading characters and generates a key. Instead of placing this key in a frequency structure, the key is loaded into a buffer. When the buffer fills the GPU seamlessly launches and bins all the information in less than a millisecond. This can be done via threads binning without contention then

atomically updating the final map with loads of avoided work. The CPU may struggle to even feed the GPU so fast. This also frees up the CPU to filter the data. Now there is opportunity to make the software truly powerful. The user may now query that things be excluded. This allows frequencies of data relevant to the user to be neatly presented in the end data. To better this, loading the file into memory would allow for the GPU to handle the filtering as well. Since the GPU could bin faster than the CPU can feed, the GPU could read the file as well and bin while the CPU prepares to share its buffer. This can allow for massive speedups since two whole processing systems are maximizing their abilities to parallelize an immense amount of work.

***THE CPU VERSION***

This version was implemented in Java and for simple reasons. Java has powerful yet easy to implement libraries built in. Most importantly concurrent hash maps and thread pools. The system starts by using a DOM XML parser. This loads the file into memory as a tree structure. XML and Web documents are tag based. For example…



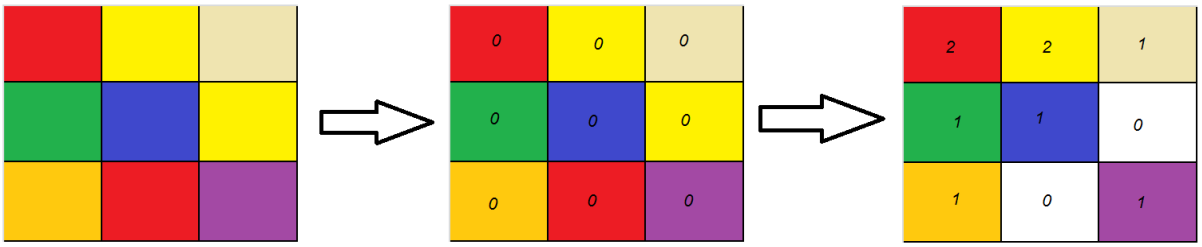

This would be loaded where the “<html class=”blue”>” becomes the root node and the following its children. To access elements this tree is traversed and any content of the nodes can exist as text making reading the document easy.

The next step in this software is the producer-consumer blocking queue. This allows access to enqueue and dequeue. Enqueues will not be blocked yet dequeues will if there is no more content. I wanted as many threads as possible consuming the document data. Since consumption is more costly, a thread pool would feed the document. A thread pool is a lightweight thread system where this task can be handled effectively. Instead of full-fledged threads performing a sequential enqueue, lighter instances can be used to feed the document. This would keep pace with consumption making it a logical move. The size of the thread pool may increase as consumer count decreases making optimization easy. If the consumer threads are reading faster than the pool can enqueue, a consumer can be eliminated to speed up the feed and typically performance. This can happen on weaker systems or with smaller documents as there is some overhead in traversing the tree structure in memory. This is due that the traversal is O(log(n)) and uses objects.

On the consumer side, things are relatively simple until the end. The queue contains tokens which can be characters or words. For each token, a hash-map's count is updated for its occurrence. Eventually the pool enqueues what is known as a "poison pill". This indicates that the feed is over with and the consumers must merge their hash-maps. The final hash map is a concurrent hash-map. This concurrent hash-map allows for bucket level atomics. As a thread attempts to update a bucket's count, it too must ensure it is not in conflict with another thread. Each consumer linearly updates this final map with their counts with only up to the virtually impossible case of the consumer thread count being the conflicts per every bucket. This is a highly efficient system, however it is about maximizing the CPU's potential. There is almost no room for improvement. One thing to keep in mind is that this can only handle ASCII without seriously lowering efficiency. This version only considers the possibility of reading characters per bit. UTF 8 is becoming the new standard and typically requires four bytes.

