15853 Project Report

Vincent Kang, David Zeng (vkang, dzeng)

July 10, 2018

1 Introduction

For our project, we integrated a SIMD integer compression technique into a graph compression library. Specifically, the SIMD integer compression was from the streamvbyte (`https://github.com/lemire/streamvbyte`) library and the graph compression library was ligra (`https://github.com/jshun/ligra`). The compression technique was used for encoding and decoding the graph within Ligra. We also implemented 1-bit codes for encoding and decoding graphs.

2 SIMD Integer Compression Benchmarks

We first wanted to investigate the theoretical speedup that utilizing SIMD integer compression would provide. We benchmarked the streamvbyte delta encoding and decoding functions from streamvbyte by running the encoding and decoding ten times on an array of sequential integers and an array of random integers with and without SIMD. We also measured the run times of the byte encoding algorithm in Ligra against these arrays. The random integers are sorted so that the difference encoding works properly. The table of measurement is below. We use the obtained measurements to gauge the theoretical best speedups we could obtain. Running the streamvbyte encode with SIMD gave a 2.4x speedup over the streamvbyte encode without SIMD and the decode with SIMD gave a 5.5x speedup over the streamvbyte decode without SIMD. When comparing the streamvbyte encoding/decoding with SIMD and the byte encoding provided by Ligra, the measurements show a 2.76x and 4.41x speedup for streamvbyte SIMD encoding and decoding the sequential array respectively over the Ligra byte encoding. For the random array, there is a 5.29x and 6.74x speedup for the streamvbyte SIMD encoding and decoding over the Ligra byte encoding. As expected, for each of the encoding schemes, the random arrays perform worse than the sequential arrays.

Type of Array / Is SIMD	Encoding	Decoding
Sequential SIMD	0.374	0.121
Random SIMD	0.380	0.181
Sequential No SIMD	0.885	0.664
Random No SIMD	2.270	1.850
Sequential Byte	0.978	0.534
Random Byte	2.01	1.22

3 Graph Compression with SIMD

We created a new compression format using the streamvbyte library. We kept most of the compression code from byte.h, but updated the encode and decode edgeSet methods by using the streamvbyte library to encode the set of edges rather than using byte encoding.

4 Graph Compression Benchmarks

We benchmarked our new compression format against byte encoding on two graphs. The first is a graph of user interactions on Wikipedia and the second is a fully connected graph. The first represents a fairly standard low diameter real world graph while the latter is designed to allow graph algorithms to be as efficient as possible. Benchmarking the latter graph is helpful for seeing how the different compression formats effect performance because it allows us to stop worrying the effects of cache misses and similar problems. We ran a variety of common graph algorithms on both graphs. We ran each algorithm as many time as was feasible and took an average over the run times.

The WikiGraph is a 2394385 node, 5021410 edge graph. The connected graph is a fully connected graph with 4000 nodes and 15996000 edges.

First, for encoding, we see that the Streamvbyte compression format is slightly less efficient than the Byte compression format. This is somewhat expected, because some sacrifices in structure are necessary to utilize SIMD instructions. We see that encoding times are roughly comparable.

Table 1: Encoding Benchmarks on Connected Graph

	Streamvbyte	Byte	Streamvbyte (no SIMD)
WikiGraph Encode Time	0.0663	0.0794	0.0662
WikiGraph Compression Size	8921647	7954231	8921647
Connected Graph Encode Time	0.032	0.0425	0.033
Connected Graph Compression Size	20007997	15999936	20007997

Next, we see that for common real world graphs like the WikiGraph, using SIMD compression might offer slight benefits in the speed of common graph algorithms. When comparing Streamvbyte to byte, the speedup ranges from 0.94x up to 1.53x speedup. However, this is far less than the theoretical speedups we found earlier.

