UNIVERSITY OF WATERLOO
Faculty of Mathematics

READING, WRITING, AND (R)OPTIMIZATIONS

Nvidia Corporation
GPU C++ Development Tools Architect
Santa Clara, California, Canada

Samuel Wanuch
20521149
3B Computer Science
May 14, 2017

Table of Contents

List of Figures

Summary

The purpose of this report is to summarize and justify the changes made to the cyclestats parser for optimization in speed and memory usage. This should be useful for anyone who has to maintain the relevant project, and make any further optimizations to it.

The analysis will cover the more important changes, and a brief analysis of how each change impacts the peformance on both metrics.

- boost::regex
- Multithreaded Writing
- Chunked Writing
- Multithreaded Loading
- File Reading
- String References
- Preformatted Strings

The final result of these optimizations is that the cyclestats parser is 20 times faster, and uses about 6 times less memory with a tighter limit on amount of memory it is possible to use.

Further improvements could potentially be found in caching the results of some string parsing and modification operations and finding a way to share the usage of commonly used strings instead of creating duplicates. Additionally it could be worth investigating something like memory pools to try and minimize wasted space in memory allocations.

1.0 Introduction

This report will cover the justifications for and changes to the cyclestats parser with the aim of making it faster, and lowering the memory usage.

1.1 Cyclestats Parser

The cyclestats parser is a program designed to read in, merge, and write out a large amount of data. From a high level perspective, the first stage is to read in data from many files, which typically takes anywhere from a minute to an hour. The next stage is to combine this data together, which rarely takes a significant amount of time. The final stage is to write out the different groupings of the data, which typically takes half an hour to a day.

1.2 Language and Libraries

The cyclestats parser is written in C++14, (the standard can be found in the references below under (C++ Foundation). This report assumes the reader is familiar with C++14 and the C++14 standard template library, more information about which can be found in the references below under (CPPReference). Additionally, this project makes use of and assumes the reader has some familiarity with the boost c++ libraries, more information about which can be found in the references below under (Boost).

This report further assumes the reader has a basic grasp of the concepts of mul-

tithreading, memory allocation, regular expressions, and, reading and writing files.

1.3 Analysis Method

This report will cover changes made to the cyclestats parser in the use of boost::regex, the addition of multithreading for reading and writing files, an improved memory usage pattern for file writing, string references, and pre-formatting strings. Each point will include some analysis of the runtime and memory impact.

The analysis of each change will be will be an explanation for how and why it might affect the performance metrics of runtime and memory usage. This analysis may be accompanied by actual timings and memory usage graphs. The timings are done by having the program itself output the time since the program started before it exits. The memory graphs are taken from the Windows 7 Task Manager's performance tab, and more specifically from the Physical Memory Usage History graph. Finally, the final implementation will be analysed. The performance metrics will be as recorded on a Windows 7 machine, with an Intel i7-5820k CPU, 16 gigabytes of RAM, and an SSD with sequential read/write capabilities of 540/520 megabytes per second respectively.

1.4 Test Cases

Analysis will be done on a test case which uses enough memory to crash the test machine using the original code, since this test case was the impetus for these changes; this test case will be referred to as the standard test case. Further analysis may be done on a small input, referred to as the small test case, or a large input, referred to as the large test case. The standard test case reads 444,987,569 bytes (424 megabytes) from 5 files, and writes 1,127,350,307 bytes (1.04 gigabytes) to 1558 files. The small test case, which is used mainly for loading tests, reads 219,849,981 bytes (209 megabytes) from 38 files. The large test case reads 7,350,499,141 bytes (6.84 gigabytes) from 38 files, and writes 16,255,564,798 bytes (15.1 gigabytes) to 7,336 files.

2.0 Analysis

2.1 boost::regex

Since the design of the cyclestats parser relies heavily on pattern matching, it also relies heavily on the boost regex libraries. While one would hope that the creators of the library would implement an efficient constructor for the boost::regex pattern object, it is undoubtebly not efficient enough when creating a creating a new regex multiple times per line, per file.

Since the regex patterns in question do not change from instantiation to instantiation, simply making each usage static so that the constructor does not keep getting rerun is a significant improvement. Admittedly this creates the risk that enough static regex patterns will use up inconvenient amounts of memory, but if you actually find this has become a problem, you likely have some serious design problems to work out.

The effect on memory is negligible since the memory has to be allocated either way. Since regular expressions are heavily used for complex parsing inside of the cyclestats parser, removing the need to construct the patterns every time halved the runtime of the program. For the most significant offender, reconstructing the regex pattern on every use slows down the standard input runtime to 20 minutes compared to the significantly improved 82 seconds of the final implementation.

