A Study of the Ska Sort Algorithm

Enda Healion

A Dissertation

Presented to the University of Dublin, Trinity College

in partial fulfilment of the requirements for the degree of

Master in Computer Science

Supervisor: Dr. David Gregg

April 2024

Declaration

I, the undersigned, declare that this work has not previously been submitted as an exercise for a degree at this, or any other University, and that unless otherwise stated, is my own work.

Enda Healion

April 15, 2024

Permission to Lend and/or Copy

I, the undersigned, agree that Trinity College Library may lend or copy this thesis upon request.

Enda Healion

April 15, 2024

A Study of the Ska Sort Algorithm

Enda Healion, Master in Computer Science

University of Dublin, Trinity College, 2024

Supervisor: Dr. David Gregg

Sorting algorithms play a very important role in software engineering, especially in use in databases where keys must be in sorted order so that search and merge functions are run more efficiently. As a result, it is important to have efficient algorithms that can sort a large number of inputs. Radix sorting algorithms are best suited to large input sizes due to their non-comparative sorting technique. Because of their importance, radix sorting algorithms are very well studied.

One of the main slow downs of in-place radix sorting algorithms, such as American Flag Sort, is the read and write dependencies when swapping elements. The Ska Sort algorithm shows significant performance improvements over American Flag Sort from its change in element swapping strategy. Despite the performance improvements, there is a gap in the literature about Ska Sort.

This paper studies the effects on the number of elements swapped with Ska Sort's new element swapping strategy. Due to the heavy usage of C++ templates and lambda functions, the changes in algorithm from American Flag Sort to Ska Sort is difficult to see. This paper takes the original implementation and extracts the algorithmic steps of Ska Sort. The original implementation contains optimisations which have not been proven to improve the runtime performance of the algorithms. This paper breaks down and analyses different sections of the algorithm into chapters to see if the optimisations provide runtime benefit or if there are any different optimisations that can be checked for performance improvements. Finally, this paper takes the learnings from Ska Sort and applies it to integer spreadsort to see if similar runtime improvements can be found.

Acknowledgments

I would like to thank my supervisor, Dr. David Gregg, for his guidance, expertise and invaluable feedback throughout this dissertation.

Enda Healion

University of Dublin, Trinity College
April 2024

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

A blog was written claiming to have created a radix sorting algorithm that out performed C++'s std::sort function (Skarupke (2016)). This new algorithm, called Ska Sort, is based on an existing radix sort algorithm called American Flag Sort with some algorithmic improvements.

The implementation of this algorithm is written in C++ with heavy usage of templates and lambda functions. Because of this, the code that controls how elements are swapped to their sorted position is split into different, separated functions, thus making it difficult to see the algorithmic change that was made to American Flag Sort's element swapping stage to create Ska Sort. In addition to this, the Ska Sort implementation may have some premature optimisations that further obfuscate the algorithm from being easily understood from the implementation.

Despite having large performance improvements over American Flag Sort, a documented and researched sorting algorithm (McIlroy et al. (1993)), no papers have been published discussing the Ska Sort algorithm and its properties. However, other papers have used Ska Sort as a fallback algorithm in their own algorithms (Obeya et al. (2019)). This dissertations aims to explain the algorithmic changes made to American Flag Sort to create Ska Sort algorithm, and to explore Ska Sort's properties compared to American Flag Sort.

1.2 Research Questions

This dissertation explores the algorithmic changes that were made to American Flag Sort to create Ska Sort, what optimisations of Ska Sort's implementation improve the runtime and what optimisations do not, and are there any new properties of Ska Sort that American Flag Sort does not?

1.3 Research Objective

1. Determine the effects of the algorithmic change in the way partitions are iterated over during the permute stage of Ska Sort, and investigate why there is a performance improvement.

2. Determine the effects of unrolling the inner for-loop in the permute stage on the runtime performance.

3. Determine what the overhead of keeping the newly added unsorted partition array is, if there are other methods of keeping track of unsorted partitions, and if this array improves overall runtime.

4. Study the properties of the changed partition iteration method in the permute stage of Ska Sort. Answer how many unsorted elements are remaining after iterating through each partition once.

5. Explore the effects of key size and input size on the runtime performance.

6. Explore existing implementations of the counting stage to see if there are any further runtime improvements.

7. Explore if the Ska Sort fallback thresholds to std::sort and American Flag Sort can be improved.

8. Apply Ska Sort's permute stage partition iteration strategy to a similar radix sorting algorithm, integer spreadsort, and study the runtime performance effects.

1.4 Dissertation Outline

Apart from being about the Ska Sort algorithm, each of the above research objectives are not necessarily directly related to each other. Having all of the methods, implementations, results, evaluations and conclusions grouped together for each separate research object

would make for a difficult read. As a result, the dissertation is split into multiple chapters, each of which will discuss their own associated research objective and have their own introduction, background, method, results and conclusion sections. However, background information that relates to all of the chapters is found in Chapter 2.

The following is an overview of the dissertation structure:

1. Chapter 1 - Introduction, presents the motivations behind why the Ska Sort algorithm is being studies in the dissertation, the research questions that will be answered, and detailed breakdown of the research questions into research objectives.

2. Chapter 2 - State of the Art, gives a detailed explanation of the necessary background information required to understand this dissertation and its results. This includes how American Flag Sort works, the algorithmic changes to American Flag Sort to create Ska Sort, and an explanation of the optimisations found in the original Ska Sort implementation.

3. Chapter 3 - Unsorted Items Remaining After One Iteration Through All Partitions, gives a novel proof for the the proportion of input elements that remain unsorted after a single iteration through each of the partitions in the permute stage, how many iterations through all partitions are required until all elements are in the correct position. The chapter also derives how the proportion of unsorted elements, for sufficiently large key sizes, approaches $\frac{1}{e}$.

4. Chapter 4 - Faster Counting, explores to what degree unrolling the for-loop body help to improve the runtime of the counting stage. The resulting performance effects are documented and analysed.

5. Chapter 5 - Unsorted Partition Tracking, examines the effects of the additional unsorted partition tracking on the runtime in the permute stage of Ska Sort. Four alternative methods are tested, to see which provides the best runtime results with different input value distributions.

6. Chapter 6 - Iteration Strategy, presents runtime and CPU performance counter results of American Flag Sort and Ska Sort. These results are analysed to see why this change in iteration method causes Ska Sort to have such a large increase in performance over American Flag Sort.

7. Chapter 7 - Permute Stage For-Loop Unrolling, presents runtime results of the permute stage unrolled for-loop, as found in the original implementation, and a

comparison to a version without for-loop unrolling. The effects of for-loop unrolling are analysed.

8. Chapter 8 - Key Size Effects, presents the effects of changing the key size, on the runtime of Ska Sort. An explanation is given for the reason different key sizes give different runtimes.

9. Chapter 9 - Fallback Threshold, determines at what input size Ska Sort algorithm should fallback to the American Flag Sort or std::sort depending on the input size. An explanation of the effects of these fallback algorithms on the total number of partitions sorted is given.

10. Chapter 10 - Improved Integer Sort, demonstrates that Ska Sort's partition iteration method found in the permute stage can be applied to integer spreadsort. Runtime results comparing the effects of these algorithms is also provided.

Chapter 2

State of the Art

This chapter will go through the necessary background information required to understand the contributions this paper provides about the Ska Sort algorithm. In order to fully understand this paper's finding, it is important to first understand the Ska Sort algorithm, and the algorithm that it is based on, American Flag Sort. In published papers about similar types of sorting algorithms, there are multiple terms that refer to the same or similar concepts. Therefore a list of terminology is also provided at the end of this chapter.

2.1 Background

2.1.1 American Flag Sort

The American Flag Sort algorithm is a most-significant-digit (MSD), non-comparative, in-place sorting algorithm that is useful for sorting large sets of data such as strings or integers in a number of passes McIlroy et al. (1993). The algorithm can be thought of in three stages: The counting stage, the permute stage, and the recursion stage.

1. **Counting & Offsets:** In the counting stage and starting at the most MSD, the algorithms begins by counting the number of distinct digit values in the input array. For implementations that use a byte as a digit, these counts are allocated onto the stack as a fixed 256 element array, where the ith index into this array gives the number of i digits found in the input array.

   This count array's values are updated using a prefix sum starting at zero to create the offset array. The values in this array represent where each of the digits should be swaps to sort the input array by the digits. While generating the prefix sum on the counts array, another array is created that stores the final offsets. Indexing into

the offset and final offset array using the *ith* digit gives the *ith* partition start and end point.

2. **Permute:** Starting at the first element in the input array, this element's digit is extracted. Using the digit as an index into the offset array tells the algorithm where to swap the element to. Because this swap location already has an element in that position, it is swapped with the current element. After this swap the element the value in the offset array for the current digit is incremented taking into account that the partition now has a sorted element.

   This continues until the element that was just swapped has a digit in the partition the algorithm is currently on. The algorithm then moves on to the next element in the partition. Once all elements are sorted, the input element's digits are in the correct position, however the array is still not fully sorted because the least significant digits still have to be sorted. This is considered a single pass of American Flag Sort.

3. **Recursion:** In the final stage, recursion is used to sort each partition on the next least significant digit. For example, on a uint32_t using a uint8_t as a digit or key, the most significant byte, bits 31 to 24, is sorted in a pass. Using these partitions, the next byte, bits from 23 to 16, are sorted and so on until the final least-significant-byte is sorted. If the partition size is less than the fallback threshold, sort the partition using insertion sort instead.

The algorithm is MSD because it starts at the most significant digit on the first pass and processes all digits until it gets to the Least-Significant-Digit the last pass. It is non-comparative because no elements are not directly compared, instead they are swapped into position using the offset array. Finally, the algorithm is in-place because the elements are swapped around in-place in the input array rather than allocating and copying to the output array.

2.1.2 Ska Sort

Ska Sort has the same stages as American Flag Sort, however there is a change in the way elements are iterated and swapped in the permute stage. As a result of this permute stage change, an additional array is created and maintained that keeps track of which partitions are unsorted Skarupke (2016).

1. **Counting & Offsets:** The counting stage in Ska Sort is identical to the one found in American Flag Sort. However, an additional array of indexes are stored. These indices keep track of which partition are sorted and unsorted. This is done by keeping the unsorted partitions on the left-hand side of this addition partition array. Unsorted partition are maintained on the left-hand side of the array using a custom implementation of std::partition as shown in 2.1. Partitions can be checked to see if they are sorted by checking if the digit's offset and final offset value are equal.

2. **Permute:** Using the additional partition array to find unsorted partitions, iterate over each element in the unsorted partition. At each element, get the element's digit, lookup it's sorted location and swap the elements. Move on to the next element in the partition array. Keep iterating through all unsorted partitions and swap the elements to the sorted position until all partitions are sorted.

3. **Recursion:** In the final stage, recursion is used to sort each of partition on the next least significant digit. For example, on a uint32_t using a uint8_t as a digit or key, the most significant byte, bits 31 to 24, is sorted in a pass. Using these partitions, the next byte, bits from 23 to 16, are sorted and so on until the final least-significant-byte is sorted. If the partition size is less than the American Flag Sort fallback threshold sort the partition using American Flag Sort instead. Or if the partition size is less than the std::sort fallback threshold, sort the partition using std::sort.

Figure 2.1: The Partition Array stores the indexes of unsorted partitions on the left-hand side of the array, marked in green, and sorted partition that no longer need to iterated over on the right-hand side.

In summary, Ska Sort makes the following two algorithmic changes, the effects of which will be studied in this dissertation:

1. In the permute stage, elements in a partition are now iterated from the partition start to the partition end and move to the next element regardless if the swapped element's key is now in the correct partition.

2. Because the new permute stage iteration method no longer guarantees that elements are sorted when all partitions have been iterated over, an additional array is needed to keep track of which partitions are unsorted.

Additionally, the Ska Sort implementation, found in (Skarupke (2024)), has an unrolled for-loop in the permute stage where elements are swapped to their correct partition. This claim is also investigated. Other background information is explained on a per-chapter bases, when they are needed.

2.1.3 Terminology

1. **Key**. A key, or digit, is the part of a data element that is used for sorting. For example, it is common to use a uint8_t as a key for sorting integers. For a uint32_t element, four passes of a uint8_t are required to fully sort the input.

2. **Partition**. A partition is a group of elements were all keys have the same value. Depending on the paper these can also be called piles, bins or buckets.

3. **Most Significant Digit (MSD)**. Processing keys of an element in passes starting with the most significant digit and ending on the least significant digit.

4. **Pass**. A pass refers to a single complete processing of the input array using the counting and offset creation stage and the permute stage. The recursive stage starts sorting the partitions in a new pass on the next digit.

2.2 Related Work

There is a gap in the literature about different methods of iteration through elements in the permute stage for non-parallel implementation. However, there exists a significant amount of literature that discusses and compares changes in this permute stage for parallel implementations (Amato et al. (1998)).

According to (Maus (2015)), the reason there has been a greater focus on multi-core sorting algorithms compared to single-core sorting algorithms in recent times is because the single-core performance, more specifically the clock speed, is not what CPU manufacturers are focused on for performance gains. CPU manufacturers have been putting more

cores on chips and improving functionality for working with multiple cores. As a result, to get the most performance out of modern CPU's, and GPU's, a multi-core sorting method must be used.

Although, there are few papers discussing the permute stag iteration method, other papers have discussed how the memory subsystems in modern CPU's can affect the performance of this stage. If the number of partitions is greater than the number of Translation Lookaside Buffer (TLB) entries, then there will be performance hit from TLB misses (Polychroniou and Ross (2014)). Memory is accessed using virtual addresses, however these need to be translated to physical addresses for the CPU to use. This translation is an expensive process, so the TLB caches a small number these translations. As a result, if there are too many partitions, then virtual address will have to re-translated to physical addresses.

Other related works are mentioned on a per-chapter bases, when they are needed.

