20332400

Sorting Algorithms using SSE

This document will explain the two algorithms I have implemented using Streaming SIMD (Single Instruction, Multiple Data) Extensions (**SSE**). The two algorithms I wrote were Sample Sort and Bitonic Sort.

Sample Sort

Theory

Sample sort is a sorting algorithm that splits a list into separate “buckets” by choosing a sample of “splitters” to define buckets. Each bucket is filled with values greater than the splitter value below them but less than or equal to the one above them, but aren’t necessarily themselves sorted. These buckets are then recursively divided until they reach a threshold value, at which point they are sorted using an algorithm better suited for smaller values, such as insertion sort. Sample sort is similar to Quicksort, which instead chooses a single pivot value to divide the algorithm into, where values are either less than or greater than the pivot.

Algorithm

Firstly, define the number of buckets to be used. As we are using SSE vectors, which can store 4 floats, it makes sense to use 4 splitters; therefore, we will use 5 buckets. Secondly, define the oversampling ratio o.

The oversampling ratio will pull more samples than there are splitters, in order to create a better distribution of splitters. For datasets with a larger overall distribution, a smaller ratio is preferred, and a larger ratio is preferred for a smaller distribution.

Next, if the size divided by the oversampling ratio is below a certain threshold, sample sort switches to a better sorting algorithm for sorting small values. In this case, insertion sort was used when the size dropped below 20.

The samples are generated randomly in this algorithm. They are then sorted, and each o value is chosen to be the splitters. These values are loaded into an SSE vector, and the buckets are allocated. Extra helper arrays are also allocated, storing the size and capacity of each array to allow for dynamic expansion. Then, the equality buckets are allocated.

If a value happens to appear more than once in the dataset, and the duplicate is one of the splitters, an equality bucket can be used to save on unnecessary recursion. All

equality buckets are sorted with the value of their splitters, and are added to the output without another recursive function call. This can help in the event that there are many similar values, but can hinder performance when there are few duplicates. Similar reasoning can be applied to choosing an oversampling ratio,

For each value in the array, the same value is loaded into all four lanes of a single SSE vector, using `_mm_set1_ps`. This vector is then subtracted from the splitters vector using `_mm_sub_ps`, and a sign mask of the result is retrieved with `_mm_movemask_ps`. The Hamming weight of this value – the sum of all its bits – will then determine which bucket the value goes in. If a bucket is full (its size is equal to its capacity), the bucket containing it is reallocated more space using `realloc` and the capacity is increased.

If the value is instead equal to one of the splitters, it is added to its respective equality bucket.

Afterwards, the main buckets are recursed through and all buckets are concatenated into a single array using `_mm_loadu_ps` and `_mm_storeu_ps`, which is then returned.

