Cache and Multi-Core Efficient Algorithms for High-Degree Permutations

Steve Linton and Ryan Gibb
University of St Andrews
{sl4, rtg2}@st-andrews.ac.uk

October 2020

Abstract

The traditional naive permutation composition algorithm is limited by memory latency and not CPU speed. Algorithms to take advantage of current memory hierarchies and multiple cores can be designed so as to be limited by memory bandwidth instead. We have implemented and benchmarked a number of such algorithms. We have found a multithreaded Rust implementation of the naive algorithm is 64% than the naive algorithm on an Intel Xeon E3-1230 v5 with 64GB RAM for permutations of degree 2^{32}. This can be adapted to improve the performance of other permutation operations, as well as other algorithms that access large amounts of main memory pseudo-randomly.

1 Introduction

Work was done on creating a permutation composition algorithm to overcome this 'memory wall' in 2002 by Cooperman and Ma using a bucket algorithm to make effect use of the cache.[1] This algorithm will be refer to as the 'Cooperman & Ma algorithm' hereafter. We will see how their algorithm holds up on modern hardware for significantly larger permutations - up to $n = 2^{32}$ compared to the previous $n = 2^{20}$.

The other algorithms considered are the naive algorithm as a baseline, a multithreaded naive algorithm, a multithreaded Cooperman & Ma algorithm, and a vector optimised naive algorithm.

2 Design

A permutation σ of degree n is a bijective function from a set $S = \{1, \ldots, n\}$ onto itself, represented as an array x of length n where the value at index i in the array is the image of i under σ. More simply, if $\sigma(i) = j$ then `x[i]=j`.

Given two permutations $i \mapsto \sigma(i)$ and $i \mapsto \pi(i)$, their composition $\sigma\pi$ is defined as $i \mapsto \pi(\sigma(i))$.

Taking two permutations `x` and `y`, the following algorithms calculate their composition `z=xy`. All of `x`, `y`, and `z` must of length `n`, with `x` and `y` containing every value from `0` to `n - 1` exactly once.

Let `perm_t` be the type of the permutation arrays; the type of the elements in the permutation image. This determines the maximum permutation degree than can be composed; as well as the number of bytes used to store the permutation `n*sizeof(perm_t)`.

2.1 Naive

```
perm_t x[n], y[n], z[z];

for (size_t i = 0; i < n; i++) {
  z[i] = y[x[i]];
}
```

Listing 1: Naive Algorithm

2.2 Cooperman & Ma

Note `CACHE_SIZE` is the number of bytes in the target cache.

```
#define BLOCK_LENGTH (CACHE_SIZE / 2 / sizeof(int))
#define NUMBER_OF_BLOCKS = (n / BLOCK_LENGTH)

perm_t x[n], y[n], z[z];
perm_t d[n];
perm_t* d_ptr[NUMBER_OF_BLOCKS];

//Phase I: distribute value, x[a], into d_ptr[block_num]
// such that block_num == x[a] / BLOCK_LENGTH
for (size_t block_num = 0; block_num < NUMBER_OF_BLOCKS; block_num++) {
    d_ptr[block_num] = &d[block_num * BLOCK_LENGTH];
}
for (size_t i = 0; i < ARRAY_LENGTH; i++) {
    block_num = x[i] / BLOCK_LENGTH;
    *d_ptr[block_num] = x[i];
    d_ptr[block_num]++;
}

//Phase II: for d[i] == x[a], replace the value x[a] by y[x[a]]
// Note that |i - d[i]| == |i - x[a]| and |i-x[a]| < BLOCK_LENGTH
for (size_t i = 0; i < ARRAY_LENGTH; i++) {
    d[i] = y[d[i]];
}

//Phase III: copy value y[x[a]] from d_ptr[block_num] to \[a]
for (size_t block_num = 0; block_num < NUMBER_OF_BLOCKS; block_num++) {
    d_ptr[block_num] = &d[block_num * BLOCK_LENGTH];
}
for (int i = 0; i < ARRAY_LENGTH; i++) {
    block_num = x[i] / BLOCK_LENGTH;
    z[i] = * d_ptr[block_num];
    d_ptr[block_num]++;
}
```

Listing 2: Cooperman & Ma Algorithm[1]

2.3 Multithreaded Naive

This algorithm splits the permutation into different slices of `x` and `z`, and runs a thread for each slice to naively compose them in parallel. The number of threads should not exceed the number of CPU cores. Note `y` is shared between all threads.

2.4 Multithreaded Cooperman & Ma

Phase 2 of the algorithm is run in parallel with the 'blocks' in `d` distributed between threads and each thread operating on a number of blocks independently. As with multithreaded naive the number of threads should not exceed the number of CPU cores and `y` is shared between all threads.