Chapter 3

Unsorted Items Remaining After One Iteration Through All Partitions

After a single iteration through all partitions in American Flag Sort's permute stage, all elements are sorted. This is because the algorithm only advances to the next element until the current element is in the correct position. However, Ska Sort does not have property since there is no guarantee that when the algorithm moves on to the next element, the previous one is sorted. This raises a question: How many unsorted elements remain after a single iteration through all of the partitions in the permute stage? This chapter presents a novel proof of the proportion of unsorted elements after an iteration through all partitions in the permute stage for a uniform random input and the number of iterations required to sort all partitions.

3.1 Terms and Lemmas

3.1.1 Terms

1. Let N be the number of element in the input.
2. Let R be the size of the key in bits.
3. Let P be the number of partitions. $P = 2^R$.
4. Let S be the average size of each partition. $S = \frac{N}{P}$.

3.1.2 Lemmas

Lemma 1. $\boldsymbol{S = \frac{N}{P}}$**.**
Since the input is from a uniform random distribution, there is a $\frac{1}{P}$ *chance of each element*

being in a given partition. This means that there are $N \times \frac{1}{P}$ chances that an element is in a given partition, and so the expected size of the partition will be: $N \times \frac{1}{P} = \frac{N}{P}$.

Lemma 2. ***All elements in a partition are iterated over.***
The algorithm iterates from the start of the partition to the end of the partition. There are no control flow statements that would cause an element not to be iterated over.

Lemma 3. ***One element is sorted per element iterated over.***
For each element iterated over, it is guaranteed to sort one element. This is because the element is swapped into it's correct partition using the offsets array which is then updated to account for an element being swapped into the partition.

Lemma 4. ***A partition of size Q will sort Q new elements.***
By combining Lemma 2 and Lemma 3, if all elements in a partition are iterated over, and each of those elements is swapped into the correct position, then the number of newly sorted elements is equal to the number of elements iterated over in the partition.

Lemma 5. ***After an iteration of a partition, all partitions will be $\frac{P-1}{P}$ of their original size.*** *From Lemma 4, if a partition is iterated over, that many elements will be sorted. All of these sorted elements will be moved to one of P partitions. Since the input values are uniformly random, there is a $\frac{1}{P}$ chance that an element will be swapped into any one of the P partitions. Considering a single partition, it means that $\frac{1}{P}$ proportion of the partition will be swapped into any one partition. As a result, the partition will be $1 - \frac{1}{P}$ of it's original size. $1 - \frac{1}{P} = \frac{P}{P} - \frac{1}{P} = \frac{P-1}{P}$.*

3.2 Proof

Using the above terms and lemmas, the proof of the proportion of unsorted of elements after a single iteration through all partitions can be found:

1. Each of the P partitions will start off being of size S according to Lemma 1.
2. The first partition will be of size S.
3. After iterating over the first partition, all partitions will be $\frac{P-1}{P}$ of their original size according to Lemma 5.
4. The second partition will be of size $S \times \frac{P-1}{P}$.
5. After iterating over the second partition, all partitions will be $\frac{P-1}{P}$ of their original size according to Lemma 5.

6. The third partition will be of size $S \times \frac{P-1}{P} \times \frac{P-1}{P} = S \times (\frac{P-1}{P})^2$.

7. This continues until the last partition where $S \times \left(\frac{P-1}{P}\right)^{P-1}$ elements are iterated over.

8. The total number of sorted elements can be found by summing all of the elements that were iterated over, which is the size of each partition per Lemma 4.

9. This gives the following sum: $S + (S \times \frac{P-1}{P}) + (S \times \left(\frac{P-1}{P}\right)^2) + \cdots + (S \times \left(\frac{P-1}{P}\right)^{P-1})$.

10. This is a finite geometric series, the sum of which can be found using the following formula:

$$\sum_{k=1}^{n} ar^{k-1} = \frac{a(1-r^n)}{1-r}$$

, where a represents the starting value in the series,
r represents the common ratio between the elements and,
n represents the number of terms in the series.

11. Substituting a for S, r for $\frac{P-1}{P}$, and n for P:

$$\sum_{k=1}^{P} S\left(\frac{P-1}{P}\right)^{k-1} = \frac{S(1-\left(\frac{P-1}{P}\right)^P)}{1-\frac{P-1}{P}}$$

12. The following steps can be taken to simplified the equation further:

$$= \frac{S(1-\left(\frac{P-1}{P}\right)^P)}{\frac{P}{P}-\frac{P-1}{P}}$$

$$= \frac{S(1-\left(\frac{P-1}{P}\right)^P)}{\frac{P-(P-1)}{P}}$$

$$= \frac{S(1-\left(\frac{P-1}{P}\right)^P)}{\frac{1}{P}}$$

$$= SP(1-\left(\frac{P-1}{P}\right)^P)$$

From Lemma 1 $S = N/P$. $N = S \times P$

$$= N(1-\left(\frac{P-1}{P}\right)^P)$$

13. The proportion of unsorted elements will be one minus the proportion of sorted elements:

$$= N(1 - (1 - \left(\frac{P-1}{P}\right)^P))$$

$$= N(1 - 1 + \left(\frac{P-1}{P}\right)^P)$$

$$= N\left(\frac{P-1}{P}\right)^P$$

The above equation gives a proportion of the elements that remain unsorted after a single iteration through all the partitions. It was found that this value only depends on the number of partitions, which is determined by the size of the key in bits.

3.3 Results

3.3.1 Comparison To Real-World Results

To see if this equation is accurate compared to real world results, the equation was tested against real values using 2^{30} uniformly random elements as inputs. As the algorithm ran, information about how many elements were sorted and unsorted were kept track of. The following is a comparison of the expected values compared to the actual values for a select number of partition sizes:

Partition Count: 2		
Metric	**Expected**	**Actual**
Sorted Proportion	0.750000	0.749994
Unsorted Proportion	0.250000	0.250006

Table 3.1: This and the following tables show how the expected results almost exactly match the actual results for a range of partition counts

Partition Count: 64		
Metric	**Expected**	**Actual**
Sorted Proportion	0.635013	0.635009
Unsorted Proportion	0.364987	0.364991

From the above tables, it is clear that the above equation does accurately reflect what happens in the algorithm, since the actual results differ from the expected results by

Partition Count: 256		
Metric	**Expected**	**Actual**
Sorted Proportion	0.632840	0.632833
Unsorted Proportion	0.367160	0.367167

approximately 0.001%. However, inputs with fewer elements will see a larger difference between expected and actual results because the equation assumes that the partitions are close or exactly equal in size. This relies on the Law of Large Numbers, where the larger the input element count, the closer the partition sizes will be to the expected size. Once the input is sufficiently small, the greater the variation in partition sizes.

3.3.2 Proportions Approaching a Constant Value

As the number of partitions increase, the expected sorted proportion converges on the value: 0.632120559, which is close to $1 - \frac{1}{e}$. The reasons this happens can be seen when the equation is rearranged:

$$1 - \left(\frac{P-1}{P}\right)^P$$

$$= 1 - \left(\frac{P}{P} - \frac{1}{P}\right)^P$$

$$= 1 - \left(1 - \frac{1}{P}\right)^P$$

From Bernoulli trials in probability theory (Dr Richard Gibbens (2016)), it is known that:

$$\lim_{n\to\infty} \left(1 + \frac{x}{n}\right)^n = e^x$$

Let $x = -1$

$$\lim_{n\to\infty} \left(1 - \frac{1}{n}\right)^n = e^{-1}$$

$$= \frac{1}{e}$$

Therefore the proportion of sorted elements, when there are sufficiently enough partitions 3.2, will approximately equal $1 - \frac{1}{e}$, and the proportion of unsorted elements will approximately equal $\frac{1}{e}$. For 8 bit keys, this means that the number of unsorted elements remaining per iteration through all the partitions will approximately be $\frac{N}{e}$

n	$\left(1-\frac{1}{n}\right)^n$	$\frac{1}{e}$	Absolute Difference
10	0.348678	0.367879	0.019201
100	0.366032	0.367879	0.001847
1000	0.367695	0.367879	0.000184
10000	0.367861	0.367879	0.000018

Table 3.2: This table shows how the above equation approaches $\frac{1}{e}$ as n increases.

3.3.3 Iterations Until Fully Sorted

Based on these results, it is clear that having a small key, and therefore fewer partitions, sorts a greater proportion of elements per iteration of all the partitions compared to having a larger key based on the tables 3.1. To determine the number of sorting iteration required to fully sort an input with a given number of partitions the following equation can be used: $N \times U^k = 1$, where U is the proportions of elements that are unsorted after a single iteration over all partitions, and k is the number of iterations required to sort an input of size N until there is only one element. This last element will be sorted since all other elements will be in the correct position, thus leaving no incorrect positions for the final element to be in. Rearranging the formula to find k:

$$N \times U^k = 1$$

$$U^k = \frac{1}{N}$$

$$k = \log_U(\frac{1}{N})$$

$$k = -\log_U(N)$$

$$k = \lceil -\log_U(N) \rceil$$

Using the above formula and the number of bits in a key, the number of iterations can be found.

Comparison of Expected vs Real Iterations			
N	**Key Size**	**Expected Iterations**	**Actual Iterations**
2^{24}	1	12	12
2^{24}	2	15	16
2^{24}	3	16	16
2^{24}	4	17	17
2^{24}	5	17	16
2^{24}	6	17	17
2^{24}	7	17	17
2^{24}	8	17	17
2^{24}	9	17	17
2^{24}	10	17	17
2^{24}	11	17	17
2^{24}	12	17	17

Table 3.3: This table shows how many iterations are required to sort all keys of a range of sizes into the correct partitions for 2^{24}. Apart from when the key size is two and five, the expected iterations match the actual iterations, suggesting that the formula is accurate.

It is clear from table 3.3 that this equation can, to a reasonable degree of accuracy, predict how many iterations over all the partitions are required until all keys are in the correct position.

3.4 Conclusion

This chapter shows that, with a uniform random input, the proportion of sorted and unsorted elements can be found after a single iteration over all the partitions using the size of the key in bits. It was also found that as the size of the key increases, the proportion of elements left unsorted also increases. However, this increase in unsorted elements only goes to a maximum of $\frac{1}{e}$. Additionally, this chapter presents a method of calculating how many iterations are required to sort all keys into their correct partition based on the size of the input, and the proportion of unsorted elements.

3.4.1 Further Work

The number of unsorted elements after a single iteration through all of the partitions is found only for uniform random inputs. Different input value distributions, such as geometric random inputs, will have different properties than the ones mentioned in this paper.

The formula for the number of iterations to sort the partitions can be updated to account for the fact that all but one partition needs to be sorted in order for all elements

to be in the correct partition. This is because if all but one partition are sorted, the last remaining partition must also be sorted too because there are no more insertion points in the offsets array outside of the last partition. As a result, it must be the case that the last remaining partition is also sorted because there are no locations for the element to be swapped to. Taking this into consideration could reduce the number of expected iterations required to sort the partitions.

Chapter 4

Faster Counting

Depending on the input array type and how the key is extracted, the counting stage can take anywhere from 30% to 50% of the total runtime as seen in tables 4.1 and 4.2. As a result, it is important to see if there are any performance gains in this stage, since it has a large impact on the overall runtime performance. The goal of this chapter is to investigate other implementations of the counting stage that will decrease the total runtime of Ska Sort. This chapter uses a previously known method of increasing count performance using for-loop unrolling, and investigates what degree of for-loop unrolling provides the best runtime results.

4.1 Background

While researching similar non-comparative sorting algorithms, I came across a paper (Rahman and Raman (2002)) which suggests methods for reducing Translation Lookaside Buffer (TLB) misses. Although not directly related to this topic, the paper provides an implementation of an LSD radix sort which contains a four-time unrolled for-loop in the counting stage. To expand upon this finding, I experimented with unrolling the for-loop body two, four, six and eight times to see the performance impact.

For-loop unrolling is an optimization technique used to increase a programs performance by decreasing the number of loop condition checks and, depending on the specific code, increases the CPU's ability perform out-of-order executions if there are no a data dependency between instructions (Allan et al. (1995)).

```
size_t counts[256] = { 0 };
for (size_t i = input_start; i < input_end; ++i) {
    uint8_t byte = extract(input[i], byte_index);
    ++counts[byte];
}
```

Listing 4.1: This code goes through each element in the input, extracts a byte, and increments the number of elements for each byte that is found. If the value of 'byte' is the same from the previous iteration, the CPU must stall until the incremented value of in the counts array is propagated into memory before doing the next iteration.

```
constexpr size_t unroll_count = 4;
size_t counts0[256] = { 0 };
size_t counts1[256] = { 0 };
size_t counts2[256] = { 0 };
size_t counts3[256] = { 0 };
for (size_t i = input_start; i + unroll_count <= input_end; i += unroll_count) {
    uint8_t byte0 = extract(input[i + 0], byte_index);
    uint8_t byte1 = extract(input[i + 1], byte_index);
    uint8_t byte2 = extract(input[i + 2], byte_index);
    uint8_t byte3 = extract(input[i + 3], byte_index);
    ++counts0[byte0];
    ++counts1[byte1];
    ++counts2[byte2];
    ++counts3[byte3];
}
// Do remaining elements
// ...
for (size_t i = 0; i < 256; i++) {
    counts0[i] += (counts1[i] + counts2[i] + counts3[i]);
}
```

Listing 4.2: The for-loop is unrolled four times with four unique counting arrays. If 'byte0' and 'byte1' are the same, the CPU does not have to stall before incrementing 'count0' or 'count1' because there is no data dependency between these instructions. This reduces the amount of time the CPU stalls before doing work.

In the first code listing 4.1, the code must wait for the ++counts[byte] to finish executing before moving on to the second iteration. While the incremented value propagates back input counts[byte], the CPU does not have work to perform and so stalls. By unrolling the loop as shown in code listing 4.2, the CPU can schedule the instructions for

extracting byte0, byte1, byte2 and byte3 since there are no data dependencies between these instructions. By the time byte0 has been extracted and propagated into a register, it can schedule the ++counts0[byte0] and other increment instructions without stalling. This reduction in stall leads to an increase in runtime performance.