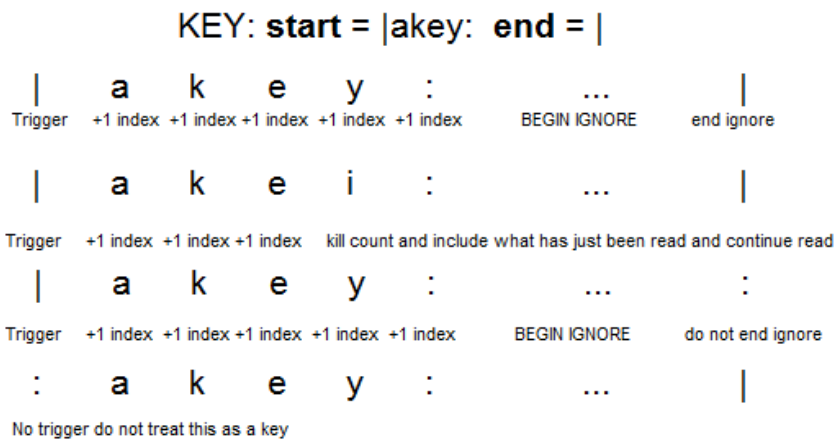
***THE GPU VERSION EVOLUTION***

To start, there was no GPU version to actually serve the purpose defined above. I determined that the CPU version needed to be well along so that I did not build something that could be obsoleted right off the bat. Instead I formulated the general idea. I determined that a sizeable thread block should be able to contention free reduce a block of results into keys and counts. This then could be shot off to the result histogram where each key location would be treated like the concurrent hash-map. This is due that each location in the histogram could be atomically locked by the GPU. I was concerned that this may not work fast enough so I performed basic tests. I used a dummy kernel where two dummy buffers of 9216 random integers were generated. This is due that my objective was to use shared memory enough to hide kernel launch overhead as much as possible. This consumed 36864 KB of 48000 KB available on the test platform discussed in metrics. 384 threads would each read 24 buffer elements linearly into shared memory. 384 locations would then be compared by 384 threads in a loop. Thread amounts were limited as the anti-contention method increases in complexity following the function 2^((threads - 1) * 2) * (total_elements / threads). This is due that a thread updates its occurrence count by reading the thread sized chunk in shared one element at a time. This operation also checks if it reads a same key previous to itself. In this case, a nullify bit is tripped killing that threads location in shared memory. This way I could flush threads trying to contend for the zeroth position in the histogram eliminating a great deal of contention.




The results were promising. I tested 384 thread increments against 192 and 160. I realized that 192 won out with a time of 0.8 milliseconds. The 384 thread version took 1.3 milliseconds and the 160 took about 0.81 milliseconds. I also tested this with a double buffer test against a single buffer test for two buffer loads. The change was not noticeable due that the CPU speed varied too much. I looped the operation to find that the double buffer system essentially hid the kernel operations. The single buffer system was the sum of CPU and GPU time. I expected this. However it was wise to ensure I understood how to implement the feature. An important note was that this kernel was considered to be my fallback plan. I had ambitions of moving much more onto the GPU. At this point I moved back to the work CPU side.

The CPU end is massive. I decided that the user must be able to configure their exclusion query, precompute it, and select exactly where to handle in a document. I also determined that C was required to eliminate any language overhead. See the metrics for a quick comparison. I ended up implementing the configuration customization, pre-computation, and simply included a tag amount system to read a desired amount of tags. There was no need for a more complex seek system since the goal was to test efficiency per document size. This was done via two memory maps, a map of offsets per index into the memory maps, a character trigger map, and a trimming array to lessen sizes and protect against UTF 8 character lookups since the data formats lucky for me use ASCII. The system is simple; a key has two dirty bits, the start and the end character. Running over a start character will trigger a flush that will also begin updating a token return buffer. Later this was modified to directly generate a key, but for the example imagine a string of text. If the first five characters match the key memory map, the variables will hold until a trigger bit or the final sequence into the key memory map is made at which the buffer will stop excluding reads. This example shows the basic idea. In truth it utilizes every map in combination to accomplish this operation.




The repeating sequences are similar however rely on unique dirty bits and uses its own memory map. This preys on the fact that reciprocating tags are used in these datatypes to indicate sections. By defining this correctly, odd or even occurrences of the start sequences will simply flip the write bit. For example, no write on a sequence “[ … ]” will write for “[[ … ]]” but not for “[[[ … ]]]”. Finally characters can be excluded to simply ignore pesky useless results.