2.2 Multithreaded Writing

The majority of the runtime for the cyclestats parser is spent on writing files. Since writing a file does not modify any of the data, it is a natual decision to try and multithread the writing. While it is possible that the multiple threads could write faster than the storage medium's write speed, number to string formatting and compression (both heavily used in typical runs) prevent this. The number of threads used is set to however many cores the system has.

In terms of runtime, this is a fairly straightforward win. With the multithreading system set to use only one thread during writing, the standard input takes 9.5 minutes to run to completion. This is approximately 6 to 7 times slower, which is mildly disappointing considering the system in question supports 12 threads. Some overhead though is not entirely surprising, and this is still an obvious improvement.

For memory usage, any memory used for writing each file will be multiplied by however many threads on the system. The original design of the system requires writing the entire file into memory, in order to then write the compressed file to disk, which ends up requiring multiple continuous segments of memory often over a gigabyte in size. This is mitigated by the next change. In practice, the final implementation uses approximately 200 megabytes more memory when using 12 threads than when using 1.

2.3 Chunked Writing

Anytime file compression is required the original design writes the entire file into memory before being compressed and written to the disk. The uncompressed files can often get over a gigabyte in size and require a continuous slab of memory to hold. In order to solve both the continuous memory problem and the massive memory usage problem, we instead write the uncompressed file in chunks which are periodically written out to disk.

If each chunk of memory is of equivalent size, they can be easily reused and it becomes easier to reason about how the changes affect memory and runtime. A size of 32 megabytes was chosen as a good medium between small and large sizes. It provides a large enough chunk of memory to write out most small files. At the same time is is small enough that at most a reasonably small amount of memory can be wasted. Since systems are expected to have 8 to 32 threads supported, 32 megabytes will waste at most 256 to 1024 megabytes of space. Given that the original design typically uses tens of gigabytes of memory, one gigabyte of waste seemed like a reasonable choice.

Each file is written as instances one by one. After each instance is written, if more than one chunk of memory was used, all the current chunks are written out and then recycled. All allocated chunks are shared behind the scenes, so that once a file is finished writing the chunks allocated for it can be reused for later files. This helps minimize memory allocation and freeing.

This improves the runtime performance by avoiding the constant reallocation

for storing the ever growing files in memory. The memory usage however is where this change really shines. Instead of massive spikes of potentially unlimited memory usage, the new system has a much lower and stricter limit on total memory usage. Each thread should use at most as many chunks as is required to hold the longest string of instance data plus one. In practice this gives a memory pattern with some growth at the start of file writing and then stable memory usage. A comparison of the memory usage patterns can be seen below, where the left side shows the memory usage of a prototype of the chunking system, and the right side shows the memory usage of the original code.

This graph clearly shows how the original code would struggle to fit large files in memory, and in fact neither graph shows a complete run since the original code crashed at the end, and the prototype code was modified to stop at the same point for this comparison. The graph runs from 0 gigabytes used at the bottom, to 16 gigabytes used at the top.

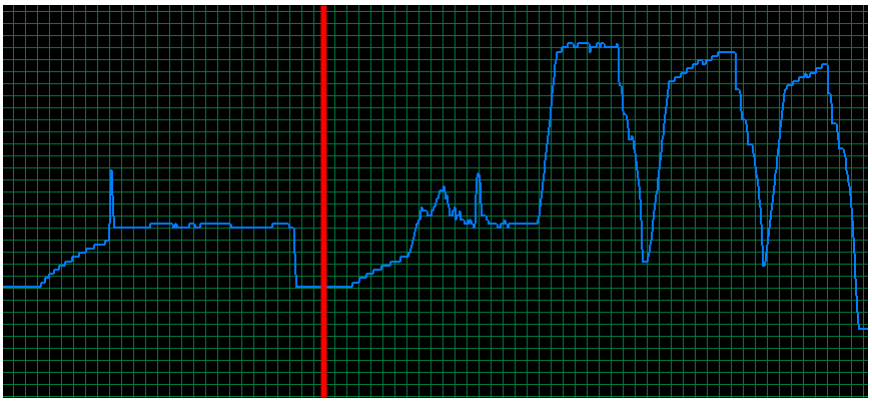

Figure 1: Final implementation on the left, original on the right

2.4 Multithreaded Loading

This is probably the most problematic change introduced to the cyclestats parser. On top of that, it has limited benefit given that file loading is a comparitively quick problem compared to file writing. However, since it is extremely valuable to be able to examine file writing in debug mode, and all the files may need to be loaded to examine a given problem, it is just as valuable to be able to load the files in debug mode in a reasonable timeframe.