4.2 Method

To compare the Ska Sort counting method against the four unrolled variants: unrolled two, four, six, and eight times, each algorithm's runtime was measured across a range of input sizes, type sizes, and value distributions.

The runtime tests were run using the following input types: uint8_t, uint16_t, uint32_t, and uint64_t. Each of the algorithms were tested using random input sizes from 2^{12} to 2^{29}, doubling each time. The values of these inputs used two distributions: uniform random, and geometric random variables. C++'s uniform_int_distribution function (cppreference (2024b)) and geometric_distribution (cppreference (2024a)) function with a value of 0.8 for the distribution parameter, were used was used to generate the random values. To remove random variations in the timings, each algorithm was run five-hundred times and timed using C++'s chrono library nanosecond high resolution clock.

4.3 Results

The results for unrolling the loop a number of times against various input sizes for uniformly random inputs is found in figure 4.1. Unrolling the loop provides runtime improvements in the counting stage. However, the difference between unrolling the for-loop four, six, or eight times is small. As a result, the four times unrolled counting stage would be preferred because it has approximately the same speed benefits as six and eight times unrolled while using less memory.

The results for unrolling the loop a number of times against various input sizes for geometric random inputs is found in figure 4.2. With geometric random input, unrolling the counting loop has a large effect compared to uniform random inputs. This is because with a geometric random input, all values are in the range 0 to 5, most of which are either 0, 1 or 2. This means that for a majority of loop iterations, the previous increment of the count leads to a stall in order for the updated value to propagate into memory, before incrementing the counts again using the same digit and therefore index in the

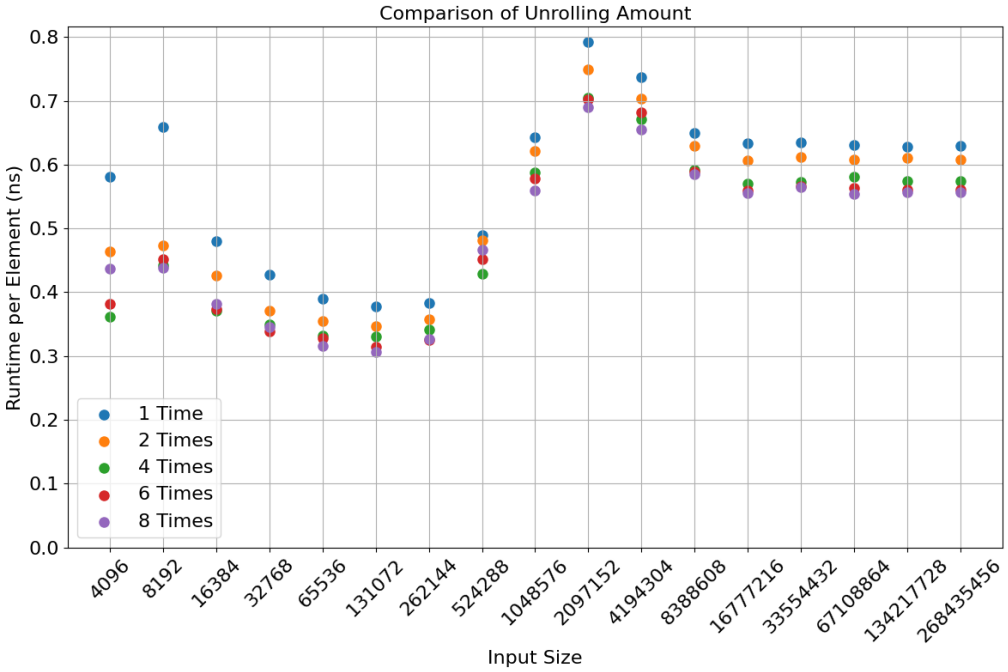

Figure 4.1: Graph of average nanoseconds taken to count a single element of a uniform random input, for a range of unrolling amounts. The more unrolled the input, the faster the runtime with 4, 6, and 8 times unrolled being approximately the same speed. At approximately 250,000 input elements, all times get significantly slower due to the input no longer fitting inside the CPU's 1.25 MB L2 cache. As a result, the values have to be loaded from main memory which is slower than from cache.

next iteration. By having multiple count arrays, data dependencies can be avoided by doing multiple increments across different arrays. This can be seen in the graph where the time for the two-time unrolled implementation is approximately twice as fast as the non-unrolled version. However, the four, six and eight timed unrolled implementations, also an approximate four-time speed increase compared to the non-unolled version.

Percent of Time Counting - Uniform Random					
Size	**Non-Unrolled**	**2x Unrolled**	**4x Unrolled**	**6x Unrolled**	**8x Unrolled**
2^{24}	44.63%	33.90%	26.18%	24.16%	24.13%
2^{25}	45.76%	33.59%	25.27%	23.98%	24.40%
2^{26}	45.53%	33.23%	25.57%	23.73%	24.55%
2^{27}	45.66%	33.22%	25.35%	24.07%	24.39%
2^{28}	45.56%	33.23%	25.55%	24.03%	24.48%
2^{29}	45.63%	33.30%	25.52%	24.16%	24.48%

Table 4.1: This table shows the amount of time spent in the counting stage for a single pass of Ska Sort on uniform random input. The implementations that have an unrolled counting stage take up less time as a percentage of total runtime.

Percent of Time Counting - Geometric Random					
Size	**Non-Unrolled**	**2x Unrolled**	**4x Unrolled**	**6x Unrolled**	**8x Unrolled**
2^{24}	46.92%	33.35%	25.49%	24.65%	25.04%
2^{25}	46.89%	33.12%	25.29%	24.37%	25.22%
2^{26}	47.07%	33.22%	25.56%	24.19%	24.78%
2^{27}	46.83%	33.29%	25.49%	24.18%	24.91%
2^{28}	46.87%	33.19%	25.48%	24.14%	24.98%
2^{29}	46.99%	33.33%	25.54%	24.08%	25.07%

Table 4.2: This table shows the amount of time spent in the counting stage for a single pass of Ska Sort on geometric random input. The implementations that have an unrolled counting stage take up less time as a percentage of total runtime.

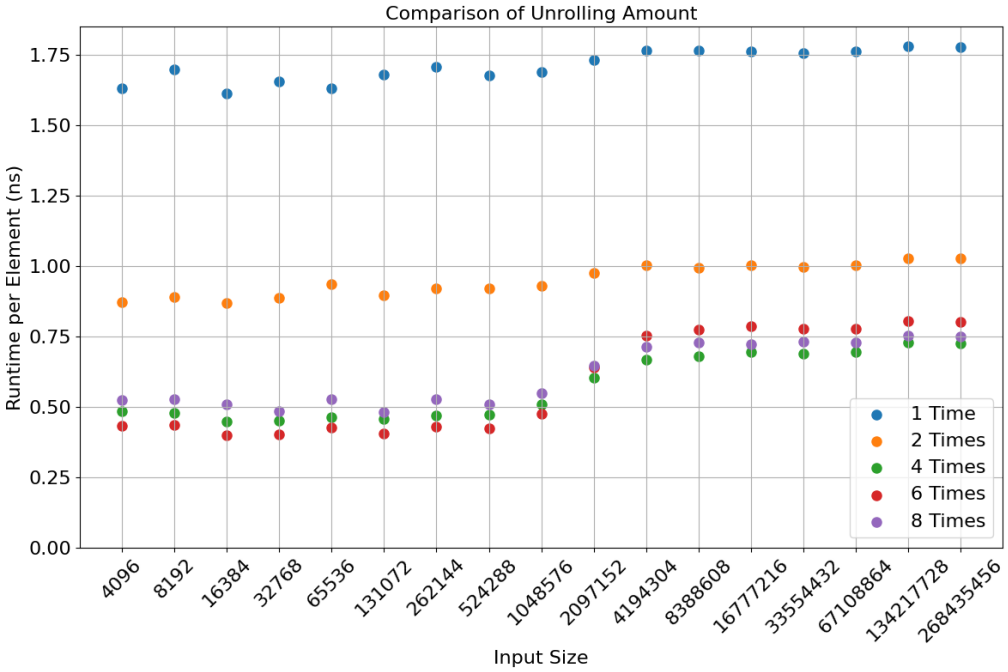

Figure 4.2: Graph of average nanoseconds taken to count a single element of a geometric random input, for a range of unrolling amounts. The more unrolled the input, the faster the runtime with 4, 6, and 8 times unrolled being approximately the same speed. At more than 250,000 input elements, the time per element grows because the input no longer fits inside the CPU's 1.25 MB L2 cache. As a result, the values have to be loaded from main memory which is slower than from cache.

4.4 Conclusion

From the results, it is clear that unrolling, regardless of how many times and the input distribution, decreases the time taken to count a single element, thus increasing the overall runtime performance. The results suggest that unrolling four-times is the best overall unrolling amount because of its similar runtime performance of the eight-timed unrolled implementations, while having half of the memory usage. Using four-times unrolling, the total amount of time spent counting can be reduced from approximately 45% to 27% according to tables 4.1 and 4.2.

4.4.1 Further Work

For unsigned integer inputs, the use of SIMD instructions (Intel Corporation (2024a)) could be used to speed up the runtime further. Multiple elements can be loaded using a single instruction and some considerations must be to extract the key from each SIMD

lane. In the case of uint64_t integers, and using a 256 bit wide register, 4 elements could be processed at a time. However four independent arrays must still be used to remove the loop data dependency.

Chapter 5

Unsorted Partition Tracking

The American Flag Sort algorithm only moves on to the next element in the input array when the current value is in the correct location. As a result, after a single iteration through all of the partitions in American Flag Sort, all elements are sorted. However, Ska Sort does not have this property since there is no guarantee that when the algorithm moves on to the next partition, the previous one is sorted. In order to keep track of unsorted partitions, the original Ska Sort implementation has an additional array that stores the unsorted partitions and keeps track of which have been sorted using a custom implementation of std::partition. The custom std::partition keeps unsorted partitions on the left of the array and sorted partitions on the right as shown in figure 5.1. The objective of this chapter is to answer if this additional array and the overhead of the custom std::partition is necessary and if there is a faster way of keeping track of unsorted partitions.

Figure 5.1: The Partition Array stores the indexes of unsorted partitions on the left-hand side of the array, marked in green, and sorted partition that no longer need to iterated over on the right-hand side.

5.1 Background

This section will give an introduction to each of the unsorted partition tracking methods.

5.1.1 Custom std::partition

When the offsets and final offsets are created using the prefix sum, an additional array is created that stores all of the partitions that contain more than one element. Partitions that contain one element or less are already sorted. The unsorted partitions are appended to the fix sized array from left to right, therefore keeping all unsorted partitions on the left. The custom version of std::partition, which separates the array into two groups, takes a lambda function to determine if a given partition is sorted or unsorted. When a partition is determined to be sorted, the custom std::partition is called and the array is split into unsorted and sorted groups. This is the algorithm used in the original Ska Sort implementation (Skarupke (2024)).

5.1.2 No Partition Tracking

This implementation does not require an additional array to store unsorted partitions, or any additional overhead by maintaining the unsorted arrays. Instead, the algorithm keeps doing a single sort iteration through all of the partitions. If the partition is sorted, the partition is skipped and the total number of sorted partition is increased. If after all partitions have been iterated over, and the number of sorted partition is not equal to the total number partitions, then the number of sorted partitions is reset to zero and the algorithm does another sorting pass through all of the partitions. This keep happening until all the number of sorted partitions equals the number of partitions.

5.1.3 Swap-to-end

This implementation is similar to the custom std::partition implementation where unsorted partitions are kept on the left-hand side of the additional unsorted partition array. However, the method of keeping the unsorted partitions on the left-hand side is different. After doing the sort iteration through a partition, the partition is checked to see if is is sorted or not. If it is sorted, the now-sorted partition is swapped with the last element in the unsorted elements and size the array containing unsorted elements decreases in size by one.

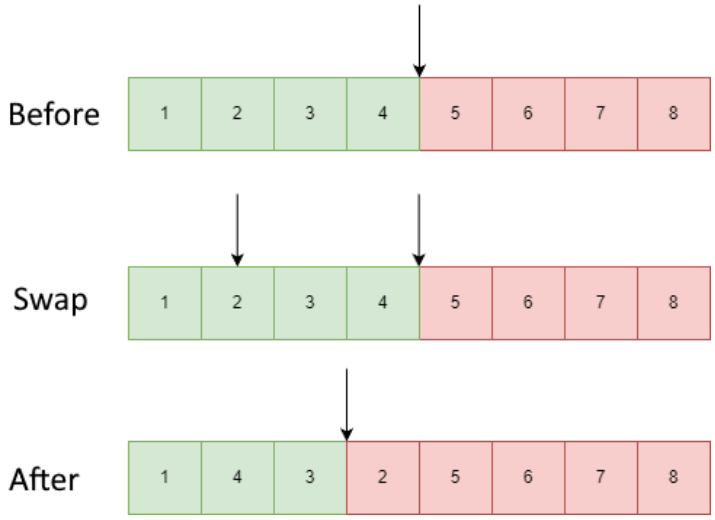

Figure 5.2: The top line "Before" shows the unsorted partitions marked in green, all of which are on the left-hand side of the array, and the sorted on the right-hand side. In the middle line "Swap", partition 2 was determined to be sorted. As a result, this partition is swapped with the last unsorted element, and the unsorted array subsection shrinks as shown in the bottom line "After."

5.1.4 C++ bitset

Another method of keeping track of the unsorted partitions is to have a bit either set or unset depending on if the partition is sorted or unsorted. When the ith partition is sorted, the ith bit of the bitset can be set to zero. The sorting is terminated when all bit are set, meaning all partitions are sorted.

5.2 Method

The effects of each partition tracking strategy on runtime performance were determined by running each of the strategies across a range of input sizes and input value distributions. To determine the overhead of each method, the total time in nanoseconds was taken to sort a single pass of the input. This time was then normalised with the input sizes, thus giving the average amount of time to sort one item with each strategy.