Table 2: Decoding Benchmarks on WikiGraph

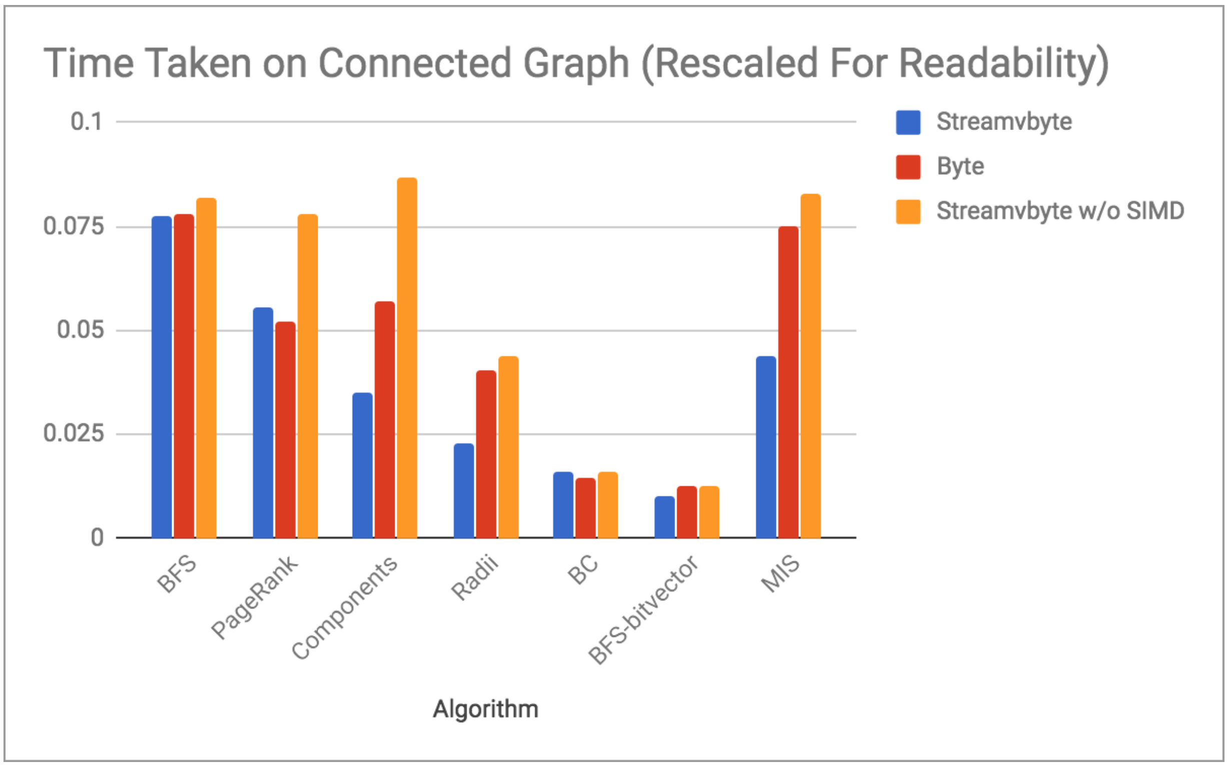
Algorithm	Streamvbyte	Byte	Streamvbyte (no SIMD)
BFS	0.0029513	0.0027835	0.0028005
Bellman Ford	0.0060697	0.0060930	0.0068904
PageRank	15.5051441	16.7447720	20.8143091
Components	0.1725565	0.2610078	0.2470632
Radii	0.0152404	0.0171209	0.0165262
BC	0.0221979	0.0228601	0.0236212
BFS-bitvector	0.0031500	0.0031805	0.0036089
MIS	0.4496172	0.5137358	0.5019633

Finally, for some graphs like the connected graph, using SIMD compression offers greater benefits for speeding up common graph algorithms. For example, when comparing Streamvbyte to byte, for the Radii graph algorithm, we approach 1.76x. In addition, we see good speedups on a greater number of graph algorithms. We also see that the sequential version of the Streamvbyte algorithm performs much worse the the SIMD version. We see a nearly 2x speedup when comparing the

SIMD and the sequential version for the Radii and MIS benchmarks. Finally, for some algorithms like BFS, it is possible the reasons we see no speedup is that the algorithm runs too quickly for our benchmarks to capture any difference made by using a different graph compression method.

Table 3: Decoding Benchmarks on Connected Graph

Algorithm	Streamvbyte	Byte	Streamvbyte (no SIMD)
BFS	0.0000774	0.0000779	0.0000818
Bellman Ford	N/A	N/A	N/A
PageRank	0.0556760	0.0519879	0.0777950
Components	0.0350172	0.0569550	0.0869790
Radii	0.0229461	0.0405780	0.0438209
BC	0.0001591	0.0001448	0.0001620
BFS-bitvector	0.0000997	0.0001255	0.0001251
MIS	0.0437257	0.0748379	0.0830691



5 Gamma Codes

We also implemented 1-bit codes, also known as gamma codes. Given an integer x, the gamma code algorithm first writes out the highest power of 2 smaller than x, say N, in unary (with 0's), then writes a 1 as a delimiter, then writes out x % N. Thus for an integer x, gamma coding uses $2\lfloor log_2(x)\rfloor + 1$ bits. A difficulty in implementing gamma coding was correctly implementing bit arrays using char arrays as there could be multiple compressed integers within one char. We did not optimize for performance so we omit performance figures. However, we expect performance to be a lot worse because we are working at the bit level rather than the byte level, which is inefficient for modern machines. The bit array was implemented using bit operations on the chars. Table 4 illustrates a comparison of the number of bytes that each of the different encoding algorithms outputs when compressing the WikiGraph and the connected graph (same graphs as in the previous section). Since the gamma coding is performed after difference coding the edges and the completely connected graph always has differences of 1, the gamma code does 14.97x better than the streamvbyte and 7.98x better than byte codes. However, on the wikiGraph the differences are not always 1 and they could be high enough such that gamma coding uses more bytes than a simple byte code. This explains why on the WikiGraph, we have have the gamma code outputting 1.02x more bytes than streamvbyte and 1.14x more bytes than the byte encoding. It is possible we could further optimize the compression size by using Delta encoding instead of gamma encoding.

Table 4: Encoding Benchmarks

	Gamma	Streamvbyte	Byte
WikiGraph Compression Size	9065214	8921647	7954231
Connected Graph Compression Size	2003999	20007997	15999936

6 Code

View the code at Project Github Page (https://github.com/vin01188/SIMD-and-Gamma-graph-compression). Instructions for running code are included. We believe the only build requirements is a version of g++ that supports c++14. (It should build on MacOS using the default clang compiler).

7 References

https://github.com/jshun/ligra

https://github.com/lemire/streamvbyte

https://people.csail.mit.edu/jshun/ligra.pdf

http://people.csail.mit.edu/jshun/ligra+.pdf

https://arxiv.org/abs/1709.08990

8 Acknowledgements

Thanks to Laxman and Guy for their help of project ideas.