To make this portion fly, the C code used no if statements per token parse. Using all these maps the method relied on resetting counters and changing write bits. This allows the system to write and overwrite a buffer seamlessly with O(n) complexity while filtering, n being the read size to fill the feed buffer. This was written into the feed buffer which could be fed to the GPU. Once this was completed, I began designing the new kernel. I decided I wanted to process the keys on the GPU. Hopefully this could alleviate the CPU. This was a dead end. In order to do this, memory had to be wasted to pad the end of tokens. Worse than that the best and likely never to happen case would be 4608 tokens loaded to the GPU in one run. The average ended up becoming around 1000. Worse than this I attempted to load these words to the registers which failed. The programmer has no fine control in this grey area. I ended up with spills and redundant global memory accesses. Even the CPU ended up under more stress managing these tokens and providing the padding. I used a single buffer to see the overall overhead. I went from twenty seconds to run the CPU end with one thread to thirty eight for the 128 MB XML tester I primarily tested this with. I immediately reverted to my fallback kernel which ended up being my final one.

The CPU feeder changed to generating keys during parsing. This way the CPU could grab a token and load the buffer size amount of tokens to the GPU. This also cut the buffer write complexity

immensely. The previous implementation had to perform an O(n) pad operation n being the operation to check the available memory left to be loaded as a word buffer of 48 integer representations to the GPU registers. I performed testing and found that the original kernel from experimentation kept the single buffer test around twenty one seconds. I immediately implemented multi-threading alongside double buffering and managed to get this down to slightly above eight seconds.

The CPU version was about 0.02 milliseconds under 5 therefore I optimized the C code and buffering system. I also began loading the file to memory. This cut the time in half down to an average of 4.3 milliseconds. At this point it was late in the game and I needed to finish up. Since I had not intended to move the file to memory for development sake over the summer I did not get to implement optimizations I had only thought of. For example, I could have offloaded filtering on the GPU since everything had been moved from a file read to a buffer system. This would help since the CPU buffer load requires an immense average of 20 milliseconds. The GPU however had been optimized down to 0.7 milliseconds. It has a great deal of idle time to utilize which can be used. The last move I made was to play with unrolling. Even though the kernel performance was so fast that this did not matter, it still would be wise to understand the kernel performance for future updates. I had played with more dummy kernels testing loads and unrolled computations. These unrolls were small in code size and showed speed ups across the board. For example, thirty two fetches and 32 computations with each thread multiplying against every other in shared managed to go from 0.2 ms to 0.16 ms. One thing I realized was that the final code's size directly influenced performance decrease unlike these kernels. From this I found the optimum unrolling would be the memory fetch while the calculation was either partially or not unrolled. I stuck with the unrolled approach.

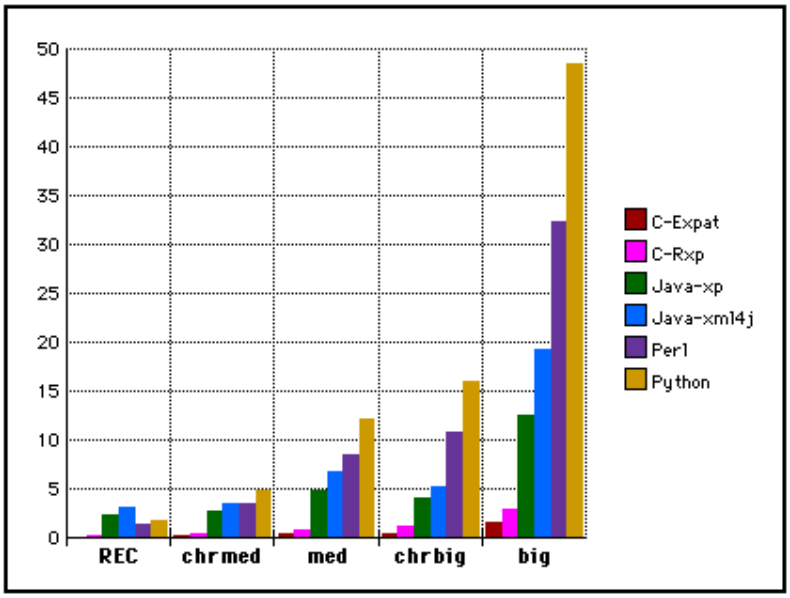
This definitely serves as proof that GPU's can serve as acceleration in frequency mapping of big data. Even without maximizing the GPU potential the implementation had much more intensive features and was still capable of winning out on time. With further tweaking there is no doubt this could be made extremely fast. Lastly, a final note about why I went about developing the system like this. I avoided using outside knowledge to the most extreme extent possible. I wanted this to be a learning experience from the ground up, and wanted this to be my own implementation unique or not. This has given me incite I likely would not have as to how to further improve this software and why some changes I have made were important in the first place.