The runtime tests were run using the following input types: uint8_t, uint16_t, uint32_t, and uint64_t. Each of the algorithms were tested using input sizes from 2^{12} to 2^{28}, doubling each time. The values of these inputs used two distributions: uniform random, and geometric random variables with a value of 0.8 for the distribution parameter. To remove random variations in the timings, each algorithm was run thirty times and timed using C++'s chrono library high resolution clock.

5.3 Results

It was found that the default implementation for tracking the unsorted partitions is consistently the slowest method on uniform random inputs as found in figure 5.3. This is because the entire additional partition array has to be re-partitioned each time a partition is no longer unsorted. This re-partitioning step requires a scan over all partitions to determine if they should be placed on the left or right side of the array. For smaller input sizes, this takes a significant amount of time relative to the total runtime. The Swap-To-End, Break and C++ bitset methods have the fastest runtime performance across all ranges of input size due to the minimal overhead to either update the additional partition array in the case of Swap-To-End or the minimal time taken to determine if a partition is sorted in the case of the Break and C++ bitset methods.

However, the Break and C++ bitset methods perform poorly on geometric random inputs as found in figure 5.4. This is because the geometric values are only spread across six different partitions. As a result, the Break method spends most of its time iterating over partitions that are already sorted. This is time spent iterating over partitions, when it could be spent swapping elements and so there is large overhead per element swapped as shown in the graph by the data points being relatively high compared to the other implementations. The C++ bitset method also has the same issue where already sorted partitions are iterated over multiple times unnecessarily. Both the default and Swap-To-End methods perform approximately as well as each other, with Swap-To-End having slightly less overhead per element at large input sizes. In both the uniform random and geometric random inputs across all ranges of input sizes, the Swap-To-End method of tracking unsorted partitions has the least overhead cost per element sorted.

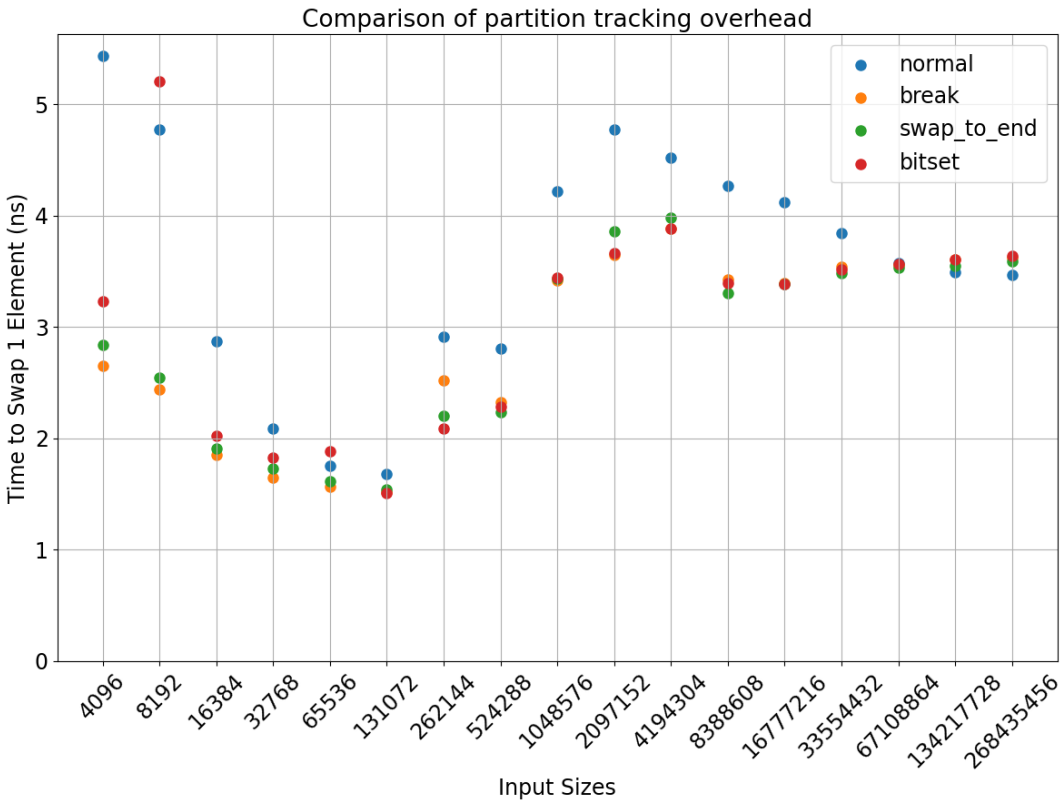

Figure 5.3: This figure shows the average time taken in nanoseconds to sort one element for a single pass over on uniform random values. The 'normal' data point is the one found in the default implementation of Ska Sort. At most input sizes, except 2^{28}, this implementation is either the slowest or the second slowest method of keeping track of unsorted partitions. All elements see an increase in overhead because the input element no longer fit in the CPU's L2 cache, which results in longer load times.

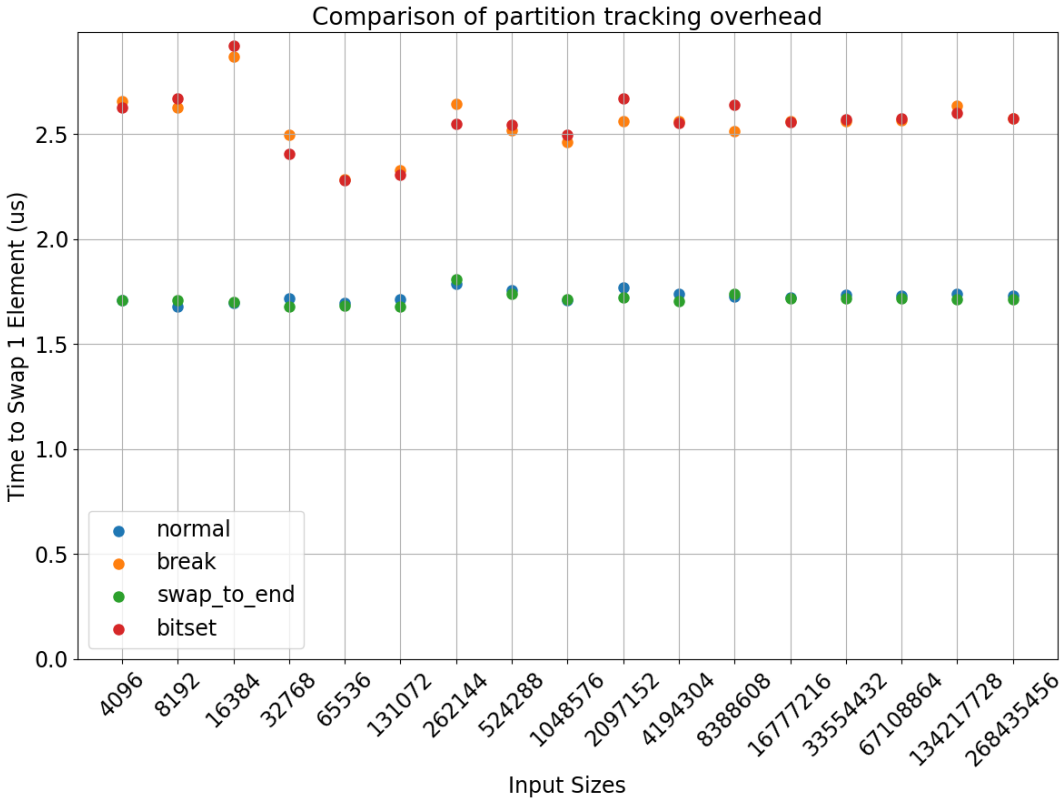

Figure 5.4: This figure shows the average time taken in nanoseconds to sort one element for a single pass over on geometric random values. The graph shows that the 'normal' and the 'swap-to-end' methods are approximately equal for all input sizes.

5.4 Conclusion

In conclusion, this chapter has found that, for uniform random values, keeping track of unsorted partitions does not degrade performance as measured as average time taken to sort one element. However, for geometric random values, or any other distribution that contains fewer than the maximum number of partitions, the additional unsorted partition array does improve the runtime by reducing the number of already-sorted partitions that are iterated over. It was found that the Swap-To-End method has less overhead compared to the method found in the original Ska Sort implementation for both uniform and geometric random input values.

Chapter 6

Iteration Strategy

In the Ska Sort blog post (Skarupke (2016)), the author suggests that the main performance increase from American Flag Sort to Ska Sort is because of a change in the way elements are iterated over when doing swaps in the permute stage. In American Flag Sort's permute stage, the algorithm keeps swapping elements to their final position until an element with the same key as the current partition is swapped into the current partition. The algorithm then moves on to the next element. Ska Sort changes this iteration strategy by swapping each element in a partition to their correct position and moving on to the next element regardless of where the elements were swapped. The Ska Sort author claims that this change in iteration strategy increases the Instructions Per Cycle (IPC) by approximately 40% from 1.61 to 2.24, thus improving runtime performance. The objective of this chapter is study the change of iteration strategy from American Flag Sort to Ska Sort, to see if the above performance claims are correct and to explain why there is a change in performance.

6.1 Background

6.1.1 CPU Pipelining

Instruction pipelining is a technique used in modern CPU architectures to improve instruction throughput. This is done by dividing the execution process into different stages each of which are performed by a dedicated piece of hardware on the CPU. A highly simplified version of modern CPU pipelining is the following: Instruction Fetch, Instruction Decode, Execute, Memory Access, Register Write Back.

For each CPU clock cycle each of the following stages can be executed once. This means that as one instruction is being executed, another instruction can be decoded or

fetched, have a memory access or registers write back. As a result, CPU's can execute instructions that are overlapped. This is called instruction level parallelism (Hennessy and Patterson (2011)), typically measured in Cycles Per Instruction (CPI) or Instructions Per Cycle (IPC). IPC = $\frac{1}{CPI}$. The approximate CPI for a program can be found using the equation: Pipeline CPI = Ideal pipeline CPI + Structural stalls + Data hazard stalls + Control stalls.

6.1.2 Data Dependencies/Hazards in a Pipelined CPU

Data dependencies between instructions occur when one instruction depends on the data output of an executing instruction. Since the data has not yet been calculated, or has been propagated back to memory, data read-after-write dependencies, or sometimes called true dependencies, can lead to pipeline stalls where one pipeline stage has to wait for another one to complete. As shown in 6.1.1, this increases the overall CPI, thus reducing the IPC.

6.1.3 CPU Performance Counters

Execution ports in modern CPU's are a part of the micro-architecture that handle specific instruction types. There are different ports that can handle different operations such as integer arithmetic operations, floating-point arithmetic operations, or memory access or a combination of these operations.

- **Core Bound Slots:** This is the percentage of CPU execution slots that contain an arithmetic operation such as an addition, or a bitwise operation like a shift instruction or an and instruction. It does not necessarily mean that these instructions caused the stall, only that these instructions were being executed while the CPU stalled.
- **Memory Bound Slots:** This is the percent of CPU execution slots that contain a memory operation such as a load or a store instruction. It does not necessarily mean that these instructions caused the stall, only that these instructions were being executed while the CPU stalled.
- **Total Stalls:** This is the total number of stalls due to either structural stalls, data dependency stalls, or control flow stalls.

6.2 Method

To study the change of iteration strategy from American Flag Sort to Ska Sort several CPU performance counters were gathered using Intel VTune and the Perf program (IBM (2024)). The second way of measuring the effects of change iteration strategy is to compare Intel VTune and Perf CPU performance counters for Instructions Per Cycle (IPC), and how memory bound the CPU backend is.

6.3 Results

The results found in table 6.1 use CPU performance counters gathered on an Intel 12th generation CPU using Intel VTune. According to the Intel VTune release notes (Intel Corporation (2024b)) for the version, the 12th generation CPUs are not yet supported. However, the VTune IPC values for American Flag Sort and Ska Sort, 1.151 and 2.410 respectively, approximately match the Perf results found in the Ska Sort blog post (Skarupke (2016)) 1.61 and 2.24. As a result, this chapter's results and analysis could potentially be incorrect.

The American Flag Sort algorithm stays in the same location in the input array, swapping this element element to where it should go with whatever element is in that position, which likely causes a data dependency between the current swap and the next swap that takes place. This is because in order for the second swap to take place, the two swapped elements have to be fully propagated into memory, which may take some time because of potential cache misses and potential TLB misses. The Ska Sort implementation reduces this data dependency by moving on to the next element in the partition regardless of the values that were swapped. As a result, Ska Sort can minimise this data dependency because most of the time the current iteration is unlikely to use the same elements from the last iteration.

From the table of CPU performance counters 6.1, American Flag Sort has a higher percentage of memory bound execution slots when the CPU stalls compared to Ska Sort. The main difference between these algorithms is the change in how elements are accessed. This means that it seems likely that the reduction of Memory Bound Slots in Ska Sort is actually from reducing memory dependency stalls.

The reduction in total stalls, likely due to a reduction of memory dependencies, in Ska Sort explains the overall increase in IPC as shown in table 6.1. This performance

increase is seen in both the original article (Skarupke (2016)) and in the VTune IPC figures. However, the Perf figures do not show this change in IPC despite the runtime improvements.

Input Size: 268,435,456 uint64_t elements		
CPU Metric	**American Flag Sort**	**Ska Sort**
Perf - IPC	1.455	1.456
VTune - IPC	1.151	2.410
VTune - Core Bound Slots	8.6%	29.6%
VTune - Memory Bound Slots	67.7%	25.4%
VTune - Total Stalls	16,824,050,472	3,048,009,144

Table 6.1: This table shows the performance counters of American Flag Sort and Ska Sort. Ska Sort is shown to have a higher IPC value, and a reduced Memory Bound Slots value.

6.4 Conclusion

In conclusion, this chapter presents CPU performance counter figures that suggest that the reason why Ska Sort is faster than American Flag Sort is due to the 5.52 times reduction of total stalls. Most of removed stalls were likely from a removal of data dependency stalls due to the change of element iteration strategy. This higher IPC allows more work to be done per cycle, thus increasing the runtime performance of the algorithm compared to American Flag Sort.