2.5 Naive Optimised

Signed `perm_t` values, the restrict type qualifier for permutation array arguments to the function, and compilation with the `-march=native` flag (to target the compilation machine's hardware), results in assembly code using vector optimisations.

3 Implementation

The naive and Cooperman & Ma algorithms were implemented in C and Rust to ensure using Rust wasn't effecting performance. The multithreaded algorithms were written in Rust for efficient memory safety. The naive optimised and unrolled algorithms were implemented in C for fine grained optimisation and compilation control.

The implementations here aren't included for the sake of brevity as they are either essentially the same as the algorithm design or are obfuscated by optimisations.

Timings were done with calls to `clock_gettime`. Wall clock time was used, as opposed to CPU time, due to the latter's complications with multithreading. If the composition took less than 10 milliseconds the number of times the composition was done was doubled (growing exponentially) with a loop in between the `clock_gettime` calls until it took 10 milliseconds or more. This was done to remove issues with the accuracy of the system call for very small intervals. It's possible this may give faster than actual results for small permutations due to caching, but the effects should be consistent across algorithms.

The repository can be found at github.com/RyanGibb/uras-permutations, containing the implementations, assembly, testing script, benchmarking script, and benchmark CSV file.

4 Results

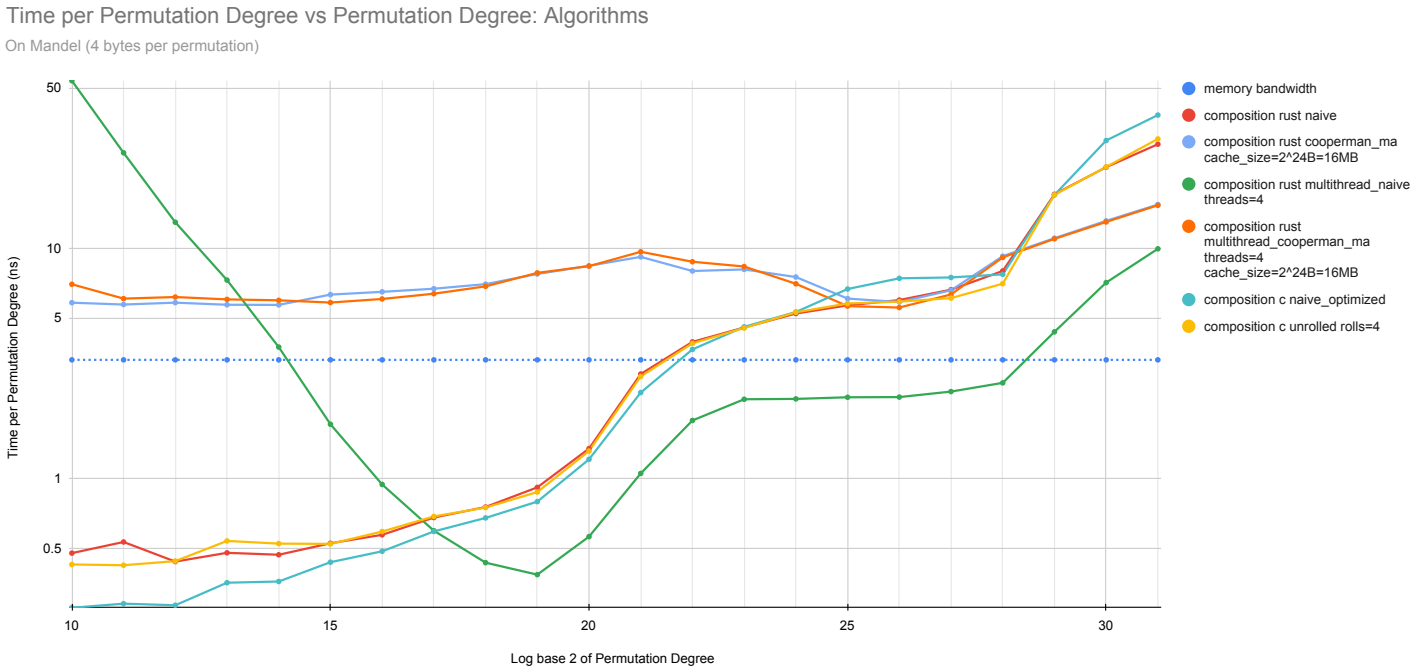

Figure 1: Time per Permutation Degree vs Permutation Degree

'On Mandel' refers the the machine that the benchmarks were done on. It has an Intel Xeon E3-1230 v5 and 64GB RAM. 4 bytes were used per permutation, giving a maximum permutation size of $2^{4*8=}2^{32}$. The amount of memory used per algorithm can be calculated by 3 (for of `x`, `y`, and `z` permutations) times the degree of the permutation times 4 bytes per permutation. For example, the largest included here is for permutations of degree 2^{32}, so $3 * 2^{32} * 4 \approx 51GB$.

