Eiffel Sum algorithm performance comparison

Fernando Pelliccioni <fpelliccioni@gmail.com>

December 2015

I was trying to make an analysis of some features of modern CPUs and I chose Eiffel as programming language.

It turned out that classical Sum algorithm implementation is extremely slow in Eiffel, which caught my attention. First, I need solve this problem before I can continue with my original goal.

The Sum algorithm is very simple. We take an array of 32-bit signed integers, sum them all and return the result. I am using an array because I wanted to use the simplest data structure possible, because I do not want a more complex data structure interferes on expected results.

After Alex's response, I decided to do some measurements on variations of the algorithm.

Algorithm implementations

These are the different implementations of the Sum algorithm:

- C: *Canonical* implementation: using *ARRAY* class and loop. Eiffel function: sum_canonical.
- A: *Across* implementation: like Canonical implementation, but using across instead of loop. Eiffel function: sum_across.
- LC: *LocalCount* implementation: like Canonical implementation, but array.count is moved into a local variable. Eiffel function: sum_local_count.
- TM: *TypeMismatch* implementation: like Canonical implementation, but array.item is moved into a 32-bit Signed Integer local variable assigned inside the loop. Eiffel function: sum_type_mismatch.
- LCTM: *LocalCountTypeMismatch* implementation, it is a combination of LC and TM implementations. Eiffel function: sum_local_count_type_mismatch.
- JS: *JustSpecial* implementation, like Canonical implementation, but using SPECIAL class instead of ARRAY class. Eiffel function: sum_just_special.
- K: *Kogtenkov* (Alex) implementation, using *SPECIAL* class. It is a combination of LC, TM and JS implementations. Eiffel function: sum_kogtenkov.

Another implementations:

They were not taken into account because they do not represent a significant improvement over base implementations.

- C32: Like Canonical implementation, but the sum (Result) is accumulated in a 32-bit Signed Integer variable. Eiffel function: sum_canonical_32.

- A32: Like Across implementation, but the sum (Result) is accumulated in a 32-bit Signed Integer variable. Eiffel function: sum_across_32.
- TM64: Like TypeMismatch implementation, but using a 64-bit Signed Integer local variable instead of a 32-bit one. Eiffel function: sum_type_mismatch_64.
- LCTM64: Combination of LC and TM64 implementations. Eiffel function: sum_local_count_type_mismatch_64.
- JS32: Like JustSpecial implementation, but the sum (Result) is accumulated in a 32-bit Signed Integer variable. Eiffel function: sum_just_special.
- K64: Like Kogtenkov implementation, but array.item is moved into a 64-bit Signed Integer local variable assigned inside the loop. Eiffel function: sum_kogtenkov_type_mismatch_64.

About the benchmarks

Implementation functions were called with arrays filled with random 32-bit signed integer values.

The array size varies between 8 and 16,777,216 elements.

All functions were called with the same data.

Each function was called several times with the same input. Each resulting time was registered. The worst and best times were eliminated and the average time was calculated.

The times shown below are in nanoseconds per element.

I use the following function to get the number of nanoseconds elapsed between Unix Epoch and the current time:

```
nanoseconds_since_epoch(): INTEGER_64
    -- Returns a 64-bit signed integer representing the number of nanoseconds elapsed
between Now and the clock's Unix epoch.
    external
        "C++ inline use <chrono>"
    alias
        "{return std::chrono::duration_cast<std::chrono::nanoseconds>(
                    std::chrono::high_resolution_clock::now().time_since_epoch()
                ).count();}"
    end
```

You can see the full code here: https://github.com/fpelliccioni/EiffelStudioOptimizationMeasurements