***THE METRICS***

Before continuing, note the hardware was a Xeon E5-2670 and NVidia Grid K520 with 15 GiB of RAM on Amazon's AWS. The test file was an XML document 128 MB in size. This code was optimized for this configuration. With the increasing power of systems it will become easier to update this code with higher portability.

The processing is linear; the naïve file load system requires that there be enough ram to contain the file. Performance will be maintained as demonstrated in the results if the file is loaded in chunks. This will be a future update. Performance loss per file size with this version depends on the test platforms available memory.

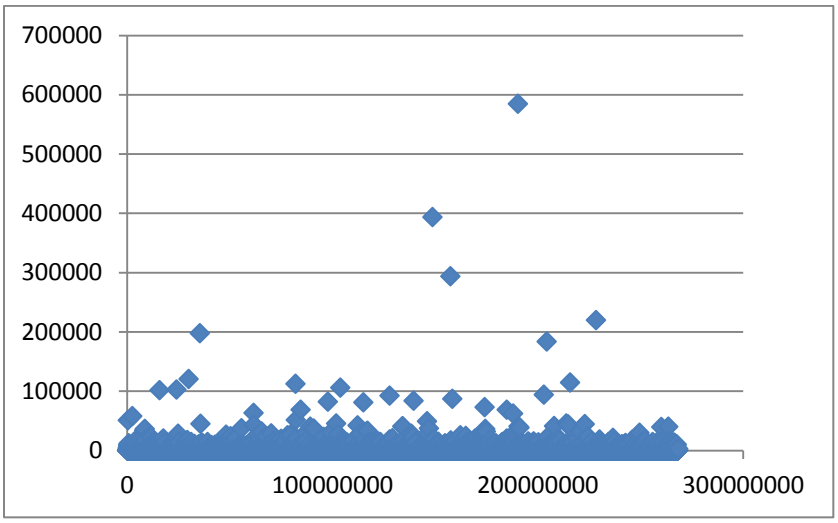
A reminder, this was programmed in C. The amount of operations required to process the file is immense. Next to assembly which eliminates readability and portability, there was no other choice. This software was to squeeze every bit of speed possible. This is well demonstrated in this XML parsing benchmark rated in seconds. A160 KB XML file war repeated from six to thirty two times.