The blue dotted line shows the memory bandwidth shows as 19.45 bytes per nanosecond for all reads, as measured by Intel® Memory Latency Checker v3.9[2]. This is the theoretical limit of the performance of the algorithms on this hardware as the permutation size approaches infinity.

When the algorithms outperform this it is due to caching. For example in figure 1, this limit is crossed for permutations of degree 2^{21}, at $2^{21} * 4 \approx 8MB$ per permutation. This is explained by the CPU having a shared L3 cache size of 8MB[3]. The reason why this is the size of one permutation, and not three, is because only `y` is being accessed randomly; `x` and `z` and read and written respectively sequentially.

The exception to this is `rust multithreaded_naive`. We can see the overhead of this is high for small permutations, but outperforms the non-multithreaded algorithms for degrees larger than 2^{18} which is about $1MB$. It plateaus from $2^{23} * 4 = 16MB$ to $2^{28} * 4 = 1GB$ where it finally fails to break the barrier.

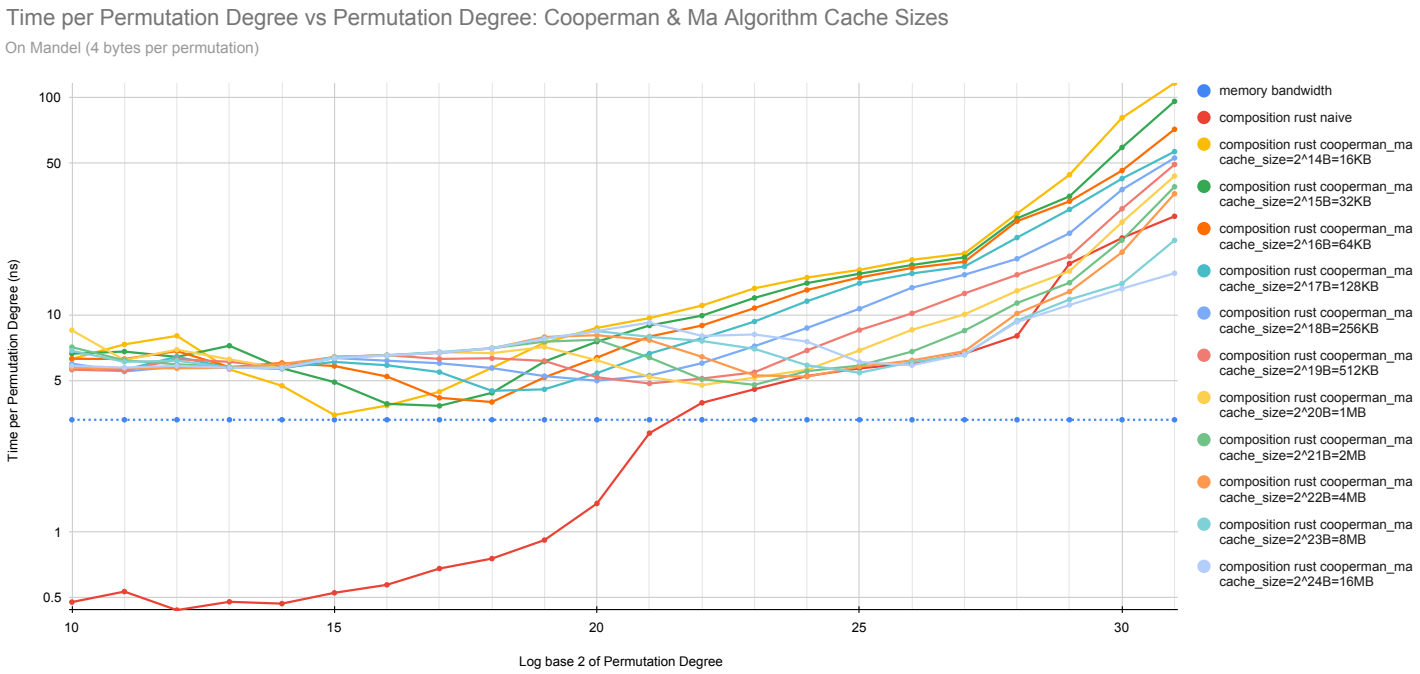

Figure 2: Cooperman & Ma Algorithm Cache Sizes

Figure 3: C

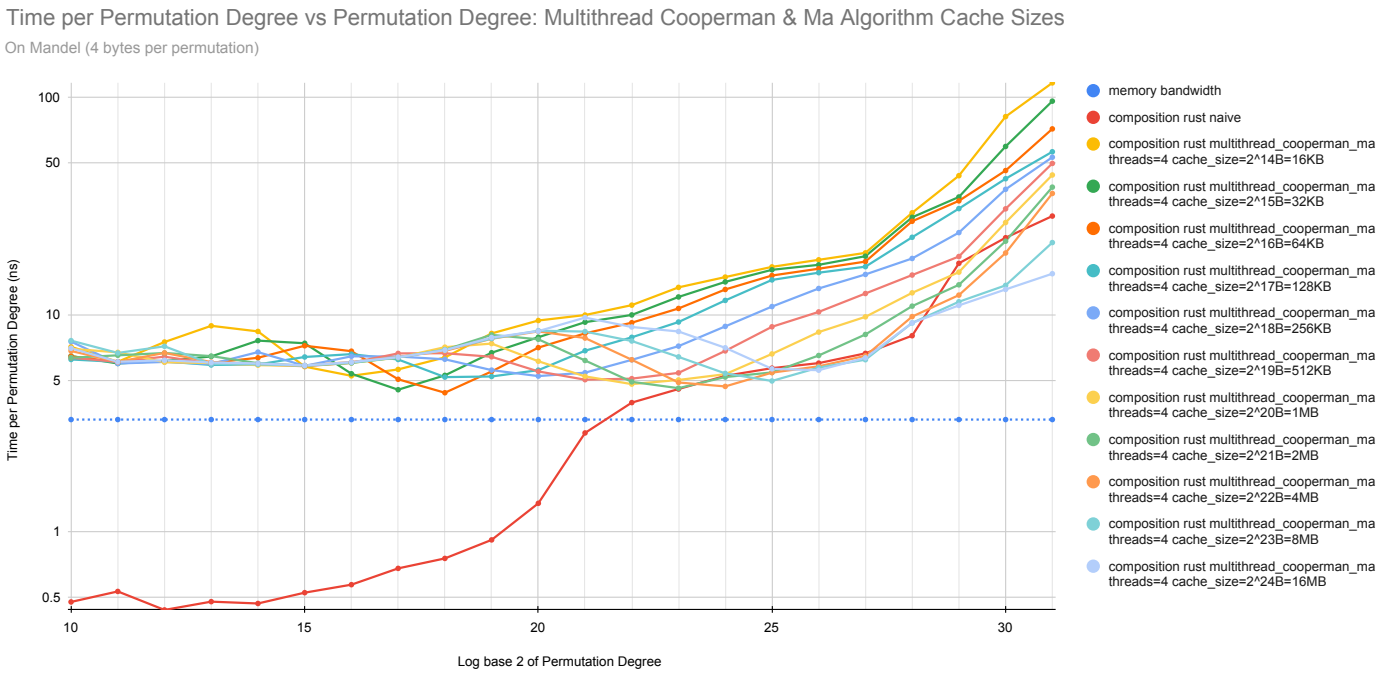

Figure 4: Multithread Cooperman & Ma Algorithm Cache Sizes

Figures 3 and 4 were used to determine the optimal cache size for single threaded and multithreaded Cooperman & Ma algorithms respectively. The naive algorithm was included as a comparison.

5 Conclusion

As can be seen the multithreaded naive algorithms outperform the bucket Copperman & Ma algorithm. Further work to be done includes exploring the effects of a multi-level bucket algorithm to take advantage of L1 and L2 caches and different memory latency's in NUMA systems; and creating an optimised multi-core bucket algorithm, building on the multithreaded Cooperman & Ma algorithm.

More investigation into the multithreaded algorithm outperforming the bandwidth limit for permutations up to 2^28 is required. It is possibly a result of the non-NUMA architecture. It is also possible that the latency results are simply slightly pessimistic.

Investigation into dynamically choosing the algorithm for different hardware and permutation sizes is another avenue of interest.

References

[1] G. Cooperman and X. Ma. Overcoming the memory wall in symbolic algebra: a faster permutation multiplication. *SIGSAM Bull.*, 36:1–4, 2002.

[2] Intel® memory latency checker v3.9. https://software.intel.com/content/www/us/en/develop/articles/intelr-memory-latency-checker.html.

[3] Intel® xeon® processor e3-1230 v5. https://www.intel.com/content/www/us/en/products/sku/88182/intel-xeon-processor-e31230-v5-8m-cache-3-40-ghz/specifications.html.