Results

# Elements	C	C32	A	A32	LC	TM	TM64	LCTM	LCTM64	JS	JS32	K	K64
8	9.06	7.90	73.09	69.47	6.07	8.21	8.22	6.18	6.21	6.62	4.41	2.24	4.31
16	8.95	7.60	58.76	58.28	5.53	7.92	8.04	5.51	5.50	6.21	3.51	1.29	3.40
32	8.30	6.90	53.02	52.35	5.45	7.18	7.42	5.46	5.45	5.48	3.28	0.81	3.18
64	7.97	6.52	49.82	49.27	5.50	7.32	7.36	5.46	5.47	5.10	2.92	0.56	2.88
128	7.77	6.20	48.09	47.98	5.72	7.26	7.28	5.76	5.73	4.93	2.72	0.47	2.68
256	7.66	6.30	47.26	46.78	5.67	7.17	7.21	5.78	5.89	4.87	2.63	0.38	2.55
512	7.24	6.56	46.65	47.62	5.66	7.16	7.15	5.64	5.65	4.82	2.58	0.33	2.57
1,024	7.45	6.41	45.75	46.38	5.63	7.15	7.16	5.64	5.65	4.78	2.54	0.30	2.58
2,048	7.50	6.39	45.83	46.48	5.62	7.12	7.08	5.60	5.63	4.80	2.60	0.30	2.55
4,096	7.56	6.40	45.75	46.26	4.84	6.45	6.70	4.70	4.79	4.80	2.57	0.29	2.56
8,192	8.01	6.87	46.49	48.22	5.67	7.44	7.38	5.61	5.62	4.72	2.53	0.29	2.55
16,384	7.35	6.89	46.18	46.02	5.30	7.13	7.25	5.32	5.33	4.66	2.55	0.29	2.55
32,768	7.27	6.38	45.43	45.75	4.76	6.39	6.43	4.79	4.78	4.72	2.52	0.28	2.48
65,536	6.97	6.31	45.05	44.59	4.81	6.48	6.70	4.84	4.83	4.78	2.53	0.29	2.53
131,072	7.30	6.43	45.77	45.83	4.84	6.43	6.66	4.79	4.77	4.77	2.55	0.29	2.54
262,144	7.10	6.38	45.74	45.76	4.80	6.47	6.62	4.79	4.81	4.79	2.54	0.29	2.53
524,288	7.06	6.38	45.86	45.99	4.84	6.47	6.70	4.84	4.86	4.80	2.55	0.29	2.57
1,048,576	7.19	6.50	46.11	46.34	4.81	6.55	6.72	4.89	4.89	4.76	2.54	0.31	2.55
2,097,152	7.09	6.30	45.92	45.88	4.90	6.55	6.74	4.90	4.86	4.84	2.54	0.35	2.55
4,194,304	7.08	6.38	45.99	45.84	4.89	6.48	6.70	4.84	4.81	4.79	2.53	0.36	2.53
8,388,608	7.05	6.36	45.86	45.64	4.74	6.47	6.67	4.81	4.84	4.73	2.47	0.35	2.47
16,777,216	6.95	6.28	44.57	45.55	4.81	6.43	6.62	4.80	4.80	4.76	2.53	0.37	2.53

The table represents the time spent for each algorithm as array size grows. The first column is the number of elements of the array. The time is in nanoseconds per element.

Kogtenkov (K) function is the fastest one, so is taken as a reference to compare with others functions.