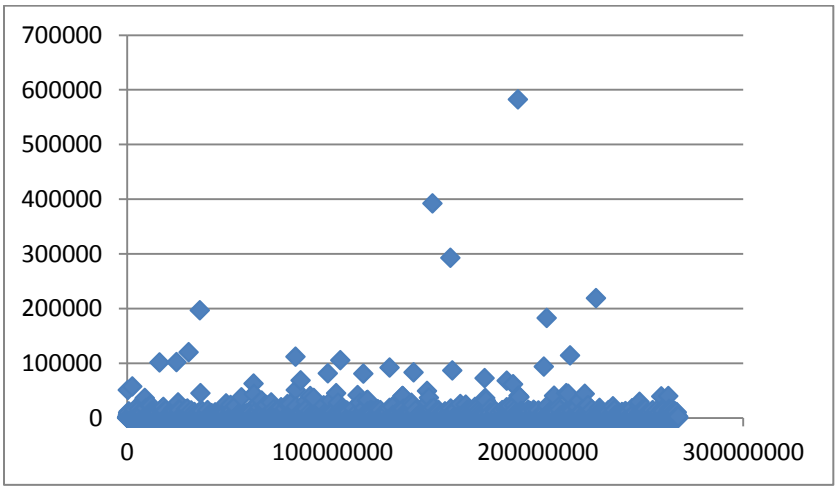

http://www.xml.com/pub/a/Benchmark/article.html?page=3

*HISTOGRAM COMPARISON*

*Full Version*



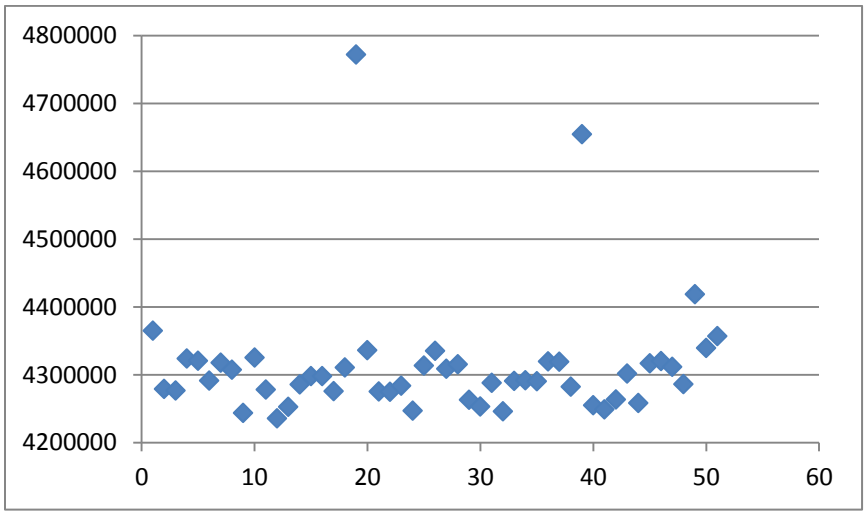

*Gold Version*




**Analysis**: The data proves the GPU version is working. One thing I have noticed however is that the CPU hash was not powerful enough to avoid some conflicts. For example the hits above 100,000 were either correct and simply a character left for testing purposes, or consisted of common collisions. This can be addressed in the future algorithmically as there only are about 300,000 unique words on average in a language. I do not believe there will be too many cross language conflicts nor should there be too many conflicts between the languages given the UTF 8 local can be handled to improve accuracy. The above test contained multiple locales and the spread appears to be relatively fair.