Pseudocode

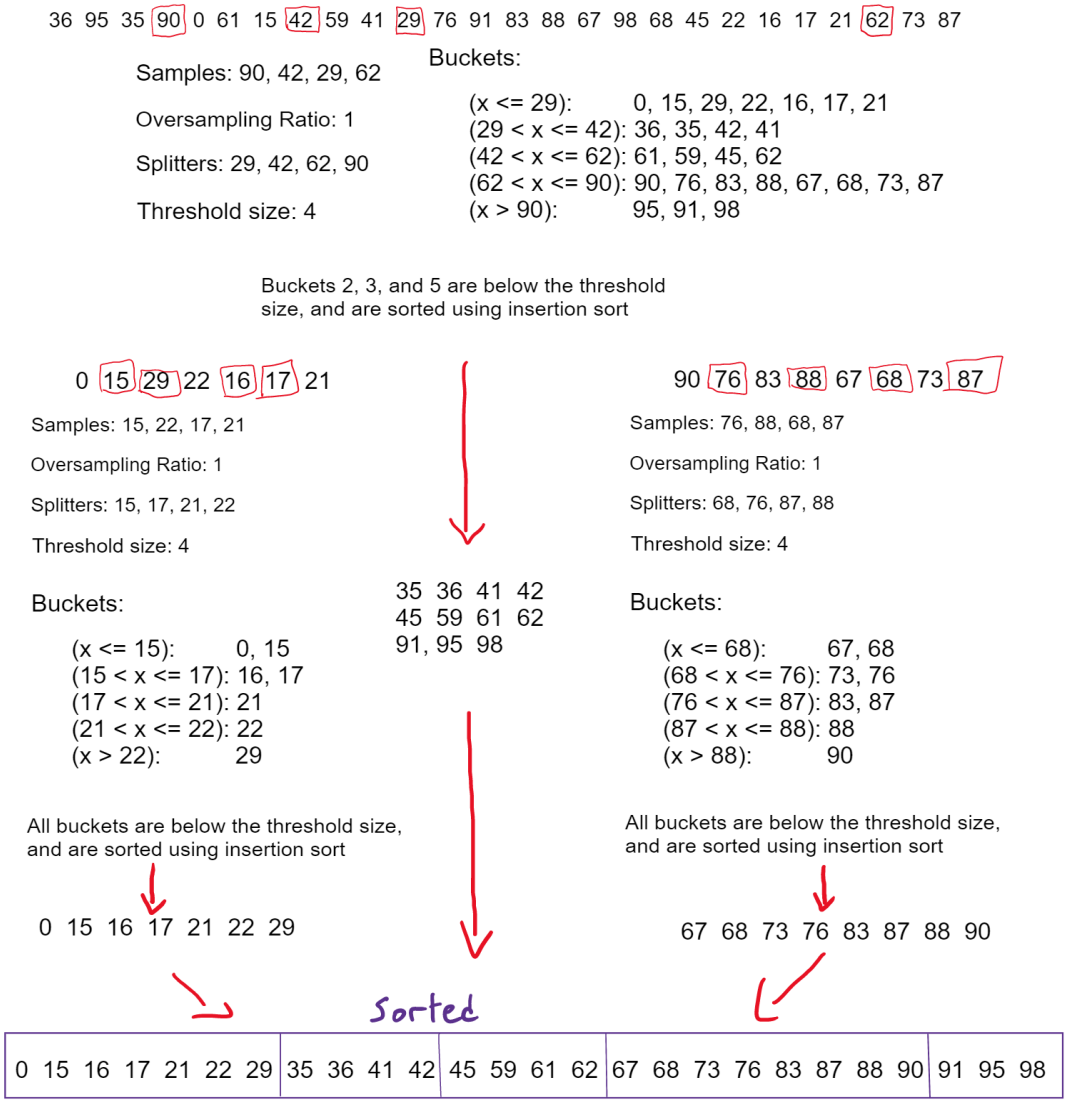
[1]
(This code is 0-indexed.)
N = size
o = oversampling ratio
p = number of buckets

```
func sample_sort(A, o, p):
    N = A.size
    // switch to other sorting algorithm when average bucket size is small enough
    if (N/o <= threshold_value):
        other_sort(A)
    samples = [a0, a1, ..., a(p-1)k] // randomly select (p-1)k samples
    sort(samples) // sort the samples
    // select splitters from samples
    splitters = vector[samples[k], samples[2k], ..., samples[(p-1)(k)]]

    buckets = [p][] // create p buckets

    for vl in A:
        if v1 in splitters:
            V = vector[vl, vl, vl, vl] // load vl into all lanes of a SIMD vector
            sub = vector_sub(splitters, V) // subtract V from the splitters
            mask = sign_mask(sub) // obtain a sign mask from the subtraction result
            idx = hamming_weight(mask) // calculate the Hamming weight of the mask
            append(buckets[idx], vl) // add vl into the correct bucket
       else:
            zidx = find(splitters, v1)
```

```
            append(eq_buckets[zidx], v1)

    // concatenate buckets and return
    return concat(sample_sort(buckets[0]), eq_buckets[0], sample_sort(buckets[1]),
eq_buckets[1], ..., eq_buckets[p-2], sample_sort(buckets[p-1]))
```

Graph of Algorithm

The use of equality buckets is not shown below for simplicity.