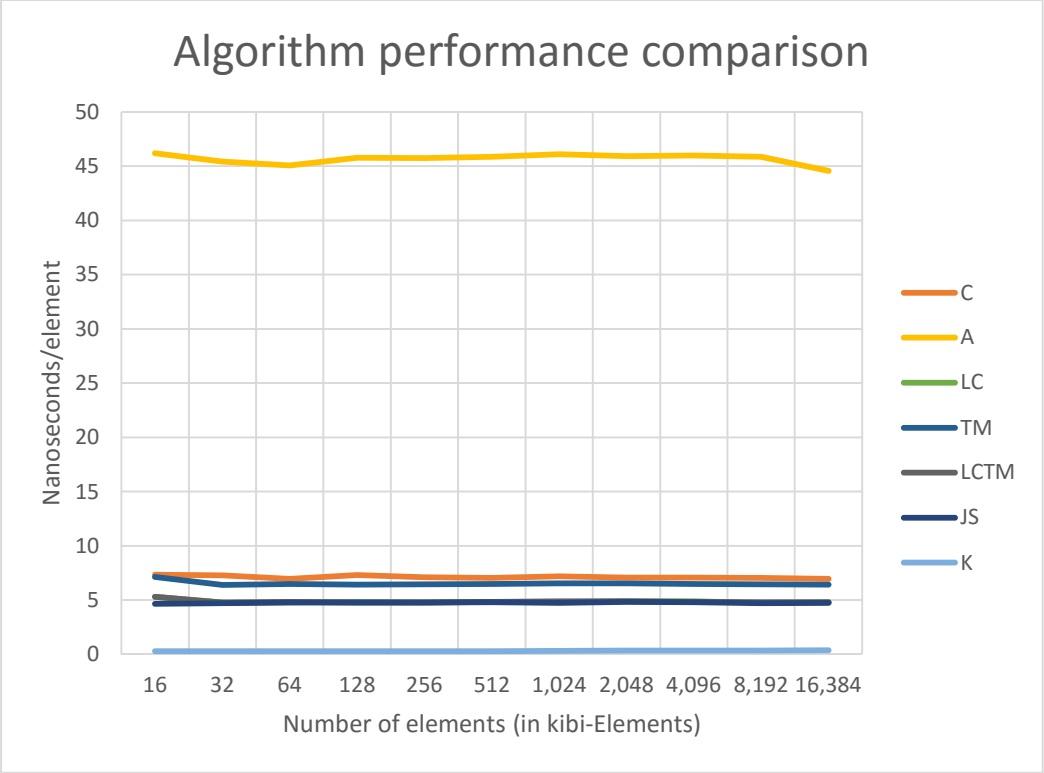


Figure 1.
Y axis: time in nanoseconds/elements.
X axis: number of elements * 1024. IE: 32 means an array of 32,768 elements.

Figure 1 shows the times of the most important algorithms, where you can see the notorious distance between *Across*, *Kogtenkov* and the rest of the functions.