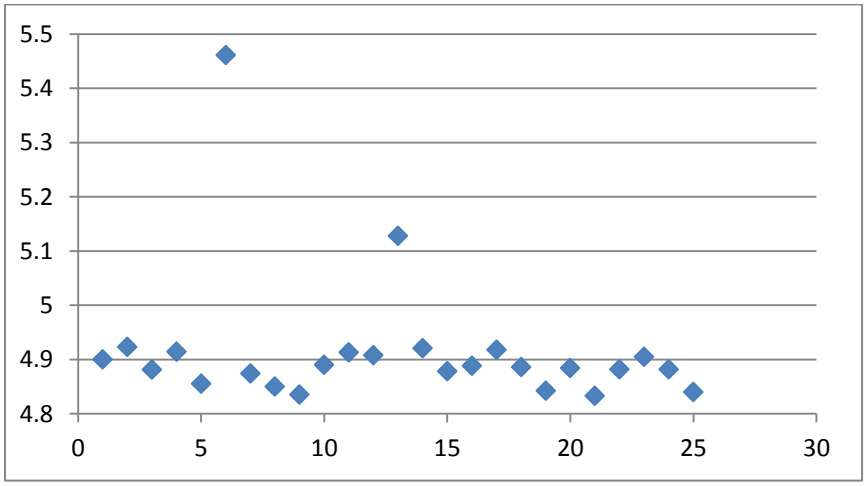
*OVERALL TIME*

*Full Version*



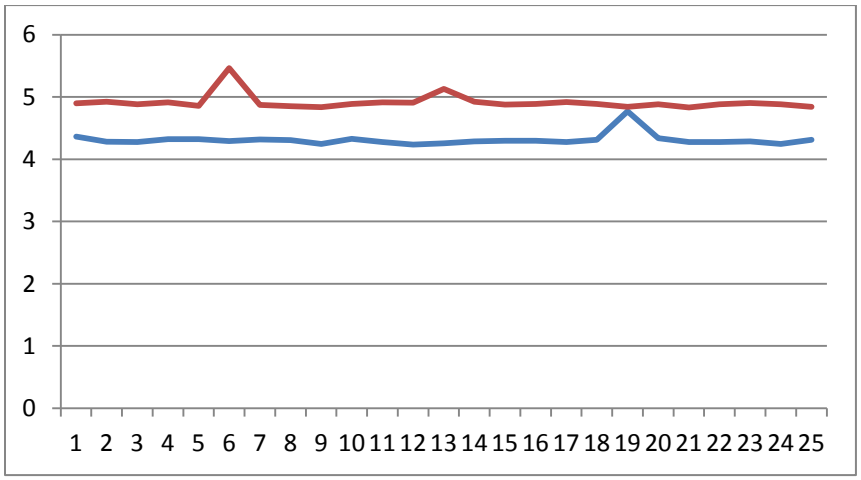

Average(s): 4.312242

*CPU Version*




Average(s): 4.915640

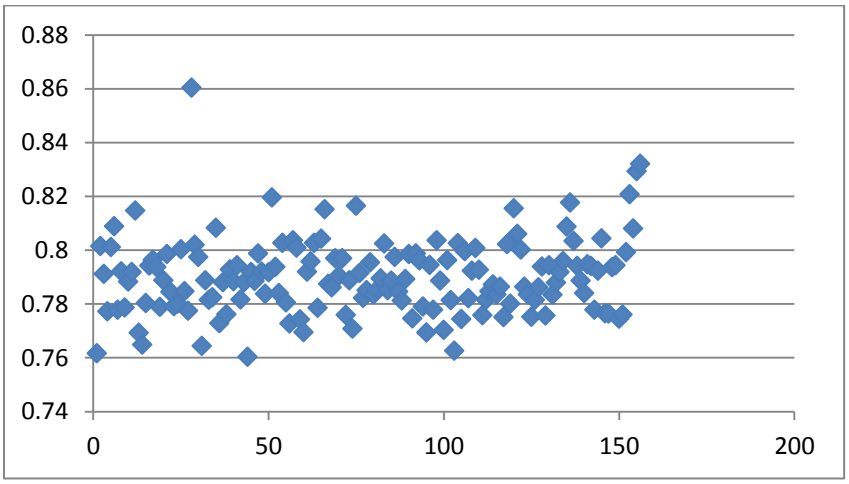
*Comparison*



**Analysis**: The CPU version is not even performing half the work of the new version. Unlike the one I've implemented using the GPU, this version does not filter content. There is a terrible amount of garbage that it finds. Worse than this is only reads ASCII characters. This is unacceptable in big data since UTF 8 is rapidly becoming a standard for the web in many regards.