Times

Times are an average of five tests and are timed in milliseconds, with N = 100,000 and an oversampling ratio of 1.

Sort order	Sample Sort (ms)	Quicksort (ms)
Randomised	62.1598	34.8084
Same value	3.3742	1.0183
Increasing/Ordered	46.4184	10.058
Decreasing/Reversed	47.6693	13.0804

Bitonic Sort

Theory

Bitonic sort is a sorting network that splits its values into “bitonic sequences”, which are sequences of monotonically increasing and then decreasing sequences of values. These sequences are then recursively merged by comparing values at gaps of the largest power of 2 smaller than the length of the array. The algorithm has a similar structure to merge sort.

Sorting networks are only really valuable for small N, such as N < 50. This made using SSE vectors difficult, as there isn’t always enough space to load 4 values from the array at a time. Additionally, bitonic sort typically requires N to be a power of 2, though that isn’t a requirement for this algorithm.

Algorithm

Firstly, the array is split into 2. The function recurses on both halves, with one in an increasing order and the other in a decreasing order. Both recursions have a `lo` index, which defines which section of the array to sort. On the increasing recursion, `lo` is the start of the second half of the array, where it is the start of the array on the decreasing recursion. Then, the `bitonic_merge()` is called on the current recursion.

The largest power of 2 less than `N` is calculated. This value, `p`, will serve as the gap between the index of the current value, `i`, and the value to be swapped with.

These two values are used as the addresses to load the next 4 values using `_mm_loadu_ps`. The minimum and maximum values from these vectors are calculated into two more vectors using `_mm_min_ps` and `_mm_max_ps` respectively.

It is important to note that this only happens when `p` is large enough to accommodate loading two separate vectors. That is, since `p` is the gap between the two vectors' starting indices, SSE intrinsics will only be used when the gap is greater than or equal to 4, the width of an SSE vector.

If the direction for this recursion is increasing, then the minimum vector is stored at index `i`, and the maximum vector at index `p`. Otherwise, the vector locations are reversed. Then `bitonic_merge()` is called twice more, using `p` as the size for the first function call and `N-p` for the size and `lo+p` for the starting value for the second call. This is repeated without using vectors for any leftover values.

Pseudocode

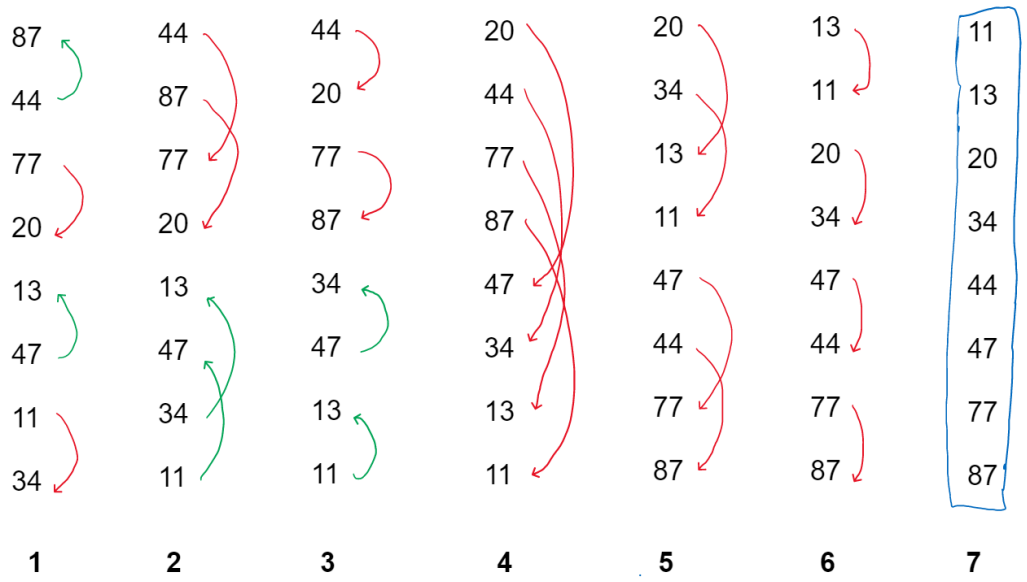
[2]
(This code is 0-indexed.)

```
func bitonic_sort(A, N):
    increasing = true
    bitonic_helper(A, N, 0, increasing)

func bitonic_helper(A, N, lo, dir):
    if (N > 1):
        m = N/2
        bitonic_helper(A, m, lo, !dir)
        bitonic_helper(A, N-m, lo+m, dir)
        bitonic_merge(A, N, lo, dir)

func bitonic_merge(A, N, lo, dir):
    if (N > 1):
        p = largestPowerOf2LessN(N)

        for i = 0 -> N:
               ma = vector_load[A[i]] // load values at i, i+1, i+2, i+3
               mb = vector_load[A[i+p]] // load values at p, p+1, p+2, p+3
               min = vector_min(ma, mb)
               max = vector_max(ma, mb)

               if (dir):
                      // ascending
                      vector_store(A[i], min)
                      vector_store(A[i+p], max)
               else:
                      // descending
                      vector_store(A[i], max)
                      vector_store(A[i+p], min)

               bitonic_merge(A, p, lo, dir)
               bitonic_merge(A, N-p, lo+p, dir)
        // then deal with leftovers
```