Chapter 7

Permute Stage For-Loop Unrolling

In the original implementation of Ska Sort (Skarupke (2024)), the inner for-loop in the permute stage that swaps elements from their current position to their sorted position is unrolled four times. According to the author, the goal of unrolling the for-loop is to decrease the amount of time it takes to sort all elements in a partition. This chapter presents runtime results of the permute stage unrolled for-loop, as found in the original implementation, and a comparison to a version without for-loop unrolling. The effects of for-loop unrolling are analysed.

7.1 Background

The background section in Chapter 6 contains important information, particularly about the CPU performance counters, which is also relevant to this chapter.

7.2 Method

An adjustment to the original algorithm, found in code listing 7.1, was made so instead of doing four swaps per iteration, only swap is done per iteration. To make sure that the only changes to the code are from the unrolling, the same function structure is kept. The unrolled implementation is found in code listing 7.2.

The runtime results were found by running both the default unrolled and non-unrolled versions of the permute stage for a range of uniform random uint64_t inputs of sizes from 2^8 to 2^{28}. Each input size was run twenty-five times to remove the effects of outliers on the data. The average time in nanoseconds to sort one element in a single pass is found by dividing the total runtime per input size by the number of runs and the input size.

Both of these versions were run using Intel VTune to gather CPU performance counters for IPC, Core Bound Slots, Memory Bound Slot, and Total Stalls.

```
template <typename It, typename Func>
inline void unroll_loop_four_times(It begin, size_t iteration_count, Func &&
    to_call)
{
    size_t loop_count = iteration_count / 4;
    size_t remainder_count = iteration_count - loop_count * 4;
    for (; loop_count > 0; --loop_count) {
        to_call(begin);
        ++begin;
        to_call(begin);
        ++begin;
        to_call(begin);
        ++begin;
        to_call(begin);
        ++begin;
    }
    switch (remainder_count) {
    case 3:
        to_call(begin);
        ++begin;
    case 2:
        to_call(begin);
        ++begin;
    case 1:
        to_call(begin);
    }
}
```

Listing 7.1: This is the unrolled for-loop implementation. It takes in a function, 'to_call', that takes an iterator and swaps the value found in the iterator to its sorted partitions with the element that is already in that position. This version does four loops of this process per iteration, therefore the number of iterations is divided by four. There may be some remaining elements that are not sorted in the unrolled loop, so are handled separately with the switch statement.

```
template <typename It, typename Func>
inline void for_loop(It begin, size_t iteration_count, Func &&to_call)
{
```

```
4       for (size_t i = 0; i < iteration_count; i++) {
5           to_call(begin);
6           ++begin;
7       }
8   }
```

Listing 7.2: This is the non-unrolled for-loop implementation. It takes in a function, 'to_call', that takes an iterator and swaps the value found in the iterator to its sorted partitions with the element that is already in that position. This version only does a single swap per iteration and moves on to the next element in the partition. The iteration_count variable is the number of elements in the partition.

7.3 Results

It is important to note that the same potential data issues about the Intel VTune CPU performance counter data from Chapter 6, section 6.3 applies to this chapter's results as well.

From the runtime results comparing the default unrolled against the non-unrolled version, as shown in figure 7.1, there is little to no difference between the runtimes across most input sizes. From input sizes from 262,144 to 1,048,576 the unrolled method is slightly faster compared to the non-unrolled version. These results suggest that there is no difference between the versions. The CPU performance counters, found in table 7.1, suggest a similar conclusion as well. Both the IPC and total stalls are very close to each other 2.66 and 2.76 for unrolled and non-unrolled respectively. There is a larger difference between the percent of core and memory bounds slots, however this value is more variable because it depends on what instructions are being executed when VTune samples the CPU counters.

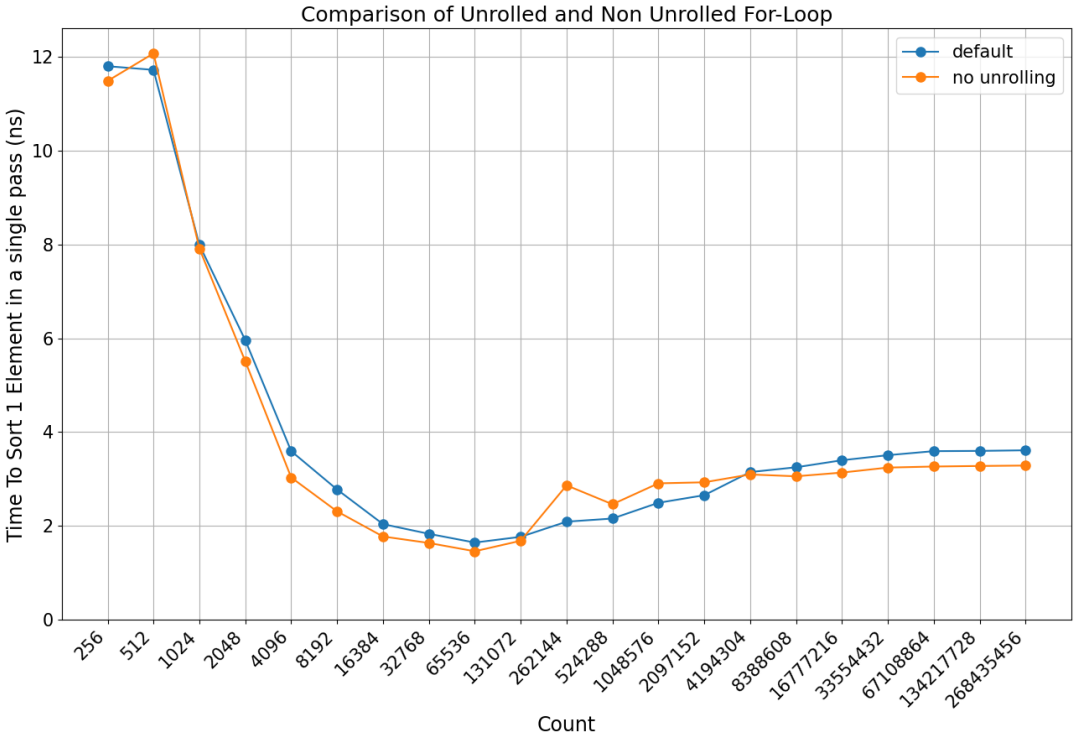

Figure 7.1: This is a graph of the average of the nanoconds taken to sort a single element of uniform random inputs for a range of input sizes. There is little to no difference between having the swapping for-loop unrolled or not unrolled.

Input Size: 268,435,456 uint64_t elements		
CPU Metric	**Unrolling**	**No Unrolling**
VTune - IPC	2.66	2.76
VTune - Core Bound Slots	17.2%	10.4%
VTune - Memory Bound Slots	34.1%	39.7%
VTune - Total Stalls	2,157,840,000	2,149,280,000

Table 7.1: This table shows a comparison of the IPC, percent of core bound execution slots, percent of memory bound execution slots, and the total number of stalls for each algorithm. Each algorithm was run on 2^{28} input uint64_t elements. There is only a slight difference between the two version's CPU performance counters. The percent of core and memory bounds have a greater difference, but this is likely down to randomness in how often the CPU takes performance measurements.

7.4 Conclusion

In conclusion, this chapter has found that the for-loop unrolling found in the permute stage has little to no impact on runtime performance. Additionally, no benefit is found in the CPU performance counter metrics because both versions have approximately the same IPC. Unlike in Chapter 4, where unrolling the counting for-loop gives faster runtimes, this version does not. Unrolling the for-loop in the counting stage provides faster runtimes because a dependency between iteration loops is being broken by unrolling the loop and having additional count arrays. However, this is not the case when unrolling the swapping loop because no dependencies between loop iterations are broken because all elements are being swapped to and from the same array.

Chapter 8

Key Size Effects

The object of this chapter is to evaluate the effect of key size on the performance of Ska Sort. From Chapter 4, it is known that the permute stage of the algorithm takes the majority of the runtime. It is therefore important to see if the runtime can be improved by changing the size of the key. In addition to this, from Chapter 3, it is clear that the size of the key determines the number of partitions, therefore the number of iterations needed to sort all partitions in the permute stage. The question this chapter answers is: What key size results in the best runtime performance and why?

8.1 Background

This chapter uses a model of the Translation Lookaside Buffer (TLB) found in the following paper (Rahman and Raman (2002)). Modern CPU's use virtual memory which is partitioned into 4Kb chunks called pages. Each virtual address has to be translated to physical memory addresses before it can be used. To avoid constantly translating virtual pages, a TLB keeps stored recently translated pages for later lookup. The number of entries in the TLB is small, typically only containing 64 to 128 entries. The paper suggests that in order to improve the runtime of the radix sort algorithm, the working set of the algorithm must be reduced to minimise TLB misses. The working set of an algorithm is how many memory pages it accesses at any one time. Applying this to Ska Sort, it means that to improve the runtime of the algorithm by minimising TLB misses, fewer pages need to be accessed. When the input is sufficiently large, each partition's insertion point will be on a different page, therefore needing its own TLB entry. As a result, decreasing the key size, and therefore the number of partition insertion points, should help performance for large input sizes.

8.2 Method

To test the relative performance of Ska Sort with different key sizes, the average nanoseconds taken to sort a single element in one single pass was gathered for different key sizes. This measurement is taken across a range of input sizes using a uniform random and geometric random input. These runtime values are then normalised by size of the key. This is done to take into account the fact that in a full algorithm that does all the required passes to fully sort the array, smaller sized keys will have a larger number of required passes compared to larger sized keys. For example, a that is four bits will require 8 passes to sort a uint32_t, while a sixteen bit key will only require two passes. The key sizes ranged from four to sixteen bits.

Two different methods of extracting the key were used. The first method extracts exactly a byte or a short using the storage types, uint8_t and uint16_t respectively. The second method, which uses the more general bit range approach uses type uint16_t type to store the key regardless of the size of the key. The effects of these approaches will be explained in the Results section.

8.3 Results

The results for uniform random inputs, found in figure 8.1, indicate that decreasing the key size does improve runtime performance for large input sizes. When the number of elements is greater than 2^{22}, using a six bit key leads to greater performance compared to all other key sizes. This fits with the TLB model presented in (Rahman and Raman (2002)). Using a key with six bits results in $2^6 = 64$ partition insertion points. For large input sizes each of these insertion points will be on different memory pages, thus requiring a separate TLB entry for each. In general, it is guaranteed that each partition insertion point will be on a separate page of memory when the total partition size in bytes is greater than the page size.

At the 2^{22} input size, it is faster to use a seven bit key instead of a six or eight bit key. It is important to note it is always faster to extract a key that is exactly a 'Byte' compared to a '8 bit' key. Although the C++ code for extracting the key is the same, the resulting optimised assembly output is not. This is because the compiler knows at compile-time, for the 'Byte' implementation, that a whole byte is being extracted and not an arbitrary number of bits. The compiler makes the following optimisation to the code:

```
uint8_t byte = (input[i] >> (byte_index * 8)) & 0xFF;
```

Listing 8.1: This code loads a uint32_t input element, shifts out bits of information that is not needed in the key. The bitwise-and zeroes all bits that are not in the 'byte' key.

```
MOV     eax, DWORD PTR [rdx]
SHR     eax, cl
MOVZX   eax, al
```

Listing 8.2: The above C++ code results in this assembly code when compiled with the G++ compiler with the -O2 optimisation level. The first instruction loads a uint32_t value from rdx into eax. The byte key is shifted into the lowest byte of eax. However, no AND instruction is generated, instead the MOVZX instruction loads a value from a register and zeroes out the remaining bits. In this case, it loads the lowest byte, referenced using 'al', from the eax register and zeroes out the remaining bits. The compiler found that a bitwise AND instruction using this byte value and 0xFF always results in the original byte value. It is then safe for the compiler to remove this instruction because the output of the operation does not change.

```
using KeyType = uint16_t;
KeyType key = (input[i] >> range.end) & key_mask;
```

Listing 8.3: This is the code segment for extracting the key from an input element. It loads a uint32_t input element, shifts out bits of information that are not needed in the key. The bitwise-and zeroes all bits that are not in the 'key' value.

```
MOV     eax, DWORD PTR [rdx]
SHR     eax, cl
AND     eax, esi
MOVZX   eax, ax
```

Listing 8.4: The above C++ code results in this assembly code when compiled with the G++ compiler with the -O2 optimisation level. The first instruction loads a uint32_t value from rdx into eax. The key is shifted into the lowest short of eax. An AND instruction is required to remove the potentially unnecessary bits of information from the short. For example, if the key is twelve bits in size, the AND instruction must remove the remaining four bits from the sixteen bit key. The MOVZX instruction loads the short, referenced using 'ax', and zeroes out all other bits.

When testing the time for extracting the key in the counting stage, the 8 bit implementation was approximately 1.076 times slower than the byte implementation. This explains why the byte implementation is always used over the 8 bit approach.

The results for geometric random inputs, found in table 8.2, show different results. For geometric inputs, the fastest method of sorting is by having a sixteen bit key, which seems to go against the TLB model already presented. However, the geometric random inputs only range in values from approximately 0 to 5. This means that there are only going to be six partition insertion points regardless of key size. All implementations are doing approximately the same amount of work in the same amount of time, however larger key sizes are being discounted to a greater extent compared to smaller key sizes. As a result, the fastest times are all large key sizes. To see the fastest key size to sort a geometric random input, the run times can be normalised by the number of partitions generated as shown in table 8.3. The table shows that the fastest key to sort the input ranges is to use a byte key. This is likely due to the byte key resulting in more optimised code generation.