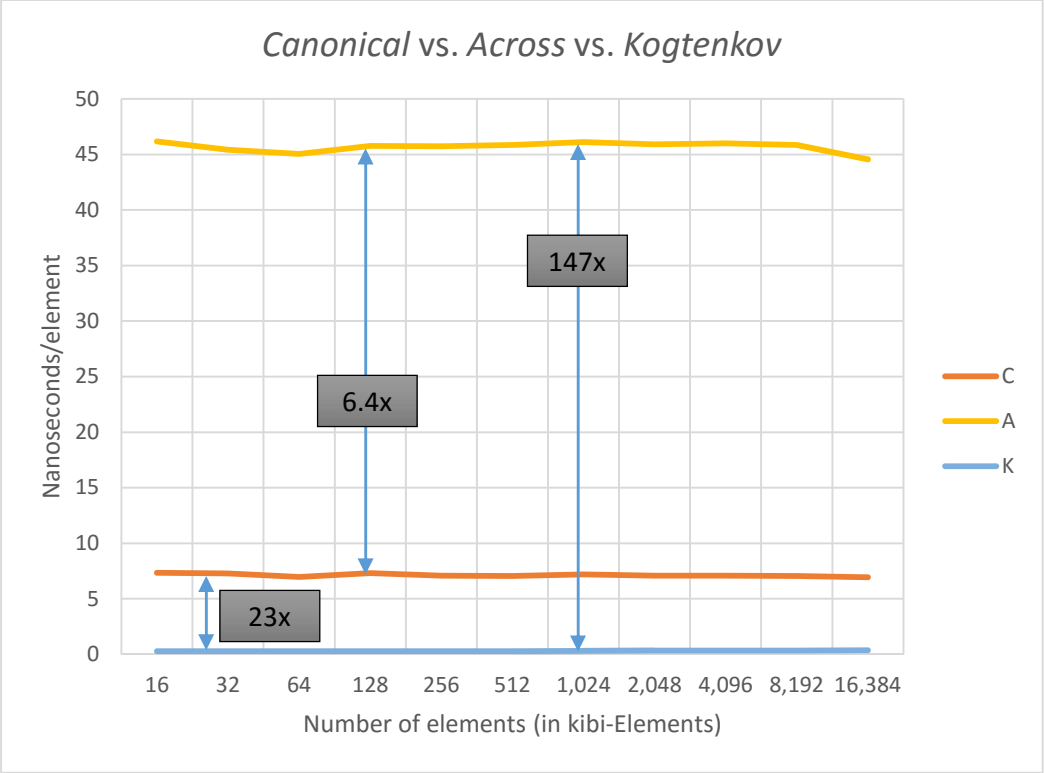


Figure 2.

Figure 2 shows a performance difference in favor of *Kogtenkov* of 23x with respect to *Canonical* and 147x with respect to *Across*.

Conclusions

The Eiffel programming language is proclaimed as a simple, clear, elegant and powerful language.

While performance issues mentioned in this document are not directly related with the Eiffel programming language, they are present in the main and most popular Eiffel compiler: Eiffel Studio.

Compilers are the visible face of programming languages. If compilers fail or generate bad code, languages are avoided.

The performance difference between the *Kogtenkov* function and the most natural functions, such as *Canonical* and *Across*, is huge. Then, the latter functions should be avoided in programs where efficiency is a key factor.

Novice programmers are doomed to suffer a performance penalty greater than 20x in their programs, or to unnecessarily write verbose code to obtain reasonable performance.

Then, it turns out that Eiffel is either powerful or simple, clean and elegant, but not both at once. (At least in the cases described in this document).

To reclaim the status of simple and powerful at the same time, compiler optimizations must be improved, such that the *Kogtenkov*, Canonical and *Across* functions have equivalent efficiency.

Platform

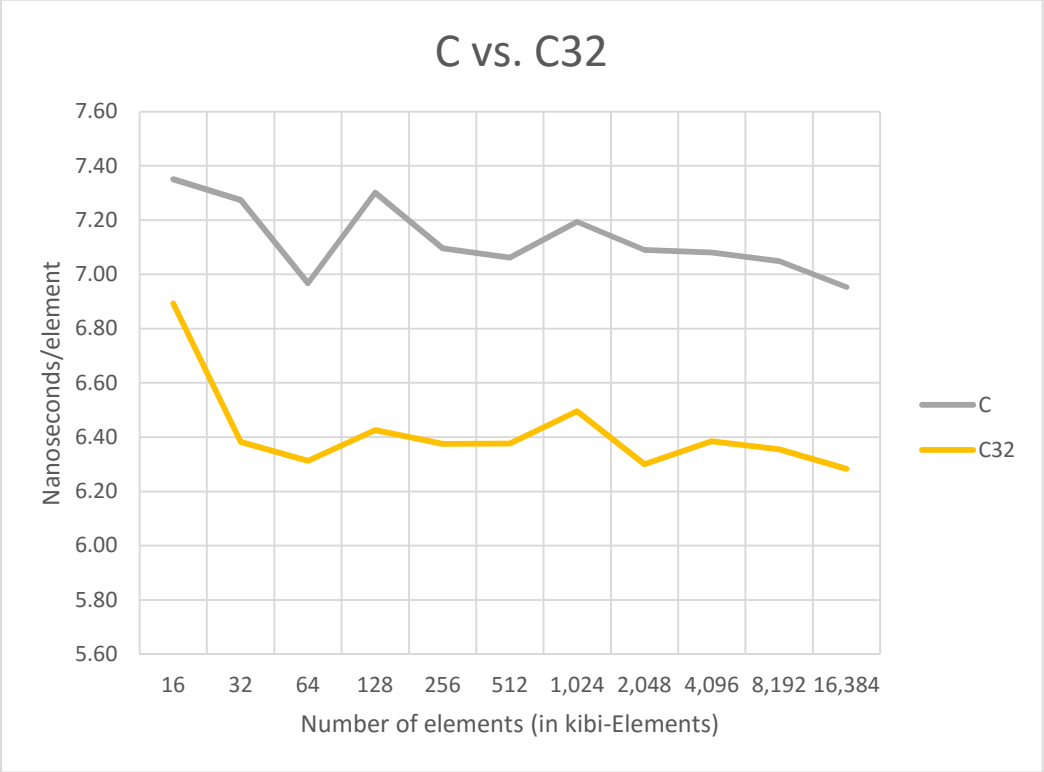
Time measurements presented in this document were obtained using the following platform:

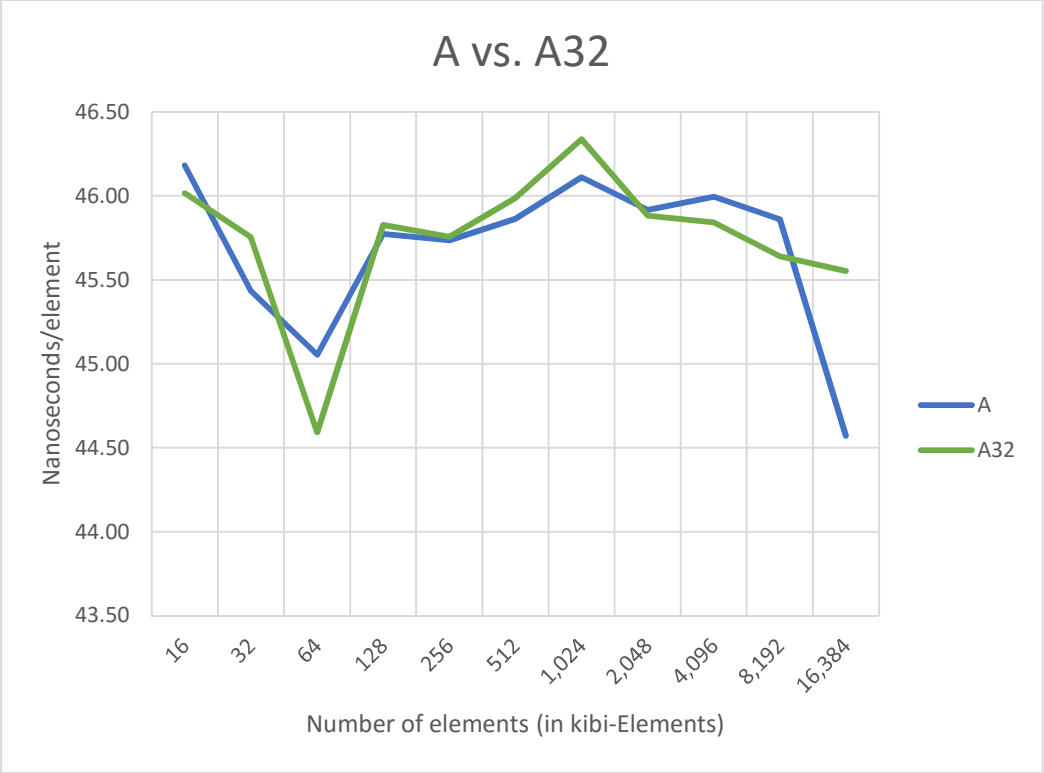
- Intel Core i7 4700MQ (Haswell) @ 2.40GHz.
  4 Cores with Hyper Threading (8 threads).
  8 GB of RAM
- Windows 10 Home, 64-bit.
- Eiffel Studio 15.08.9.7862 GPL Ed - win64, using MSVC12 (Visual Studio 2013)