The original design is slow enough in debug mode that the best approach to debugging the file writing is to give up, or at the very least approach the problem from a different direction. The multithreaded loading addition makes debugging the file writing significantly more feasible.

Since the original design assumes the files are loaded sequentially, the code had to be adapted. Each file now gets its own context to load in. While some objects, like states, are supposed to be shared between files while loading to avoid creating duplicates, the new design instead creates these duplicates. An additional step is added after the files have finished loading to merge the results of loading the files and combine any duplicates into single objects.

This improves runtime performance for the loading step by approximately a factor of however many threads are being used, with a small additional cost required for merging the results at the end. The merge is reasonably efficient and usually insignificant compared to the improvements. The memory impact of this is relatively minimal, as there is some amount of overhead used

to create the seperate loading contexts. In practice the standard input uses approximately 12 megabytes more memory.

2.5 File Reading

The standard template library and boost libraries for C++ are designed to be able to handle any and all possibilites. While this is very useful in general, for some cases it means a significant amount of unneccessary overhead. In specific, each line of each file must be read from the disk, and tokenized based on whitespace which are either spaces or tabs.

The std::ifstream, for all its marvels, can be rather slow. On the other hand, the lower level FILE pointers from C do the minimum work necessary to maximize speed for reading data from the disk.

The next problem is to read the file in units of lines. The standard approach to this is to use std::getline to read one line from the file into a string. This is also super adaptable at the price of speed. A more specialized function was designed to read data into a reusable buffer.

The final problem in this pipeline is to tokenize the line. This was done originally using the boost::tokenizer which is a reliable and generalized approach. Of course a generalized approach can always be beaten by a specialized approach. In this case, aggressive use of memchr, predicated on the fact that whitespace is expected to be only tabs or spaces.

Memchr is a C function which searches a stretch of memory for a certain byte value and returns the location, if found. This is presumed to be the most efficient way to find the next space or tab in a string (and inspecting the actual performance data of the original code shows that the visual studio compiler uses memchr inside of std::string::find).

Due to a less optimized memory allocation pattern, removing these improvements increases memory usage for the loading stage of the standard input by about 100 megabytes. Additionally, simply loading the files for the standard input ends up taking 28 seconds instead of the much improved 7 seconds.

2.6 String References

The cyclestats parser, by its nature, requires a significant number of unique strings. However, some of the strings are actually just substrings of others which remain constant once set. In order to save on memory, we can simply use a string reference, which requires only two pointers compared to an entire string copy. While the C++17 standard is rumoured to include string references, the best available solution is currently boost::string_ref.

Anytime strings will be reliably substrings of another string already being stored, the string was replaced with a string reference. While this would appear to be just a memory optimization, the fact that we do not have to copy strings also helps save significantly on precious cpu cycles which would otherwise be wasted copying strings. And of course it serves as a pretty nice

memory optimization.

2.7 Preformatted Strings

One of the problems with outputting numbers as text is that it is slow. When writing out the files, a lot of numbers have to be formatted into strings, many of them the same number being formatted again and again. Since these values rarely, if ever, change once set, but are formatted multiple times, it becomes cheaper to create and store formatted strings anytime the values are set.

Any of the values which might have to be formatted multiple times while writing files had a preformatted string added for the explicit purpose of writing out. On the one hand, this obviously increases memory usage. Fortunately, number strings are relatively small, and so the overall memory price is not hugely costly. This is mainly used on instances, for about 10 values. For the most part the strings are limited to about 10 characters, so the total impact is about $10 \times 10\times$ the number of instances bytes. For the standard input, this works out to about 150 megabytes.

While this does increase runtime minutely during loading to create these preformatted strings, it is an almost negligible impact. When writing out the files, removing only four uses of preformatted strings costs about 20 seconds on the standard input. This is about 5 seconds per usage. This is admittedly not a huge improvement, but the more files which need to be written out, the more impact this change will have.

2.8 Final Implementation Performance Comparisons

Unfortunately, the original impetus for these performance improvements was that the program was crashing on a certain input after running out of memory. Due to this, a proper comparison of a full run on that input is not possible. As a reasonable approximation, the program was run up to the crash point, then rerun, with the exception that it would skip all the files written up to, and including the file it crashes on. The final implementation is represented by the red line on the memory usage graph, with the blue line being the original implementation. Look closely, the red line is there, in the bottom left corner. The graph runs from 0 to 16 gigabytes vertically, and 0 to 30 minutes horizontally.