Uniform Random Input - uint64_t										
Size	**Byte**	**Short**	**11 bit**	**10 bit**	**9 bit**	**8 bit**	**7 bit**	**6 bit**	**5 bit**	**4 bit**
2^{12}	0.392	5.674	0.884	0.720	0.521	0.431	0.359	0.354	0.386	0.448
2^{13}	0.321	3.738	0.641	0.486	0.391	0.342	0.326	0.338	0.411	0.579
2^{14}	0.288	2.661	0.613	0.440	0.378	0.370	0.424	0.451	0.480	0.679
2^{15}	0.240	1.549	0.354	0.296	0.302	0.280	0.310	0.357	0.460	0.670
2^{16}	0.221	1.032	0.258	0.236	0.235	0.237	0.281	0.341	0.460	0.746
2^{17}	0.219	0.734	0.218	0.210	0.212	0.230	0.263	0.331	0.460	0.785
2^{18}	0.245	0.646	0.302	0.295	0.323	0.332	0.359	0.440	0.599	0.921
2^{19}	0.258	0.558	0.322	0.354	0.354	0.345	0.358	0.449	0.556	0.812
2^{20}	0.325	0.543	0.386	0.414	0.429	0.393	0.413	0.451	0.538	0.811
2^{21}	0.414	0.542	0.448	0.480	0.526	0.459	0.459	0.454	0.554	0.853
2^{22}	0.469	0.528	0.483	0.533	0.587	0.522	0.460	0.471	0.544	0.853
2^{23}	0.497	0.560	0.547	0.574	0.630	0.556	0.506	0.467	0.558	0.881
2^{24}	0.566	0.694	0.620	0.643	0.689	0.616	0.547	0.528	0.636	0.989
2^{25}	0.589	0.738	0.652	0.663	0.706	0.635	0.579	0.544	0.651	0.988
2^{26}	0.601	0.760	0.653	0.667	0.722	0.651	0.587	0.555	0.671	1.013
2^{27}	0.565	0.794	0.629	0.637	0.692	0.623	0.570	0.525	0.647	0.998
2^{28}	0.564	0.971	0.621	0.630	0.689	0.616	0.558	0.512	0.635	1.001
2^{29}	0.530	1.015	0.619	0.609	0.661	0.586	0.533	0.473	0.594	0.963

Table 8.1: The table shows the number of nanoseconds per bit it takes to sort a single uniform random uint64_t element given a key of a specific size. The fastest times are marked with a green background. For inputs larger than 2^{21}, reducing the size of the key, and therefore the number of partition insertions points, increases the runtime performance.

Geometric Random Input - uint64_t									
Size	**Byte**	**Short**	**12 bit**	**11 bit**	**10 bit**	**9 bit**	**8 bit**	**7 bit**	**6 bit**
2^{12}	0.531	5.569	0.977	0.741	0.533	0.605	0.678	0.598	0.495
2^{13}	0.455	2.010	0.562	0.393	0.377	0.364	0.483	0.400	0.558
2^{14}	0.388	1.116	0.409	0.353	0.335	0.342	0.646	0.417	0.532
2^{15}	0.397	0.780	0.340	0.361	0.345	0.353	0.455	0.390	0.466
2^{16}	0.402	0.583	0.335	0.338	0.366	0.362	0.458	0.519	0.529
2^{17}	0.429	0.414	0.342	0.359	0.347	0.392	0.423	0.476	0.692
2^{18}	0.424	0.354	0.367	0.454	0.376	0.479	0.534	0.554	0.666
2^{19}	0.456	0.275	0.385	0.426	0.471	0.472	0.566	0.657	0.753
2^{20}	0.458	0.301	0.399	0.421	0.486	0.518	0.587	0.668	0.673
2^{21}	0.441	0.280	0.401	0.444	0.472	0.495	0.604	0.653	0.755
2^{22}	0.463	0.278	0.385	0.433	0.459	0.522	0.591	0.664	0.757
2^{23}	0.432	0.279	0.386	0.427	0.487	0.522	0.544	0.621	0.775
2^{24}	0.446	0.283	0.392	0.432	0.460	0.517	0.567	0.655	0.728
2^{25}	0.450	0.283	0.397	0.423	0.469	0.512	0.572	0.646	0.733
2^{26}	0.437	0.289	0.392	0.425	0.466	0.516	0.577	0.637	0.732
2^{27}	0.439	0.279	0.397	0.429	0.465	0.509	0.571	0.641	0.732
2^{28}	0.445	0.281	0.394	0.420	0.464	0.512	0.570	0.637	0.729
2^{29}	0.443	0.282	0.395	0.428	0.468	0.513	0.572	0.645	0.734

Table 8.2: The table shows the number of nanoseconds per bit it takes to sort a single geometric random uint64_t element given a key of a specific size. The fastest times are marked with a green background. This table shows the opposite results of the uniform random input 8.1. This is because for a geometric random distribution, most elements will be in the range of 0 to 5. As a result, there will only be approximately six partition insertion points. This means that all key sizes are doing the same amount of work, while the larger key sizes are being normalised to a greater extent, leading to comparatively lower times.

Geometric Random Input - uint64_t - Partition Normalised								
Size	**Byte**	**Short**	**9 bit**	**8 bit**	**7 bit**	**6 bit**	**5 bit**	**4 bit**
2^{12}	0.708	14.851	1.017	0.797	0.578	0.903	0.529	0.529
2^{13}	0.607	5.360	0.725	0.533	0.651	0.541	0.573	0.525
2^{14}	0.517	2.976	0.969	0.556	0.621	0.596	0.541	0.529
2^{15}	0.529	2.080	0.682	0.520	0.544	0.601	0.567	0.557
2^{16}	0.536	1.555	0.687	0.692	0.617	0.585	0.576	0.673
2^{17}	0.572	1.104	0.634	0.635	0.807	0.761	0.647	0.620
2^{18}	0.565	0.944	0.801	0.739	0.777	0.792	0.778	0.803
2^{19}	0.608	0.733	0.849	0.876	0.878	0.867	0.784	0.854
2^{20}	0.611	0.803	0.880	0.891	0.785	0.875	0.877	0.863
2^{21}	0.588	0.747	0.906	0.871	0.881	0.884	0.864	0.857
2^{22}	0.617	0.741	0.887	0.885	0.883	0.883	0.833	0.865
2^{23}	0.576	0.744	0.816	0.828	0.904	0.885	0.858	0.849
2^{24}	0.595	0.755	0.850	0.873	0.849	0.872	0.855	0.837
2^{25}	0.600	0.755	0.858	0.861	0.855	0.843	0.853	0.880
2^{26}	0.583	0.771	0.866	0.849	0.854	0.848	0.849	0.857
2^{27}	0.585	0.744	0.857	0.855	0.854	0.859	0.854	0.855
2^{28}	0.593	0.749	0.855	0.849	0.850	0.862	0.858	0.863
2^{29}	0.591	0.752	0.858	0.860	0.856	0.857	0.858	0.858

Table 8.3: The table shows the number of nanoseconds per partition it takes to sort a single geometric random uint64_t element given a key of a specific size. It is clear from the table that a key that is a byte is the fastest per partition than any other key size.

8.4 Conclusion

In conclusion, this chapter found that having a six bit key for inputs greater than 2^{22} elements results in faster runtime performance compared to any other key size on uniform random inputs 8.1. For uniform random inputs smaller than 2^{22} elements or geometric random inputs 8.3, the more optimised byte key size is preferred for runtime performance. The effect of the key size can be explain by the additional cost of TLB misses per element load and store. Additionally, the TLB model does seem to be useful for predicting the size of key to use for Ska Sort and radix sort.

Chapter 9

Fallback Threshold

Radix sorting algorithms such as American Flag Sort and Ska Sort perform relatively poorly on small input sizes in comparison to comparative sorting algorithms such as std::sort (Knuth (1998)). This is because just under half of the runtime of these radix sorting algorithms is counting the number of each digit in the input as found in Chapter 4. This time spent counting, is time that could be spent swapping elements to their sorted position. It is therefore common to see fallback algorithms found in radix sorting algorithms. For example, American Flag Sort fallback to insertion sort below a certain threshold (McIlroy et al. (1993)).

In addition to this, using a comparative sorting algorithm sorts elements into their final sorted positions. This means that no further Ska Sort passes are required on the input ranges that were sorted using std::sort. With Ska Sort, sorting uniform random 32-bit numbers with an 8-bit key takes four passes. The first pass generates 256 partitions, each of which will be sorted in a second pass. In the second pass, each of these 256 partitions will create 256 more partitions. By the time the algorithm is on its last pass, 256^4 partitions will have to be sorted, most of which will contain a small number of elements. By using a comparative sorting algorithm for small partitions, no additional partitions will be created within that input range, therefore decreasing future work.

The objective of this chapter is to answer the following questions: At what input size is it quicker to fully sort an array with std::sort compared to a partition using Ska Sort? Additionally, American Flag Sort is used as a fallback to Ska Sort in the original implementation. Is it faster to use American Flag Sort to sort a partition compared to Ska Sort, and if so at what input sizes?

9.1 Method

To answer these two questions, two different tests were run. The first of which is to find at what input size is std::sort quicker at sorting the entire input compared to Ska Sort sorting a single pass. To do this the average time in nanoseconds to sort a single element using std::sort and a single element using one pass of Ska Sort were compared for a range of input sizes. By creating these runtimes, the crossover point can be determined and thus the std::sort fallback threshold can be set.

To answer the second question, I ran a similar experiment to compare the average time taken to sort a single element of one Ska Sort pass to a one American Flag Sort pass. Using these runtimes, the crossover point can be used as the fallback threshold for one pass of American Flag Sort.

9.2 Results

After generating the runtimes for the std::sort and one pass Ska Sort, as found in figure 9.2, it was found that the std::sort fallback threshold should be set at 88 or 96 elements. However, it is likely that a higher threshold would result in better runtime performance. This is because when one pass of Ska Sort is done, more partitions are created which have to be sorted again. This does not happen with std::sort, where all sorted elements are in their final position and do not have to be swapped around in another pass. As a result, it is likely that a higher fallback threshold would improve performance since this method of getting the fallback threshold does not account for std::sort's removal of future passes. The original Ska Sort (Skarupke (2024)) implementation uses 128 input size as a fallback threshold for std::sort which is 33.33% higher fallback threshold.

The above std::sort fallback threshold reduces the total number of partitions sorted from 277,065,198 to 16,843,006, a 16.45 times reduction, on 2^{28} uint64_t uniform random elements. In addition to this, the runtime is reduced by a factor of 3.59 times from 17.52 seconds with no fallback algorithms to 4.88 seconds using the 96 element std::sort fallback threshold.

After generating the runtimes for one pass of American Flag Sort and one pass of Ska Sort, as shown in figure 9.2, it was found that the American Flag Sort fallback threshold should be set at 1500 or 1600 elements. Since both the counting stage and offset generation stage of these algorithms are both the same, this means that the sorting stage of the

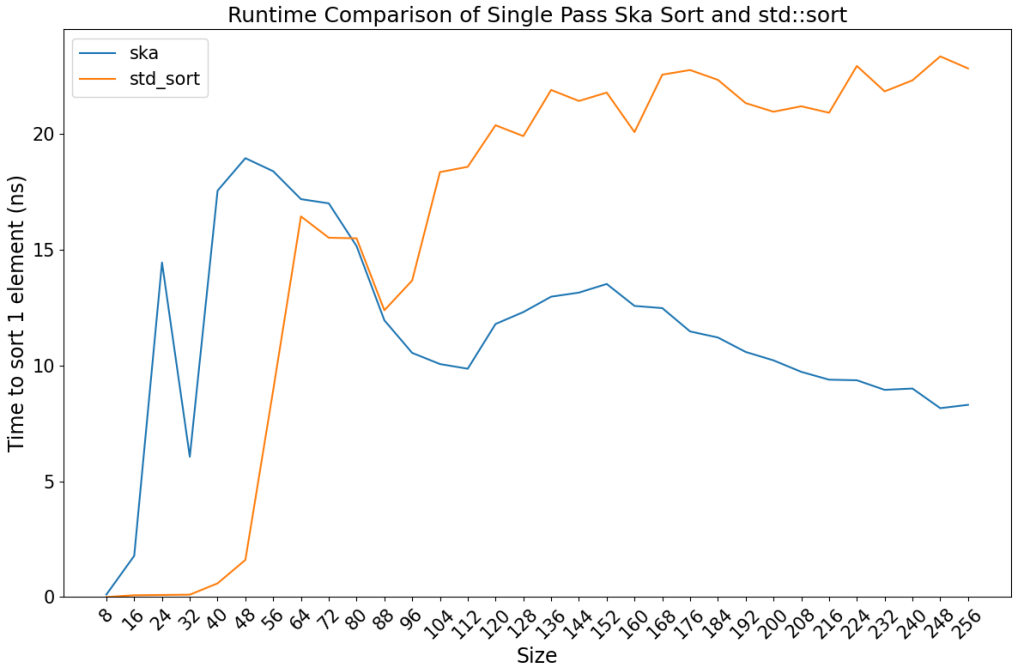

Figure 9.1: This graph shows the average time to sort one element of input across a range of input sizes for std::sort and one pass of Ska Sort. The runtimes of both of these algorithms cross at approximately 88 to 96 elements, meaning std::sort is faster than a single pass of Ska Sort for inputs smaller than 96.

American Flag Sort for values below this threshold is faster compared to Ska Sort.

Using these fallback thresholds, the runtime effects of the custom and default fallback thresholds can be compared as shown in figure 9.3. The custom threshold shows a light improvement over the current fallback thresholds.