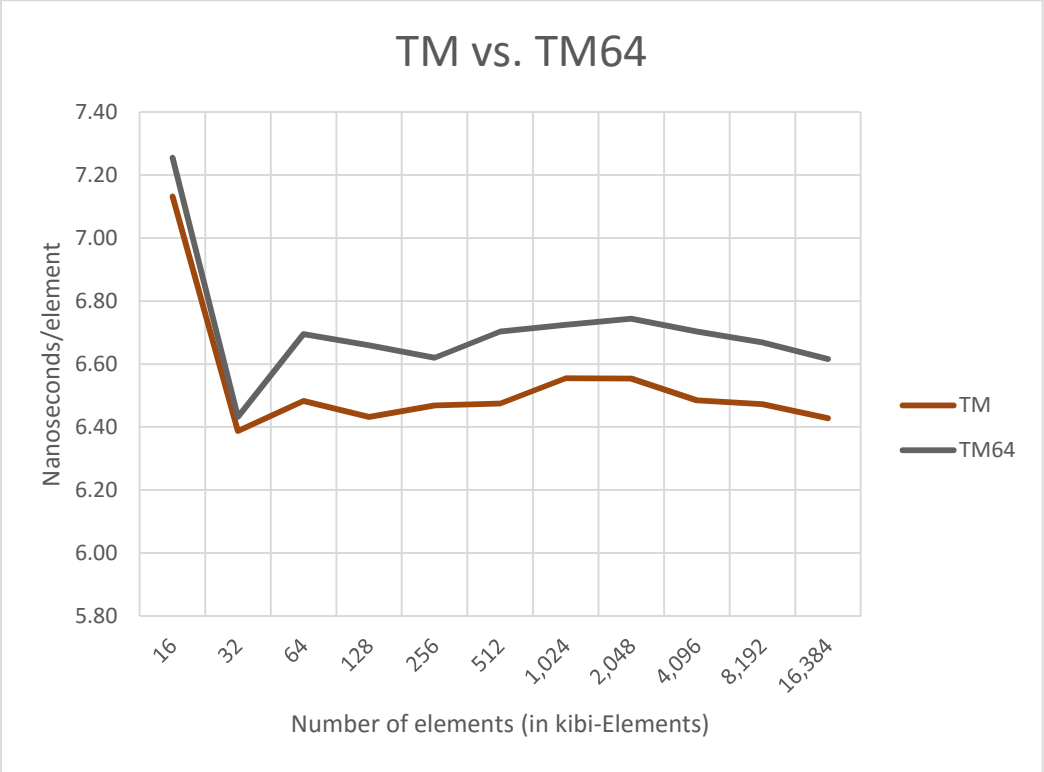
Source code

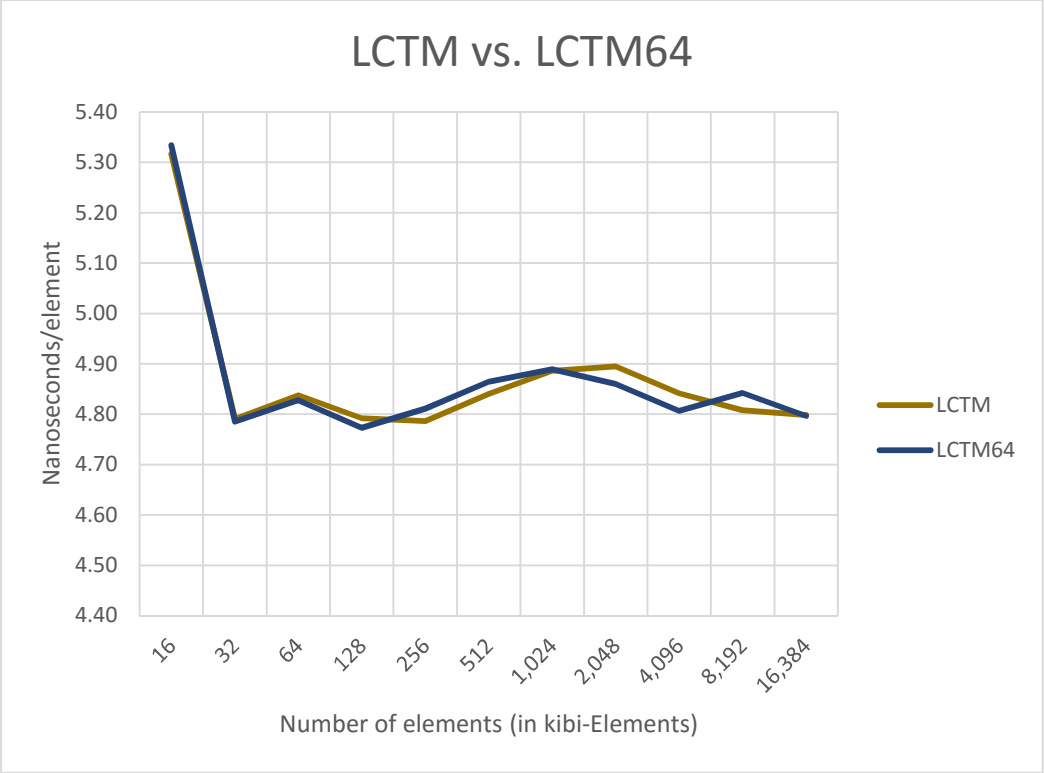
https://github.com/fpelliccioni/EiffelStudioOptimizationMeasurements