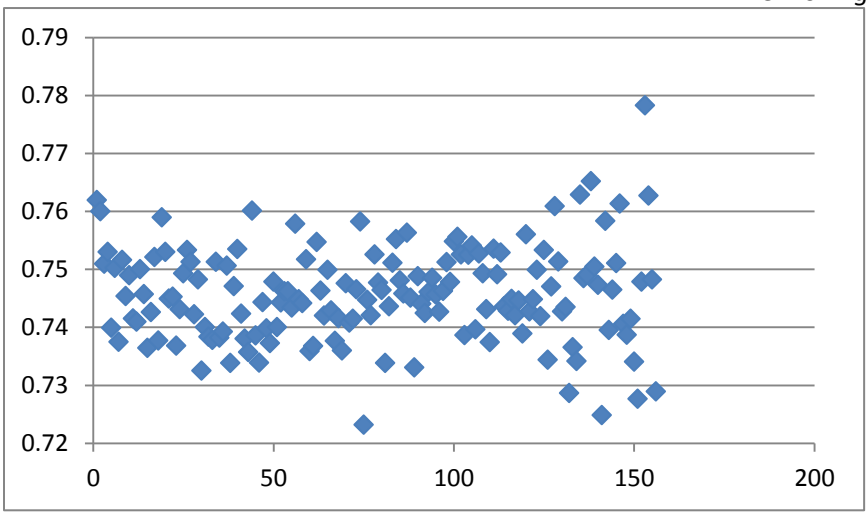
*OPTIMIZATION VIA UNROLLING*

*No unrolling*



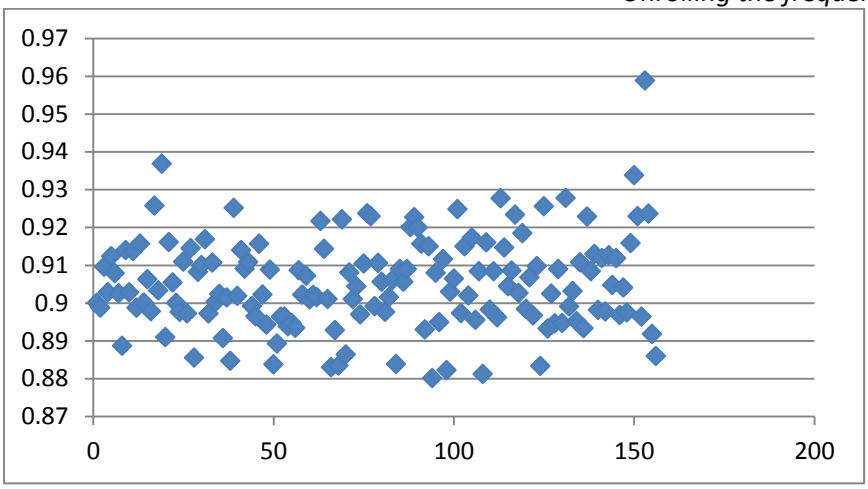

Average(ms): 0.79015

*Unrolling the buffer fetch*



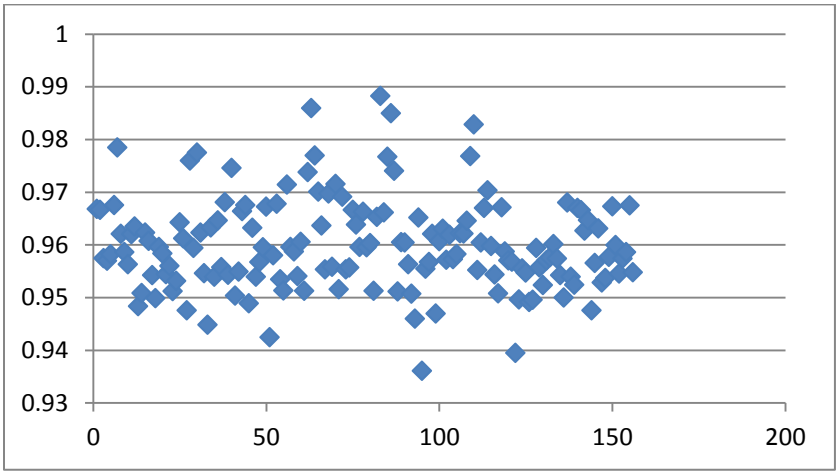

Average(ms): 0.745715

*Unrolling the frequency computation*




Average(ms): 0.905206

*Unrolling Everything*



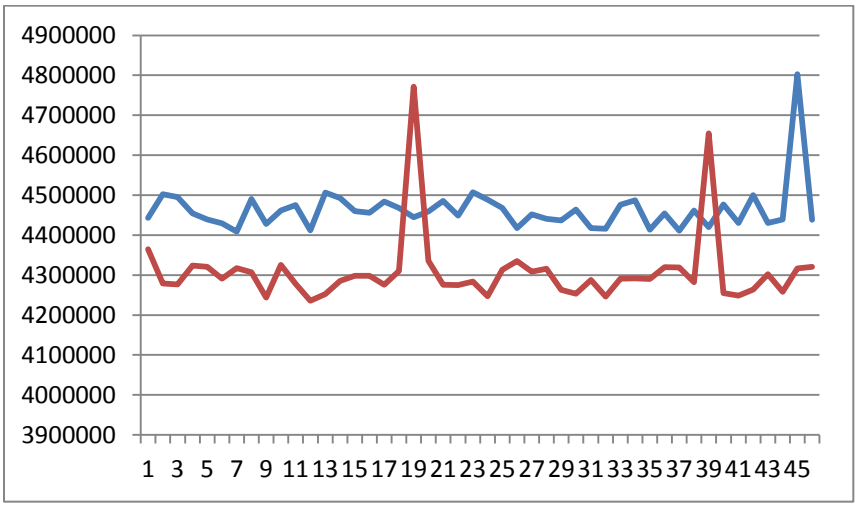

Average(ms): 0.959922

**Analysis**: I had only written the project after experimentation with all the algorithms I needed. For example I made unrolling kernels performing dummy computations and memory fetches. From these I derived unrolling offered an improvement all around. However, this data stunned me at first until I realized what could be happening. I discovered that increased code size was detrimental to performance in the end. It can be seen that unrolling just the memory fetch gives rise to notable improvement. However, by raising the code size with the computation unrolling efficiency drops due to this increase in the code size. This can be confirmed by unrolling everything and noting that the fetch unroll becomes detrimental where before it assisted in decreasing run time. This does not surprise me. It is likely that prediction on top of optimizations made to update the variables per iteration is hard to emulate unless the GPU compiler is better understood. I am only beginning to understand how to tweak higher level code to improve the lower level result.