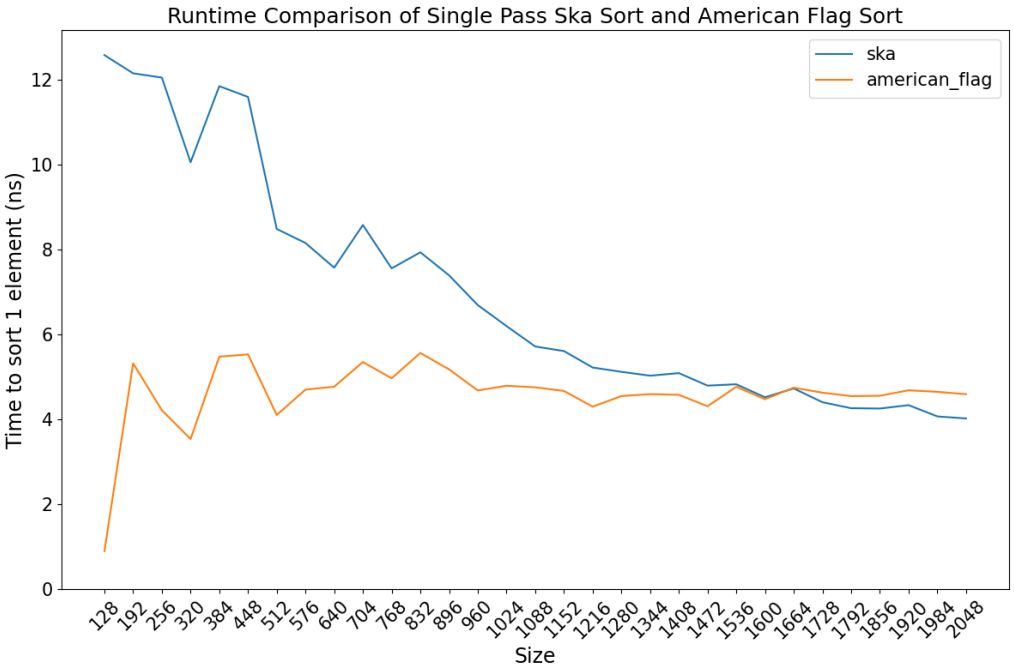

Figure 9.2: This graph shows the average time in nanoseconds to sort one element of input across a range of input sizes for one pass of American Flag Sort and one pass of Ska Sort. The runtimes of both of these algorithms cross at approximately 1500 to 1600 elements.

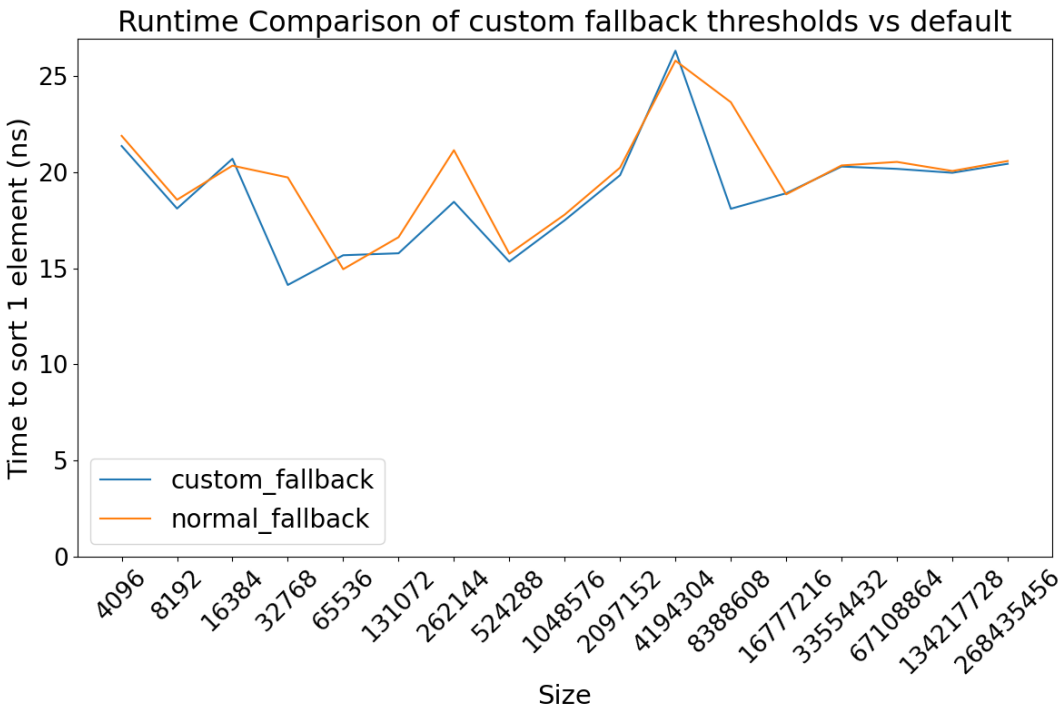

Figure 9.3: This graph shows the average time in nanoseconds to sort one element of input across a range of uniform random input sizes for Ska Sort with a custom and default fallback threshold. The custom thresholds are 96 elements for the std::sort fallback, and 1550 elements for the American Flag fallback. The custom fallback threshold is marginally faster than the default implementation across most input sizes.

9.3 Conclusion

In conclusion, this chapter has found that, at approximately 100 element sized inputs, it is beneficial to fallback to std::sort for two reasons. The first reason is because std::sort is faster at sorting the input array compared to doing one Ska Sort pass. And secondly, it reduced the total number of partitions to sort by a factor of sixteen leading to an increase in runtime performance. However, there is only a small difference between the runtimes for the custom and default fallback thresholds.

9.3.1 Further Work

Due to time restraints, the reason why American Flag Sort is faster than Ska Sort for inputs sizes less than approximately 1500 to 1600 has not been investigated. Additionally, there could be another method of determining what fallback thresholds would give the fastest runtime, that is not mentioned in this paper.

Chapter 10

Improved Integer Sort

The Boost (Boost (2024)) library contains a number of highly optimised sorting algorithms. This library contains a specialised sorting algorithm for integers called integer spreadsort which is the go-to algorithm for sorting integers because of its runtime performance. Integer spreadsort's permute swapping algorithm is fundamentally very similar to the American Flag Sort strategy, where elements are swapped to their correct partition and only move on to the next element when a key is moved to the current partition. This paper creates and studies an updated algorithm that swaps integer spreadsort's current permute iteration strategy with the strategy found in Ska Sort to see if similar performance gains can be found.

10.1 Background

Integer spreadsort is similar to the American Flag Sort algorithm in that the mains stages are similar: Generate the histogram of the digits in the input array, creating the offsets and final offsets using a prefix sum, swapping elements into the correct position based on the offsets, and finally recursively sorting the partitions using the next digit or using a fallback sorting algorithm. The main difference between these algorithms is that instead of having a partition for each unique key value as in American Flag Sort, integer spreadsort puts a range of values into the one partition. This is done using integer division. For example, if all key values are divided by 100, then values from 0 to 99 will result in a partition index of 0, values from 100 to 199 will result in a partition index of 1 and so on. Using this method a range of values can be put into the one partition.

The background section in Chapter 6 contains important information, particularly about the CPU performance counters, which is also relevant to this chapter.

10.2 Implementation

The code listing for the full implementation of this algorithm can be found in the appendix entry 1.

10.3 Method

To test if this Ska's sorting iteration method provides a runtime improvement, the two algorithms were tested using three datasets: random uniform values, random geometric values with a distribution parameter of 0.8, and sorted values. The algorithms were tested using the following types: uint8_t, uint16_t, uint32_t, and uint64_t, using a range of input sizes from 2^{12} to 2^{29}. In addition to this, single pass versions of the algorithms were also tested to show the per pass performance. The algorithms were run ten times and timed using the C++ chrono library's high resolution clock.

10.4 Results

It is important to note that the same potential data issues about the Intel VTune CPU performance counter data from Chapter 6, section 6.3 applies to this chapter's results as well.

This section presents the results of the testing mentioned in the Method section. Both the single pass 10.1 and the full sort 10.2 on uniform random values are graphed showing the time taken to sort one element on the y-axis, and across a range sizes on the log-scaled x-axis. These results show that the change of iteration method to the Ska Sort method leads to a significant performance increase.

The runtime results for geometric random inputs, as shown in figure 10.3 and 10.4, show that the Ska Sort permute stage partition iteration method performs better than the original integer spreadsort implementation even when the elements are primarily in a single partition. The already sorted results, as shown in figure 10.5, show no performance difference. This is because there is a check to see if the input array is already sorted before swapping the elements.

Table 10.1 shows a list of performance counters. The number of instructions the CPU can execute per cycle for the original implementation is significantly 2.85 times lower than the implementation with the Ska Sort permute iteration method as shown by the IPC

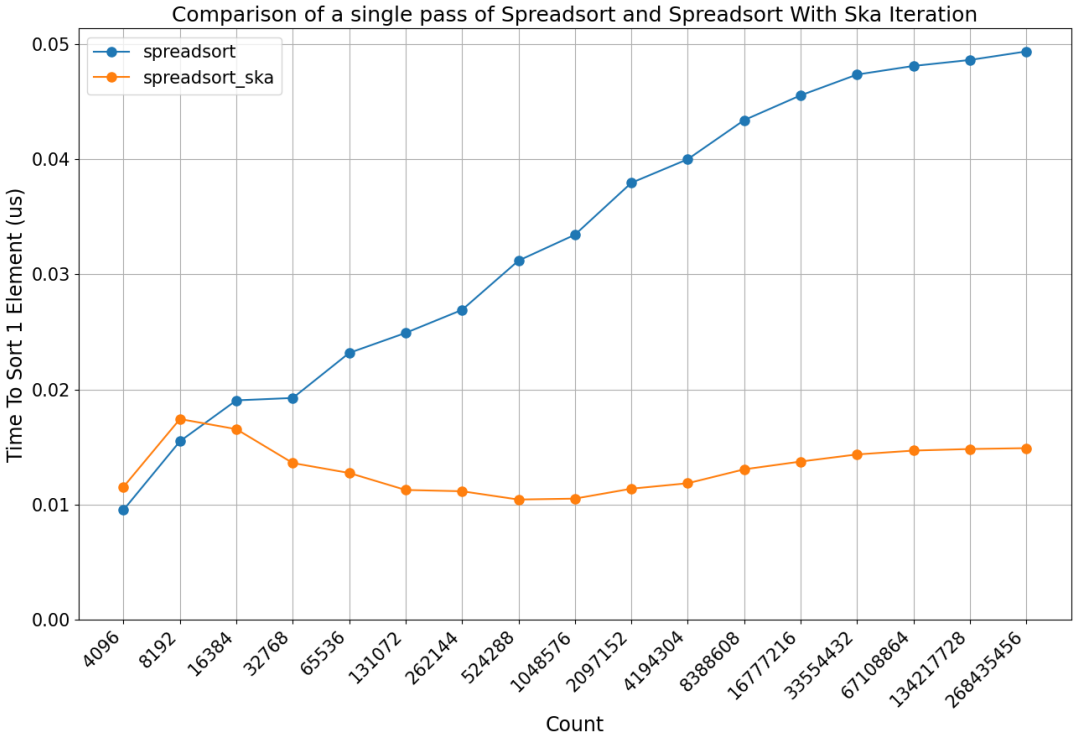

Figure 10.1: The average time to sort one element in a single pass on uniform random uint64_t inputs against various input sizes is graphed. On inputs greater than 10,000, the average time to sort one element when doing a single pass is faster with the Ska Sort iteration method compared to the original implementation. On inputs with multiple million elements, the custom spreadsort is approximately three times faster.

values. According to the Memory Bound Slots metric, the original implementation also has a significantly higher percentage of execution units stalled while there were memory instructions being executed compared to the implementation with the Ska iteration method. This suggests that the original implementation stalls more because it is waiting for swapped elements to fully propagate into their memory locations before reading the current value again. This issue is reduced with the Ska iteration method because in most cases the algorithm moves on to read a different element. Therefore, the current read instruction does not have to wait for the previous store instruction value to be propagated into memory, because they are loading and storing from different locations, thus reducing the number of data dependency stalls. The original implementation reports 7.15 times more CPU stalls compared to the implementation with the Ska iteration method. This suggests that stalls from data dependencies can be avoided using the Ska iteration method, thus increasing runtime performance.

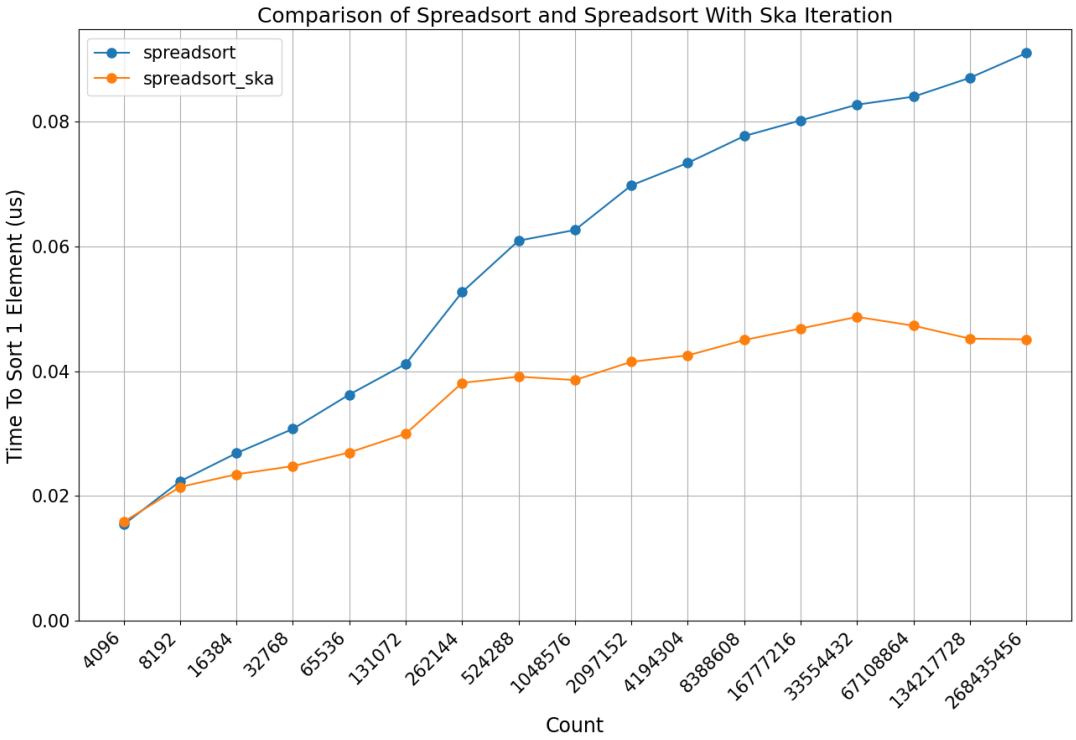

Figure 10.2: The average time to sort one element on uniform random uint64_t inputs against various input sizes is graphed. The spreadsort implementation that uses the Ska Sort iteration method is faster for all the graphed input sizes.