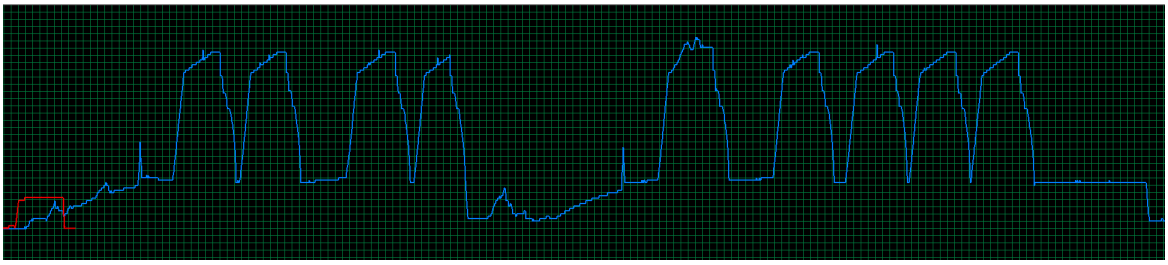

Figure 2: The red line is the final implementation, the blue is the original

For runtime performance, the final implementation takes a mere 80 seconds to run the standard input. While the original code cannot actually finish the standard input without crashing, it is a reasonable estimate that it takes roughly 30 minutes to complete. That makes the final implementation roughly 22.5 times faster. Additionally, the original code used roughly 12 gigabytes

of memory at its peak, while the final implementation uses only about 1.8 gigabytes of memory, which is about 6.5 times less.

In terms of simply loading, which is a metric useful for maintenance and debugging, the small input can be loaded in approximately 1 second. It is interesting to note that since the input files are themselves about 210 megabytes, and the disk read speed is 550 megabytes per second, the cyclestats parser is able to read in and merge the data in only twice as much time necessary to only read the raw data from the disk.

The large input takes a whopping 21 minutes to complete in the final implementation. Although to be fair, the runtime in the original code was roughly one "leave it running overnight", and so having it done within the hour is certainly a step up. Additionally, it uses 25 gigabytes of memory, and the test system has a mere 16 gigabytes, so there is some amount of paging to be done. This ends up being roughly 600 hard faults per second, and while finding the exact performance cost of this is beyond the scope of this report, it probably is not cheap.

3.0 Conclusions

Based on the above analysis, the cyclestats parser is now considerably faster, and should have a much stabler and smaller memory usage pattern. The most important change was the chunked file writing, which puts a soft limit on the amount of memory the program can consume.

Multithreading the file writing was a relatively safe change which improves performance by an approximate factor of the number of cores available. While this improvement is limited by the write speed of the disk being used, in practice it seems that limitation is still well above our current performance.

For further improving file write speed, adding preformatted strings to the data was a useful way to avoid repeated and wasteful string formatting. While this adds some extra memory usage and upfront runtime costs, the overall savings on speed and limited memory costs make the change worth it.

Similarly, changing boost regular expression patterns to be static is a significant improvement to performance for relatively little risk.

Since even loading a file can have significant memory impact, the use of string references throughout for both temporary and permanent strings helps cut down on the memory required, and provides a not inconsequential improvement in runtime to boot.

For overall runtime, multithreaded loading is a nice if not overly valuable improvement. However, since the cyclestats parser is expected to require active

maintenance in the future, the ability to quickly load files even while running in the significantly slower debug mode of the program is well worth it. The risk this change brings is mitigated by making it an optional setting.

For general loading performance, the file line reading and tokenizing improvements are a useful improvement in speed. Since the nature of the problem is such that the code should require little to no maintanence in the future, the added code complexity seems like a reasonable price.

Overall, the cyclestats parser is now 20 times faster, and uses about 6 times less memory, in a much more bounded manner.

4.0 Recommendations

While the final implementation of the cyclestats parser is an improvement on the original version, there is still room for improvement.

A significant amount of string parsing and manipulation happens inside the parser, and there may be room for improvement in caching the results of some of these operations. Additionally, it is expected that a lot of the strings being stored are less than unique, and there may be some memory savings to be had by finding such duplicates and storing only a single copy, possibly through the use of std::shared_ptr.

A more useful memory improvement may be found in smarter memory allocation, perhaps through memory pools or similar, in order to minimize wasted memory on multiple allocations.

References

1 Standard C++ Foundation: The Standard. Retrieved 27 April, 2017 from

`https://isocpp.org/std/the-standard`

2 CPP Reference. Retrieved 27 April, 2017 from

`http://en.cppreference.com`

3 Boost. Retrieved 27 April, 2017 from

http://www.boost.org/