20332400

Graph of Algorithm

The below is an example of sorting 8 values using bitonic sort.



At step 4, the values have all been sorting in bitonic order. The red arrows represent an ascending comparison (swapping if the value at the arrow is lower) and green arrows represent descending comparisons.

Times

Times are an average of five tests and are timed in milliseconds, with N = 32.

Sort order	Bitonic Sort (ms)	Quicksort (ms)
Randomised	58.1028	.0072
Same value	.0516	.0264
Increasing	.0022	.0024
Decreasing	5.3664	2.7892

Conclusions

From these tests, it is clear that quicksort is a superior sorting algorithm to both sample sort and bitonic sort in most situations. Sample sort is outperformed by quicksort, even when given solely duplicate values, in spite of its implementation of equality buckets.

Something to note is that the added equality buckets actually slowed the algorithm down roughly 50%. However, the removal of equality buckets results in a very deep recursion cycle, as all values would be placed in one quickly expanding bucket. Sample sort may be more useful when used in parallel processing, allowing each processor to take control of a bucket, but when only vectorised, it doesn't have much of an advantage.

Similarly, bitonic sort, while better suited for small quantities of values, is still outperformed by quicksort in nearly every category. Its quickly growing time complexity makes it a terrible choice for even moderately sized sets of data.

The implementation of the used bitonic sort algorithm is something to take into consideration. In it, I used SSE intrinsics to utilise vectorisation, but they weren't used as efficiently as possible. In fact, when testing how many times vectors were actually used – which was previously stated to have a requirement of an index gap of 4 between swapped values – it was underutilised extremely heavily. A random sample of N = 16 shows it using SSE vectors an astonishingly low 14 times, whereas regular swapping was used over 6000 times.

This is something that could be fixed by reducing the gap required to use SSE vectors. If there was ever an overlap less than 4, swapping only the first 4-p values of the first vector and the last 4-p values of the second vector would still allow for SSE vectors to be used. However, I felt that this was an unnecessary addition to the algorithm as it would make it needlessly complex.

In general, quicksort has certainly earned its name, as it outperforms even situation specific sorting algorithms with ease. While better implementations of sample sort and bitonic sort can definitely be achieved, especially with better use of SSE intrinsics, quicksort stands to be one of the most useful sorting algorithms in the C language.

Citations

1. https://www.inf.hs-flensburg.de/lang/algorithmen/sortieren/bitonic/oddn.htm, for creating the bitonic sorting algorithm for arbitrary N.
2. https://en.wikipedia.org/wiki/Samplesort#Pseudocode, for sample sort pseudocode.