Input Size: 268,435,456 uint64_t elements		
CPU Metric	**Spreadsort**	**Spreadsort + Ska Iteration**
VTune - IPC	0.92	2.62
VTune - Core Bound Slots	11.3%	23.9%
VTune - Memory Bound Slots	74.8%	38.3%
VTune - Total Stalls	29,328,087,984	4,104,012,312

Table 10.1: This table shows a comparison of the IPC, percent of core bound execution slots, percent of memory bound execution slots, and the total number of stalls for each algorithm. Each algorithm was run on 2^{28} input uint64_t elements. The spreadsort shows worse performance in all measurements except for percent of core bound execution slots.

Figure 10.3: The average time to sort one element in a single pass on geometric random uint64_t inputs against various input sizes is graphed. The average time to sort one uint64_t element when doing a single pass is consistently faster with the Ska sorting iteration method.

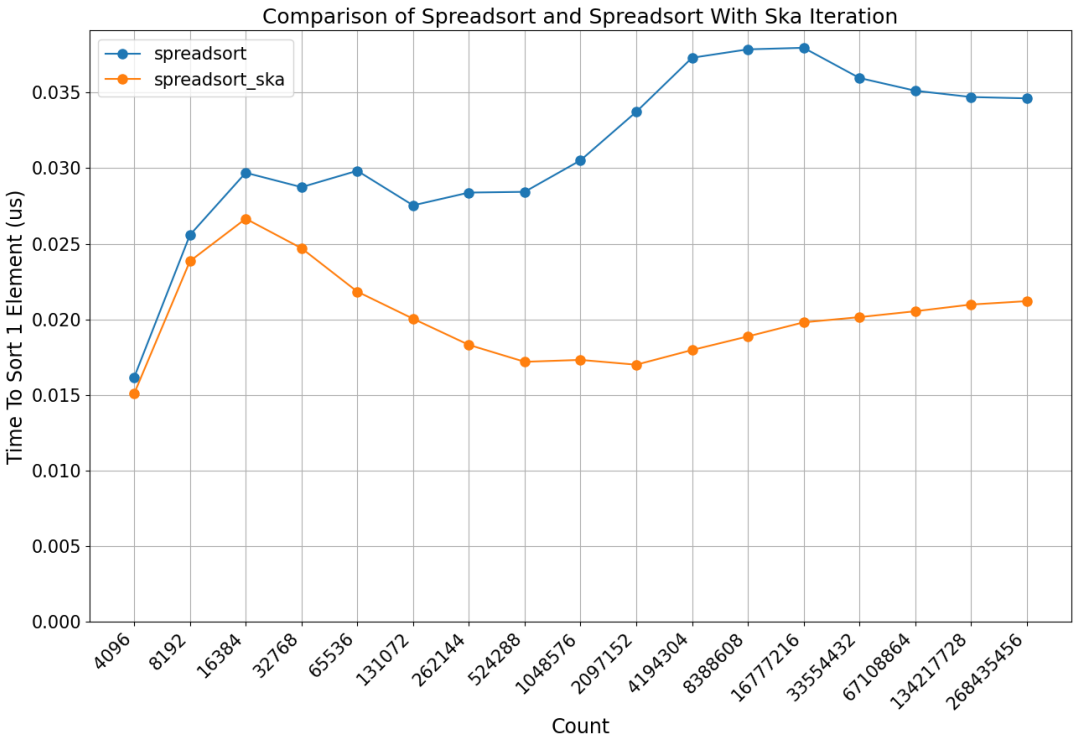

Figure 10.4: The average time to sort one element on geometric random uint64_t inputs against various input sizes is graphed. The average time to sort one uint64_t element when doing a single pass is consistently faster with the Ska sorting iteration method.

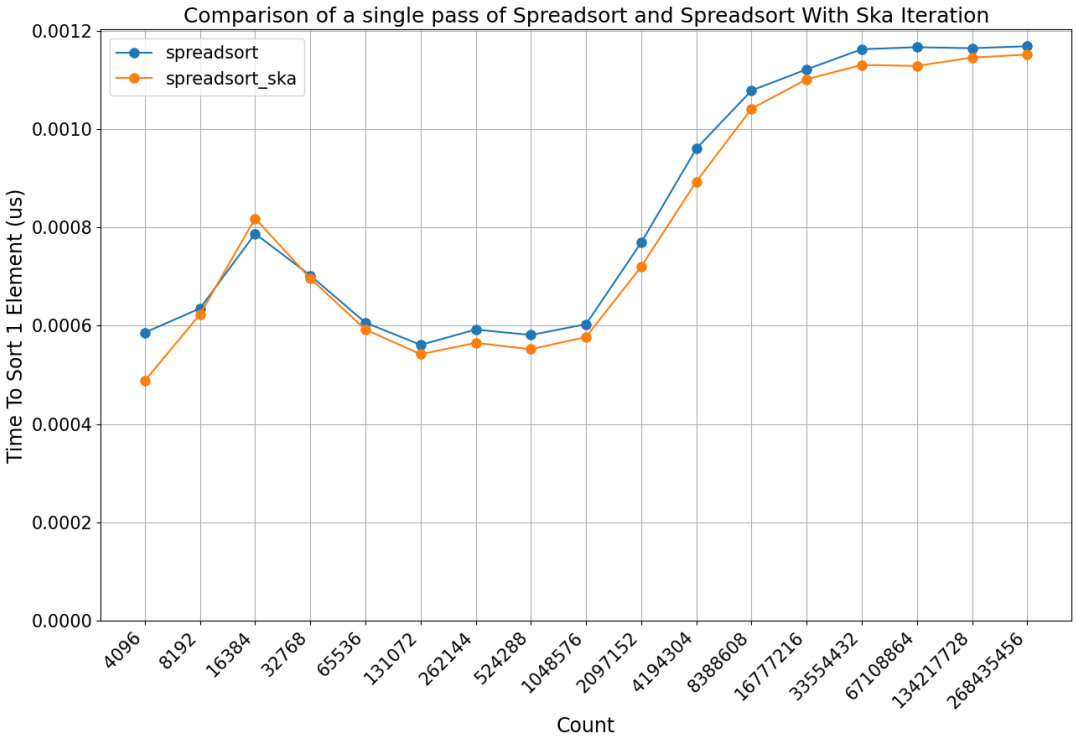

Figure 10.5: The time taken to sort already sorted inputs for both algorithms are approximately equal. This is because the spreadsort algorithms checks to see if the input is already sorted before it starts the sorting process. Therefore, no runtime improvements can be realised with already sorted inputs.

10.5 Conclusion

It was found that, on a range of input sizes and distributions, replacing integer spreadsort's permute stage iteration method with Ska Sort method increases the runtime performance by up to three times. This performance improvement can be seen in both the single pass and full passes data and figures. Based on the CPU performance counters, it is likely that the Ska iteration method is faster because it reduces the number of stalls from data dependencies.

10.5.1 Further Work

From the results, it seems like the change in permute stage iteration is not an American Flag Sort or Integer Spreadsort specific improvement. Other radix sorting algorithms could potentially be improved by changing the iteration strategy to Ska Sort's.

Bibliography

Allan, V. H., Jones, R. B., Lee, R. M., and Allan, S. J. (1995). Software pipelining. *ACM Comput. Surv.*, 27(3):367–432.

Amato, N., Iyer, R., Sundaresan, S., and Wu, Y. (1998). A comparison of parallel sorting algorithms on different architectures. Technical report, Texas A & M University, USA.

Boost (2024). Boost.org: https://www.boost.org/.

cppreference (2024a). std::geometric_distribution - cppreference.com.

cppreference (2024b). std::uniform_int_distribution - cppreference.com.

Dr Richard Gibbens (2016). University of cambridge teaching resources: https://www.cl.cam.ac.uk/teaching/1516/compsysmod/expfn.pdf.

Hennessy, J. L. and Patterson, D. A. (2011). *Computer Architecture, Fifth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th edition.

IBM, I. (2024). Getting started with the perf command.

Intel Corporation (2024a). Intel intrinsics guide: https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html.

Intel Corporation (2024b). Intel vtune release notes: https://www.intel.com/content/www/us/en/developer/articles/release-notes/vtune-profiler-release-notes.html.

Knuth, D. E. (1998). *The art of computer programming, volume 3: (2nd ed.) sorting and searching*. Addison Wesley Longman Publishing Co., Inc., USA.

Maus, A. (2015). A full parallel quicksort algorithm for multicore processors. In *A full parallel Quicksort algorithm for multicore processors*.

McIlroy, P. M., Bostic, K., and McIlroy, D. (1993). Engineering radix sort. *Comput. Syst.*, 6:5–27.

Obeya, O., Kahssay, E., Fan, E., and Shun, J. (2019). Theoretically-efficient and practical parallel in-place radix sorting. In *The 31st ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '19, page 213–224, New York, NY, USA. Association for Computing Machinery.

Polychroniou, O. and Ross, K. A. (2014). A comprehensive study of main-memory partitioning and its application to large-scale comparison- and radix-sort. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, page 755–766, New York, NY, USA. Association for Computing Machinery.

Rahman, N. and Raman, R. (2002). Adapting radix sort to the memory hierarchy. *ACM J. Exp. Algorithmics*, 6:7–es.

Skarupke, M. (2016). Ska sort article: https://probablydance.com/2016/12/27/i-wrote-a-faster-sorting-algorithm.

Skarupke, M. (2024). Ska sort source code: https://github.com/skarupke/ska_sort.

Appendix

```
template <typename T>
static void my_spreadsort_ex(T *input, size_t input_start, size_t input_end,
    size_t *offsets, std::vector<size_t> &final_offsets_cache, size_t
    cache_offset)
{
  // Find smallest and largest values
  T *max, *min;
  bool already_sorted = is_sorted_or_find_extremes<T *>(input + input_start,
    input + input_end, max, min);
  if (already_sorted) {
    return;
  }

  // Calcualte number of bins (partitions)
  unsigned int log_divisor = get_log_divisor<int_log_mean_bin_size>(input_end -
    input_start, rough_log_2_size(*max - *min));
  size_t div_min = (*min) >> log_divisor;
  size_t div_max = (*max) >> log_divisor;
  unsigned int bin_count = (unsigned int)(div_max - div_min) + 1;

  // Allocate partition final offsets
  size_t cache_end = cache_offset + bin_count;
  if (cache_end > final_offsets_cache.size()) {
    final_offsets_cache.resize(cache_end);
  }
  size_t *final_offsets = final_offsets_cache.data() + cache_offset;

  // Count number of elements in each partition
  memset(offsets, 0, bin_count * sizeof(*offsets));
  for (size_t i = input_start; i < input_end; i++) {
    ++offsets[(input[i] >> log_divisor) - div_min];
  }
```

```
29
30  // Calculate offsets and final offsets using prefix sum
31  size_t total = 0;
32  for (int i = 0; i < bin_count; ++i) {
33    size_t count = offsets[i];
34    offsets[i] = total;
35    total += count;
36    final_offsets[i] = total;
37  }
38
39  // Sort elements into their partitions
40  while (true) {
41    int sorted_bin_count = 0;
42    for (unsigned int i = 0; i < bin_count; ++i) {
43      size_t local_bin_start = offsets[i];
44      size_t local_bin_end = final_offsets[i];
45      if (local_bin_start == local_bin_end) {
46        ++sorted_bin_count;
47        continue;
48      }
49
50      for (size_t p = local_bin_start; p < local_bin_end; ++p) {
51        T current = input[input_start + p];
52        size_t bin_index = (current >> log_divisor) - div_min;
53        size_t swap_location = input_start + offsets[bin_index];
54        input[input_start + p] = input[swap_location];
55        input[swap_location] = current;
56        ++offsets[bin_index];
57      }
58    }
59
60    if (sorted_bin_count >= bin_count - 1) {
61      break;
62    }
63  }
64
65  // Return if all digits have been sorted
66  if (!log_divisor) {
67    return;
68  }
69
```

```
70    // Recursively sort partitions
71    size_t max_count = get_min_count<int_log_mean_bin_size,
        int_log_min_split_count, int_log_finishing_count>(log_divisor);
72
73    size_t next_partition_start = input_start;
74    for (size_t i = 0; i < bin_count; ++i) {
75      size_t next_partition_end = input_start + final_offsets_cache[cache_offset +
          i];
76      size_t next_partition_size = next_partition_end - next_partition_start;
77      if (next_partition_size <= 1) {
78        next_partition_start = next_partition_end;
79        continue;
80      }
81
82      if (next_partition_size < max_count) {
83        boost::sort::pdqsort(input + next_partition_start, input +
        next_partition_end);
84      } else {
85        my_spreadsort_ex<T>(input, next_partition_start, next_partition_end,
        offsets, final_offsets_cache, cache_end);
86      }
87
88      next_partition_start = next_partition_end;
89    }
90  }
91
92  template <typename T>
93  static void my_spreadsort(T *input, size_t count)
94  {
95    constexpr size_t offset_count = 4096;
96    size_t offsets[offset_count] = { 0 };
97    std::vector<size_t> final_offset_cache = {};
98    my_spreadsort_ex(input, 0, count, offsets, final_offset_cache, 0);
99  }
```

Listing 1: Full code listing for integer spreadsort with Ska Sort's permute stage partition iteration method. Function implementations that can be found in the Boost library such as is_sorted_or_find_extremes, get_log_divisor, and boost::sort::pdqsort are not mentioned in the code listing.