*OPTIMIZATION VIA DOUBLE BUFFERING DOUBLED BUFFERED VS BUFFERED*

(As seen above, the double buffered code runs in 4.312242 seconds)

*Comparison*



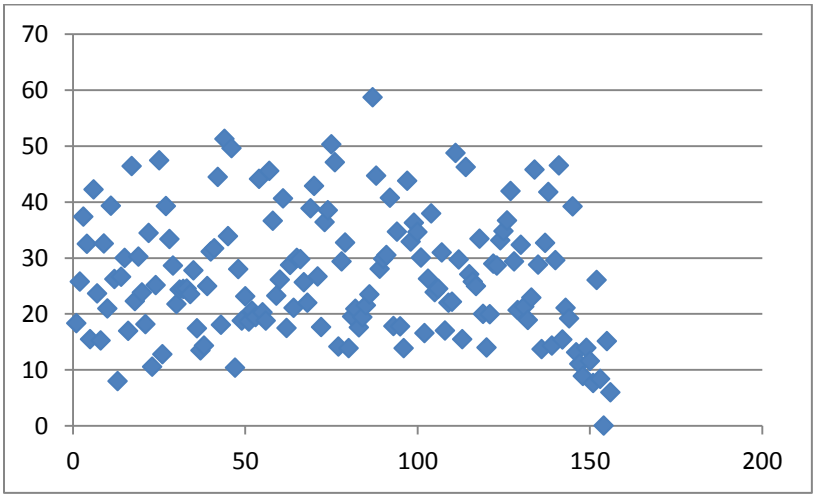

Average(s): 4.462947

**Analysis**: I never thought to finalize this program without double buffering for good reason. So long as the kernel runs faster than the CPU, it would be masked. This is due that the CPU can supply information to a buffer that is immediately available to the GPU as soon as it launches. This is a demonstration of how the lack there of such an optimization forces the final time to become the total of the kernel run times plus the total run times of the CPU buffer loads.

*GPU KERNEL TIME VS CPU KERNEL LAUNCH PREPERATION*

(As seen above, the GPU manages to run in 0.745715 milliseconds)

*CPU Kernel Launch Preparation*




Average(ms): 26.80327

**Analysis**: The CPU is no doubt a bottleneck. The GPU with the exception of calling the GPU functions which return control immediately to the CPU, there is no addition to the total CPU time. This leads me to believe that there needs to be immensely more work loaded onto the GPU while the CPU co-processes. There really is not more to be done for the CPU version other than writing offload management to filter on the GPU. Its efficiency is almost maximized for its task.

***PROJECT ANALYSIS***

My goal was to mask bin contention for the result histogram with the CPU's ability to feed it information. I then began changes. For example, a crucial speedup was loading the file into memory. I had designed this with the hope of portability for when my cloud resource was no longer free to run. I realized this was not practical and this opened up opportunities too late in the game to make useful changes. I ended up with a product that eliminates the time binning, yet in implementing a filtering system on the CPU only that I have not reached the programs maximal potential. While it runs faster than the simple CPU version which implements lots of efficient features while avoiding all the heavy work I've included in the GPU version, it absolutely can run faster. The GPU needs to perform the filter operation in parallel. While the CPU is processing pages, the GPU could be made to also process pages. This can be accomplished since I made the change that the file is loaded to memory.

This project was an incredible learning experience and I enjoyed developing this idea over time. I see now how to take this implementation to the next level and am excited to continue it during the summer. I plan to make this well developed and deployable as a library to my machine learning applications for training and actual data. I will work on brining the computational time below one second. This will take considerable time.

## Works Cited

Cooper, Clark. *Benchmarking XML Parsers*. n.d. webpage. 1 May 2015.