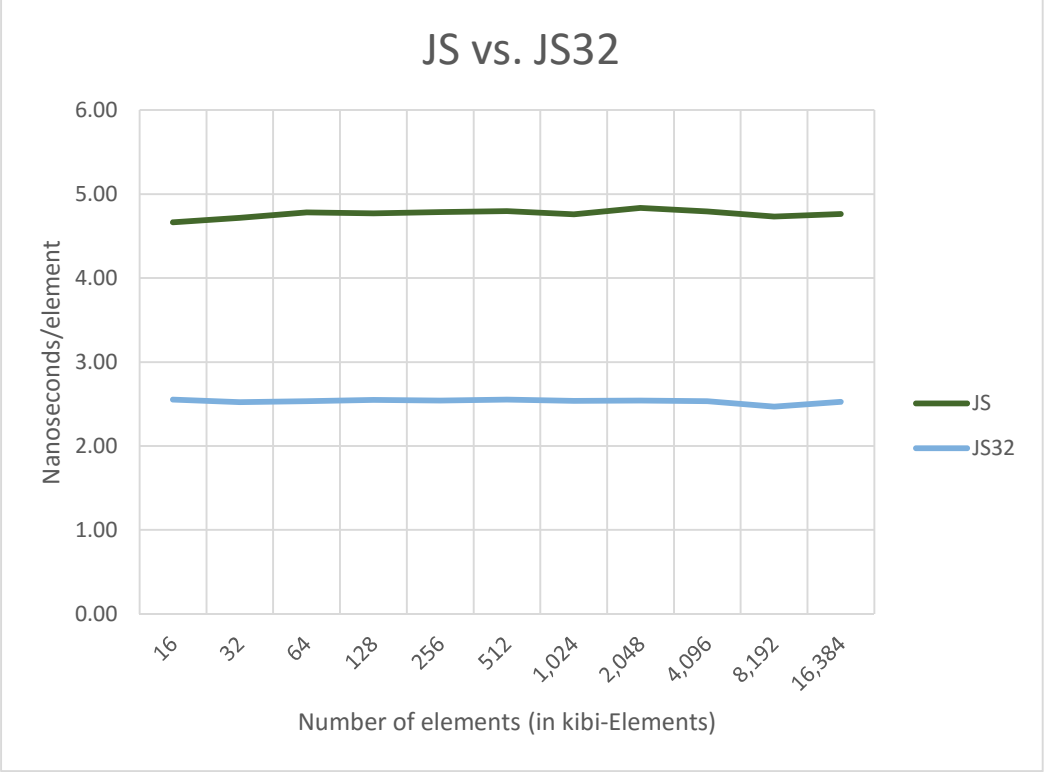
Appendix A: Less relevant figures

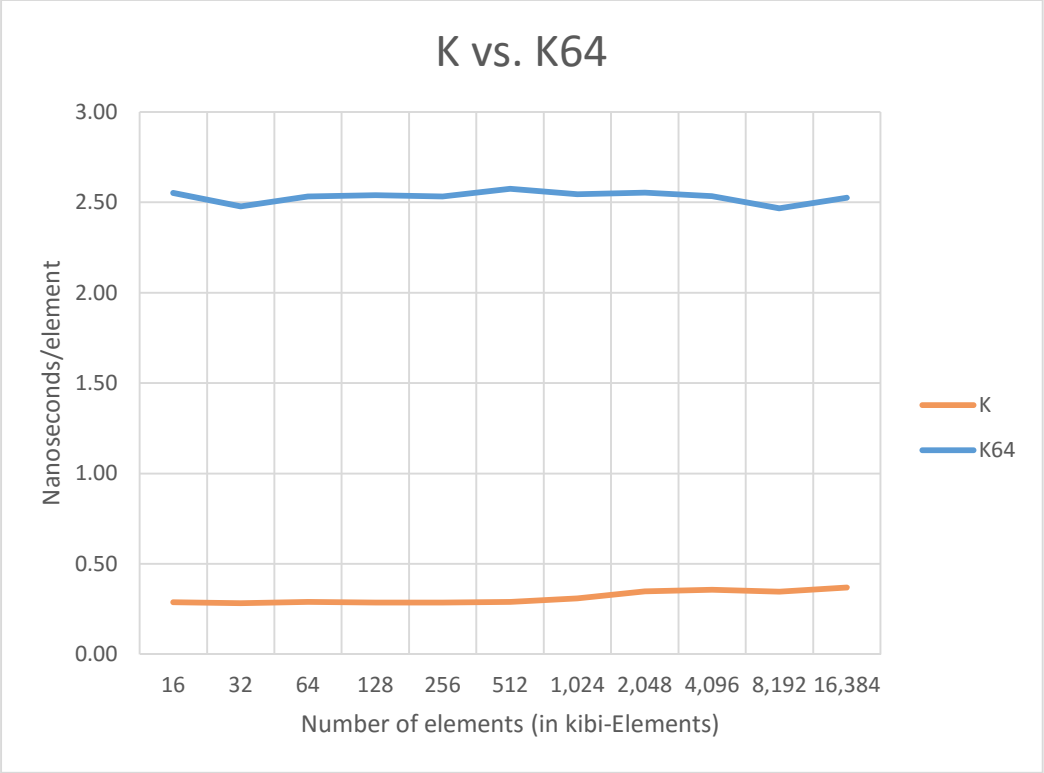



A vs. A32
46.50
46.00
45.50
45.00
44.50
44.00
43.50
Nanoseconds/element
16
32
64
128
256
512
1,024
2,048
4,096
8,192
16,384
Number of elements (in kibi-Elements)
A
A32


TM vs. TM64
7.40
7.20
7.00
6.80
6.60
6.40
6.20
6.00
5.80
Nanoseconds/element
16
32
64
128
256
512
1,024
2,048
4,096
8,192
16,384
Number of elements (in kibi-Elements)
TM
TM64


LCTM vs. LCTM64
Nanoseconds/element
5.40
5.30
5.20
5.10
5.00
4.90
4.80
4.70
4.60
4.50
4.40
16
32
64
128
256
512
1,024
2,048
4,096
8,192
16,384
Number of elements (in kibi-Elements)
LCTM
LCTM64


JS vs. JS32
Nanoseconds/element
6.00
5.00
4.00
3.00
2.00
1.00
0.00
16
32
64
128
256
512
1,024
2,048
4,096
8,192
16,384
Number of elements (in kibi-Elements)
JS
JS32


K vs. K64
3.00
2.50
2.00
1.50
1.00
0.50
0.00
Nanoseconds/element
16
32
64
128
256
512
1,024
2,048
4,096
8,192
16,384
Number of elements (in kibi-Elements)
K
K64